FORMAL TECHNIQUES FOR CYBER RESILIENCE: MODELING, SYNTHESIS, AND VERIFICATION

by

Mohammed Noraden Alsaleh

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2018

Approved by:

Dr. Ehab Al-Shaer

Dr. Bei-Tseng Chu

Dr. Jinpeng Wei

Dr. Mohamad-Ali Hasan

ABSTRACT

MOHAMMED NORADEN ALSALEH. Formal Techniques for Cyber Resilience: Modeling, Synthesis, and Verification. (Under the direction of DR. EHAB AL-SHAER)

As cyber vulnerability and complexity increases, cyber-attacks become highly sophisticated and inevitable. Therefore, cyber resilience is necessary to make cyber capable of misleading attackers in reaching their goals, resisting their progress, and mitigating the consequences by timely responding to attack activities and preserving the mission integrity. Cyber resilience includes various techniques such as isolation/segmentation, diversity, deception, adaptive response, and others. This dissertation focuses on addressing many challenges that limit the effectiveness and deployment of these four key resilience techniques. First, the lack of theoretical foundations that allow for formulating and integrating isolation and diversity limits the capability of optimizing resilience by composition. Second, the initiation of many mitigation actions simultaneously requires techniques to support safe and efficient courses of action (CoA) orchestration to guarantee correct and consistent defense actuation while allowing for the maximum concurrency. Third, the lack of automated planning techniques for cyber deterrence and deception based on malware code significantly limits the effective deployment of these techniques against such innovative attacks. In this thesis, we address each of these challenges in three chapters as will be described below.

In the second chapter of this dissertation, we provide a formal synthesis framework that automatically generates a resilient network configuration, integrating available software diversity and isolation measures to meet user-defined cyber risk and budget constraints. We provide a formal specification for two key resilience techniques: *isolation* that defines the network access control and countermeasures between services, and *diversity* that assigns many software variants to network services. In our model, we consider the interdependence between isolation and diversity to maximize

the impedance of the attack propagation and optimize cyber resilience. The isolation and diversity configurations are computed according to estimated risk after considering all possible attack paths to network assets, and all potential software variants and countermeasures in each path. To make our approach scalable to large network sizes, we developed model reduction and network decomposition techniques and evaluated our framework using networks of thousands of nodes.

In the third chapter, we address the safety and efficiency of Active Cyber Defense (ACD) policies that initiate courses of investigation and configuration actions to mitigate attacks automatically. We present a formal specification for ACD policies and develop formal techniques and algorithms to maximize concurrency of action execution while guaranteeing that defense actions are conflict-free, executed correctly, and satisfying the mission requirements. We model and verify the CoA orchestration using satisfiability modulo theories, and bounded model checking.

In the fourth chapter, we present a new analytical framework to analyze the malware behavior and extract the agility parameters using symbolic execution to enable automated planning of deterrence and deception. The agility parameters are system variables on which attackers depend to discover the target system and reach their goals; Yet, they can be reconfigured or misrepresented by the defender in the cyber environment to mislead attackers or significantly increase the attack cost. We first develop a symbolic execution engine to execute Microsoft Windows malware and characterize their attack behavior based on their interactions with the environment. We then analyze the attack behavior to extract the set of agility parameters that can deliver effective deterrence and deception based on well-defined criteria. Our analysis of many recent malware instances shows that our framework has successfully identified various critical parameters that are effective for cyber deterrence and deception.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Ehab Al-Shaer, for his support, patience, and encouragement throughout my doctoral studies. His deep knowledge, interests, and passion towards research in network and system security make him an exemplary researcher and mentor.

I would also like to express my sincere gratitude to my dissertation committee members: Professor Bill Chu, Professor Jinpeng Wei, and Professor Mohamad-Ali Hasan for serving on my dissertation committee and their precious comments and suggestions. I would particularly like to single out Professor Jinpeng Wei for his excellent cooperation and support throughout the design and implementation of our malware analysis project. Besides, I would like to thank the faculty members and the staff of the Department of Software and Information Systems in the University of North Carolina at Charlotte, my fellow graduate students, and my research collaborators for their academic and administrative support throughout my doctoral study.

I would also like to acknowledge the NSA Science of Security Lablet community for their inspiring comments and suggestions during my involvement in the Lablet. Special thanks to Professor Laurie Williams, North Carolina State University, and Steve Danko and Ahmad Ridley, NSA, and my colleague in the Lablet Md. Mazharul Islam.

Last, but not least, my most profound gratitude goes to my parents, my wife Duaa Alansari, and my two kids, Malak and Yamin, for their great encouragement, understanding, and love during the past few years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The recent incidents of data breaches, such as the attacks on SnapChat, Yahoo, the SWIFT system, and others reported in Verizon Data Breach Investigation Reports [1, 2] emphasize the fact that cyber breaches have become inevitable. This requires high assurance of resilient defense techniques that go beyond the attack detection and prevention to include mitigating attacks and managing their consequences after being partially or entirely successful. In addition to fortifying the system assets, this includes cyber deception, incident response, and recovery mechanisms. Many different techniques have been proposed to accomplish this and enforce cyber resilience. However, the lack of formal modeling and analysis to compose, verify, and synthesize cyber techniques limits their effectiveness significantly.

1.1 Cyber Resilience

The concept of resilience has been adopted by and extended to various domains, including systems engineering [3, 4, 5, 6], software engineering [7, 8, 9, 10], and recently cyber security domain [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. As a result, it has been defined in various ways based on the domain and what needs to be resilient. The word "Resilience" is defined as "*an ability to recover from or adjust easily to misfortune or change*" by Merriam Webster dictionary. In control systems, a system is resilient if it "*maintains state awareness and an acceptable level of operational normalcy in response to disturbances including threats of an unexpected and malicious nature*" [21, 22]. In [12], the resiliency of networks is defined based on the University of Kansas ResiliNets project as "*the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation.*" The resiliency of

a cyber infrastructure system is defined in [11] as "*the ability of the system to recover from a cyber intrusion and to assume close to normal operations within an acceptable time and at an acceptable total encompassing cost.*"

The scope of this dissertation covers cyber networks and we follow MITRE's definition [23], which defines Cyber Resilience as "*The ability of a nation, organization, or mission or business process to anticipate, withstand, recover from, and evolve to improve capabilities in the face of, adverse conditions, stresses, or attacks on the supporting cyber resources it needs to function.*". This definition broadly incorporates the four goals of cyber resilience: *anticipate* (i.e., to maintain a state of informed preparedness in order to forestall compromises of mission functions from adversary attacks), *withstand* (i.e., to continue essential mission functions despite successful execution of an attack by an adversary), *recover* (i.e., to restore mission functions to the maximum extent possible subsequent to successful execution of an attack), and *evolve* (i.e., to change mission functions and the supporting cyber capabilities, so as to minimize adverse impacts from attacks). In this dissertation, we investigate the resilience techniques related to the goals of *withstand* and *recover* in the face of cyber-attacks.

There is a wide spectrum of resilience techniques that can contribute to one or more of the cyber resilience goals. These techniques have been categorized by MITRE [12, 23] into the following major categories:

- Isolation/Segmentation. The isolation technique aims at separating the dubious components from the critical ones in a cyber network to reduce the exposure of the system's critical assets and limit the damage from potential exploits [24, 25, 26, 27, 28]. The isolation might be performed on different layers of network components, either by physically isolating network assets, creating virtual islands for conducting sensitive operations through well-established virtualization technologies, or providing appropriate access control and attack

countermeasures, such as firewalls, intrusion detection and prevention systems, and VPN gateways, in the networking infrastructure.

- Diversity. The diversity technique is implemented by using heterogeneous technologies for the different components, such as firmware, operating systems, and software programs, instead of using the same technology [29, 30, 31, 32]. Diversity is a well-known practice for resilience [12, 33] that is used to limit the impact of known exploits and disrupt the attackers by forcing them to learn new exploits and attack multiple technologies.

- Adaptive Response. The adaptive response is the practice of taking cyber actions in response to security events that reflect ongoing attacks in the cyber network. This technique requires self-awareness of the system's operational state and potential attack indications. The objectives of the adaptive response include containing the damage, maintaining the critical services in the system, and recovering to an acceptable state. This process is normally governed by Active Cyber Defense (ACD) policies that select and execute the appropriate Course of Action (CoA) based on the received security event [34, 35, 36, 37, 38, 39, 40].

- Coordinated Defense. Practicing this technique requires managing and coordinating multiple defense mechanisms against different attack activities, such as multi-layer defense systems and reactive policies [35, 37, 41]. There are two objectives for these techniques. First, it covers different attack activities and potential attack patterns, which makes it harder for the attacker to successfully attack critical assets because she needs to defeat multiple defense mechanisms. Second, it ensures that one defense mechanism does not introduce adverse consequences by interfering and conflicting with another mechanism.

- Deception. Deception techniques misrepresent the network configuration and parameters to confuse the attacker. Deception can be used for various objec-

tives: distorting the attacker's reconnaissance process and making her uncertain about the network assets, depleting her time and resources pursuing false information or exploiting decoy assets, and discovering her attack tactics and techniques and, potentially, her identity and attack intents. Examples of deception techniques in literature include honey-nets and honeypots [42, 43, 44], mimicking and decoy technologies [45, 46].

- Deterrence using Dynamic Positioning. This technique dynamically mutates network configuration and relocates critical assets promptly to make it hard for the attackers to locate them, introduce uncertainty about the targeted system, and force the attacker to spend more time and effort understanding the environment and locating critical assets [47, 48, 49, 50]. It is essential for these techniques to apply randomness, such that it is hard for attackers to discover mutation patterns and anticipate the new locations.

- Other Techniques. There are many other techniques to achieve resilience objectives [23], including redundancy [51, 52, 53], Non-Persistence [54, 55, 56], Substantiated Integrity [57, 58, 59], and Privilege Restriction [60, 61]. These techniques are outside the scope of this dissertation, and we will only define them briefly. Redundancy techniques introduce back up components for critical services to maintain their operation if the original ones are disrupted. Non-Persistence techniques refresh the critical information and services periodically, such that if they are exploited, attackers cannot establish persistent foothold. Substantiated Integrity techniques continuously verify that critical information and services are not corrupted by performing integrity checks. Finally, Privilege Restriction (i.e., least privilege) minimize the exposure of critical assets using fine-grained access control and trust-based privilege assignments.

In this dissertation, we directly address many challenges that limit the effectiveness

Table 1.1: Mapping Resilience Techniques to Broad Attack Phases

Resiliency Technique	Pre-Attack	During-Attack	Post-Attack
Isolation/Segmentation	★		
Diversity	★		
Adaptive Response	★	★	★
Coordinated Defense	★		
Deception	★	★	
Deterrence (Dynamic Positioning)	★	★	

and deployment of isolation, diversity, adaptive response, deception, deterrence using dynamic positioning, and we touch base on coordinated defense as will be shown in the following chapters. These techniques serve multiple resiliency goals, and they can be effectual during multiple phases of attack kill-chain [62]. We classify these techniques based on the attack phases, at which they can be effective, into the three broad phases: *pre-attack*, *during-attack*, and *post-attack*. Our classification is shown in Table 1.1.

1.2 Motivation: Cyber Resilience Challenges

Cyber resilience techniques objectives include misleading attackers in reaching their goals, resisting their progress, and mitigating the consequences by timely responding to attack activities and preserving the mission integrity. However, there are many key challenges that limit their effectiveness and deployment. We discuss these challenges in the following:

- Lack of formal frameworks for cohesive integration and composition of isolation and diversity resilience techniques. Although there has been a significant body of research for automated synthesis of isolation and diversity techniques, existing works study each technique independently and not as an integrated defense system. Therefore, they do not provide the necessary modeling and specification to characterize the interdependence between them. To devise cost-efficient resilient configurations integrating both diversity and isolation, we need a uni-

fied model to assess their decisions and impacts with respect to overall resilience of the system against cyber-attacks.

- Adaptive response can jeopardize the network mission. Automation of Active Cyber Defense (ACD) policies and techniques poses serious problems that might jeopardize its effectiveness. Active cyber defense techniques execute certain courses of action in response to security events and incidents. As a result, multiple courses of action, which operate on shared or interdependent cyber resources, may be triggered and executed concurrently. This can lead to conflicting actions competing over the same resources. Besides, ACD actions can change the network configuration state, possibly after each action. It is crucial to ensure that the network mission requirements are not violated during, as well as, after the complete execution of CoA. Thus, there is a need for provable orchestration models that can execute the ACD courses of action safely and efficiently, without posing high processing overhead.

- Lack of frameworks for automated deterrence and deception planning. Cyber deterrence and deception require careful planning to be effective because they are highly dependent on the attacker's intents and strategies. Deception techniques that are effective against specific attacks can be ineffective against others. Hence, these dynamic techniques need to be planned and dynamically adapted based on the attack behaviors. Due to the large number of malware captured every day, reliance on manual effort and human intelligence in planning and developing deterrence and deception techniques can dramatically slow down the response and limit its effectiveness. There is a need for automated tools that can analyze real malware instances and devise appropriate and effective deception and deterrence techniques.

1.3 Work Objectives

The broad objective of this dissertation is to ensure safe and effective planning, integration, and implementation of cyber resilience techniques. We model various types of resilience techniques, measure their effectiveness against cyber threats, and address their implementation challenges in order to accomplish the following specific objectives:

- Risk-aware Composition of Isolation and Diversity. We discussed in the previous sections the isolation and diversity techniques, which are typically planned and implemented independently. Isolation determines the placement of countermeasures in the network infrastructure, where diversity determines the assignments of software variants at the end-points (i.e., services). Hence, these two techniques are fundamentally interdependent and must be consistent for both the correct operation and the resilience of the network. If these two techniques are integrated correctly, they can provide multi-layer defense systems. The first objective of this dissertation is to answer the question of how isolation and diversity can be combined and how we can make the interdependence and correlation between them a leverage, rather than a liability, to provide the optimal resilience against propagating cyber-attacks.

- Efficient and Provably Safe Active Cyber Defense. Multiple active cyber defense strategies may take place in the same enterprise and execute concurrent courses of action on the same system. A naive implementation of such strategies can create conflicts that inhibit the proper response to security events or violate the network mission invariants. The second objective of this dissertation is to orchestrate the implementation of concurrent courses of action in such a way that (1) avoids the conflicts between the actions that might simultaneously operate on shared system resources, (2) ensures that one action does not invalidate the

conditions required to execute the others successfully, (3) fulfills the requirements of the network mission, and (4) minimizes the total time of executing the courses of action.

- Extraction of Critical Parameters for Agility Planning. Resilience through deterrence and deception requires effective planning of a sequence of mutations or misrepresentations of system parameters to steer the adversary according to the desired defense goals. Implementing such techniques depends on the accurate identification of the most appropriate parameters against specific attacks. The third objective of this dissertation is to provide a systematic framework that can automatically analyze given attack binaries and identify the system configuration parameters that can manipulate the decisions of the attacker for effective deception and deterrence.

- Cost-effective Cyber Resilience. This is the fourth objective of this dissertation and it goes along the three previous objectives as we explain in the following. First, a cost-effective composition of isolation and diversity takes into consideration the costs of software variants and isolation countermeasures to reduce the total composition cost while meeting the risk-reduction objectives. Second, cost-efficient ACD orchestration must minimize the total execution time of concurrent courses of action. Third, agility parameters extraction must consider the cost of using each available configuration parameter in deterrence or deception and select the ones that minimize the cost.

1.4 Research Challenges

To fulfill our research objectives, we need to address many research challenges. The major challenges are as follows:

- Modeling the interdependence between isolation and diversity. Isolation and diversity operate on different layers of network components, namely application-

level software and networking infrastructure configuration. Despite that, it is crucial for a consistent resilient configuration to model the interdependence between them and how the decisions of one technique can influence the decisions of the other. The major challenge in this regard is to find a common basis and a metric that can quantify the impact of these different techniques to weigh and priorities their decisions.

- Modeling advanced multi-step and multi-path cyber-attacks. As resilience techniques are more about managing cyber-attacks and their consequences after the initial breaches, we need to model multi-step attack behaviors, over multiple network paths, in order to generate an effective resilient configuration that takes into account all possible attack scenarios to critical network assets and evaluate the aggregate effect of isolation and diversity for each of these scenarios.

- Courses of action interdependence. Active cyber defense policies and techniques are not guaranteed to be independent. They might react to the same set of events and update a shared set of configuration and control variables. Moreover, there is no specific order of executing their actions, which makes it challenging to analyze them under multiple levels of dependencies.

- Model checking with arithmetic constraints. To model and verify QoS constraints as part of the network mission requirements, we need to develop a model checking approach that supports arithmetic theories. Arithmetic constraints can represent infinite sets, which makes traditional symbolic model checking approaches, such as Binary Decision Diagrams and SAT solvers insufficient to support them.

- Scalability. Networks may consist of thousands of interdependent devices and services. Therefore, it is challenging to model the complete configuration of such large-scale complex networks to synthesize resilient configuration or verify the

mission's requirements. For resilient diversity and isolation configuration synthesis, we need to model all the paths between connected devices and consider all possible combinations of diversity and isolation decisions. For the bounded model checking verification of the mission requirements with QoS and security constraints, we need to model new variables and track the satisfaction of these constraints in all possible paths between communicating devices.

- Customization of Dynamic Malware Analysis Tools. To identify the agility parameters for effective deterrence and deception techniques, we need to analyze attack binaries (i.e., malware). Malware developers use multiple programming strategies to evade detection and prevention techniques and avoid static and dynamic code analyses. The first challenge in this regard is selecting the toolset and the analysis environment that are appropriate for our agility-oriented malware analysis. The second challenge is extending these tools to intercept relevant interactions between the attacker and the system, study its dependency on the system parameters, and analyze the complete set of execution paths given the attack binaries without high-level specification or the source code of the attack. The third challenge is selecting the set of interactions that can successfully influence the decisions of the attacker to reveal her dependency on the environment and her perception of the targeted system.

1.5 Contributions

The key contribution of this dissertation is developing novel formal techniques and tools that can address the limitations in composing resilience techniques, orchestrating active cyber defense actions, and systematic planning of deception and deterrence schemes for resilient cyber systems. We address the research challenges mentioned above and contribute to the efforts of cybersecurity to become cyber resilient through the following specific contributions:

- Automated configuration synthesis for composing isolation and diversity techniques. We develop a new formal synthesis framework to generate a consistent configuration for a given cyber network that employs both diversity and isolation resilience techniques. We model the interdependence between all possible decisions and find a configuration exhibiting harmony of software diversity and isolation countermeasures that reduces the risk of spreading cyber-attacks without exceeding certain budget constraints.

- Model reduction to improve the scalability of isolation and diversity composition. Given the increasing complexity of cyber networks, which makes formal configuration synthesis a challenge, we developed two model reduction techniques to scale our synthesis framework for large networks. In one technique we leverage the fact that some isolation countermeasures are bidirectional in reducing the decision space because the same decision will be applied to traffic flowing in both directions between a pair of communicating services. In the other technique, we decompose the network and synthesis a resilient configuration for the smaller sub-networks. We then combine them into one configuration and prove that it is a valid resilient configuration for the whole original network.

- Formal modeling for active cyber defense policies. We present a formal model for ACD policies that describes the organization and execution modes (i.e., concurrent or sequential) of ACD courses of action and the fine-grained components of their cyber actions.

- Safe and Efficient CoA orchestration. We define a set of formal properties that guarantee conflict-free execution of concurrent courses of action. Then, we develop a novel formal synthesis framework that enforces these properties and orchestrates the implementation of CoAs, such that all the actions are executed successfully in the minimum possible time.

- Model checking for mission requirements verification in OpenFlow-based Software Defined Networks. We provide a bounded model checking framework to verify that network data plane updates caused by the CoAs of ACD policies do not violate the network mission requirements at any point during their execution.

- Agility-oriented dynamic malware analysis. We develop a dynamic analysis framework based on symbolic execution engine to execute malware, intercept its interactions with the environment, and analyze the execution traces to extract the environment parameters that can invalidate the attacker's assumptions and perceptions of the system and mislead her according to deterrence and deception goals. We then select the agility parameters that constitute consistent, resilient, and cost-efficient deterrence or deception. That is, it does not reveal the deterrence or deception scheme by neglecting interdependence between system parameters, covers all possible execution paths and potential evasion maneuvers of the attacker, and minimizes the changes and investments required at the defender side, respectively.

1.6 Background

1.6.1 Satisfiability Modulo Theories (SMT)

In the past decade, Boolean formal methods (e.g., SAT and Binary Decision Diagrams [63, 64, 65, 66, 67, 68, 69]) has been used successfully in network security analysis, especially for verifying security policy, which is defined as a sequence of propositional logical constraints. However, due to the increasing complexity of network security and business requirements, Boolean propositional logic constraints are not suitable to develop security and resiliency analytics that requires verifying complex arithmetic properties. SMT has been developed to overcome this shortcoming by offering various theories that efficiently deal with integers, real numbers, arrays, un-

interpreted functions, linear and non-linear arithmetic, etc. In addition, SMT solvers provide a much richer formal modeling platform compared to SAT solvers.

SMT solvers have been shown to be powerful tools in solving constraint satisfaction problems that arise in many diverse areas including software and hardware verification, type inference, extended static checking, test-case generation, scheduling, planning, graph problems, etc. [70, 71, 72, 73, 74]. An SMT instance is a formula in first-order logic, where some functions and predicate symbols have additional interpretations according to the background theories. SMT is the problem of determining whether a formula is satisfied or not. SMT solvers are efficiently applied in solving large and complex problems. It has been shown that modern SMT solvers can check formulas with hundreds of thousands of variables, and millions of clauses [75]. In this research, we primarily use theories of integers, real numbers, and linear and non-linear arithmetic for modeling, and we find that our models are highly efficient in verifying resiliency metrics and techniques.

1.6.2 Bounded Model Checking

The Model Checking is the problem of deciding whether a certain property ϕ holds on all the paths of a transition system that is normally specified as the Kripke structure M (formally, $M \models \boldsymbol{A}\phi$). Looking from a different perspective, systems can be verified by checking the existential problem $M \models \boldsymbol{E}\neg\phi$, which decides on the existence of a path in which the property ϕ does not hold. The existence of such path means that the system does not satisfy the property (i.e., $M \models \boldsymbol{E}\neg\phi \rightarrow M \not\models \boldsymbol{A}\phi$).

In bounded model checking, we look for property violations in a bounded version of the system (i.e., $M_k \models \boldsymbol{E}\neg\phi$). In the bounded system M_k, only execution paths with at most k transitions are considered. If a violation is discovered in the bounded version, this also proves that a violation exists in the complete system (i.e., $M_k \models \boldsymbol{E}\neg\phi \rightarrow M \models \boldsymbol{E}\neg\phi$). The bounded existential problem can be reduced to a satisfaction problem by encoding the system along with the property as a quantifier

free formula that is satisfied if a violation exists within the bounded model. The satisfiability of the quantifier free formula can be determined using state of the art SMT solvers, such as Z3 [76], YICES [77], or CVC4 [78].

1.6.3 Symbolic Execution with S^{2}E Platform

Many threat analysis techniques are used to analyze and detect potential threats by understanding the attack behavior based on formal attack description [79, 80, 81, 82]. Static analysis tools study the source code or the binary execution files of the attack. However, in many cases, the source code may not be available, and attackers often use some binary obfuscation techniques [83] to hide the malicious behavior, which makes it hard to detect them statically. To overcome these challenges, other malware analysis techniques use dynamic analysis by running the attack executable files in a controlled environment and capturing the execution traces [84, 85, 86]. The traces are then analyzed to look for patterns of malicious behaviors. Both static and dynamic malware analyses have been traditionally used to build behavioral profiles for malware, often represented by system call graphs, to detect new malware that is likely to exhibit the same behavior.

The idea of symbolic execution was first introduced in [87] as a means for program testing and debugging. The intuition of the symbolic execution is to use symbolic values for the program inputs instead of their actual values. As a result, the program's internal variables will be described as symbolic expressions in terms of the symbolic inputs. The symbolic execution platforms keep track of logical expressions for all the possible execution paths in a program, often referred to by path constraints. The path constraints are conditions in terms of symbolic inputs that need to hold for the execution to follow the corresponding path. A new execution path is forked for each outcome of the conditional statements in terms of the symbolic inputs.

The Selective Symbolic Execution (S^{2}E) [88] is one open-source platform that can be used for symbolic execution, and it allows developers to write tools for analyzing

the behavior of software systems. It provides the means to install software systems in controlled environments, symbolically execute them, and analyze their execution traces. S^2E utilizes the idea of selective symbolic execution, which automatically minimizes the amount of code that must be executed symbolically given a target analysis. This enables the simultaneous analysis of entire families of execution paths, instead of just one execution path at a time. S^2E provides multiple ways to define symbolic inputs and variables based on the targeted software. If the source code of the software is available, the code can be instrumented using a particular API to mark symbolic variables programmatically. If the code is not available, S^2E provides a plugin that can be used to mark specific memory locations or CPU registers as symbolic at execution time.

1.7 Overview of the Technical Approach

We present in this section a high-level description of our technical approaches to pursue our research objectives.

1.7.1 Automated Synthesis of Isolation and Diversity Configuration Composition

The automated generation of resilient network configuration requires accurate and precise modeling of the isolation and diversity resilience techniques. In this work, we model the network as a set of services connected by virtual links. A virtual link between any two services may consist of multiple physical links in the underlying networking infrastructure (i.e., between hosts, switches, and routers). Multiple software variants can fulfill each service. Software variants of the same service may be different software (e.g., developed by different vendors) or different implementations of the same software. Different software variants have different attack surfaces or exploitability scores [89, 90, 91, 92, 93, 94] and different costs. We assume that multiple isolation countermeasures, such as firewalls and intrusion detection systems, can be placed at any virtual link in the network. The countermeasures may have different

effectiveness for different software variants. This means that the effectiveness of the same countermeasure may vary based on the software variant selected for the targeted service. Besides, different countermeasures usually have different costs.

Our objective is to find the optimal software variants assignments (i.e., which software variants are assigned to services) and isolation countermeasures placements (i.e., which isolation countermeasures are assigned to virtual links), such that certain cyber risk and budget thresholds are not exceeded. The total cost is merely the sum of all the selected software variants and isolation countermeasures. We define a metric to measure the cyber risk of the propagating attacks to drive our resilient configuration synthesis. We model the risk imposed on a target service considering the following criteria: the *exploitability* score of the selected software variant for the target service, the *similarity* between the software variants of the target and the source services, and the *effectiveness* of the isolation countermeasure implemented in between. We consider the similarity between their software variants a factor of risk because similar variants are likely to exhibit the same vulnerabilities. Hence, attackers might learn vulnerabilities of one software variant by exploiting another. Our risk model considers all the attack paths that lead to network services. If different attack paths impose different risks on the service, we take the most conservative option and make our decisions based on the one of the maximum risk (i.e., least attacking effort [95]). Moreover, we consider direct and indirect reachability, such that an attack path consists of a sequence of attack hops, where different software variants are selected for different hops and different isolation countermeasures are deployed at each attack step.

We formalize the problem of software variants assignment and isolation countermeasures placements as a constraints satisfaction problem. The decision variables of the problem determine which software variant should be selected for each service and which countermeasure should be placed at each virtual link. Based on that, we

encode the constraints required to calculate the risk imposed at each service based on diversity and isolation decision variables. In addition, we encode the constraints to calculate the cumulative risk and cost for each possible configuration. We implemented this problem utilizing the Z3 SMT solver that will select the configuration that satisfies the given risk and budget constraints. We evaluated our solution for networks of up to 600 services and attack paths of 8 hobs. However, we experienced insufficient memory space problems when exceeding these number. To solve this scalability issue, we developed model reduction and network decomposition algorithms. In the model reduction algorithm, we leverage the fact that some countermeasures are bidirectional. Hence, if a bidirectional countermeasure is selected for one direction of communication between a pair of services, we do not need to consider the other possible countermeasures for the opposite direction of communication. This reduces the number of possible solutions and enhances the performance of our framework. In the network decomposition algorithm, we split the network into two portions, where we ensure that this decomposition does not invalidate our assumptions and requirements to calculate accurate risk scores. We present a proof that a solution to the problem after decomposition is also a solution to the original problem if the whole network is considered as one unit. This improves the scalability of our synthesis approach to accommodate networks in the order of thousands of nodes.

1.7.2 Provably Safe and Efficient Course of Action Orchestration for Active Cyber Defense Policies

For safe and efficient deployment of Active Cyber Defense (ACD) policies, we present a formal specification for them, and we develop formal techniques and algorithms to maximize concurrency of actions execution while guaranteeing that they are conflict-free, executed correctly, and do not violate the mission requirements. The ACD policies consist of rules that associate specific security events and incidents (e.g., infected hosts, flooded links, and detecting untrusted communication sessions)

to courses of configuration and investigation actions that mitigate the potential consequences of the events. A Course of Action (CoA) is an ordered set of cyber actions (commands) organized for parallel or sequential execution. Each action specifies a cyber command (e.g., block traffic, migrate a virtual machine), an actuator that will carry the execution of the action (e.g., a switch or a virtualization controller), an object (e.g., particular traffic flows or virtual machines), and it can optionally have pre- and post-conditions that are expressed as arithmetic logic formulas in terms of *control variables*. The values of the control variables represent the dynamic state of the network attributes, such as servers' capacities or links utilization. Thus, the pre-condition describes the system's dynamic states that can lead to the successful execution of the action, such as whether sufficient capacity exists to migrate a VM. The post-condition reflects the system's dynamic state after performing the action (e.g., only 50 GB remain in the desk space of the destination server).

Since CoAs are interdependent and can influence the execution of each other due to sharing and operating the same set of objects or control variables, we identify a set of conflicts that might prevent the actions from being executed successfully within an acceptable time. We define formal properties to guarantee conflict-free, complete, and efficient execution of CoAs. The first property ensures that concurrent actions that share objects or control variables be executed sequentially. The second property ensures that cyber actions whose pre- and post-conditions overlap are composed such that the pre-condition of one action is satisfied by the post-conditions of the prior actions. The third property ensures the minimum execution time for all action of the actively scheduled CoAs. We developed an orchestrator that takes a set of CoAs and finds a Global Orchestrated CoA Workflow (GOAL) that determines the optimal time to execute each action to satisfy the three properties defined above. We leverage the optimization problems modulo theories provided in Z3 to implement our orchestrator. We evaluated the performance of the orchestrator, and we managed to orchestrate up

to 500 concurrent actions within 90 seconds.

While the orchestrator guarantees conflict-free, complete, and efficient execution of the actions, the CoA orchestration will not be safe without verifying that the network mission requirements are also satisfied during the execution of the ACD policy. For this purpose, we develop a specification language to define the network mission requirements as reachability, QoS, and security requirements. Then, we model the network data plane configuration along with the network mission requirements as SMT constraints and verify the satisfaction of the requirements using bounded model checking. We track the configuration updates after each action in the ACD policy and efficiently verify the satisfaction of the network mission requirements.

1.7.3 Automated Extraction of Agility Parameters for Cyber Deterrence and Deception Planning

For the automated creation of deception and deterrence resilience techniques, we analyze the malware behavior and extract candidate agility parameters that are necessary for planning effective deterrence and deception techniques. To execute the malware symbolically, we used a tool called Selective Symbolic Execution (S^2E) engine. S^2E provides a controlled environment to run binary code symbolically and supports a basic set of features, such as monitoring certain functions and tracing the execution, with the ability to develop custom plugins for various analysis objectives. Since our objective is to discover the attacker's dependence on the system parameters and since malware interacts with its environment through system and library API calls, we configured the (S^2E) through special scripts, written in LUA language [96], to intercept API invocations and mark the relevant output information as symbolic. Marking the symbolic information is crucial for the correctness of symbolic execution because they determine how the malware will follow different execution paths based on the decisions taken in terms of the symbolic information. We have selected more than 130 APIs that are commonly used by Malware, and we mark their return

and output arguments as symbolic. Further, we mapped these APIs to the system parameters that determine or influence the values returned by the APIs.

After instrumenting S2E with the appropriate scripts and plugins required for our agility analysis, we run the malware to build an attack behavior model that captures all the invoked APIs and the control and data dependency between them. We analyze this behavior model with respect to deception and deterrence by (1) identifying the execution paths that are relevant to the desired resilience technique (i.e., the path that can mislead the attacker for successful deterrence or deception), and (2) eliminating parameters that do not influence the attacker with respect to the desired technique. We end up with a set of relevant paths, each of which is populated with system parameters that represent the necessary conditions for the attacker to follow the path. We select a subset of those parameters that are *consistent* to preserve the integrity of the deterrence and deception from the attacker's point of view, resilient to provide deterrence and deception in all possible execution paths that lead to the desired goals, and *cost-efficient* to mutate or misrepresent with the minimal cost. In the evaluation, we present case studies for four major malware families: Worms, Cryptocurrency-mining malware, Ransomware, and Credential-stealing malware. For each case study, we modeled its behavior, extracted candidate deception parameters, and show how they can be used to design different deception schemes for different goals.

1.8 Organization

The rest of the dissertation is organized as follows. Chapter 2 presents our automated formal framework for synthesizing the diversity and isolation configuration. We present a risk metric to assess the quality of both isolation and diversity decisions, formalize the problem as a constraint satisfaction problem given risk and budget constraints, and we implement our solution using the Z3 SMT solver. In Chapter 3, we present our solution for active cyber defense orchestration and verification as a

way to implement a resilient adaptive response. Chapter 4 presents our automated agility-oriented malware analysis framework and demonstrates its effectiveness using real malware case studies. We discuss our technical approach for symbolically executing malware and analyzing the execution results with respect to concrete properties for resilient deception and deterrence. In Chapter 5, we present the summary of this dissertation, emphasizing the contributions and evaluation results, and we close this dissertation with some directions for future work.

CHAPTER 2: AUTOMATED SYNTHESIS OF ISOLATION AND DIVERSITY CONFIGURATION COMPOSITION

The rapidly increasing rate of cyber-attacks against computers and critical infrastructure mandates cyber defense strategies that go beyond preventing attacks on the network perimeter. Resilient defense strategies should be put in place to resist ongoing attacks and contain their propagation. Isolation and Diversity are resilience techniques that can proactively accomplish this objective.

Isolation is used to separate critical assets in the network from untrusted components by hardening the network configuration through the use of security countermeasures, such as firewalls and intrusion detection systems. This reduces the exposure of the critical assets to potential threat sources, resists the propagation of cyber-attacks, and guarantees that the critical functions of the network remain operational even if some nodes in the network are compromised [97, 98, 99, 25, 26]. Diversity techniques refer to the informed selection of software variants for the different services in the network to place heterogeneous software in potential attack paths. This includes, the selection of operating systems, software applications, and third-party libraries [95, 100, 29, 30, 31, 32] to variegate the software attack surface over attack paths. Software diversity limits the impact of discovered exploits and disrupt the attackers by forcing them to learn new exploits and attack multiple software programs. Both isolation and diversity techniques can increase the system's resistance by disrupting the propagation of sophisticated attacks through the network. However, the question that still stands is "how can these techniques be combined together to provide the optimal resilience against attacks?".

2.1 Motivation

Isolation and diversity are highly correlated techniques because one controls the network infrastructure where the other manages the software at the end-points. Nonetheless, their correlation represents an opportunity and a challenge at the same time. It is an opportunity because they can complement each other to provide complete protection against attack propagation at different layers of the network. It is a challenge because they operate on different interdependent layers and they must be consistent to guarantee the correct operation of the network. To take advantage of the opportunity and address the challenge, we need to find a safe and effective composition that weighs the different decisions of both techniques, especially under a limited budget and finds a consistent configuration that reduces the risk of attack propagation. A few shy attempts have been made for the optimal composition of isolation and diversity [95, 101]. However, it was made not clear how isolation and diversity contribute to the overall resilience of the network. They calculated the overall resilience of the network as the weighted sum of diversity and isolation metrics, where arbitrary weights are given to both the diversity and isolation. There are no scientific bases for selecting the optimal weights that maximize the overall network resilience.

In this work, we evaluate both the diversity and isolation techniques using the same metric; that is, the reduction in the likelihood of cyber-attack. We link each possible diversity or isolation action to a change in the attack likelihood and select the appropriate actions accordingly. The following key observations inspire our approach. First, diversity decisions are more effective if taken based on path-oriented analysis rather than pairwise or entropy-based approaches [100, 102, 103, 104]. Let us assume that we have two software variants that are completely different software programs (i.e., developed by different vendors), A and B. If we only consider the pairwise diversity between connected pairs of assets in the network, then the attack path that exhibits the software variants' pattern $A \rightarrow B \rightarrow A \rightarrow B$ is considered a secure path

since each asset has a completely different software variant from the preceding and following assets it in the path. However, if we follow a path-oriented approach, this path does not provide sufficient diversity because the third asset is the same as the first. Once the attacker exploits the first two assets, she will learn the vulnerabilities of both software variants A and B, and she can exploit the remaining two. Second, a specific isolation countermeasure might have different effectiveness scores for different variants of software. This implies that there is dependency between diversity and isolation decisions with respect to their effectiveness in reducing the cyber risk in the network. Placing a particular software variant in one asset (diversity action) in the network mandates a specific set of countermeasures (isolation actions) that are effective against the exploits directed towards that variant.

2.2 Related Work

The problem of automated synthesis of isolation and diversity configuration has attracted many researchers, who developed metrics and synthesis techniques to find the optimal deployment of these techniques. However, most of the existing works focus only on either isolation or diversity independently.

There are two lines of work related to our diversity model: the automated generation and assignment of diverse software implementations. For automatic generation of diverse software, researchers developed frameworks that can produce multiple implementations of the same software program, but with different attack surface. The idea behind this line of work is to randomize different aspects of software implementation to produce multiple and different instances while each instance implements the desired functionality of the software. This includes instructions substitution and reordering [105, 106, 107, 108], basic block shuffling and control flow flattening [109, 110, 111], and stack layout and functions randomization [112, 113, 114, 115] among others. The output of these automated diversity techniques is the input for our work in this chapter. We take the diverse software instances and assign them

to the services in the network, considering the available isolation countermeasures to reduce the risk and increase the network resilience. In the other line of work related to diversity, researchers provided diversity metrics and software assignments techniques to network services. In [95], the authors presented a metric to measure the diversity of a network, based on the least attacking effort, and they discussed how it could be applied in instantiating network diversity models. Other metrics, such as [116, 117], have been introduced to calculate the attack likelihood, considering the diversity of network components. The diversity principle has been used for implementing intrusion-tolerant web servers in [118]. Authors in [29, 30, 31, 32] investigated different typologies and architectures for cyber networks that counteract malware and worm attacks employing the diversity principle. These works focus only on diversity, and they do not consider the interdependence between software variants with the isolation countermeasures.

Isolation and segmentation is also a well-known technique for resilience. It has been widely adopted in the networked control systems as a means for fault isolation [119, 120, 121, 122]. Besides, researchers provided metrics that quantify the risk imposed of cyber-attacks, considering isolation and segmentation techniques [24, 25, 26, 27, 28]. In [97], the authors proposed a framework that automatically generates isolation configuration in terms of access control policies based on given usability and cost constraints. In [123, 124], the authors used attack countermeasures trees, one variant of attack graphs, that takes into account the attack actions as well as countermeasures and finds the optimal placement of countermeasures to block the critical attack paths. These isolation techniques only consider the isolation actions, and they do not evaluate the effectiveness of these countermeasures with respect to different software variants.

A few shy attempts have been made for the fine-grained composition of isolation and diversity. In [101], the authors presented a framework to synthesize network

resilience configuration considering variations of diversity and isolation configurations. However, it was made not clear how isolation and diversity contribute to the overall resilience of the network. They calculated the overall resilience of the network as the weighted sum of diversity and isolation metrics, where arbitrary weights are given to both the diversity and isolation. There are no scientific bases for selecting the optimal weights that maximize the overall network resilience. In our previous work [125], we developed a framework to verify the effectiveness of isolation and diversity techniques in OpenFlow-based SDN. However, the framework can only be used for verification, not synthesis, and the user must provide separate properties for each of diversity and isolation. In this chapter, we use the effectiveness of both isolation and diversity configurations in reducing the risk in the network as a unified measure for both of these techniques. Also, we compute the risk considering all the possible paths that lead to the network's critical assets, and we evaluate all the possible diversity and isolation actions in all the paths.

2.3 Problem Statement and Contributions

This chapter addresses the problem of automatic synthesis of isolation and diversity configuration to proactively achieve network resilience. Given the network Topology, network services placement (which services run on which hosts for a successful operation), available software diversity options (which software variants can be used for each service along with the variants costs and exploitability scores), available isolation options (attack countermeasures along with their costs and effectiveness scores against attacks), and a certain financial budget, the problem is to find appropriate (1) software variants assignments and (2) isolation countermeasures placement in the network, such that the global and local cyber risk, against multi-step cyber-attacks, is bounded within by a certain threshold.

To accomplish this, we first develop a metric for measuring the risk of multi-step persistent cyber-attacks in the network. Then, we use the risk metric to formalize

the selection of software variants and the placement of isolation countermeasures as a constraints satisfaction problem and solve it using the Z3 SMT solver [126] to find a satisfiable resilience configuration. We summarize our contributions in this chapter as follows:

- We model the isolation and diversity resilience techniques and define their effectiveness as a composite measure that represents the reduction in the attack likelihood, and the resulting network risk.
- We develop an SMT-based formal synthesis framework that generates a resilient network configuration, which includes the diversity and isolation configurations to satisfy the risk and budged constraints.
- We develop two optimization techniques to improve the scalability of our approach: a model reduction heuristic and a network decomposition algorithm.

We consider the global impacts of all diversity and isolation decisions as several attack paths might share a single decision and affects the risk of various assets in the network. Since we have budget constraints and it may not be feasible to select all the recommended isolation and diversity configurations, we find a balance between them with respect to the global risk.

2.4 Network Model

The scope of this work encompasses the configuration of the network infrastructure and the software variants of the services running in the network. To capture the impacts of isolation and diversity techniques in the network infrastructure and software, we first present abstract models for the network infrastructure configuration, isolation, and diversity. Then, we integrate them together in one model that represents the network augmenting isolation and diversity actions. This final model is used later to assess the risk and devise appropriate diversity and isolation configurations.

In the abstract model, a cyber network consists of a set of services connected through end-to-end communication links and is represented as the graph $G = (\mathcal{H}, \mathcal{S}, \theta, \mathcal{E})$, where $\mathcal{H}$ is a set of hosts, $\mathcal{S}$ is a set of services hosted in the network, $\theta(.)$ is a mapping function that maps each service to one host, and $\mathcal{E} = \{\langle s_i, s_j \rangle \mid s_i \in \mathcal{S}, s_j \in \mathcal{S}, i \neq j\}$ is the set of edges that represent the required end-end connections between services in the network (i.e., the reachability requirements). Note that we define $\mathcal{E}$ to model the end-to-end connections rather than physical links. An end-to-end connection is practically a path that may be composed of multiple physical links. For example, if the traffic from service a needs to go through a router in order to reach service b. The set $\mathcal{E}$ will include only one edge that abstractly represents the complete path from a to b, which is composed of two physical links in this case: a physical link from service a to the router and another from the router to service b.

2.4.1 Software Diversity Model

To increase uncertainty to stepping stone adversaries, who move from one machine in the network to another, organizations can use heterogeneous software variants, which exhibit different attack surfaces, for network services so that attackers cannot reuse that same exploits for different services in an attack path. Two different software programs are considered *variants* of a service when they have different implementations but can fulfill the functions of that service. On the one hand, software variants can be completely different software programs developed by different vendors. For example, Apache HTTP Server, Internet Information Services (IIS), and H2O HTTP server are all valid software variants for an HTTP web service. In addition, there exists artificial and automated software diversity techniques that take a program as an input and produce multiple variants whose implementations are different from the original program [127, 128, 129, 130]. The new variants are generated by randomizing implementation aspects, such as the code layout, in a way that is not predictable by the attacker. Examples of software implementation aspects that can be randomized

include code layout, address space layout, stack layout, register allocation, and system call mapping. This randomization process can change the location and/or the representation of the potential software flaws and significantly reduces the chance of reusing the same exploits again by the attackers.

The generation of software randomized implementations is out of the scope of this work. We assume that several software variants are given for each service regardless of whether they are completely different software programs or randomized implementations of the same program. In addition, we assume that a cost for each software variant is given, and the similarity score between any two software variants can be estimated. In Definition 1, we define the *Diversity Space*, which captures the software variants, their costs, and similarity scores.

Definition 1. ***(Diversity Space)*** *We define the diversity space of a cyber network as the tuple* $(V, \sigma, \epsilon, \rho, \delta)$*, where:*

- V *is the set of all software variants that are available to be used for all the services in the network.*
- $\sigma(.) : \mathcal{S} \rightarrow 2^V$ *is a function that determines which software variants are available for a particular service. Recall that* $\mathcal{S}$ *is the set of services in the network.*
- $\epsilon(.) : V \rightarrow [0, 1]$ *is the exploitability function that returns the intrinsic likelihood of exploiting software variants.*
- $\rho(.) : V \times V \rightarrow [0, 1]$ *is the similarity function that determines the similarity between any two software variants, where a value of* 1 *means that they are completely similar and a value of* 0 *means they are completely different from each other.*
- $\delta(.) : V \rightarrow \mathbb{Z}^*$ *is the cost function that determines the cost of each software variant, where* $\mathbb{Z}^*$ *is the set of non-negative integers.*

The exploitability function, $\epsilon(.)$, reflects the ease and technical means by which the software variant can be exploited. Its value is estimated based on the CVSS scores [89] of the variant's known vulnerabilities. The similarity, represented by the function $\rho(.)$, with respect to the attack surface, between a pair of software variants is a measure of the common security weaknesses among them. It is obvious that randomized implementations of the same software program are likely to have a high degree of similarity since they may share a significant portion of code. However, it is also possible to have a high degree of similarity between different software programs due to modular programming, software reuse, iterative development, and open-source packages. One technique that can be used for similarity estimation is to compute the share gadgets between software variants. Gadgets are short instruction sequences from various library functions that can be leveraged by attackers to perform malicious actions [131, 132]. The cost, $\delta(.)$, is an important factor for selecting an appropriate combination of software variants that meets the budget constraints. Note that randomized implementations of the same software program are likely to have the same costs. However, the software variants of each service may be completely different software programs, which will have different costs.

2.4.2 Isolation Model

Isolation is achieved by deploying appropriate attack countermeasures, such as firewalls, VPN gateways, and intrusion detection systems, in the potential attack paths from threat sources to critical network assets. Traffic inspection, encryption, and access control actions are examples of countermeasures actions. In this work, we do not restrict the type or the number of countermeasures that can be considered in our framework. However, we assume that the effectiveness of countermeasures for resisting attacks with respect to particular services can be estimated. Specifically, the effectiveness is measured by the probability of the countermeasure to prevent exploits directed towards particular services. This measure can be measured statistically from

previous incidents. In addition, each attack countermeasure is assigned a cost that reflects both the operational cost of deploying the countermeasure and the usability cost that is incurred due to the service interruptions that might be experienced during and after deploying it. In Definition 2, we define the *Isolation Space*, which captures all the available countermeasures, their costs, and effectiveness scores.

Definition 2. ***(Isolation Space)*** *We define the isolation space of a cyber network as the tuple* (C, η, δ)*, where:*

- C *is the set of all countermeasures that can be deployed in the network.*
- $\eta(.) : C \times V \rightarrow [0, 1]$ *is the effectiveness function that determines the effectiveness of each countermeasure with respect to software variants,* V*.*
- $\delta(.) : C \rightarrow \mathbb{Z}^*$ *is the cost function that determines the cost of each countermeasure, where* $\mathbb{Z}^*$ *is the set of non-negative integers.*

The effectiveness function, $\eta(.)$, returns the effectiveness of each isolation countermeasure with respect to a software variant. Note that it is a function of both the countermeasure and the software variants because we assume that countermeasures will have different effectiveness scores for different software variants based on its attack surface. For example, intrusion detection systems may be effective against exploits related to improper input validation weaknesses, while they may not be effective against exploits related to inadequate encryption. This shows the dependency between isolation and diversity decisions that we need to consider simultaneously to optimize the network resilience. Both software variants and isolation countermeasures should be selected such that we gain the maximum effectiveness against software exploits and decrease the risk as will be shown in the following sections.

2.4.3 Resilient Network Model

Within the context of isolation and diversity as attack resistance techniques, the resilient network configuration is the one that implements the optimal composition of

isolation (i.e., countermeasures placement) and diversity (software variants selection). In the following definition, Definition 3, we extend the network abstract model to include the isolation and diversity configurations.

Definition 3. ***(Resilient Network Configuration)*** *We model a cyber network as the graph* $G = (\mathcal{H}, \mathcal{S}, \theta, \mathcal{E}, \mu, \lambda)$*, where:*

- $\mathcal{H} = \{h_1, h_2, ..., h_n\}$ *is a set of hosts.*
- $\mathcal{S} = \{s_1, s_2, ..., s_m\}$ *is a set of services hosted in the network.*
- $\theta(.) : \mathcal{S} \rightarrow \mathcal{H}$ *is a mapping function that maps services to hosts. Each host may run up to* $|\mathcal{S}|$ *services.*
- $\mathcal{E} = \{\langle s_i, s_j \rangle \mid s_i \in \mathcal{S}, s_j \in \mathcal{S}, i \neq j\}$ *is the set of edges that represent the required end-to-end communication links between services in the network.*
- $\mu(.) : \mathcal{S} \rightarrow V$ *is the diversity function that determines which software variant is selected for a particular service in a particular host.* V *is the complete set of software variants.*
- $\lambda(.) : \mathcal{E} \rightarrow C$ *is the isolation function that determines which countermeasure is applied at each end-to-end communication link.* C *is the complete set of countermeasures.*

Based on this definition, the problem we are addressing in this work is finding the diversity and isolation functions, $\mu(.)$ and $\lambda(.)$, that reduces the risk of propagating cyber-attacks. Note that the diversity function selects a software variant for each particular service located in a particular host. This means that if two different hosts in different locations in the network run the same type of service, web service for example, it is possible to select a different software variant for each of them based on the attack paths that can reach them.

2.5 Risk Assessment Under Isolation and Diversity Configurations

In this section, we present a metric that measures the residual risk in the network for given isolation and diversity configurations. The risk metric considers the propagation of cyber-attacks through lateral movement in a chain of stepping stones, and the effectiveness of the isolation and diversity configurations at each chain link to obstruct the propagation. Before presenting the risk metric formally, we discuss the attack propagation model.

2.5.1 Attack Propagation Model

We present in the following our main assumptions about the attack propagation and its relation to the isolation and diversity configurations.

- Attacks can propagate from one victim service to another target only if (1) they are connected through an end-to-end connection, (2) the isolation countermeasures that is deployed in the connection are not sufficient to obstruct the propagation, and (3) the attacker can exploit the vulnerabilities in the target service (i.e., she possesses the required knowledge and skills).

- Compromising a service increases the attacker's capability of exploiting other services that use similar software variants. That is because compromising a service indicates that the attacker is familiar with its weaknesses and increases her knowledge and experience, which makes it easier to exploit similar ones. Hence, the software weaknesses that an be exploited belong to two independent classes: (1) known vulnerabilities that are being scored in public scoring systems like CVSS [89] and (2) vulnerabilities that are learned by compromising similar software variants. We assume that these two classes are disjoint.

- The attacker's capability of exploiting a service can be estimated based on the intrinsic exploitability scores of the selected software variant and its similarity

with all the software variants that the attacker encountered in the previous stepping stones in the attack path.

Based on this attack propagation model, isolation and diversity configurations are tightly coupled with the conditions of attack propagation. Effective isolation can obstruct the propagation by deploying appropriate countermeasures between services, and effective diversity can limit the capability of attackers to exploit services by selecting software variants with the least possible similarity.

2.5.2 Risk Metric

The risk imposed on a particular service in the network depends on the likelihood of compromising it and its asset's monetary value. We follow an attack path-oriented approach to calculate the risk, in which there might be multiple paths an attacker can use to reach network services. We consider all possible attack paths and we calculate the risk based on the weakest path (i.e., the path in which the service is more likely to be compromised).

To calculate the risk imposed on a particular service s in the network, we first build a set of cycle-free paths. Each path is represented by a sequence of services that starts at a service that can directly or indirectly reach the service s and ends at s. We build this set using a depth first search algorithm for a network configuration modeled according to Definition 3. The depth of the search is bounded by a given limit, k. Then, we compute the likelihood of compromising the service over each path. If there are multiple paths to reach a particular service, we select the one that has the maximum likelihood of compromise because it is the weakest path posing the maximum risk. Formally, let $\mathbb{Q}_s$ be the set of all possible attack of the service s and let $p^Q(s)$ denote the likelihood of compromising the service s through the path $Q \in \mathbb{Q}_s$. Further, let $\nu(.) : \mathcal{S} \rightarrow \mathbb{Z}^*$ be the value function that maps each service to

its asset value. Then, the risk, R_s, imposed on the service s is computed as follows:

$$R_s = \nu(s) \times \max_{Q \in \mathbb{Q}_s} \{p^Q(s)\} \tag{2.1}$$

The path $Q \in \mathbb{Q}_s$ can be defined as $Q = \{q_1, q_2, ..., q_k\}$ where $\forall_{i \in [1,k-1]} : \{q_i \in \mathcal{S},\ \langle q_i, q_{i+1} \rangle \in \mathcal{E}\}$, and $q_k = s$ since all the paths in $\mathbb{Q}_s$ end at service s. We compute the likelihood of compromising any service q_i in the path Q recursively as follows:

$$p^Q(q_i) = \begin{cases} \epsilon(\mu(q_i)) & i = 1 \\ p^Q(q_{i-1}) \times g(q_i \mid q_{i-1}) & i > 1 \end{cases} \tag{2.2}$$

Recall that $\mu(q_i)$ returns the software variant of the service q_i and $\epsilon(.)$ is the exploitability function defined in Definition 1. Hence, for the first service in the sequence (i.e., when $i = 1$), the likelihood of compromise is simply its intrinsic exploitable score, $\epsilon(\mu(q_i))$. For the following services, $g(q_i \mid q_{i-1})$ represents the probability of compromising q_i given that its predecessor in the sequence, q_{i-1}, is already compromised. According to our attack propagation model, $g(.)$ depends on the effectiveness of the countermeasure, between q_i and its predecessor, and the ability of the attacker to exploit q_i. Formally,

$$g(q_i \mid q_{i-1}) = (1 - \eta(\lambda(\langle q_{i-1}, q_i \rangle), \mu(q_i))) \times \left(\epsilon(\mu(q_i)) + \max_{j \in [1,q-1]} \rho(\mu(q_i), \mu(q_j)) \right) \tag{2.3}$$

where $\lambda(\langle q_{i-1}, q_i \rangle)$ is the isolation countermeasure deployed over the end-to-end connection from q_{i-1} to q_i, and $\eta(.)$ is the effectiveness of that countermeasure with respect to the software variant used for the service q_i (i.e., $\mu(q_i)$). Note that we subtract the effectiveness of the countermeasure from 1 since what we calculate here is the likelihood of the attacker bypassing the countermeasure, the opposite of the effectiveness. The ability of exploiting q_i is computed as the sum of two values: the

intrinsic exploitability of the software variant, $\epsilon(\mu(q_i))$, and the maximum similarity of the software variant of q_i to all software variants preceding it in the path Q. In our attack propagation model, we assumed that the set of known vulnerabilities, which is used to compute the exploitability score of a specific software variant, is disjoint from the vulnerabilities learned by compromising other software variants. Hence, the sum of a software variant's exploitability score and its similarity with the preceding software variants will lie in the interval $[0, 1]$ since they are disjoint. Moreover, we assume that the probability of exploiting a service is proportional to the similarity of its software variants with the previously compromised services. Thus, for simplicity, we assume in this formalization that they are equal, and we use the similarity as the probability of exploit.

Since the service s is the last service in the sequence Q (i.e., $q_k = s$), the likelihood of compromising s through the sequence Q is equal to $p^Q(q_k)$. We compute the risk of each service in the network according to Equation 2.1 and we compute the global risk of the entire network, R_G, as follows:

$$R_G = \sum_{s \in \mathcal{S}} R_s \tag{2.4}$$

Constraining the global risk, R_G, is our objective in the following formal resilient configuration synthesis problem.

2.6 Resilient Configuration Synthesis

In this section, we formalize the problem of diversity and isolation configurations synthesis as a constraint satisfaction problem. We show how we encode the network configuration and the risk metric as SMT constraints in order to find a satisfiable resilient configuration within budget constraints.

2.6.1 Decision Variables

We first define the decision variables, whose values represent the output of the synthesis problem. Our objective is to find the selection of software variants for the network services and the placement of attack countermeasures. Hence, we define two sets of decision variables as follows:

$$\mathbb{V} = \{v_1, v_2, ..., v_m\} \qquad \forall_{i \in [1,m]} : v_i \in V$$

$$\mathbb{C} = \{c_{ij} \mid \text{for each connection } \langle s_i, s_j \rangle \in \mathcal{E}\} \qquad \forall_{i,j \in [1,m]} : c_{ij} \in C$$

where m is the number of services in the system. We define a variable in $\mathbb{V}$ for each service in $\mathcal{S}$, whose value is the index of the software variant that is recommended to be used for the service (i.e., $\forall_{i \in [1,m]} : \mu(s_i) = v_i$, where $\mu(.)$ is the diversity function in Definition 3). Similarly, we define a variable in $\mathbb{C}$ for each service end-to-end connection in $\mathcal{E}$, whose value is the index of the countermeasure that is recommended to be used over the connection (i.e., $\forall_{i,j \in [1,m]} : \lambda(\langle s_i, s_j \rangle) = c_{ij}$, where $\lambda(.)$ is the isolation function in Definition 3).

2.6.2 Risk Computation

Now, we show how we encode the appropriate assertions that will calculate the risk imposed on each service in the network based on our risk metric, with respect to the decision variables. First, we define a variable e_i, where $i \in [1, m]$, for each service in the system to hold its intrinsic exploitability score. Note that the exploitability depends on the software variant that is selected for the service. Hence, we assign a value for this variable based on the corresponding decision variable in $v_i \in \mathbb{V}$ as follows:

$$\forall_{i \in [1,m]} : \forall_{j \in \sigma(s_i)} : (v_i = j) \rightarrow (e_i = \epsilon(j)) \tag{2.5}$$

where $\sigma(.)$ is the mapping function defined in Definition 1 to determine the alternative software variants for each service s_i. The function $\epsilon(.)$ returns the fixed intrinsic exploitability score of a particular software variant (denoted by j in the equation). The intrinsic exploitability, e_i, of the service s_i is fixed regardless of the attack path(s) it resides in. Hence, one variable per service is sufficient to model its intrinsic exploitability.

Next, we add the variables to calculate the likelihood of compromise for each service over all possible attack paths. Since the same encoding is used for all the paths, we will show how to do this in detail for one path only. Let us consider the attack path $Q = \{q_1, q_2, ..., q_u\}$, we define a variable, denoted by p_i^Q for the i-th service in the attack path Q, which represents the likelihood of compromising that service over that particular path. Based on our risk metric, the likelihood of compromising the first service in a path depends only on its intrinsic exploitability. Hence, we need to encode only the following assertion:

$$p_1^Q = e_j \ , \text{ where } q_1 = s_j \tag{2.6}$$

For the following services in the attack path, it is more complicated since the calculation of their likelihood of compromise must consider all the isolation and diversity decision variables along the path. For each service q_i in the path, where $i > 1$, we define the variable $t_{i-1,i}$ that represents the effectiveness of the countermeasure deployed between q_{i-1} and q_i. Let c represents the decision variable in $\mathbb{C}$ that corresponds to the isolation countermeasure between q_{i-1} and q_i and let v_i be the diversity decision variable for the software variant of the service q_i. Then, the value of $t_{i-1,i}$ is assigned according to the following expression:

$$\underset{j \in C}{\forall} : \underset{l \in \sigma(q_i)}{\forall} : (c = j) \wedge (v_i = l) \rightarrow (t_{i-1,i} = \eta(j, l)) \tag{2.7}$$

where $\eta(c, l)$ is the effectiveness function that returns the effectiveness of the countermeasure c with respect to the software variant l. This constraint assigns the value $\eta(c, l)$ to the variable $t_{i-1,i}$ if the countermeasure j is deployed between q_{i-1} and q_i and if the software variant l is selected for the service q_i.

Next, we define the variable g_i^Q to represent the likelihood of exploiting any service q_i in the path Q due to its similarity with all the services preceding it in the path. Note that this depends not only on the software variant selected for q_i, but also on the software variants selected for all services $\{q_1, q_2, ..., q_{i-1}\}$. To model this, let the variables $\{v_1, v_2, ..., v_i\}$ represent the software variants of all the services $\{q_1, q_2, ..., q_i\}$, where $\forall_{k\in[1,i]} : v_k \in \sigma(q_k)$. We compute the set $\mathbb{O}$ of all the possible permutations, where each permutation $O \in \mathbb{O} = \{o_1, o_2, ..., o_i\}$ is a vector of assignments for all the variables in $\{v_1, v_2, ..., v_i\}$. The variable g_i^Q is then assigned a value based on the following expression:

$$\underset{O\in\mathbb{O}}{\forall} : \left(\bigwedge_{k\in[1,i]} v_k = o_k \right) \rightarrow \left(g_i^Q = \max_{k\in[1,i-1]} \rho(o_i, o_k) \right) \tag{2.8}$$

where $\rho(v_i, v_k)$ is the similarity function that returns the similarity between the the two software variants v_i and v_k. This assertion will evaluate the similarity between the service q_i to all the individual services preceding it in the path $\{q_1, ..., q_{i-1}\}$ and assign the maximum similarity to the variable g_i^Q.

Now that we have computed the values of $t_{i-1,i}$, e_i, and g_i^Q for each service q_i in the attack path Q, we add an assertion for each of the services in $\{q_2, q_3, ..., q_u\}$ that represents its likelihood of compromise according to the formula that is defined in Equation 2.2 as shown in the following:

$$\underset{i\in[2,u]}{\forall} : p_i^Q = p_{i-1}^Q \times (1 - t_{i-1,i}) \times \left(e_i + g_i^Q \right) \tag{2.9}$$

We start from $i = 2$ because we have calculated p_1^Q of the first service in the path in Equation 2.6. This assertion calculates the likelihood of compromise recursively for each service in the path Q based on the decision variables and according to Equation 2.2 in our risk model.

Based on this encoding, the variable p_u^Q holds the likelihood of compromising the target service q_u since it is the last service in the attack path Q. We now define a variable, r_i, to represent the risk for each service s_i in the network based on the values of the isolation and diversity decision variables.

$$\underset{i\in[1,m]}{\forall} : r_i = \nu(s_i) \times \max_{Q\in\mathbb{Q}_{s_i}} p_{|Q|}^Q \tag{2.10}$$

Where $\nu(s_i)$ is a constant that represents the asset's monetary value of the service s_i and Q_{s_i} is the set of possible attack paths that lead to s_i. We use $p_{|Q|}^Q$ to represent the likelihood of compromising the last service in the attack path Q. Note that if there are more than one path leading to the service s_i (i.e., $|\mathbb{Q}_{s_i}| > 1$), we calculate the risk based on the weakest one (i.e., the one having the maximum likelihood of compromise).

2.6.3 Constraints

The generation of effective isolation and diversity configurations is driven by a specific set of constraints on the global risk and cost of recommended configurations. To define these constraints, we define the variables $RISK$ and $COST$ that hold the global risk and the total cost of implementing recommended configurations, respectively. Since we defined the set of variables $\{r_1, r_2, ..., r_m\}$ that represent the risk of individual services in the network. The $COST$ is computed in terms of those variables calculated based on Equation 2.4 in our risk model as follows:

$$RISK = \sum_{i\in[1,m]} r_i \tag{2.11}$$

To calculate the cost of diversity decisions, we define the variables $\{d_1, d_2, ..., d_m\}$ that represent the costs of the software variants selected for the services in the network. The values of those variables are calculated based on the diversity decision variables, $\mathbb{V}$, as follows:

$$\underset{i\in[1,m]}{\forall} : \underset{j\in\sigma(s_i)}{\forall} : (v_i = j) \rightarrow (d_i = \delta(j)) \tag{2.12}$$

Similarly, we define a set of variables to hold the cost of the recommended isolation measures. For each end-to-end link $\langle s_i, s_j \rangle \in \mathcal{E}$, let a_{ij} represent the cost of the isolation measure that is recommended for that link. The values of those variables are computed based on the isolation decision variables, $\mathbb{C}$, as follows:

$$\underset{\langle s_i,s_j\rangle\in\mathcal{E}}{\forall} : \underset{l\in C}{\forall} : (c_{ij} = l) \rightarrow (a_{ij} = \delta(l)) \tag{2.13}$$

Finally, we compute the total cost of all the isolation and diversity decisions as follows:

$$COST = \sum_{i\in[1,m]} d_i + \sum_{\langle s_i,s_j\rangle\in\mathcal{E}} a_{ij} \tag{2.14}$$

The risk and cost thresholds that constrain our configuration synthesis can now be defined in terms of the $RISK$ and $COST$ variables. Let us assume that we have the risk and cost thresholds, τ_R and τ_C, respectively. We add the following constraints to our constraints satisfaction problem:

$$RISK \leq \tau_R \tag{2.15}$$

$$COST \leq \tau_C \tag{2.16}$$

We add another set of operational constraints that are related to the consistency between the different software variants. Different software variants may have different

requirements for the Hardware and Software Platform running them. Since a single host in our network can run multiple services, all the selected software variants that are selected for its service should be able to operate on the same platform. To encode these operational constraints, we first build a set of neighbors for each service in the network. The neighbors of a service are the services that are co-located with it in the same host. Formally, for each service $s_i \in \mathcal{S}$, we define the set of neighbors $N_i = \{j \mid s_j \in \mathcal{S}, s_j \neq s_i, \theta(s_j) = \theta(s_i)\}$, where $\theta(.)$ is defined in our network model as the mapping function that returns the host running a specific service. In addition, for each software variant $i \in V$, we define a set of compatible variants as $W_i \subset V$. This set is static and is provided as input along with the diversity space of the network. Based on this, the operational constraints are encoded as the following assertions:

$$\underset{i\in[1,m]}{\forall} : \underset{j\in\sigma(s_i)}{\forall} : (v_i = j) \rightarrow \bigwedge_{n\in N_i} \left(\bigvee_{w\in W_j} (v_n = w) \right) \tag{2.17}$$

This constraint intuitively specifies that for any service in the network, its software variant is compatible with the software variants selected for the other services that exist on the same host.

2.7 Implementation and Evaluation

We have implemented our framework using the Z3 SMT solver 4.5.0. We used the Z3 C++ API to build the assertions required to model the network configuration and the constraint. The configurations generated using our formal synthesis framework are proven resilient by construction against any attack that follows our attack propagation model. In this section, we present our scalability evaluation for networks of various parameters, such as network size, the depth of attack paths, and the number of software variants and isolation countermeasures. We generated a number of synthetic networks of up to 600 hosts and evaluated the time required by our framework to generate a resilient configuration.

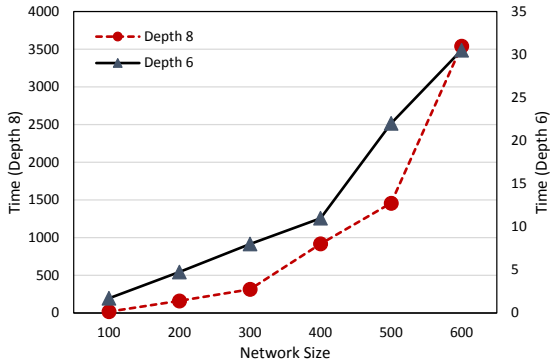


Figure 2.1: The impact of network size (number of hosts).

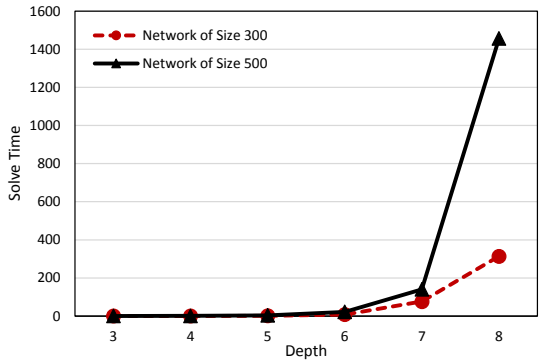


Figure 2.2: The impact of attack path depth.

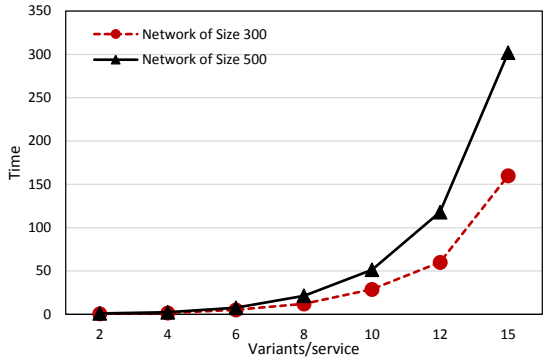


Figure 2.3: The impact of number software variants.

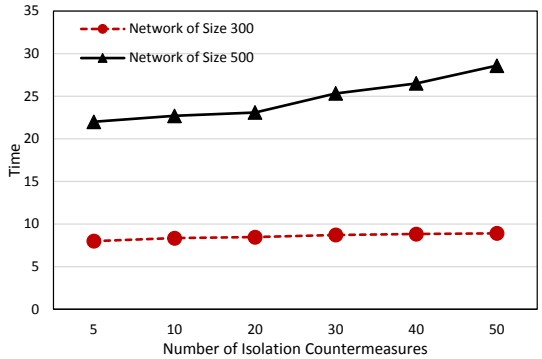


Figure 2.4: The impact of number of countermeasures.

2.7.1 Evaluation Results

To generate the synthetic networks, we used aSHIIP [133], a random topology generator that allows one to generate networks using various generation models. The authors in [133] show that the general linear preference model (GLP), with certain parameters, generates topologies, which most appropriately represent the Internet based on the internet topologies gathered by the Center for Applied Internet Data Analysis (Caida) [134]. We used the GLP model with the recommended parameters (the probability parameter set to 0.55 and the parameter responsible for the choice of attachment nodes for edges set to 0.75) to generate all the networks for the following experiments. All experiments were conducted on a standard PC with an Intel Core i7 CPU and 16 GB of RAM.

The Impact of Network Size. We generated a number of networks of sizes that ranged from 100 to 600 nodes and we evaluated our framework for attack paths of up to 8 intermediate nodes. The results are shown in Figure 2.1. For networks beyond 600 nodes and a depth of 8 or greater, our synthesis engine does not return a result because of insufficient memory. As shown in the figure, the time required to generate resilient configuration exhibits quadratic growth with respect to the network size.

The Impact of Attack Path Depth. In this experiment, we evaluated our framework using two networks of sizes 300 and 500 nodes, but with varying attack path depth. As depicted in Figure 2.2, the depth varies between 3 and 8 intermediate nodes. The synthesis time exhibits exponential growth in terms of the attack path depth. We believe that an attack depth of 8 intermediate nodes is feasible. Consequently, our framework will run for networks of up 600 hosts under this configuration.

The Impact of Number of Software Variants. In this experiment, we evaluated our framework using two networks of sizes 300 and 500 nodes under a fixed attack depth of 6, but with varying number of software variants for each service. As depicted in Figure 2.3, the number of software variants is between 2 and 15 variants per service. The synthesis time also exhibits exponential growth, but the growth rate is slow compared to the growth rate with respect to the attack path depth. This growth is a result of considering the similarity between each software variant in an attack path with all the software variants preceding it in the path. Increasing the number of software variants significantly increases the number of permutations we need to consider (Equation 2.8), which increases the solve time exponentially.

The Impact of Number of Isolation Countermeasures. We evaluated our

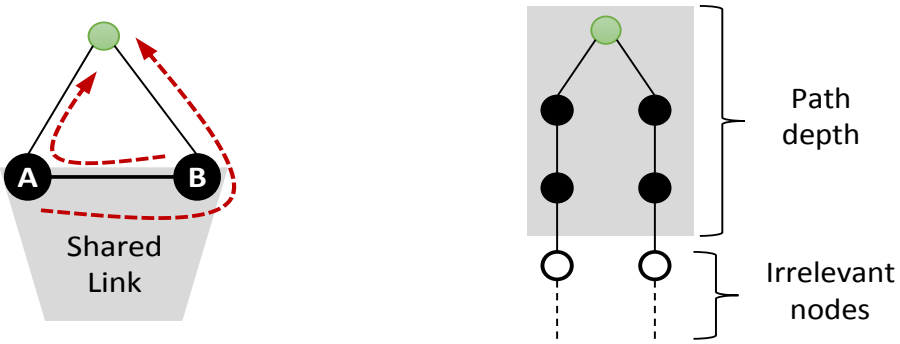
framework using the same networks from the previous experiment with respect to the number of possible countermeasures for each end-end link in the network. As depicted in Figure 2.4, the number of countermeasures was increased from 5 to 50. The results show a minor impact on the synthesis time with respect to increasing number of countermeasures. The number of countermeasures will only increase the possible values for the isolation decision variables, and it affects how we compute the effectiveness of each isolation decision in the network, this is reflected in Equation 2.7. However, increasing this number does not require adding new variables or additional nonlinear computations.

2.7.2 Discussion

The evaluation results display that our framework is not affected by the number of countermeasures and it can consider a reasonable number of 15 software variants per service. Moreover, it can scale to large networks considering short attack paths. However, the time complexity is very sensitive to the attack depth as shown in Figures 2.1 and 2.2, where it takes around one hour for a network of 600 nodes compared to less than a minute for the same network with an attack path depth of 6. This high complexity limits the applicability of our framework for large-scale networks. To overcome this challenge, we provide two heuristics that make our framework scale to large networks.

2.8 Model Reduction and Network Decomposition for Scalable Synthesis

In this section, we present two heuristics to enhance the scalability of our synthesis framework. These solutions are inspired by the two observations that are depicted in Figure 2.5 and discussed in the following sections.



(a) Shared end-to-end links. (b) Irrelevant nodes.

Figure 2.5: Two observations that are leveraged for scalability.

2.8.1 Model Reduction

Based on our risk model, we enumerate all possible attack paths that lead to each service in the network. Then, during the synthesis process, we define variables for each end-to-end link that assign an appropriate countermeasure to each end-to-end link. However, as depicted in Figure 2.5a, we may have the case in which a bidirectional end-to-end link is shared among multiple attack paths. The red dashed lines in Figure 2.5a represent two attack paths that consists of the same nodes and share the bi-directional end-to-end link connecting nodes A and B.

We leverage this observation to reduce the number of variables needed to model our risk metrics (Equation 2.7) and reduce the synthesis time as a result. Both the attack paths shown in the figure consist of the same services, but with different order. From the diversity point of view, both paths are identical because the attacker will encounter the same number of distinct software variants on both paths regardless of the order. However, from the isolation point of view, the paths are different because the isolation countermeasure that is required on one direction may be different than the one required on the opposite direction. Since some of the isolation measures may be bi-directional (e.g., if a firewall denies communication from client x to the web, it is unlikely that it will allow the web to connect to the client x), it is sufficient to consider only one direction and use the same set of variables for both directions.

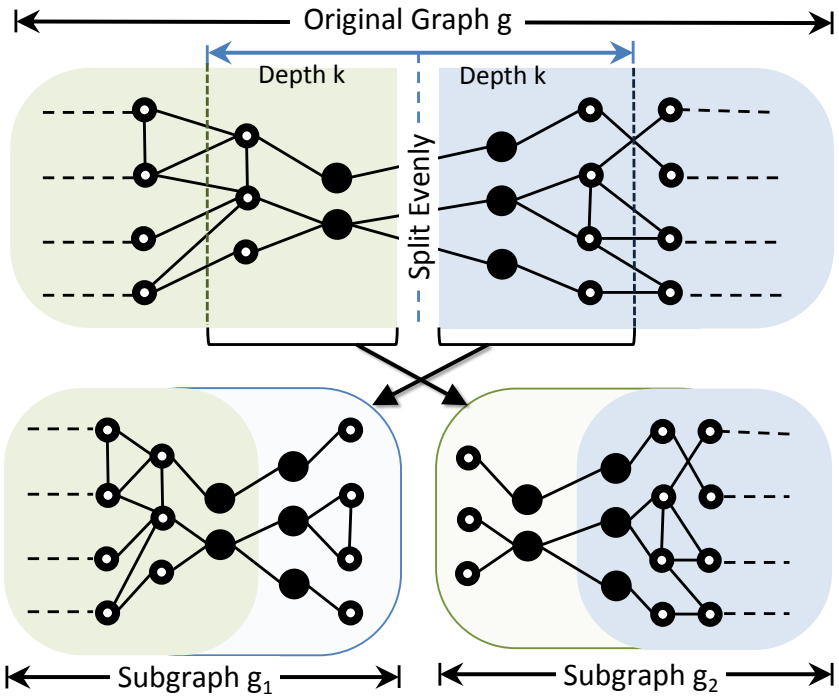


Figure 2.6: Graph Decomposition.

We implemented this reduction technique and evaluated the performance of our synthesis framework with and without it. The exact times for solving the synthesis problems for network of variants size are shown in Figure 2.7. Roughly, this technique reduces the time requirements by 30%.

2.8.2 Network Decomposition

Our risk metric considers the exposure of network services to potential threat sources within attack paths of a bounded depth. As depicted in Figure 2.5b, the threat sources that are further than the attack path depth from a particular victim have no effect on its risk, which implies that they can be in a completely separate sub-network without affecting our risk calculations. Based on this observation, we decompose the network into two or multiple sub-networks, solve them separately, and then combine the results to have one resilient configuration that satisfies the risk and budget constraints.

To decompose a network into two sub-networks, we developed Algorithm 1 (a visual

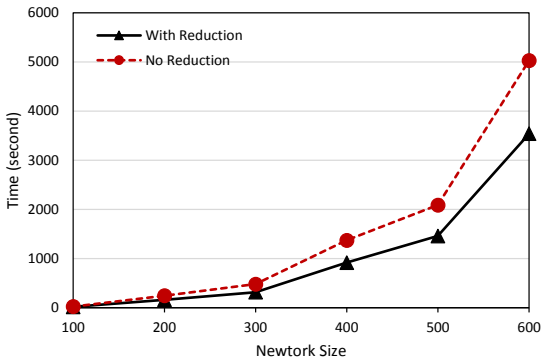


Figure 2.7: Synthesis time with model reduction.

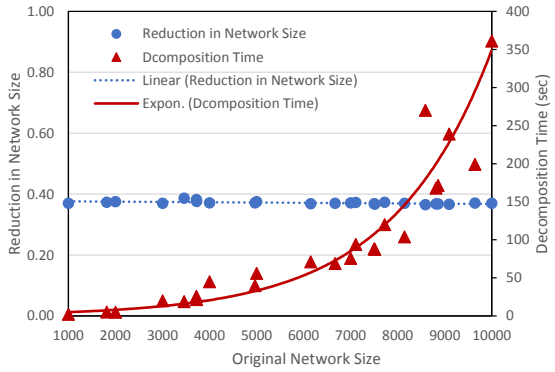


Figure 2.8: Network Segmentation Time and Value.

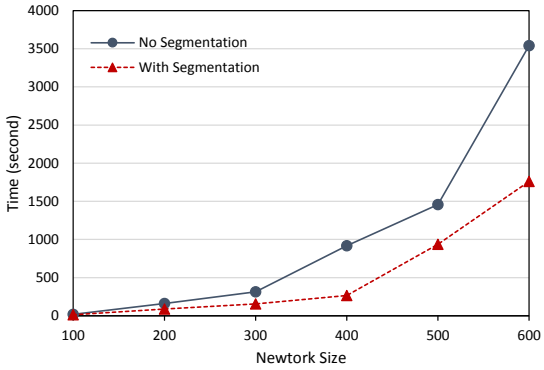


Figure 2.9: Synthesis time with network decomposition.

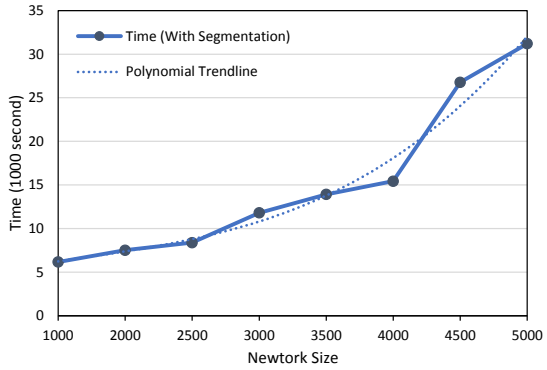


Figure 2.10: Synthesis time with network decomposition (for large networks).

demonstration for the decomposition process is provided in Figure 2.6). The input of this algorithm is the original network represented as the graph $g = (\mathcal{S}, \mathcal{E})$. The algorithm consists of two steps. First, we leverage an existing heuristic algorithm for finding partitions of graphs, known as *Kernighan–Lin* [135], to split the network into two disjoint sub-graphs of equal, or nearly equal, sizes (Line 1 in Algorithm 1). The *Kernighan–Lin* algorithm will find an edge cut-set that splits the network evenly into the two sub-graphs g_1 and g_2. In addition to the sub-graphs themselves, we record two sets of nodes, b_1 and b_2 that contain the border nodes in both sub-graphs (i.e., those nodes that were originally connected to the edge cut-set and they are represented as the dark circles in Figure 2.6). Second, taking each sub-graph separately, we run a depth first search starting from each of the border nodes to enumerate all nodes

Algorithm 1: Network Decomposition Algorithm

Input : graph g and the depth k
Output: graphs g_1, g_2

```
1  g_1, g_2, b_1, b_2 = Kernighan_lin(g)
2  foreach node n in b_1 do
3  |   e = find_foreign_end(n)
4  |   f_1 = DFS(g_2, e, k − 1)
5  |   append_subgraph(g_1, f_1, n, e)
6  end
7  foreach node n in b_2 do
8  |   e = find_foreign_end(n)
9  |   f_2 = DFS(g_1, e, k − 1)
10 |   append_subgraph(g_2, f_2, n, e)
11 end
12 return g_1, g_2
```

that can be reached within the attack path depth k from the border nodes. We add those nodes to the other sub-graph. This step will create an overlap between the two sub-graphs, but it is required to preserve the soundness of our risk calculations. Since the risk imposed on a particular service depends on all the other services that can reach it within the depth k, this overlap guarantees that all the relevant threat sources of each service exist in the same sub-graph. This step is implemented by the lines 2-11 in Algorithm 1. The function *find_foreign_end(n)* returns the other end of the cut-edge that is connected to the border node n. The depth first search function, *DFS*$(g_i, e, k-1)$, starts from node e and traverses the sub-graph g_i to enumerate all nodes that can be reached within $k-1$ steps. We use $k-1$ because we have already called *find_foreign_end(n)* once, which means there are $k-1$ out of k steps remaining. The function *append_subgraph*(g_i, f_i, n) appends the sub-graph f_i to g_i and adds an edge between the node n in g_i and e.

We evaluated Algorithm 1 with respect to the reduction in the network size and the time it takes to decompose networks of up to $10,000$ nodes. The reduction in the network size is simply the difference between the size of the resulted sub-graph and the size of the original graph divided by the size of the original graph. As shown

in Figure 2.8, the algorithm reduces the size of the network by at least 36%. The segmentation time grows exponentially with respect to the network size, but it takes a shorter time compared to the time required to solve the original network without segmentation, as will be shown later.

Decomposing a network based on Algorithm 1 results in two sub-network that can be solved separately. Although we have two separate sub-networks, the risk and cost thresholds are set for the entire original network and not for each segment separately and the cumulative risk and cost of both solutions must meet these thresholds. We have developed Algorithm 2 that can consider multiple solutions the sub-networks until it finds a pair of solutions that satisfies the global risk and cost constraints.

Algorithm 2: Synthesis

Input : graphs g_1, g_2 and the thresholds R, C
Output: solution for s_g

```
1  m_sub = solve(g_1, R, C)
2  while m_sub.hasMoreSolutions() do
3  |   s_sub = m_sub.getSolution()
4  |   r = R − s_sub.eval(RISK)
5  |   c = C − s_sub.eval(COST)
6  |   m_2 = solve(g_2|_{s_sub}, r, c)
7  |   if m_2.hasMoreSolutions() then
8  |   |   return s_sub ∪ m_2.getSolution()
9  |   end
10 end
11 return null
```

The inputs of Algorithm 2 are the two sub-networks generated by Algorithm 1 along with the global risk and cost constraints and the output is a solution for the whole original network, s_g. In line 1 of the algorithm, we solve the first sub-network g_1 with the global risk and cost constraints. If no satisfying solution exists given the global risk and cost constraints, then there is no solution for the entire original network. If solutions are found for the sub-network g_1, we start enumerating them one by one (Line 3) and evaluating the actual risk and cost of implementing each on the first sub-

network. Recall that $RISK$ and $COST$ are two variables defined to hold the global risk and configuration cost according to equations 2.11 and 2.14. The actual risk and cost of a solution, $RISK$ and $COST$, may be less than or equal to the global risk and cost thresholds, R and C, respectively. We calculate the remaining risk and cost by subtracting the actual ones from the global risk and cost thresholds (Lines 4 and 5). The remaining risk and cost, r and c, represent the local risk and cost thresholds for the other sub-network. Hence, we attempt to solve sub-network g_2 given those thresholds (Line 6). The existence of a solution for the second sub-network, g_2, given the local risk and cost constraints, means that we found a solution for the whole original network, which is the union of the two solutions of the individual segments. If no solution is found for the second sub-network at Line 6, we loop through all other possible solutions of the first sub-network until a solution is found. If the loop finishes without finding a solution, this means that no solution is possible for the whole original network.

Since the two sub-networks may have overlap, the final solution must select one countermeasure for the overlapped links and one software variant for the overlapped service. We guarantee this by restricting the values of the decision variables that are associated to the overlapped links and hosts in the second sub-network to their values in the solution found for the first sub-network. In Line 6 of Algorithm 1, the first input to *solve* is $g_2|_{m_{sub}}$, which denotes the second sub-network g_2 restricted by the solution of the first sub-network, m_{sub}.

Theorem 1. *Any solution for the two sub-networks returned by Algorithm 1 is a valid solution for the entire original network.*

Proof Sketch. The risk of any service in the network depends on all other services that can reach it within a path of k steps. For any service $i \in \mathcal{S}$ in the original network, its risk depends on the set S_i. After the initial decomposition (Line 1 in Algorithm 1) of the original network g into two sub-networks g_1 and g_2, the service i ends up in

either g_1 or g_2. Let N_1 and N_2 denote the set of services in the initial sub-networks g_1 and g_2, respectively. If $i \in N_1$, then for any service j, such that $(j \in S_i) \wedge (j \in N_1))$, j will be in the same graph and it will be considered in calculating the risk of i. For any service l, such that $(l \in S_i) \wedge (l \notin N_1)$, it will be consider as an overlapped node and appended to the sub-network g_1 (Line 5 in Algorithm 1). As a result, $S_i \subset g_1$. The same applies if $i \in N_2$. This will result in equal risk values for each service i in both cases if we solve the entire network in one shot or if we decompose and solve every sub-network separately.

We evaluated the performance of our synthesis framework with decomposition. In Figure 2.9, we show the time requirements to synthesis resilient configuration for the same networks we evaluated earlier (Figure 2.1), decomposing the each network into two sub-networks. In Figure 2.10, we show the required time to solve large networks of up to 5000 hosts. It was impossible to solve the problem for those networks without using our decomposition algorithm because we ran out of memory. Note that for networks larger than 1000 hosts, we used multi-layer decomposition. That is, we fist decompose into two sub-networks, then each sub-network is decomposed into two smaller sub-networks, and so on, until we reach sub-networks of sizes less than 500 nodes.

CHAPTER 3: PROVABLY SAFE AND EFFICIENT COURSE OF ACTION ORCHESTRATION FOR ACTIVE CYBER DEFENSE POLICIES

In addition to the proactive resilience techniques that we presented in the first chapter, ensuring the resilience of computer networks against dynamic cyber-attacks requires active defense strategies for attack prevention, mitigation, or recovery. These dynamic defenses may change the network configuration according to high-level policy, in response to specific security events. In this chapter, we present a formal framework to model such policies, and orchestrate the attack prevention, mitigation, and recovery procedures to act in an effective and safe manner that does not break the network mission.

3.1 Motivation

To cope with numerous potential incidents and security events in real-time under dynamically changing environments, cyber often adopts special processes and tools for an automatic and fast response. These processes and tools are governed by a policy that determines the appropriate course of action in response to particular security events. We refer to such policy by the *"Active Cyber Defense (ACD) Policy"*. Multiple security incident response and handling platforms provide the means to define and manage ACD policies, such as Phantom Security Automation and Orchestration Platform [40], IBM Resilient [38], AlientVault USM Anywhere [39], and Security Incident Response Orchestration by ServiceNow [136]. These platforms, among others, provide some interface to define security events and link them to the desired courses of action. Thus, a Course of Action (CoA) is an ordered set of cyber actions (commands) that are triggered and executed in response to a particular security event.

The ACD policy may consist of several rules that modify different, yet interdependent, layers of network elements, including the configuration of hosts, the services hosted by them, the virtualization servers and data centers, and the networking infrastructure configuration. Moreover, multiple rules, which might have different competing actions, might be triggered and executed simultaneously. For such diverse reactions to preserve and enrich the resilience of the network, they must satisfy the following two properties:

- Effectiveness. This property requires that the different policy rules, which might be executed concurrently, implement the desired responses successfully without conflicting with each other.

- Safety. This property requires that the policy rules, whether they are executed concurrently or sequentially, do not introduce violations to the high-level network mission.

Meeting these properties in complex and dynamic networks is challenging due to the following reasons. First, the successful execution of a cyber action that belongs to a CoA within the ACD policy may require changing various resources in the network. In this work, we consider three types of resources: actuators, objects, and control variables. The actuators are network entities that perform the actions, such as OpenFlow controller, virtualization server, and service managers. The objects are the network entities controlled by the actions, such as the traffic transmitted in the network infrastructure, the virtual machines hosted in a data center, and the services running on top of hosts. The control variables are the dynamic system attributes, which the execution of the actions depends on or affects, such as links utilization, servers' availability, space capacity, and CPU utilization. Since the CoAs in the reactive policy are not mutually exclusive, multiple actions might operate on the same set of resources simultaneously, which might cause access conflicts and, as a result,

execution failures. Second, even if cyber actions are not executed simultaneously, they might still be interdependent and affect the execution of each other if they rely on the same system attributes (control variables). For example, you cannot migrate a virtual machine to a server unless it does have the appropriate capacity to accommodate the virtual machine. However, the capacity of the server is a control variable that can be affected by the preceding migrate actions, which indicates a dependency between the consecutive migrate actions or any other type of actions that influence the capacity of servers. Third, the cyber actions can change the network data plane configuration in a way that violates the high-level network mission requirements. The network data plane configuration includes the access lists that reside in switches and firewalls, and control the flow of traffic in the network. The network mission requirements are logical properties that describe the network reachability, QoS, and security requirements for accomplishing the mission.

In the current practice, there are no guarantees that the reactive policy is composed and executed in a way that satisfies both the effectiveness and the safety properties. In this chapter, we address this gap through our automated ACD policy orchestration and verification framework that is presented in the following sections.

3.2 Related Work

In this section, we discuss the works related to two research directions: the ACD policy specification and conflicts resolutions, and the network requirements verification.

3.2.1 Active Cyber Defense Policies and Conflicts Resolution

With the current complexity of cyber networks and the increasing number and severity of security incidents, it becomes mandatory to automate the cybersecurity incident response and attack mitigation through ACD. Multiple proprietary incident response and handling platforms [38, 40, 39, 136] provide the tools and processes to

accomplish this, but they do not provide any formal specification or analysis techniques to resolve potential conflicts and ensure safe implementation of CoAs.

There have been a few recent attempts in research literature that define high-level formal specification languages to describe verifiable CoAs and ACD policies, such as Nettle, Pyretic, Procera, and Kinetic [137, 138, 36, 37]. Procera [36] is a high-level network control language for writing high-level policies, which is based on Functional Reactive Programming [139]. Procera at the policy layer provides simple constructs to direct the network controller's responses to signals and network events. Procera allows operations for filtering, merging, transforming, and joining event streams by providing event algebra. While Procera provides a rich set of operations to aggregate the events that will trigger the reactions, it has limited options for the reactions that are specific to the SDN networks. The Pyretic [138] language, from the Frenetic [140] family, is another network programming language that allows network programmers to describe flow entries for OpenFlow switches dynamically. It provides the constructs to specify a packet forwarding policy that will govern the operation of the network controller. However, it does not provide the ability to describe CoAs. In other words, it can describe how the traffic flows should be forwarded through the network, but it cannot describe the required changes in the network due to specific events. kinetic [37], which is an extension of pyretic, is the latest network programming language that is targeting reactive and ACD polices. In Kinetic, dynamic policies are described using state machines, where several parallel policies can run throughout the system life cycle. Kinetic also provides the means to verify the correctness of the reactive policy using model checking. The CTL language is used to write the desired property, and the NuSMV model checker is used to model and verify the system. The verification in Kinetic is limited to the correctness of the reactive programs themselves. It does not provide the constructs to verify the satisfaction of the network mission requirements under multiple reactive policies. In this work, we model the operation of the complete

system and verify the fulfillment of the network mission requirements.

3.2.1.1 Tools for Network Invariants Verification Under ACD

In another line of research, a substantial body of research has been conducted in the area of network invariants verification for both enterprise and software defined networks. We classify the related work in this section into two groups: static verification tools and real-time verification tools.

FlowChecker [67], HSA [141], and FLOVER [142] are static verification tools for software defined networks. FlowChecker encodes the OpenFlow flow tables using Binary Decision Diagrams (BDD) to verify security properties. HSA verifies the data plane configuration correctness by modeling the network as a geometric model to discover violations in reachability and traffic isolation. FLOVER [142] is another model checker that checks the OpenFlow configuration for security violations using Yices SMT solver. ConfigChecker [68], Anteater [65], and SecGuru [143] are also static verification tools for enterprise networks. ConfigChecker and Anteater use model checking to verify network requirements expressed in temporal logics. These works targets only traditional enterprise networks configuration and they use binary analysis platforms (BDD and SAT), which make it hard to verify properties with arithmetic constraints. SecGuru is another tool that is based on the bit-vectors theory in the Z3 solver for checking network invariants.

VeriFlow [144], NetPlumber [145], and FlowGuard [146] are real-time verification tools for software defined networks that verify the satisfaction of specific properties after each configuration update control messages. VeriFlow proposes to slice the OF network into equivalence classes to efficiently check for reachability violations. NetPlumber is a real-time policy checking tool that utilizes a dependency graph between flow entries to incrementally check for loops, black holes, and reachability properties. FlowGuard examines dynamic flow updates to detect firewall policy violations, and it provides violation resolution approaches. Although these works can check the com-

pliance of OpenFlow network updates with specific invariants, their applications are limited to reachability or related analyses in VeriFlow and NetPlumber and firewall policy verification in FlowGuard.

While these works provide multiple platforms to verify the end-to-end reachability in enterprise and software defined networks, they do not consider a dynamic environment under multiple CoAs. Even the real-time verification tools, such as VeriFlow and NetPlumber, they verify invariants against one update at a time, and they do not consider multi-step strategies. They also have limited support for QoS requirements except for NetPlumber that provides path length constraints. In this work, extend our previous work [34] and provide the ability to verify complete ACD techniques and we utilize the arithmetic theory in SMT to verify quantitative QoS in addition to reachability and security requirements.

3.3 Active Cyber Defense Policy Specification

As we mentioned earlier, multiple existing security incident response and handling platforms provide interfaces to define ACD policies. Although their interfaces are different, they all allow users to define ACD policies that associate security events with courses of actions, which can mitigate potential threats and recover from their potential consequences. For the purposes of this chapter, we do not dictate a specific language, but we assume that the ACD specification language meets the following criteria:

- The ACD policy can include multiple rules to respond to different security events, where each rule associates an event with a CoA.
- Actions within a single CoA can be organized sequentially, in parallel, or we can select between two CoAs based on some conditions on the dynamic state of the system.
- A CoA can be composed of two types of actions: configuration actions and

$$
\begin{aligned}
\textit{Policy } \Pi &::= \{\langle \textit{event} \rangle \twoheadrightarrow \Lambda\} \\
\textit{CoA } \Lambda &::= \alpha \mid \Lambda \ ; \ \Lambda \mid \Lambda \parallel \Lambda \mid \psi \ ? \ \Lambda : \Lambda \\
\textit{Action } \alpha &::= [\boldsymbol{A}\ \psi]\, \boldsymbol{DO}\ f(\{b = \langle \textit{number} \rangle\})\ \boldsymbol{BY}\ u\ \boldsymbol{ON}\ o\ [\boldsymbol{G}\ \psi] \\
\textit{Expression } \psi &::= v['] \mid v['] \bowtie \langle \textit{number} \rangle \mid v['] \bowtie \psi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \\
\textit{Operator } \bowtie &::= \ == \mid > \mid < \mid \leq \mid \geq \mid + \mid - \mid \times \mid / \\
\textit{Command } f &::= \pi \mid \tau \\
\textit{Configuration } \pi &::= \textit{block} \mid \textit{forward} \mid \textit{limit} \mid \textit{inspect} \mid \textit{encrypt} \mid \textit{tunnel} \mid \\
&\quad\ \textit{enable} \mid \textit{disable} \mid \textit{migrate} \mid \textit{reroute} \mid \langle \textit{other} \rangle \\
\textit{Investigation } \tau &::= \textit{SNMPGet} \mid \textit{LogAudit} \mid \textit{SplunkActions} \mid \\
&\quad\ \textit{MITRE-ATT\&CK/CWE/CAPEC-InvActions} \mid \langle \textit{other} \rangle \\
\textit{Argument } b &::= \textit{portno} \mid \textit{threshold} \mid \textit{ccuid} \mid \langle \textit{other} \rangle \\
\textit{Actuator } u &::= \langle \textit{list of unique actuators} \rangle \\
\textit{Object } o &::= \langle \textit{list of unique objects} \rangle \\
\textit{Variable } v &::= \textit{capacity} \mid \textit{bandwidth} \mid \textit{resource utilization} \mid \langle \textit{other} \rangle
\end{aligned}
$$

Figure 3.1: ACD Policy Language Syntax.

investigation actions. Configuration actions dictate direct cyber commands to be performed by specific actuators (e.g., firewalls, virtualization servers, and SDN Controllers) on specific objects in the network (e.g., traffic flows, virtual machine, ports/queues in SDN switches). Investigation actions query and return particular information about the system's dynamic state.

- Actions can be associated with pre- and post-conditions (also known as assumptions and guarantees in other formal platforms [147, 148], respectively). The pre-condition must be satisfied for the correct execution of an action, and the post-condition is guaranteed to be satisfied after the action is executed correctly.

We use a language called *CLIPS*, which has been developed by our research group to describe ACD policies. The syntax of CLIPS is shown in Figure 3.1. The reactive policy Π according to this syntax consists of a set of ACD rules, where each rule associates an event $\langle \textit{event} \rangle$ to a CoA (denoted by Λ).

The CoA is represented as a process that executes single or multiple actions composed in different modes. A CoA can be one action, multiple consecutive actions using

the sequential composition operator (;), multiple parallel actions using the parallel composition operator ($\|$), or a conditional statement (ψ ? Λ_1 : Λ_2), which executes Λ_1 if the condition ψ evaluates to true, otherwise, it executes Λ_2. Cyber actions denoted by α are the basic building blocks, where each cyber action specifies that a command (f) (e.g., blocking or forwarding packets, migrating virtual machines, and disabling/enabling services in the network) be executed by a certain actuator (u) (e.g., a firewall, an SDN switch, a virtualization server or controller) on a specific object (o) (e.g., a specific traffic flow or a virtual machine). In Figure 3.1, we list only a subset of the commands, actuators, and objects and we leave it to the users to define their own. Each command f can also take a set of arguments, if needed, in the format ($argument\ name = value$). Examples of these arguments include the output port number of the forward commands and the destination server of the migrate command.

CLIPS allows cyber actions to have optional pre-conditions (denoted by the expression ψ following the $\boldsymbol{A}$ keyword in the syntax) and post-conditions (denoted by the expression ψ following the $\boldsymbol{G}$ keyword). The pre- and post-conditions are arithmetic logic expressions in terms of a set of control variables $\mathcal{V} = \{v_1, v_2, ...\}$. Let us assume that we have the control variables $v_a \in \{true, false\}$ that represents the availability of a particular server, and the variables $v_c, v_p \in \mathbb{Z}^*$ that represent the capacities of two servers in the system. A pre- or a post-condition can simply be whether a Boolean control variable evaluates to *true* or *false* (e.g., $\psi = v_a$) or it can be a comparison between a control variable and a constant number (e.g., $\psi = v_c \geq 500$). It can be also expressed as a comparison between a variable and another arithmetic expression in terms of one or multiple other variables (e.g., $\psi = v_c \geq v_p + 500$). In addition, an arithmetic expression can consist of other expressions linked using the negation, conjunction, and disjunction operators (e.g., $\psi = (v_c \geq 500) \vee (v_p \geq 500)$). Moreover, we add the ability to express the post-conditions in terms of the previous values of the

control variables because the post-conditions can be used to describe the changes and effects of the cyber actions. For example, migrating a virtual machine to a server will decrease its capacity (represented by v_c) by a specific value (e.g., 500). Since it is not likely to know the absolute available capacity in advance, we allow the post-condition to be specified as $\psi = (v_c == v'_c + 500)$, where the primed variable v'_c denotes the previous value of the capacity before executing the action. Hence, the syntax of CLIPS allows us to optionally use the prime symbol with each control variable to represent its previous value.

3.4 Problem Statement and Contributions

This chapter addresses the need for automated orchestration and verification of ACD policies. Given a set of CoAs that belong to an ACD policy, the current state of the dynamic system attributes (represented by the control variables), the current state of the network data plane configuration, and the network mission requirements, our objective is to find a Global Orchestrated CoA workflow (GOAL) that satisfies the following properties:

- Resource Integrity. No resource conflicts between concurrently executed actions due to simultaneous access to shared resources (i.e., actuators, objects, and control variables).

- Action Integrity. Every action will be executed correctly (i.e., its pre-condition is not violated by a previously executed interdependent action).

- Mission Integrity. No violations for the network mission requirements are introduced during the GOAL execution.

- CoA Concurrency. The total execution time is minimized by safely maximizing the concurrency in executing the cyber actions while preserving the temporal order between actions that belong to the same CoAs.

The ACD policy is specified according to the CLIPS language discussed in the previous section and it clearly specifies all the CoAs, including the organization of actions within each CoA, their commands, actuators, objects, and pre- and post-conditions. The network data plane configuration is realized through the flow tables in an OpenFlow-based Software Defined Network. The mission requirements include reachability requirements (i.e., which hosts and services must be connected to each other), performance and QoS guarantees on particular flows, and security requirements to preserve the confidentiality, availability, and the integrity of specific critical assets in the network. The performance and QoS requirements are typically described in the Service Level Agreement (SLA) and translated to the QoS parameters of the networking infrastructure in terms of bandwidth, transmission data rates, delays, jitter, etc.

To solve this problem, we have made the following contributions in this chapter:

- We present a formal specification for ACD policies as a set of heterogeneous CoAs and we develop formal properties that are required for safe and effective execution of these CoAs.

- We develop a formal CoA orchestration framework that digests a set of CoAs and produces a global orchestrated CoA workflow (GOAL) that determines the best time to execute each cyber action, such that the properties of safe and effective execution are satisfied. We model this problem as a constraints optimization problem and solve it using the Z3 SMT solver.

- We model the complete data plane configuration of OpenFlow-based software defined networks and provide a model checking approach to verify the satisfaction of the network mission requirements during the CoAs execution. Our model considers the configuration of middle-boxes and the QoS parameters of the networking infrastructure, such as the data rates and of the forwarding

queues, to verify a variety of QoS and security requirements in addition to basic reachability.

3.5 Framework Overview

To orchestrate a safe and efficient CoA execution workflow, we follow the two-step approach depicted in Figure 3.2. The input for our approach is a set of CoAs that belong to multiple rules in the system's ACD policy. These CoAs are triggered as a result of specific security events, and they are set to be executed simultaneously by the different control agents that exist in the system. As illustrated in the figure, the CoAs are expressed using the CLIPS language discussed earlier.

The first step in our approach takes multiple CoAs and produces a single global orchestrated CoA workflow (GOAL). Recall that the GOAL determines the execution time for each of the input actions. The produced GOAL is provably safe because it is synthesized formally to ensure the *Resource Integrity*, *Action Integrity*, and *CoA Concurrency* properties that guarantee a conflict-free and fast execution of the different and interdependent input actions. We designed and implemented our orchestrator as a constraints optimization problem utilizing the optimization problems modulo theories in Z3 [126]. The actions in the output GOAL will then be executed by the appropriate control agents, each at its specific time. We do not enforce a particular set of control agents, but we assume that if an action is included in the ACD policy, there is a mechanism and designated control agent that can execute it. Examples of control agents include generic SDN controllers, such as Floodlight [149], special-purpose SDN controllers, such as ActiveSDN [150], generic enterprise network middle-boxes, such as firewalls and intrusion prevention systems, and virtualization server and data centers controllers. In addition to being safe, the produced GOAL is optimized to have the minimum possible execution time.

The second step in our approach is to verify the satisfaction of network mission requirements during and after the execution of the GOAL produced in the first step.

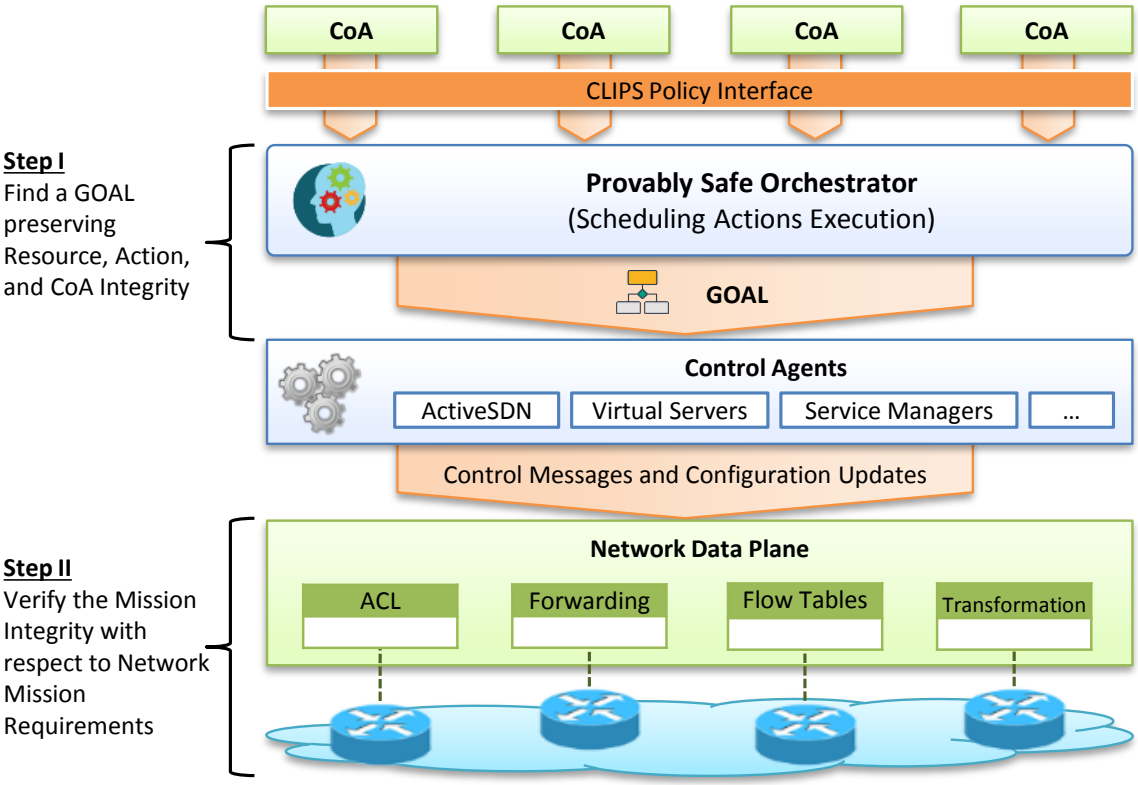


Figure 3.2: Overview of the CoA orchestration and verification framework

While the previous step ensures that the actions within a GOAL do not conflict with each other, it cannot guarantee that the configuration updates, which result from the actions' execution, do not introduce any violations to the network mission requirements. Starting from an initial state, the network data plane configuration will be updated incrementally according to the actions in the GOAL. We use a bounded model checking approach to model the entire data plane configuration and track the changes during the execution of the GOAL. After each configuration update, we verify that the new configuration state satisfies all the network mission requirements (i.e., the reachability, QoS, and security requirements).

3.6 Course of Action Orchestration Safety Properties

In this section, we present the formal models of the reactive policies and the global orchestrated CoA workflows (GOAL). We also define, formally, the properties that

need to be satisfied in the automatically generated GOAL in order to achieve effective and safe CoA execution. We start by defining the CoAs as follows:

Definition 4 (CoA)**.** *We define a CoA as the 7-tuple* $\langle \mathcal{A}, \mathcal{E}, \theta, \sigma, \epsilon, \rho, \varrho \rangle$*, where:*

- $\mathcal{A}$ *is a set of nodes representing the cyber actions. Each action will be associated with a command, actuator, object, pre-, and p/ost-conditions.*
- $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$ *is the precedence relation, that if* $e_{ij} = (a_i, a_j) \in \mathcal{E}$*, then the action* a_i *must be executed before the action* a_j*.*
- $\theta(.) : \mathcal{A} \rightarrow \Sigma$ *associates each action in* $\mathcal{A}$ *to a command from the pre-defined set of commands,* Σ*.*
- $\sigma(.) : \mathcal{A} \rightarrow \mathcal{U}$ *associates each action in* $\mathcal{A}$ *to a unique actuator from the pre-defined set of actuators,* $\mathcal{U}$*.*
- $\epsilon(.) : \mathcal{A} \rightarrow \mathcal{O}$ *associates each action in* $\mathcal{A}$ *to a unique object from the pre-defined set of objects,* $\mathcal{O}$*.*
- $\rho(.) : \mathcal{A} \rightarrow \mathcal{L}_{\mathcal{V}}$ *associates each action in* $\mathcal{A}$ *to a pre-condition expressed in the arithmetic logic* $\mathcal{L}_{\mathcal{V}}$*, where* $\mathcal{V}$ *is the set of control variables that can appear in formulas of this logic.*
- $\varrho(.) : \mathcal{A} \rightarrow \mathcal{L}_{\mathcal{V}}$ *associates each action in* $\mathcal{A}$ *to a post-condition expressed in the arithmetic logic* $\mathcal{L}_{\mathcal{V}}$*.*

Since the first objective of this work is to find a GOAL given a set of CoAs, we define the GOAL in Definition 5. Intuitively, the GOAL is a schedule that determines when to execute the actions in the given set of CoAs, where actions are set to be executed in parallel whenever it is safe. Formally, we define a GOAL as follows:

Definition 5 (GOAL)**.** *For any set of CoAs* $\mathcal{W} = \{w_1, ..., w_n\}$*, a GOAL can be defined as the 2-tuple* $\langle \mathcal{G}, \tau \rangle$*, where:*

- $\mathcal{G} = \bigcup_{w_i \in \mathcal{W}} \mathcal{A}_i$ *is the complete set of actions in all the CoAs. The set* $\mathcal{A}_i$ *is the set of actions in the CoA* w_i*.*

- $\tau(.) : \mathcal{G} \rightarrow \mathbb{Z}^*$ *is the timing function (i.e., execution schedule) that maps each action in* $\mathcal{G}$ *to its execution starting time.*

In the following discussion, we will frequently reference actions in $\mathcal{G}$ using two subscripts, where the first subscript represents the CoA, to which the action belongs, and the second subscript represents the index of the action within its original CoA (i.e., a_{ik} denote the k-th action in the set $\mathcal{A}_i$ of the i-th CoA, w_i).

Based on Definition 5, we can have multiple GOALs for the same set of CoAs with different timing functions (τ). However, not all GOALs are safe and effective. We discuss in the following the properties that need to be met in a GOAL in order to be safe and effective with respect to a given set of CoAs.

3.6.1 Resource Integrity

We identify the following three cases in which the integrity of the resources in the network can be violated due to the concurrent execution of actions that share the same resources (i.e., control variables, objects, and actuators).

- Case I: concurrent modifications to the same control variables, which represent the dynamic system attributes, by different actions that belong to the same or different CoAs may result in access violations and cause some actions to fail. For example, two actions simultaneously migrate two virtual machines to the same server and modifying the control variable related to the destination server's capacity or two actions simultaneously rerouting different flows to the same link modifying the control variable that represents the link's utilization.

- Case II: concurrent actions taken on the same objects can result in conflicting commands, which might cause failures or non-deterministic outcomes. For ex-

ample, two simultaneous actions disable and enable the same service or block and forward the same traffic flows.

- Case III: concurrent actions taken by the same actuator may also result in failures if the actuator is not capable of handling concurrent actions. Actuators may have a limit on the number of simultaneous commands (i.e., requests) that they can handle. For example, two or more actions send control messages to the same OpenFlow switch simultaneously while the switch cannot handle more than one control message at a time.

In order to preserve the resource integrity, the CoAs must be executed in a way that prevents any of these cases. To formally define this property, let us define the function $\varepsilon(.) : \mathcal{G} \rightarrow 2^{\mathcal{V}}$, which maps each action to a set of control variables in $\mathcal{V}$, such that the control variable $v \in \varepsilon(a_{ik})$ if v appears in the post-condition of the action a_{ik} (i.e., if the action modifies the value of that control variable). This function can be constructed statically since the post-conditions of the actions are specified in the action definition as part of the reactive policy and they do not change during the execution. For simplicity, we will also assume the function $\delta(.)$ that represents the execution finishing time of each action. Note that this function is derived from the timing function $\tau(.)$ as follows:

$$\underset{a_{ik} \in \mathcal{G}}{\forall} : \delta(a_{ik}) = \tau(a_{ik}) + d_{ik}$$

where d_{ik} is the constant time period that is required to execute the action a_{ik}. Further, we assume that actuators in the network can handle a limited number of actions concurrently. We call this constant number the *Actuation Threshold*. Based on this, we define the *Resource Integrity* property as follows:

Property 1 (Resource Integrity)**.** *The GOAL $\langle \mathcal{G}, \tau \rangle$ satisfies the resource integrity property if the following conditions hold:*

- *There is no pair of actions that are executed simultaneously and share a control variable or an object. Formally:*

$$\underset{a_{ik},a_{jl}\in\mathcal{G}}{\forall} : (\epsilon_i(a_{ik}) = \epsilon_j(a_{jl})) \rightarrow ([\tau(a_{ik}), \delta(a_{ik})) \cap [\tau(a_{jl}), \delta(a_{jl})) = \emptyset) \quad (3.1)$$

$$\underset{a_{ik},a_{jl}\in\mathcal{G}}{\forall} : (\varepsilon_i(a_{ik}) \cap \varepsilon_j(a_{jl}) \neq \emptyset) \rightarrow ([\tau(a_{ik}), \delta(a_{ik})) \cap [\tau(a_{jl}), \delta(a_{jl})) = \emptyset) \quad (3.2)$$

- *At any point of time, no actuator is executing more actions than its actuation threshold. Formally, let h_u be the actuation threshold for the actuator $u \in \mathcal{U}$, then:*

$$\underset{u\subseteq\mathcal{U}}{\forall} : \underset{b\subseteq\mathcal{G}}{\nexists} : \left(\underset{a_{ik}\in b}{\forall} : \sigma(a_{ik}) = u\right) \wedge \left(\bigcap_{a_{ik}\in b} [\tau(a_{ik}), \delta(a_{ik})) \neq \emptyset\right) \wedge (|b| > h_u) \quad (3.3)$$

The first condition in Property 1 guarantees that if any two actions, a_{ik} and a_{jl}, share the same object (i.e., $\epsilon_i(a_{ik}) = \epsilon_j(a_{jl})$) or any control variable (i.e., $\varepsilon_i(a_{ik}) \cap \varepsilon_j(a_{jl}) \neq \emptyset$), their execution intervals will never coincide with each other. Recall that $\epsilon(.)$ returns the objects of actions and $\varepsilon(.)$ maps an action to a set of control variables and any two intervals $[a, b)$ and $[x, y)$ coincide if $[a, b) \cap [x, y) \neq \emptyset$. For any action $a_{ik} \in \mathcal{G}$, the interval $[\tau(a_{ik}), \delta(a_{ik}))$ represents its execution interval because $\tau(a_{ik})$ represents its execution starting time and $\delta(a_{ik})$ represents its execution end time. The second condition guarantees that for each actuator u in the system, we will never have any set of actions that has a length greater than u's actuation threshold and the execution intervals of all its actions coincide at any point of time.

3.6.2 Action Integrity

The action integrity property is related to the potential interdependence between the consecutive actions. Recall that actions are associated with pre- and post-conditions. If two actions are executed sequential, and the earlier action invalidates the pre-condition of the later, this will cause the later action to fail. Our objective

is to find an execution order for such interdependent actions, if possible, such that their pre-conditions are valid at the time of their execution. A GOAL that satisfies the *Action Integrity* property can achieve that objective. We define this property as follows:

Property 2 (Action Integrity)**.** *The GOAL $\langle \mathcal{G}, \tau \rangle$ satisfies the Action Integrity property if for any action in $\mathcal{G}$, its pre-condition is satisfied at the time in which its execution starts. Formally, let S^t represents the state of the control variables at time t and let T be the total execution time of the GOAL, then:*

$$\underset{\substack{a_{ik} \in \mathcal{G} \\ t \in [1,T]}}{\forall} : (\tau(a_{ik}) = t) \rightarrow \left(S^t \models \rho_i(a_{ik})\right) \tag{3.4}$$

The expression in Property 2 means that for any action a_{ik}, if it is selected to be executed at time t, the state of the control variables at that point of time, S^t, satisfies the action's pre-condition given by $\rho_i(a_{ik})$.

3.6.3 CoA Concurrency

The CoA Concurrency property is related to the total execution time of the GOAL and it guarantees effective and fast GOAL execution. Given that both the Resource Integrity and the Action Integrity properties are preserved, an effective GOAL must maximize the parallelism in executing the actions. To formally define this property, we first introduce the definition of the *Latest End Time* of a GOAL according to Definition 6.

Definition 6 (Latest-End-Time)**.** *Given the GOAL $\langle \mathcal{G}, \tau \rangle$, its latest end time, denoted by $\widehat{\tau}$ is the time at which the execution of all the actions in $\mathcal{G}$ ends. Formally:*

$$\widehat{\tau} = \max_{a_{ik} \in \mathcal{G}} \delta(a_{ik}) \tag{3.5}$$

where $\delta(a_{ik})$ is the execution end time of the action a_{ik}.

Given the definition of the Latest-End-Time, we define the CoA Concurrency property as follows:

Property 3 (CoA Concurrency)**.** *The GOAL $\langle \mathcal{G}, \tau \rangle$ satisfies the CoA Concurrency if there is no other possible GOAL for the same set of CoAs, $\langle \mathcal{G}, \mu \rangle$, that satisfies both the Resource Integrity and the Action Integrity properties and its Latest-End-Time, $\widehat{\mu}$, is less than $\widehat{\tau}$ (i.e., $\widehat{\mu} < \widehat{\tau}$).*

3.6.4 Mission Integrity

The network mission requirements include reachability, QoS, and security requirements. To formally specify the mission requirements, we provide the language shown in Figure 3.3. According to this syntax, a mission requirement, R, can be specified in terms of the two constructs *CanReach* and *Protect*. *CanReach* is used to specify reachability with optional QoS constraints between pairs of locations in the network. The *QoS Constraints* are composed as a set of conditions on the aggregate values of the network infrastructure QoS parameters. The aggregate functions *max*, *min*, *sum*, and *avg* calculate the maximum, the minimum, the sum, and the average values of the parameter κ in each possible path between the specified locations. For example, let the set N be a set of clients in an organization that need to access a particular service s. And let P be a pool of servers that are running that service. Let us assume that the mission of this organization requires that: (1) *"each client should be able to reach at least one server with a data rate greater than or equal to h_{dr}"* and (2) *"the response from any server to any client should not be delayed more that h_{dl}"*. These requirements can be represented using our language as follows:

$$\bigwedge_{n \in N} \bigvee_{p \in P} CanReach(n, p : s, min(D\_RATE) \geq h_{dr})$$

$$\bigwedge_{p \in P} \bigwedge_{n \in N} CanReach(p : s, n, sum(DELAY) \leq h_{dl})$$

$$
\begin{array}{rcl}
\textit{QoS Param } \kappa & ::= & \textit{BW} \mid \textit{D\_RATE} \mid \textit{DELAY} \mid \ldots \\
\textit{Operator } \bowtie & ::= & > \mid < \mid \geq \mid \leq \mid == \\
\textit{predicate } \Phi & ::= & \kappa \mid \mathbb{Z}^+ \mid \textit{max}(\kappa) \mid \textit{min}(\kappa) \mid \textit{sum}(\kappa) \mid \textit{avg}(\kappa) \\
\textit{QoSCond } \Psi & ::= & \Phi \bowtie \Phi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \\
\textit{Location } \mathcal{L} & ::= & < \text{(ip:port)} > \\
\textit{Requirement } R & ::= & \textit{CanReach}(\mathcal{L},\ \mathcal{L},\ \Psi) \mid \textit{Protect}(\mathcal{L}, \mathcal{L}) \mid R \vee R \mid R \wedge R
\end{array}
$$

Figure 3.3: Mission Specification Language Syntax. $\mathbb{Z}^+$ is the set of positive integers.

The construct *Protect* is the opposite of the *CanReach*, and it is used to specify that a threat source should never reach a critical asset, those that if compromised, the system mission will be in jeopardy and they should only be reached from particular trusted locations in the network. Thus, it takes two arguments, the first one is a location that is not trusted and can be considered as a threat source and the second argument is a location that represents a critical asset or service.

The actions of the CoAs make changes to the network data plane as they are executed over time. However, there are no guarantees that these changes will not cause violations to the network mission requirements. Hence, we define the *Mission Integrity* property, which must be met by a GOAL to be considered a safe GOAL. To formally define this property, let $\{C^1, ..., C^T\}$ represent the data plane configuration states at each time unit in the interval $[1, T]$ and let $\mathcal{R}$ represent a set of network mission requirements, the Mission Integrity property is defined as follows:

Property 4 (Mission Integrity)**.** *The GOAL $\langle \mathcal{G}, \tau \rangle$ satisfies the mission integrity property if all the network mission requirements are satisfied in all the intermediate and the final configuration states during and after the GOAL execution. Formally:*

$$\underset{t \in [1,T]}{\forall} : \bigwedge_{r \in \mathcal{R}} \left(C^t \models r \right) \tag{3.6}$$

In section 3.8, we show how we model the complete data plane configuration using bounded model checking and verify the network mission requirements.

3.7 Safe Course of Action Orchestration Synthesis

In this section, we formalize the problem of CoA safe orchestration as a constraint optimization problem. We show how we encode the CoAs and the safe orchestration properties we presented before as SMT constraints in order to orchestrate an effective and safe GOAL.

3.7.1 Formalization of the Course of Action Orchestration

Given a set of cyber actions that belong to a set of CoAs, the problem we formalize in this section is to find the execution starting time for each action, such that the *Resource Integrity*, the *Action Integrity*, and the *CoA Concurrency* properties are satisfied.

Decision Variables. We start our formalization by defining the decision variables, whose values represent the output of this synthesis problem. Since our objective is to find a single value for each action, which represents the action's execution starting time, we define the set $\mathbb{T}$ of integer decision variables. Let $\mathcal{G}$ be a set of cyber actions that encompasses the cyber actions in the set of CoAs, $\mathcal{W}$, we define $\mathbb{T}$ as follows:

$$\mathbb{T} = \{t_{ik} \mid \text{for each action } a_{ik} \in \mathcal{G}\}$$

We follow the same notation we used in Section 3.6, where a_{ik} is the k-th action in the i-th CoA in $\mathcal{W}$. Since the Latest-End-Time of the synthesized GOAL cannot be known until a solution is found, the range of the elements in $\mathbb{T}$ is between 1 and the maximum possible time (i.e., the worst case). The maximum possible time is simply the cumulative sum of the durations of all the actions in $\mathcal{G}$ because in the worst case, all the actions will be executed sequentially. In the following, we denote the maximum possible time by the value T.

Additional Variables. In addition to the set $\mathbb{T}$, we define additional variables that will be used in formalizing our constraints as follows:

- For each action a_{ik} in $\mathcal{G}$, we define: d_{ik} to represent the action's expected duration, u_{ik} to represent its actuator, and o_{ik} to represent its object. Note that d_{ik} is a constant value for each action, and u_{ik} and o_{ik} are mapped through the functions $\sigma_i(.)$ and $\epsilon_i(.)$ of the corresponding CoA, respectively. Note that for each action a_{ik}, we use $\sigma_i(.)$ and $\epsilon_i(.)$, subscripted by i, that belong to its CoA w_i to retrieve its actuator and object.

- For each actuator u in the system, we define the integer variable h_u that represents its actuation threshold.

- We define the matrix $\mathcal{M}$, with a row and a column for each $a_{ik} \in \mathcal{G}$ to determine the existence of shared control variables between actions. Note that this matrix is static, and we can build it in advanced based on the $\varepsilon(.)$ function that we defined before to associate each action to the set of control variables that appear in its post-condition. The values of $\mathcal{M}$ are calculated as follows:

$$\underset{a_{ik},a_{jl}\in\mathcal{G}}{\forall} : \mathcal{M}[a_{ik}, a_{jl}] = \begin{cases} 1, & \varepsilon(a_{ik}) \cap \varepsilon(a_{jl}) \neq \emptyset \\ 0, & \varepsilon(a_{ik}) \cap \varepsilon(a_{jl}) = \emptyset \end{cases} \tag{3.7}$$

- To encode the control variables in the system, we define the set of variables $\mathbb{V} = \{v_1, ..., v_m\}$ that capture the values of the control variables in $\mathcal{V}$. Since the values of the control variables are expected to change over time due to executing cyber actions, one set of variables is not enough to capture all the transient states of the control variables over time. Hence, we define a set of variables for each time (i.e., $\{\mathbb{V}^1, \mathbb{V}^2, ..., \mathbb{V}^T\}$, where $\mathbb{V}^t$ represents the state of the control variables at time t).

.

Precedent Constraints. Having the variables defined, we define the first set of constraints that are related to the order of the cyber actions inside each individual CoA. According to Definition 4, the actions of a CoA are ordered based on the precedence relation $\mathcal{E}_i$ for each $w_i \in \mathcal{W}$. This relation simply entails that if an edge (a_{ik}, a_{il}) exists in $\mathcal{E}_i$, the action a_{ik} must finish before starting the action a_{il}. The precedence constraints of all the CoAs are encoded as follows:

$$\underset{w_i \in \mathcal{W}}{\forall} : \underset{(a_{ik}, a_{il}) \in \mathcal{E}_i}{\forall} : t_{il} \geq (t_{ik} + d_{ik}) \tag{3.8}$$

This set of constraints in terms of the decision variables ensures that each action is not executed until its predecessor is finished. Note that we calculate the execution end time by adding the action duration to its starting time.

Mutual Exclusion Constraints. This set of constraints is required to ensure that the execution intervals of actions that share control variables or objects never coincide in order to satisfy the Resource Integrity property. The mutual exclusion constraints are defined as follows:

$$\underset{(a_{ik}, a_{jl}) \in \mathcal{G}}{\forall} : (o_{ik} = o_{jl}) \rightarrow ((\, t_{jl} \geq t_{ik} + d_{ik}\,) \vee (\, t_{ik} \geq t_{jl} + d_{jl}\,)) \tag{3.9}$$

$$\underset{(a_{ik}, a_{jl}) \in \mathcal{G}}{\forall} : \mathcal{M}\,[a_{ik}, a_{jl}] \rightarrow ((\, t_{jl} \geq t_{ik} + d_{ik}\,) \vee (\, t_{ik} \geq t_{jl} + d_{jl}\,)) \tag{3.10}$$

The first constraint handles the case when two actions share an object. In that case, one of the actions should finish before the other one starts. The same goes for the case of shared control variables in the second constraint, in which we make use of the matrix $\mathcal{M}$ we defined earlier to determine which actions share control variables. If a shared control variable exists (i.e., $\mathcal{M}\,[a_{ik}, a_{jl}] = 1$), the actions' execution intervals

should never coincide.

Actuation Limits Constraints. These constraints are related to the maximum number of actions that can be handled by each actuator in the system. According to our definition of Resource Integrity property, exceeding this limit causes a violation for this property. We formalize a constraint for each actuator to ensure that at each point of time, not more actions than its actuation threshold are being executed. These constraints are encoded as follows:

$$\underset{\substack{u\in\mathcal{U}\\ t\in[1,T]}}{\forall} : \left(\sum_{a_{ik}\in\mathcal{G}} ((u_{ik} = u) \wedge (t \geq t_{ik}) \wedge (t < t_{ik} + d_{ik}))\ ?\ 1 : 0 \right) \geq h_u \tag{3.11}$$

To encode this constraint, we use the *if-then-else* operator, denoted by the notation $(\psi\ ?\ v_1 : v_2)$, which evaluates to the value v_1 if the condition ψ is true, and to the value v_2 if ψ is false. The constraint intuitively counts the number of active actions for each actuator u at each time unit t, those actions whose actuator is u, their execution started at t or before, and their execution will finish after t. We enforce that the number of such actions is less than the actuation threshold of the actuator, h_u.

Action Integrity Constraints. To enforce the Action Integrity property, we encode the constraints that keep track of the control variables' changes over time. We have earlier defined the function $\varepsilon(.)$ to associate each action with a set of control variables it affects. Now, we define another similar function, $\omega(.) : \mathcal{V} \rightarrow 2^{\mathcal{G}}$, that associates each variable with the actions affecting it. Similar to $\varepsilon(.)$, this association is static and can be computed in advance. Although we can encode the Action Integrity constraints without building this function, having it will make the constraints more concise. Note that the mutual exclusion constraints guarantee that a control variable

cannot be modified by more than one action at each time. Thus, for each control variable v, only one of the actions in $\omega(v)$ will change its value at each possible time. We encode the constraints that update the values of control variables as follows:

$$\underset{v\in\mathcal{V}}{\forall} : \underset{\substack{a_{ik}\in\omega(v)\\ t\in[1,T]}}{\forall} : (t = (t_{ik} + d_{ik})) \rightarrow \varrho(a_{ik})[\mathbb{V}^t/\mathcal{V}] \tag{3.12}$$

We use the substitution notation, $\varrho(a_{ik})[\mathbb{V}^t/\mathcal{V}]$, in Equation 3.12 to substitute the variables of $\mathbb{V}^t$ (i.e., the set of variables that represents the state of control variables at time t) for all the occurrences of the corresponding variables in $\mathcal{V}$ in the expression $\varrho(a_{ik})$, where $\varrho(a_{ik})$ is the post-condition of a_{ik} and it is originally expressed in terms of the variables of $\mathcal{V}$. The complete constraint will check which action finishes at each time unit and performs its post-condition in terms of the appropriate state variables. If no action belongs to the the set $\omega(v)$ finishes at that time, the value of the variable at the previous time unit is copied as is.

The previous set of constraints will take care of keeping the values of the control variables up to date over the execution time, all we need to do now is to enforce that the pre-conditions of the actions are satisfied at the time their execution starts. This can be encoded by the following constraints:

$$\underset{\substack{a_{ik}\in\omega(v)\\ t\in[1,T]}}{\forall} : (t = t_{ik}) \rightarrow \rho(a_{ik})[\mathbb{V}^t/\mathcal{V}] \tag{3.13}$$

where $\rho(a_{ik})$ is the expression that represents the pre-condition of a_{ik} that is originally expressed in terms of the variables of $\mathcal{V}$. As indicated through the substitution notation, we substitute the variables of $\mathcal{V}$ by the corresponding variables that belong to the appropriate time unit, $\mathbb{V}^t$.

Execution Time Optimization. The optimization objective of our safe orchestra-

tion is to minimize the total execution time, which is represented by the *Latest-End-Time* according to Definition 6. We first encode the calculation of the *Latest-End-Time* utilizing a chain of *if-then-else* expressions, denoted by $(\psi \ ? \ v_1 : v_2)$, which evaluates to the value v_1 if the condition ψ is true, and to the value v_2 if ψ is false. We define the variable t_{max} to hold the *Latest-End-Time* that is calculated based on the decision variables. To clarify this encoding, let $r = |\mathcal{G}|$ represent the length of the set of actions $\mathcal{G}$, we define the set of intermediate variables $\{m_1, m_2, ..., m_r\}$, where m_i will hold the Latest-End-Time among all the actions from the i-th element of the set $\mathcal{G}$ till the end of the set. For simplicity and for the purposes of calculating t_max only, we will abandon our double subscript indexing of the actions in $\mathcal{G}$ to use a single subscript, where t_i represent the time variable of the i-th action in $\mathcal{G}$ and d_i represent the duration of the same action. We calculate the values of the intermediate variables recursively as follows:

$$m_r = t_r + d_r \tag{3.14}$$

$$\underset{i \in [1, r-1]}{\forall} : m_i = ((t_i + d_i) > m_{i+1}) \ ? \ (t_i + d_i) : m_{i+1} \tag{3.15}$$

The first equation calculates the execution end time of the last action in the list $a_r \in \mathcal{G}$ as the sum of its starting time and duration. Note that this represents the base case of the recursion. For each preceding actions a_i, where $1 \leq i < r$, we compare its execution end time with the highest execution end time among the actions following it in the list $\mathcal{G}$, represented by m_{i+1}. If the execution end time of the action a_i is greater than m_{i+1}, the intermediate variable m_i, will be set to a_i's end time. Otherwise, it will be set to m_{i+1}, which means there is another action in the set $\{a_{i+1}, ..., a_r\}$ that will end after a_i. Based on this encoding, the final *Latest-End-Time* of the entire GOAL will be equal to m_1, which is the cumulative max among all the end times of all the actions in $\mathcal{G}$ (i.e., $t_{max} = m_1$).

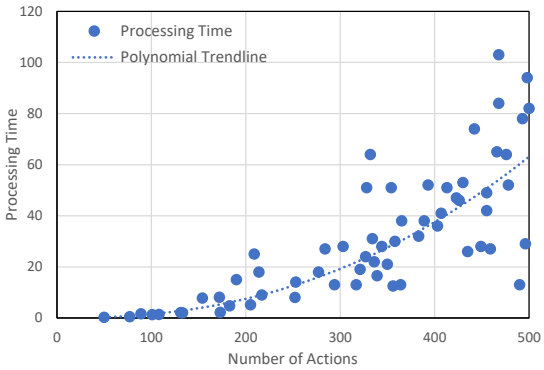


Figure 3.4: The impact of number of actions.

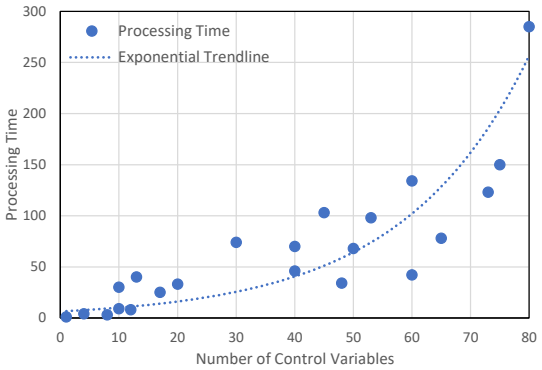


Figure 3.5: The impact of the control variables.

After calculating our objective value, t_{max}, we direct Z3 to find the solution that minimizes it using special instructions provided by the optimization problems modulo theories in Z3 ($minimize(t_{max})$). The solution will be the values of the decision variables $\mathbb{T}$ that ensure the satisfaction of the Resource Integrity, the Action Integrity, and the CoA Concurrency properties.

3.7.2 Evaluation of the Course of Action Orchestration

We evaluated the performance of our orchestration synthesis framework in terms of the time required to generate the optimal GOAL that satisfies the Resource Integrity, the Action Integrity, and the CoA Concurrency properties. We evaluated the performance with respect to the total number of actions and the number of control variables. We show and discuss the results in the following.

The impact of the number of actions. In this experiment, we evaluated the performance of our orchestration framework with respect to the number of actions. We generated random sets of actions that represent multiple CoAs and synthesized the optimal GOAL for each set. The size of the sets was up to 500 actions. In addition, we generated a set of precedence and mutual exclusion relations. The precedence relations emulate the ordered actions within single CoAs. The mutual exclusion

relations emulate the actions that share control and objects and will not be executed concurrently according to the properties we defined for the GOAL. Since we study the impact of the number of actions in this experiment, we added only one control variable to the model.

We ran our framework for more than 60 sets of actions, and the results are depicted in Figure 3.4. For up to 500 actions, the processing time was within 2 minutes, and it exhibits a polynomial growth rate. We believe that the performance of our framework with respect to the number of actions is reasonable since this processing time is required to find the optimal GOAL that is guaranteed to complete at the minimum execution time. However, when this framework is put in practice, the number of actions that should be orchestrated at the same time might be tuned for better performance. For example, if we consider only 100 actions at a time instead of 500, the optimal GOAL can be computed in less than 5 seconds.

The impact of the number of control variables. We define a set of variables in our SMT model to capture the values of the control variables at each possible time during the GOAL execution. Hence, the number of control variables can affect the space and time required to syntheses the optimal GOAL. In this experiment, we study the impact of the number of control variables for a fixed set of 100 actions. We changed the number of control variables from 1 to 80.

The processing time with respect to the number of control variables is depicted in Figure 3.5. We can see that the processing time is very sensitive to the number of control variables and it exhibits exponential growth. However, it is not likely that all the CoAs in an ACD policy are triggered and executed simultaneously. In this case, we need to model only the control variables that belong to the active subset of CoAs. We plan to investigate other modeling techniques and potential heuristics that will help us in tracking the control variables updates more efficiently and scale to support

Table 3.1: State Variables.

Location Variables	
loc	The unique switch ID
tab	The number of the flow table in a switch
Match Fields Variables ($\mathbb{F}$)	
in_port	Ingress port: A physical or logical port.
eth_dst	Ethernet destination MAC address.
eth_src	Ethernet source MAC address.
eth_type	Ethernet type of the OF packet payload.
vlan_id	VLAN-ID from 802.1Q header.
vlan_pcp	VLAN-PCP from 802.1Q header.
ip_prot	IP protocol number.
ip_src	IPv4 source address.
ip_dst	IPv4 destination address.
src_port	Source port number.
dst_port	Destination port number.
ip_dscp	Diff Serv Code Point (DSCP).
ip_ecn	ECN bits of the IP header.
Action Set Variables ($\mathbb{A}$)	
as_pop_vlan	Pop the outer-most VLAN header.
as_push_vlan	Push a new VLAN header.
*as_set_*<Var>	Set the value of *Var* ($Var \in \mathbb{F}$).
as_set_queue	Set the output queue to a specific value.
as_out	Forward the packet to a specific port.

more variables.

3.8 Mission Integrity Verification

In this section, we first present SDNChecker, a bounded model checker that encodes the entire data plane configuration of an OpenFlow-based SDN and verifies properties expressed using the generic LTL language. Then, we show how we translate the mission requirements to LTL expressions and verify them using SDNChecker in order to ensure that a give GOAL satisfies the *Mission Integrity* property.

3.8.1 Network Data Plane Configuration Model

We model the network data plane configuration as a transition system to track the transformations of traffic flows inside the OpenFlow network. The transition system consists of states, which are defined over a set of state variables, connected by transitions, which capture the changes in the state variables based on the actions taken by the networking switches.

States. In our model, the state of the network is determined by the different flows, represented by the flow match fields, that can be transferred through the network and their possible locations. Besides, our model provides the ability to incorporate special-purpose variables for particular mission invariants, such as QoS guarantees. Therefore, the state of the network is modeled by four groups of variables: the location variables, the match fields variables $\mathbb{F}$, the Action Set variables $\mathbb{A}$, and the special-purpose variables $\mathbb{L}$. Formally, the state is encoded by the following characteristic function:

$$\sigma : \boldsymbol{loc} \times \boldsymbol{tab} \times \mathbb{F} \times \mathbb{A} \times \mathbb{L} \rightarrow \{true, false\} \tag{3.16}$$

Table 3.1 shows a list of the location, match fields, and action set variables along with their meanings. In OpenFlow architecture, the Action Set contains a set of actions carried between the flow tables during the pipeline processing. Since the instructions of flow entries can modify the Action Set by inserting or removing actions, we need to keep track of its content during the transitions. The actions in the Action Set are executed when the instruction set of the matching flow entry does not contain a *Goto* instruction. Based on OpenFlow specification, the Action Set includes a maximum of one action of each type. We define a variable for each possible action in the Action Set. If the value of that variable is *null*, the associated action is not part of the Action Set.

Transitions. Transitions are built based on packet transformations across flow tables. We add a transition if we encounter an *Output* action (a transition to a new OF switch) or a *Goto* instruction (a transition to a new table in the same OF switch). Multiple transitions may be associated with the same flow rule. Let us consider the rule r_i that belongs to the table t in the OF switch s. Let $\mathbb{R}_i$ be the set of values specified in the rule r_i for the match fields indexed by the variables' names (i.e., $\mathbb{R}_i[\mathcal{F}]$ is the value of the field $\mathcal{F}$ in the rule r_i). The transitions of the rule r_i are calculated as follows:

$$S_i = (\boldsymbol{loc} = s) \wedge (\boldsymbol{tab} = t) \wedge \bigwedge_{\mathcal{F} \in \mathbb{F}} (\boldsymbol{\mathcal{F}} = \mathbb{R}_i[\mathcal{F}]) \wedge \neg \boldsymbol{S_{-i}} \tag{3.17}$$

$$S'_{i,a} = \bigvee_{k \in \mathbb{O}} \left[\begin{array}{l} (\boldsymbol{loc'} = k_{tar}) \wedge (\boldsymbol{tab'} = 0) \\ \wedge \bigwedge_{\mathcal{F} \in \mathbb{F}} (\boldsymbol{\mathcal{F}'} = \mathbb{R}'_i[\mathcal{F}]) \end{array} \right] \wedge \bigwedge_{\mathcal{A} \in \mathbb{A}} (\boldsymbol{\mathcal{A}'} = 0) \tag{3.18}$$

$$\begin{array}{c} S'_{i,g} = (\boldsymbol{loc'} = loc) \wedge (\boldsymbol{tab'} = new\_table) \wedge \\ \bigwedge_{\mathcal{F} \in \mathbb{F}} (\boldsymbol{\mathcal{F}'} = \mathbb{R}'_i[\mathcal{F}]) \wedge \bigwedge_{\mathcal{A} \in \mathbb{A}} (\boldsymbol{\mathcal{A}'} = \mathbb{R}'_i[\mathcal{A}]) \end{array} \tag{3.19}$$

$$\begin{array}{c} S'_{i,as} = \mathbb{R}'_i[AS\_OUT] \rightarrow [(\boldsymbol{loc'} = \mathbb{R}'_i[AS\_OUT]) \wedge \\ (\boldsymbol{tab'} = 0) \wedge \bigwedge_{\mathcal{F} \in \mathbb{F}} Exp[\mathcal{F}] \wedge \bigwedge_{\mathcal{A} \in \mathbb{A}} (\boldsymbol{\mathcal{A}'} = 0)] \end{array} \tag{3.20}$$

In equation 3.17, we calculate the flow space S_i of the rule (i.e., all the flows that match the values in the rule r_i and do not match another rule with higher priority). The expression S_{-i} captures the flow space for all the rules that have higher priority in the same flow table.

The flow space calculated in Equation 3.17 encodes the current state of the transitions associated with the rule r_i. To encode the next state(s), we use the primed variable $\mathcal{F}'$ to represent the next state variable of the field $\mathcal{F}$. We also define the set $\mathbb{R}'_i$ that keeps the values of the next state variables during the execution of the rule's

instructions list. Initially, the next state variables have the same values as the current state ones (i.e., $\forall_{\mathcal{F}\in\mathbb{F}} : \mathbb{R}'[\mathcal{F}] = \mathcal{F}$). There are three sources of transitions in a flow entry: (1) The *Output* action(s) in the *Apply-Actions* instruction's actions list, (2) The *Goto* instruction, which transfers packets processing from one flow table to another in the same OpenFlow switch, and (3) The *Output* action in the packet's Action Set. If the instructions list of a flow entry does not contain a *Goto* instruction, the actions in the packet's Action Set are executed, where the *Output* action is executed last. If no *Output* action exists in the Action Set, the packet is dropped. The next states of the three cases are encoded in equations 3.18, 3.19, and 3.20, respectively. The set $\mathbb{O}$ of Equation 3.18 includes the *Output* actions in the *Apply-Actions* instruction's actions list, where k_{tar} is the target switch of the *Output* action at index k. Note that in equations 3.18 and 3.20, the packet is transferred to another switch; hence, the Action Set's variables ($\mathbb{A}$) need to be cleared. The *Exp* set in Equation 3.20 captures the effects of *Set* actions in the Action Set. For example, the final value of the next state variable *ip_src'* depends on the value of the Action Set variable *as_set_ip_src'*. *Exp*[*ip_src*] captures the conditional statement shown in Equation 3.21 that encodes the value of the variable *ip_src'* depending on the value of *as_set_ip_src'* using the notation (ψ ? $v_1 : v_2$) that represents the *if-then-else* operator.

$$ip\_src' = \mathbb{R}'_i[as\_set\_ip\_src] \ ? \ \mathbb{R}'_i[as\_set\_ip\_src] : \mathbb{R}'_i[ip\_src] \tag{3.21}$$

The complete transition relation of the rule r_i based on Equations 3.17 to 3.20 is represented as

$$T(r_i) = S_i \wedge (S'_{i,a} \vee S'_{i,g} \vee S'_{i,as}) \tag{3.22}$$

Global Transition Relation. The global transition relation is the disjunction of the transition relations of all the rules. For the network N that has $\$$ switches, the

global transition relation $T_g(N)$ is calculated as

$$Tg(N) = \left[\bigvee_{s \in \mathbb{S}} \bigvee_{t=1}^{len(s)} \bigvee_{i=1}^{len(t)} T(r_{s,t,i}) \right] \wedge \bigwedge_{\mathcal{L} \in \mathbb{L}} f(\mathcal{L}, \mathcal{L}') \tag{3.23}$$

Where $len(s)$ is the number of flow tables in the switch s, $len(t)$ is the number of rules in the flow table t and $r_{s,t,i}$ is the rule i in the flow table t that belongs to the switch s. $T(r_{s,t,i})$ is calculated according to Equation 3.22. $\mathbb{L}$ is the set of special-purpose variables that will be defined later based on the desired verification along with their transition relation, where $f(\mathcal{L}, \mathcal{L}')$ denotes the transition of the special-purpose variable $\mathcal{L}$ expressed in arithmetic logic.

3.8.2 SMT-based Bounded Model Checking

Encoding the network data plane configuration model as an SMT-based satisfaction formula is done by unfolding the model for k steps starting from the initial states S_0. Let $V = \mathbb{F} \cup \mathbb{A} \cup \mathbb{L} \cup \{loc, tab\}$ be the set of variables that represent a state in the system, and let V_i denote the set of variables that represent the state i. We use $I(V_i)$ to represent a relation in terms of the variables V_i. The global transition relation computed in Equation 3.23 is expressed in terms of the current and next state variables as $T(V_i, V_j)$ for a transition from state i to state j. The network properties are encoded as a relation in terms of the variables $\{V_0, V_1 \dots V_k\}$. The formula representing the unfolded system M_k can be represented as:

$$M_k = I(V_0) \wedge \left(\bigwedge_{i=1}^{k} T(V_{i-1}, V_i) \right) \wedge I(V_0, V_1, \dots V_k) \tag{3.24}$$

The transition relation is unfolded starting from the base transition relation computed in Equation 3.23 as follows: given the base transition $T(V, V')$ and a bound k, (1) we define the variables set V_0, (2) we initialize i to 1, and we repeat the following three steps for all $i \leq k$, (3) we define the new variables set V_i, (4) we construct a

$$
\begin{array}{rcl}
[q]_i^k & \Leftrightarrow & \Psi_i \\
[\neg q]_i^k & \Leftrightarrow & \neg [q]_i^k \\
[q_1 \wedge q_2]_i^k & \Leftrightarrow & [q_1]_i^k \wedge [q_2]_i^k \\
[q_1 \vee q_2]_i^k & \Leftrightarrow & [q_1]_i^k \vee [q_2]_i^k \\
[\boldsymbol{X}\ q]_i^k & \Leftrightarrow & [q]_{i+1}^k \text{ if } i < k \text{ , and } \bot \text{ otherwise} \\
[\boldsymbol{F}\ q]_i^k & \Leftrightarrow & \bigvee_{j \in [i,k]} [q]_j^k \\
[\boldsymbol{G}\ q]_i^k & \Leftrightarrow & [q\ \boldsymbol{U}\ (\boldsymbol{loc} = sink)]_i^k \\
[q_1\ \boldsymbol{U}\ q_2]_i^k & \Leftrightarrow & \bigvee_{j \in [i,k]} \left([q_2]_j^k \wedge \bigwedge_{l \in [i,j)} [q_1]_l^k \right) \\
[q_1\ \boldsymbol{R}\ q_2]_i^k & \Leftrightarrow & \bigwedge_{j \in [i,k]} \left([q_2]_j^k \vee \bigvee_{l \in [i,j)} [q_1]_l^k \right)
\end{array}
$$

Figure 3.6: Property Encoding.

new formula $T(V_{i-1}, V_i)$ by replacing all variables of V and V' with the corresponding variables from V_{i-1} and V_i, (5) we sdd the new transition formula to the model as an assertion.

The network properties need to be translated to SMT expressions as well. We use the standard LTL specification language as generic means to specify system properties. A property in LTL can contain the temporal connectives: *next* (**X**), *eventually* (**F**), *global* (**G**), *until* (**U**) and *release* (**R**) operators. Let Ψ_i be a constraint in the property q expressed in terms of the i-th state variables (V_i); we denote the SMT expression of a property q at point i on a path given a bound of k as $[q]_i^k$. Based on this notation, the property encoding as SMT formula can be recursively defined as shown in Figure 3.6. The translation is based on the standard semantics of LTL operators except for the $\boldsymbol{G}$ operator. To encode the $\boldsymbol{G}$ operator, we forward the packets that are dropped or have reached their destinations to a designated node called *sink*. If a packet reaches the *sink* node then the path is terminated. The property $[\boldsymbol{G}\ q]$ is simplified to $[q\ \boldsymbol{U}\ (\boldsymbol{loc} = sink)]$ in our model, which means that q holds until the

Table 3.2: Examples of Mission Requirements

Property	**LTL Property Expression**
1. CanReach(s, d, _)	$P_1 = (loc == s) \wedge (IP\_SRC == s) \wedge (IP\_DST == d) \rightarrow \boldsymbol{F}[(loc == d) \wedge (IP\_DST == d)]$
2. CanReach(s, dc, min(DR) $\geq \tau$*)*	*def* $\boldsymbol{DR}$ = map ($\boldsymbol{loc}$) {{1, 512}, {2, 1024} ...} $P_2 = (loc == s) \wedge (IP\_DST == dc) \rightarrow \boldsymbol{F}\,[(min(DR) < \tau) \wedge (loc == dc))]$
3. Protect(ts, cd)	$P_3 = (IP\_DST == cd) \rightarrow ([loc \neq ts]\ \boldsymbol{U}\ [loc == cd])$

path is terminated.

After unfolding the transition relation and the LTL properties, the complete SMT formula M_k that is calculated based on Equation 3.24 is fed to Z3 SMT solver, which determines if the formula is satisfiable or not.

3.8.3 Verifying the Satisfaction of Mission Requirements in a Data Plane Configuration Snapshot

In this section, we show the steps we follow to translate the mission requirements to LTL properties and verify them using our model checker given a snapshot of the network data plane configuration.

The *CanReach* construct that we defined in the mission requirements specification language can specify reachability requirements with and without any constraints on the QoS the QoS between sources and destinations in the network. The first example in Table 3.2 represents a basic reachability requirement without QoS constraints. The corresponding LTL expression, P_1, is satisfied if packets that are initially located at location s will be located at the destination d at any future state. If the *CanReach* construct specifies a QoS constraint, the source should be able to reach the destination and the QoS, in terms of the QoS parameters, should meet particular thresholds. In this case, we follow the following steps to translate such requirements to LTL:

- The QoS parameters such as bandwidth, data rate, and delay are defined as special-purpose variables and encoded using the special keywords *def* and *map*.

We assume that every device/port in the network will have a value for each quality of service parameter. We use *def* to declare a new special-purpose variable that is not part of the basic state variables. Any variable defined in this way can be used anywhere in the LTL properties. The *map* construct is used to assign values to the special-purpose variables based on the values of other basic state variables. For example, in the second property in Table 3.2, we define a special-purpose called DR using the *def* keyword to hold the data rate of transmission queues inside OpenFlow switches. Then, we use the *map* keyword to assign values to this variable based on the variable *loc*. At any state, we check the value of the *queue* and assign a value to DR accordingly (i.e., if $loc = 1$ then $DR = 512$, if $loc = 2$ then $DR = 1024$, ...).

- If any aggregate functions are used, they will also be encoded as special-purpose variables and their values are computed accumulatively at each transition. The use of SMT allows us to compute their commutative values utilizing the basic arithmetic and conditional operators.

- Since the QoS parameters and the aggregate values are defined as special-purpose variables, the QoS constraints are encoded directly into SMT by replacing the QoS parameters defined in the constraints with the corresponding special-purpose variables.

The second property in Table 3.2 is an example of a mission requirement, at which the traffic from a particular server s to the data center dc does not pass through a location that has a data rate less than a specific threshold τ. Such a requirement is crucial in several solutions that have proposed techniques for task scheduling in MapReduce architectures [151]. Since this requirement depends only on one QoS parameter, data rate, we define the DR parameter and assign its value using a *map* between the OpenFlow switch ID and the data rate of that particular queue. Next, we

express the requirement as an LTL expression that contains a constraint in terms of the new special-purpose variable DR and the aggregate function min that is satisfied if the data rate in all the paths between the source and the destination does not drop below the given threshold.

The *Protect* requirements specify that particular untrusted locations should never reach other critical assets or services in the network. We write LTL expressions that will be satisfied if all the traffic destined to the critical locations in the network, it should never be originated or passes through the untrusted locations before reaching the critical locations. The third property, P_3 in Table 3.2, shows how we represent the *Protect* properties using the Until LTL operator. In this example, traffic reaching the critical location cd can be at any location except the untrusted locations ts.

3.8.4 GOAL Verification

We show in the previous section how we model a snapshot of the network data plane configuration and verify that it satisfies the network mission requirements. However, the *Mission Integrity* property requires the satisfaction of the mission requirements at any transient state of the data plane configuration during the GOAL execution because the cyber actions may frequently change the configuration. To accomplish this, we need to verify the network mission requirements after each configuration update incrementally.

We follow the following procedure to verify the Mission Integrity property with respect to the GOAL $\langle \mathcal{G}, \tau \rangle$. We use C^t to represent the network data plane configuration snapshot at time t.

1. We start from the initial configuration snapshot C^1 at $t = 1$.

2. We increment the time t and enumerate the subset of actions $g \subset \mathcal{G}$ that end at time t. Note that due to the concurrency in actions' execution, we might have more than one action finishing at the same time.

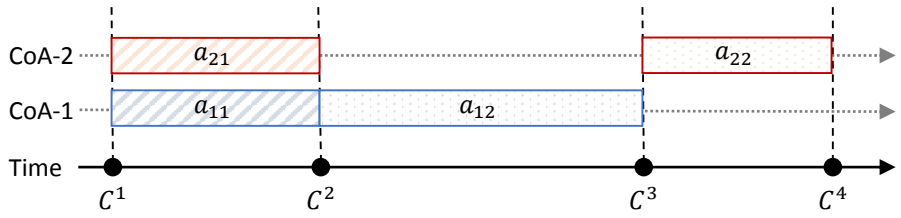


Figure 3.7: Incremental Verification of Network Mission Requirements.

3. We generate a new incremental snapshot of the network data plane configuration C^t by implementing the changes dictated by all the actions whose execution ends at time t.

4. We verify that C^t satisfies all the mission requirements according to the approach discussed in Section 3.8.3. If C^t violates any of the mission requirements, then the $\langle \mathcal{G}, \tau \rangle$ does not satisfy the *Mission Integrity* property.

5. We return to step 2 and repeat as long as t lies within the interval $[1, \widehat{\tau}]$, where $\widehat{\tau}$ is the Latest-End-Time of the GOAL $\langle \mathcal{G}, \tau \rangle$.

We illustrate this process by the example shown in Figure 3.7. We see in the figure two CoAs, where each consists of two actions. The GOAL determines that both actions a_{11} and a_{21} will start together because they do not operate on shared resources. However, the action a_{12} cannot be executed at the same time with a_{22}, which means that we need to wait until the execution of a_{12} finishes to start executing a_{22}. The actions a_{11} and a_{21} have the same duration, and they will end at the same time causing the data plane configuration to transform from state C^1 to state C^2. Then, the action a_{12} will change the data plane configuration to the state C^3. And finally, the last state C^4 will be generated after executing the action a_{22} on the state C^3. The states $\{C^1, C^2, C^3, C^4\}$ represent different snapshots of the network data plane configuration, and we verify the network mission requirements at each snapshot.

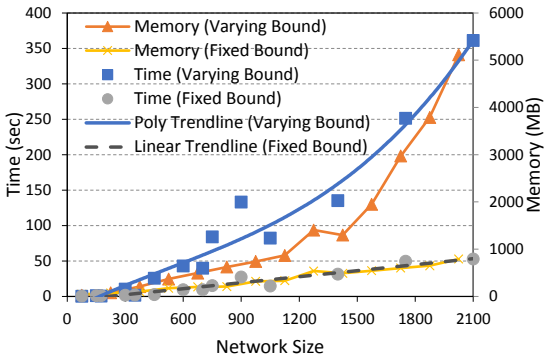


Figure 3.8: The impact of network size.

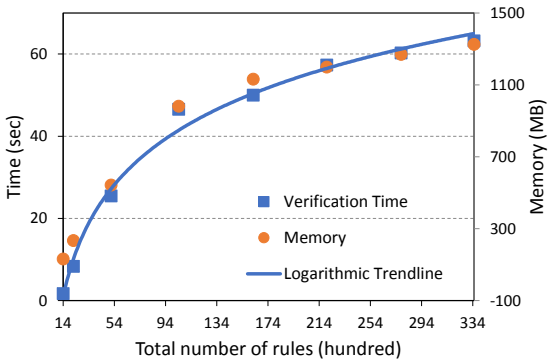


Figure 3.9: The impact of the table size.

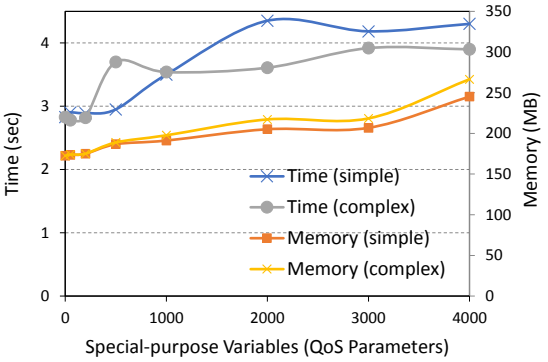


Figure 3.10: The impact of the number of QoS Parameters.

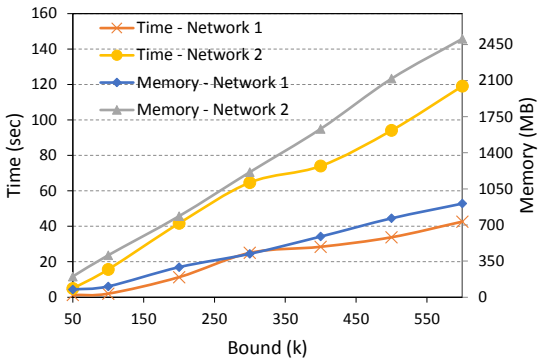


Figure 3.11: The impact of the bound.

3.8.5 Evaluation of Mission Requirements Verification

We implemented a tool using C#.NET that automatically reads the complete data plane of an OpenFlow network and provides a GUI for the user to specify the mission requirements, the resistance techniques specification, and the resistance properties. This tool uses the Z3 .NET API to compose the proper SMT expressions and generate an SMT file that contains the complete problem encoded as SMT assertions. We then feed this file to the Z3 SMT solver. We ran all experiments on a standard PC with 3.4 GHz Intel Core i7 CPU and 16 GB of RAM.

To evaluate the performance and the scalability of our framework, we measure the time required to solve the satisfaction problem with respect to multiple parameters, such as the network size, the sizes of flow tables, the number of special-purpose

variables, and the complexity of the mission requirements and isolation specifications.

The scalability of model checking tools depends on the size of the problem in terms of the state space and the number of transitions. In our case, the size of the problem depends on multiple network parameters, such as the network size and the length of flow tables. Due to the lack of real large-scale networks configurations, we synthesized a number of network instances with given parameters based on tree topology, where the leaves are hosts, and the inner nodes are OF switches. In all the generated networks, the core switches do not constitute more than 15% of the total nodes in the network. We then generate the constraints satisfaction problem and solve it using the Z3 SMT solver. We record the time and memory required to solve the problem. As presented earlier, the network mission requirements are translated to LTL expression with the support of special-purpose variables. Hence, the following evaluation applies for all the types of mission requirements.

The impact of network size. In this experiment, we study the impact of the overall number of network nodes. We generated many networks whose sizes varied from 75 to 2100 nodes. For each of them, we report the average time and space of verifying 20 reachability properties between random pairs of hosts. We experimented with two different settings of the bound k. In one setting, the bound k varies based on the number of flow tables. It was set to a constant value in the other setting regardless of the network size. Each switch in these experiments contains up to two flow tables with an average length of 50 flow rules. Figure 3.8 shows the results of this experiment. We can see that in the case of a fixed bound, the time and space requirements are linear with respect to the network size. However, in the case of varying bound, the performance is affected by both: the network size and the bound. The growth rate with respect to the network size is best described by a quadratic polynomial function.

The impact of the flow table size. In this experiment, we generated various networks with the same number of nodes (300 nodes), but with varying flow table sizes measured by the number of rules. All the rules have the same structure (i.e., the same number of instructions and the same length of actions lists). As reported in Figure 3.9, the total number of rules ranged from 1.4 to 33.5 thousand rules. The results show that the growth in time and space tends to stabilize after a threshold (i.e., increasing the flow table size does not significantly increase the requirements). We believe this behavior is due to the compact representation of assertions in Z3 that merges similar expressions.

The impact of the bound k. The bound determines the number of steps to consider in the bounded model. For each step, a new set of variables and a replica of the transition relation is added. To study the impact of the bound value, we generated two networks: *Network 1* that consists of 300 nodes and *Network 2* that consists of 600 nodes. Each switch in the network has up to 2 tables and an average of 50 flow rules per table. For each network, we ran our experiment multiple times, selecting a different bound each time. The bounds ranged from 50 to 600. Figure 3.11 shows that in both networks the time and space requirements increase linearly with respect to the bound.

The impact of special-purpose variables. We conducted an experiment to study the impact of special-purpose variables, which depends on the QoS parameters, diversity attributes, and the number of countermeasures. For this purpose, we ran our framework against a network of 300 nodes with a varying number of special-purpose variables that ranged from zero to 5000. Moreover, we defined two types of special-purpose variables, namely *simple* and *complex*. The *simple* variables were defined

as additive operations (e.g., *sum* of delays over a path), while the *complex* includes non-linear multiplication. Figure 3.10 reports the time and space for both types. We can see that the number of special-purpose variables of both types has a minor impact on the time and space requirements. They remain almost constant even with a large number of special-purpose variables.

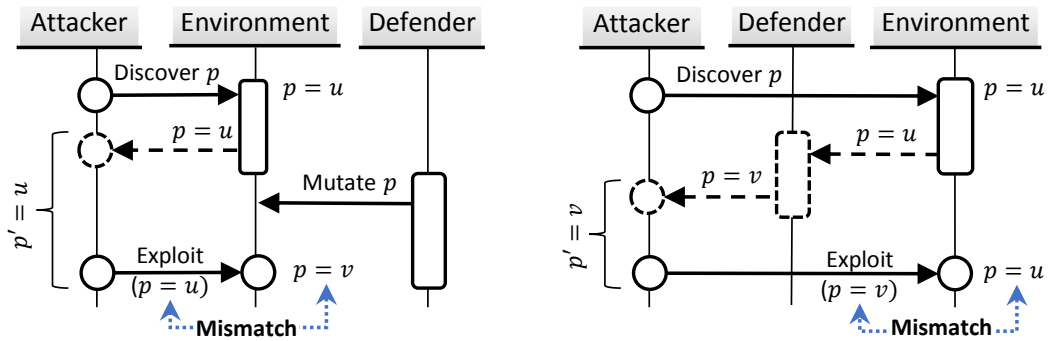
Discussion. The evaluation of mission requirements verification using bounded model checking shows that it takes a considerable amount of time to verify requirements for large-scale networks, which makes it inappropriate for real-time verification. This is due to the fact that the required incremental updates for the network configuration state after each action execution can lead to a significant delay in the execution. We have investigated a solution to this problem by considering all the actions in the GOAL and introducing state-update activation variables in our model checking approach. These activation variables will activate and deactivate the appropriate state changes based on the execution order of the actions. As a result, our network configuration model will capture all the possible updates in the system with respect to a bounded set of actions, which eliminates the need to reconstruct the model after each update. This technique makes our model checker more efficient for real-time incremental verification.

CHAPTER 4: AUTOMATED EXTRACTION OF AGILITY PARAMETERS FOR CYBER DETERRENCE AND DECEPTION PLANNING

Malware attacks have evolved to be highly evasive against prevention and detection techniques. It has been reported that at least $360,000$ new malicious files were detected every day and one ransomware attack was reported every 40 seconds in 2017 [152]. This reveals severe limitations in prevention and detection technologies, such as anti-virus, perimeter firewalls, and intrusion detection systems. Cyber deterrence and deception have emerged as effective defenses for cyber resilience [99] that can corrupt and steer adversaries' decisions to (1) *deflect* them to false targets, (2) *distort* their perception about the environment, (3) *deplete* their resources, and (4) *discover* their motives, tactics, and techniques [153, 154].

4.1 Motivation

Advanced cyber threats often start with intensive reconnaissance by interacting with cyber systems to learn the true values of its parameters, such as keyboard layout, geolocation, hardware ID, IP address, service type, OS/platform type, and registry keys to discover vulnerable targets and execute their objectives. We call such parameters "Critical Parameters". Cyber agility can be particularly effective during this phase by providing false perceptions about the configuration of the cyber environment to achieve the deterrence and deception goals [154]. There are two key mechanisms to accomplish this: (1) parameter *mutation*, to frequently change the ground truth (i.e., the real value of the system parameter) of the cyber configuration, such as IP address [155] or route [156], or (2) parameter *misrepresentation*, to change only the value returned to the attacker (i.e., the ground truth is intact). We call such



(a) Mutation. (b) Misrepresentation.

Figure 4.1: Attacker's Dependency on System Parameters.

critical parameters that can be feasibly and cost-effectively mutated or misrepresented the "Agility Parameters". Figure 4.1 shows these two mechanisms with respect to the environment parameter p. It shows that the adversary knowledge about p was falsified by either changing p to a new value (mutation) or by lying about its true value (misrepresentation). Both mechanisms are needed in cyber agility because mutation can be infeasible or uneconomical, and misrepresentation can sometimes be uncovered.

An effective planning of cyber deterrence and deception requires a sequence of mutations and/or misrepresentations of agility parameters in order to steer the adversary to the desired goals. The key challenge that we address in this research is to identify the most appropriate *agility parameters* against any arbitrary malware by symbolically executing and analyzing the malware binary.

While some previous work, such as Moving Target Defense (MTD) [157, 49, 50, 158, 156, 159, 160] and decoy technologies [42, 44, 46] attempt to invalidate attackers' perceptions, the agility parameters and techniques were engineered manually, which significantly limits the ability for creating deterrence and deception actions automatically against novel malware. The ultimate goal of this research is to automate cyber agility planning against novel malware attacks. Thus, unlike IPS/IDS, our objective is to deter and deceive, rather than detect and block, by enabling the malware to

execute in a real or virtual environment configured based on the extracted agility parameters. To the best of our knowledge, this is the first work that uses automated reasoning to infer effective agility parameters based on malware analysis.

Our framework can be incorporated in the production systems to automatically analyze, deter, and deceive malware without human intervention. Although there are various techniques to trigger the detection of malware such as signature analysis [161], behavior analysis [162], decoy software [46] and decoy bugs [42], the focus of this research is on extracting the agility parameters to automate creating deterrence and deception schemes.

4.2 Related Work

In this section, we review the recent research conducted on malware analysis and symbolic execution, highlighting the significant objectives and motivations for using malware analysis. Besides, we discuss the latest research on automated design and synthesis of cyber agility and moving target defense.

Analyzing and exposing behaviors of malware has been extensively discussed in the literature [163, 164, 165, 166, 167, 168]. Forced execution [169] and X-Force [170] were designed for brute-force exhausting path space without providing semantics information for each path's trigger condition. To discover the trigger conditions, Brumley *et al.* [171] applied taint analysis and symbolic execution to derive the condition of malware's hidden behavior. Moser [172] introduced a snapshot based approach that could be applied to expose malware's environment-sensitive behaviors. Hasten [173] was proposed as an automatic tool to identify and skip malware's stalling code. In [174], Kolbitsch *et al.* proposed a multipath execution scheme for Java-script-based malware. Other research [175, 169] proposed techniques to force the execution of different malware functionalities.

While we need to analyze malware in this work, we have a different goal: to automatically discover system parameters that can be mutated or misrepresented to

deceive malware, rather than detect it. We can benefit from all existing malware analysis techniques, and in this work, we choose symbolic execution in particular.

In another direction of research, randomization and moving target defense are well-investigated techniques toward agile cyber that can proactively disrupt advanced attacks. Randomization techniques, such as instruction set randomization [176], compiler-generated software diversity [177] and address space layout randomization [159], introduce unpredictability to confuse the attackers and invalidate their assumptions about the system. Moving target defense techniques, such as [157, 178, 179, 49, 50, 180, 156], mutate specific static system parameters proactively over time. For example, NASR [181] randomizes IP addresses based on DHCP over time. Similarly, the authors in [50] propose to periodically migrate VMs to make it harder for adversaries to locate targeted VMs. In another direction, deception techniques, such as honeynets and honeypots [42, 43, 44, 46], divert attackers away from their targets to consume their resources and protract their reconnaissance. Although these techniques and many other similar ones have been successful, within acceptable performance overheads, in deterring and deceiving the targeted attacks, they were designed in an ad-hoc manner to counteract specific attacks. Our proposed analytic framework makes this process systematic and decreases the need for manual intervention and the reliance on human intelligence to design effective active cyber deception schemes.

4.3 Problem Statement and Contributions

This chapter addresses the problem of automatic extraction of the agility parameters, whose value mutation or misrepresentation can invalidate the attacker's perception about the system in order to achieve deterrence and deception goals. Given the attack binary code and sufficient understanding of common system and library APIs through which the malware communicates with the environment, our objectives are to identify (1) *"what"* agility parameters are the most appropriate to accomplish the deterrence and deception goals and (2) *"how"* to effectively mutate or misrepresent

their values.

To accomplish this, we present a systematic approach and an automated tool to analyze any malware binary code and extract the agility parameters. This requires an agility-oriented analysis of malware behaviors that goes beyond existing dynamic analyses, which are usually tailored towards attack detection and prevention. Thus, we extended the existing dynamic analysis and symbolic execution frameworks to track the execution of malware symbolically and analyze system and library API calls that particularly entail interactions with the cyber environment. We then, identify the agility parameters that can impact the malware decision-making. Since these parameters can be inter-dependent and may exhibit varying effectiveness and cost, our analysis guarantees the selection of consistent sets of parameters that can obtain resilient and cost-effective deterrence and deception plans. We summarize our contributions as follows:

- We present *gExtractor*, an agility-oriented malware symbolic execution analysis tool that intercepts and tracks the malware interactions with the environment, and maps them to specific agility parameters.

- We developed formal constraints to extract agility parameters that constitute consistent, resilient, and cost-effective deterrence and deception.

- We implemented *gExtractor* and evaluated it using various types of malware codes. Our evaluation demonstrates the ability of *gExtractor* to extract effective agility parameters, as manually verified by experts.

To demonstrate the value of our approach, we used *gExtractor* to analyze over fifty recent malware instances. We present in the evaluation section representatives of four major malware families: Worms, Cryptocurrency-mining malware, Ransomware, and Credential-stealing malware. For each representative, we modeled its behavior, extracted candidate agility parameters, and showed how they can be used to design

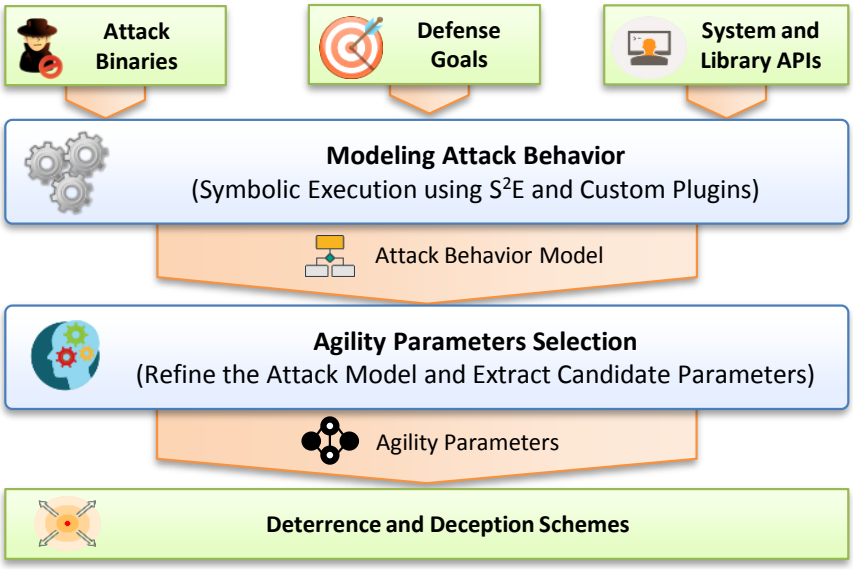


Figure 4.2: Agility Parameters Extraction Framework Overview.

different deception schemes for different goals. Our case studies presented in this chapter show that our approach can discover effective agility parameters. For example, the bitcoin miner case study reveals multiple parameters including Windows Script Host engine, win32_processor WMI (Windows Management Instrumentation) class that can be used to deflect the malware by misinforming false platform types, and the bitcoin hashing results that can be used to corrupt the results in the mining pool and deplete the adversary's resources (i.e., score).

4.4 Framework Overview

To analyze the attacker behavior and infer its dependence on the system parameters, we follow the approach depicted in Figure 4.2. Since every interaction between the attack binary and its environment, such as accessing the file system, sending a packet over the network, or launching other programs, requires the attack binary to make use of one or more of the operating system or user library APIs, our approach intercepts those interactions. The knowledge that the attacker gains about the environment is transferred through the output arguments of these APIs. Thus, we construct an attack behavior model based on the intercepted calls, the informa-

tion flow between them, and the decisions taken by the attack binary based on that information. Then, we refine the attack behavior model to prune out any information that is irrelevant to the desired deterrence and deception goals. We use the refined model to extract candidate sets of agility parameters that can be utilized in different deterrence and deception schemes.

We implement this approach through *gExtractor* on top of the Selective Symbolic Execution engine (S^2E) [88]. Specifically, *gExtractor* takes three inputs: the attack binary code, a specification of the desired deterrence and deception goals, and selected system and user library APIs. With the assistance of our custom S^2E plugins, *gExtractor* executes the attack binary in a real controlled environment, intercepts the system and library API calls, marks the relevant symbolic information, guides the path exploration of the symbolic execution (e.g., pruning back edges and limiting the number of times a loop body is executed), and collects textual logs. This facilitates the construction of a comprehensive malware behavior model, which is refined by (1) pruning out execution paths that are not relevant to the desired deterrence and deception goals and (2) eliminating the don't-care symbolic variables that have no impact on the goals. We use the remaining parameters after the refinement process to construct a constraints optimization problem, and we solve it using the Z3 solver [126] to find an optimal set of parameters that satisfies our agility criteria.

We discuss in the following our broad design decisions we have taken in this work and the motivations behind them:

- Binary files over the source code. Although static analysis of the source code provides a complete understanding of attack behavior, the source code is not available for most attacks. Hence, we chose to analyze malware binaries. There are many public repositories, such as *VirusSign* [182], that collect hundreds of thousands of malware samples from different malware families. Our approach can take any malware binary, even if it has previously not been investigated or

indexed in malware repositories before. Thus, our approach does not require any form of high-level specification or analysis reports.

- Symbolic execution over pure static and dynamic analyses. Different static and dynamic malware analyses have been proposed and used in literature. However, each of them has some limitations. In principle, dynamic techniques cannot guarantee complete coverage of execution, while code obfuscation and process injection [183] can evade static techniques. The focus of this work is how we can leverage the outcome of attack binary analysis for deterrence and deception. Hence, we used symbolic execution, which can cover all execution paths, given that symbolic variables are marked correctly. In addition, the code is executed in a real controlled environment, which makes evasion harder. We acknowledge that symbolic execution can suffer from scalability issues and we are investigating techniques to overcome this in our research.

4.5 Attack Behavior Model

The *attack behavior model* describes how the attack behaves based on the results of its interaction with the environment. The malware interacts with its environment through system and user library APIs characterized by their input and output arguments. Some of these arguments may be attacker-specific variables and cannot be controlled by the environment, while other parameters can be reconfigured or misrepresented. We assume that mapping between the selected system or library APIs' arguments and the corresponding parameters in the environment, which include *files*, *registry entries*, *system time*, *processes*, *keyboard layout*, *geolocations*, *hardware ID*, *C&C*, *Internet connection*, *IP address* or *host name*, and *communication protocols*, is given. For example, the *from* argument of the *recvfrom* API can be mapped to a system parameter that represents the IP address of the sender machine. Currently, the mapping of arguments to system parameters is performed manually.

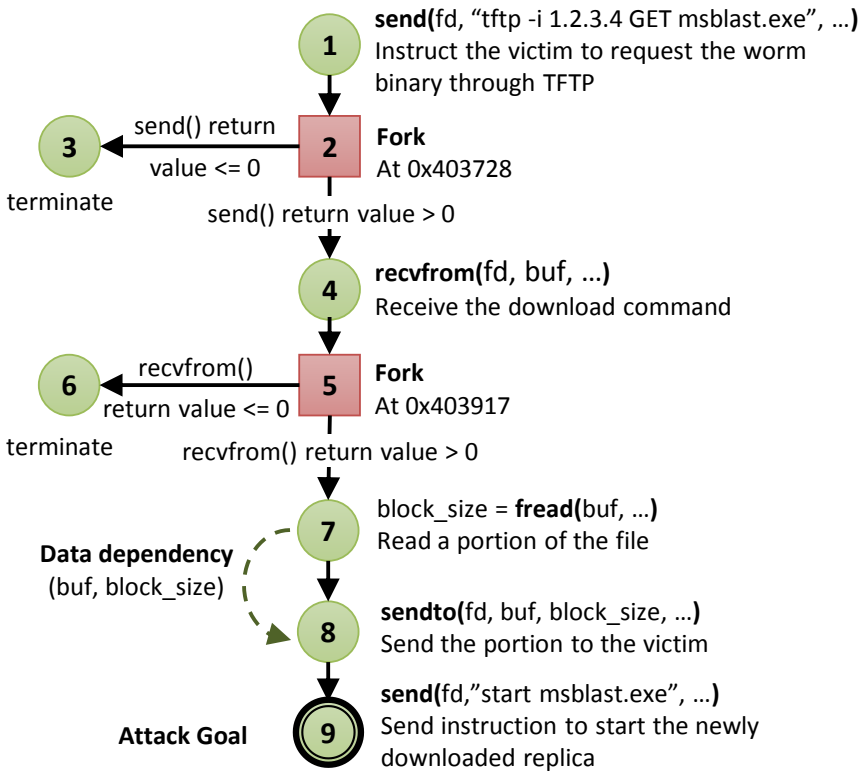


Figure 4.3: Example of Attack Behavior Model

We define the attack behavior model as a graph of *Points of Interaction (PoI)* nodes and *Fork* nodes. The PoIs refer to the points in the malware control flow at which the malware interacts with the environment by invoking system or library APIs. For example, node 1 in Figure 4.3 represents a PoI that corresponds to an invocation of a *Winsock* library API named *send*, whose input arguments include a socket handle and a pointer to a buffer containing the data to be transmitted. The fork nodes represent the points in the control flow at which the malware makes a control decision based on the results of its interactions with the environment, for example, nodes 2 and 5 in Figure 4.3.

To formally model the attack behavior, let Γ be the set of selected System and Library APIs, where each $\gamma \in \Gamma$ takes a fixed number of input augments ($I_\gamma = \{i_1, ..., i_n\}$) and returns a fixed number of output arguments ($O_\gamma = \{o_1, ..., o_m\}$). We model the attack behavior as the directed graph $G = (P, \mu, E, \nu)$, where:

- P is a set of nodes that represent the PoI and Fork nodes. The type function $\mu : P \rightarrow \{\boldsymbol{PoI}, \boldsymbol{F}ork\} \times (\Gamma \cup \emptyset)$ associates nodes with their types. If the node represents a PoI, μ further maps it to the appropriate system and library API from the set Γ.

- $E \subseteq P \times P$ is the set of edges that represents the dependency between the nodes in P. A directed edge $e = (p_i, p_j)$ is added from node p_i to node p_j if there is a control or data dependency between them. The dependency function $\nu : E \rightarrow \mathcal{L}_O$ associates each edge to a constraint expressed as a logic formula in the logic $\mathcal{L}_O$ with support for quantifier-free integer, real, and bit-vector linear arithmetic. Expressions in $\mathcal{L}_O$ are defined over the set of output arguments $O = \bigcup_{\gamma \in \Gamma} O_\gamma$.

In Figure 4.3, we show an example of attack behavior model that represents a portion of the Blaster worm that delivers a copy of the worm to an exploited victim. Round nodes represent PoIs, and square nodes represent fork points. The solid edges represent control dependency, while dashed ones represent data dependency. In this model, the worm first sends an instruction to a remote command shell process running on the exploited victim through the *send* library API, then it waits for a download request through the *recvfrom* API call. The attack code checks if these operations are executed successfully and terminates otherwise as depicted through the conditions shown on the outbound edges from the fork nodes 2 and 5. At node 7, the worm starts reading its executable file from the disk into a memory buffer, through *fread*, and sending the content of the buffer to the remote victim, through the *sendto* API. There is a data dependence between the third argument of the *sendto* call, which represents the number of bytes to transmit, and the return value of the *fread* call, which represents the number of bytes read from the worm file.

4.6 Modeling Attack Behavior using Binary Symbolic Execution

To extract the complete behavior of a cyber-attack, we execute its binaries (i.e., malware) symbolically and build a model that represents its behavior with respect to selected system parameters. Given that the correct set of system parameters is selected, symbolic execution can cover all relevant execution paths. We utilize the S^2E engine to execute malware binaries symbolically. The path coverage and the progress of the executed program depends on the correct marking of symbolic variables. Since we are interested in the interactions of the malware with its environment through the selected system and library APIs, we intercept these calls and mark their output arguments as symbolic. This allows us to capture the malware decisions based on those arguments and track the corresponding execution paths. In the current version of our implementation, we select about 130 APIs that cover activities related to networking, file system and registry manipulation, system information and configuration, system services control, and UI operations. We list the complete set of APIs in Appendix A.

We have investigated two approaches to intercept the selected system and library APIs and mark the symbolic variables. First, the use of dynamic binary instrumentation tools, specifically, Pin [184] by Intel. Pin is a dynamic instrumentation tool for IA-32 and x86-64 instruction-set architectures. It provides the means to write special purpose tools that intercept instructions and system calls for further dynamic analysis. It features an API that intercepts the system calls and reads/writes their arguments, which was appropriate for our symbolic variables selection. Second, the annotation plugin provided by the S^2E engine, which combines monitoring and instrumentation capabilities to let users annotate single machine instructions or entire function calls. It executes predefined scripts at runtime when the annotated instruction or function call is encountered in any execution path. Although Pin API provides a special function to intercept system calls, it does not provide the same for library APIs, which makes it inadequate to support our parameter extraction. Moreover, it

runs the executable in some sandbox, which adds one layer of indirection between the executable being analyzed and the symbolic execution engine, making it difficult to understand the result of the symbolic execution. Hence, we used the annotation plugin to implement our framework. The details of the steps required to accomplish that are presented in the following.

4.6.1 Marking Symbolic Variables

To intercept system and library API calls and mark the appropriate symbolic variables, we take advantage of the *Annotation* plugin provided by S^2E, which combines monitoring and instrumentation capabilities and executes user-supplied scripts, written in LUA language, at runtime when a specific annotated instruction or function call is encountered. We define an annotation entry for each API. The annotation entry consists of the module name, the address of the API within the module, and the annotation function. We identified the module names and addresses using static/dynamic code analysis tools, such as IDA and Ollydbg.

The annotation function is executed at the exit of the intercepted call. It reads the addresses of the return and output arguments of the call and marks the appropriate memory locations and registers as symbolic. To read the values of the output arguments inside the annotation function, we call a *curState:readParameter(param_no)*, which is provided by the Annotation plug-in, where *curState* represents the current execution state (i.e., path). We then call the function *curState:writeMemorySymb(sym_label, address, size)* to set the memory block of *size* bytes starting at *address* as symbolic. The address is the value of the output parameter and the *sym_label* is a name of the resultant symbolic variable. Note that output arguments may have varied sizes and structures. Hence, custom scripts are needed to mark each individual output argument of the intercepted APIs. The return values of APIs are typically held in the *EAX* register and we use a special method provided by S^2E to mark its value as symbolic. It should be noted that system calls and user library APIs are invoked

Listing 4.1: Example of Annotation Functions

```
1  function annotation_recvfrom(state, plg)
2    if (plg:isReturn()) then
       -- Mark the returned buffer as symbolic
3      buf = state:readParameter(2)
4      buflen = state:readParameter(3)
5      make_memblock_symbolic(state, "c-recvfrom-a2", buf, buflen)
       -- Mark the return value as symbolic
6      state:writeRegisterSymb("eax", string.format("c-recvfrom-ret"))
7    else
8      writeLibCallLog(state, plg, "recvfrom")
9    end
10 end
```

by all applications in the environment, not only the malware process. Therefore, our annotation functions check the name of the process that invokes them and ignore calls from irrelevant processes. More specifically, our scripts first retrieve the address of the current PEB (Process Environment Block) from *FS:[0x30]* in the guest VM and traverse this data structure to obtain the image path name; then they compare this name with a list of names relevant to the malware.

In Listing 4.1, we show the annotation functions *annotation_recvfrom*, which is called whenever the API *recvfrom* of the *ws2_32.dll* is invoked. The *if-else* statement at line 2 checks whether this annotation is invoked upon the entry (*plg.isReturn()==false*) or the exit point (*plg.isReturn()==true*) of the annotated API. On the entry point, at line 8, we only log the invocation information, which include the API name, its arguments, and the call stack. On the exit point, we read the output buffer (line 3) and its length (line 4) and we define a symbolic variable of the specified length by calling the function *make_memblock_symbolic* at line 5. The function *make_memblock_symbolic* is an alternative for *curState:writeMemorySymb* that we use in the cases where the buffer size is not known in advance. The string *"c-recvfrom-a2"* represents the name of the new symbolic variable, which indicates that it corresponds to the second argument of a call to the *recvfrom* API. Then, we declare the

return value, which is held in the register *EAX*, as another symbolic variable named *"c-recvfrom-ret"* at line 6.

4.6.2 Building the Attack Behavior Model

After preparing the appropriate annotation entries, we execute the malware using S^2E to collect the execution traces. We configured the annotation functions to record the arguments, the call stack, and other meta-data, such as the time-stamp and the execution path number for each intercepted system and library call. By design, S^2E intercepts branch statements whose conditions are based on symbolic variables and forks new states of the program for each possible branch. We collect the traces and branching conditions of all execution paths and build the attack behavior model as follows:

- We create a PoI node for each system or library API call logged by our annotation functions. Similarly, the traces contain special log entries for state forking operations. Those are used to create the *Fork* nodes in our model.

- For each node in the model, we add a control dependency edge from the node preceding it in the execution path. If the preceding node is a *Fork* node, the edge will be associated with a branching condition in terms of the symbolic variables.

- To capture the data dependency, we check the values of all the input arguments upon the entry of each API call. If the value is a symbolic expression, this implies that it is a transformation of previously created symbolic variables. Hence, we add a data dependency edge from the PoI nodes in which the symbols of the expression were created.

4.7 Agility Parameters Extraction

Given the attack behavior model generated through symbolic execution, we extract a set of system parameters that help in designing effective deterrence and deception

schemes to meet particular goals. Recall that the attack behavior model describes the complete behavior of malware with respect to selected system parameters. However, that does not mean that every parameter in the attack behavior model is a feasible candidate for deterrence or deception. That is, mutating or misrepresenting its value may not be sufficient to deter or deceive the attacker successfully. We analyze the attack behavior model to select the appropriate set(s) of agility parameters that can help in designing deterrence and deception schemes without dictating particular ones.

We present the following four criteria ($\mathcal{C}1$ - $\mathcal{C}4$) that must be considered to decide on which parameters are appropriate for effective deterrence and deception and which are not:

- $\mathcal{C}1$ (Goal Dependency): the selected agility parameters can directly or indirectly affect the outcomes of the attack in terms of whether the attacker can reach their goal. Hence, parameters that are used only in execution paths that do not lead to particular goals might be excluded.

- $\mathcal{C}2$ (Resilience): in cases where multiple attack paths lead to particular goals, selected parameters must provide deterrence or deception in all the paths, not on one path only.

- $\mathcal{C}3$ (Consistency): the selected agility parameters must preserve the integrity of the environment from the attacker's point of view. As system parameters may be interdependent, deterrence and deception schemes must take this into consideration, such that misrepresenting one parameter without misrepresenting its dependents accordingly does not disclose the deterrence or deception plans. For example, 32-bit architectures are limited to addressing a maximum of four gigabytes of memory. Hence, if in one particular execution path, we misrepresented our 64-bit architecture to appear as 32-bit for the attacker, we must also misrepresent the available memory if the attacker inquires about it later in the

execution path.

- $\mathcal{C}4$ (Cost-Effectiveness): although multiple parameters may exist in the execution paths leading to particular goals, mutating or misrepresenting different parameters may require different costs and provide various benefits from the defender's point of view. Defenders must select the most cost-effective set of parameters for deterrence and deception.

To extract the parameters that satisfy the four criteria, we follow two steps. First, we refine the attack behavior model to eliminate irrelevant execution paths and symbolic variables to ensure that the refined model contains only the agility parameters that satisfy $\mathcal{C}1$. Second, we construct a constraints optimization problem based on the refined model. We add the appropriate constraints to extract the parameters that satisfy $\mathcal{C}2$ and $\mathcal{C}3$. Further, we encode the estimated costs of using the candidate deception parameters to select the most cost-effective set that complies with $\mathcal{C}4$.

4.7.1 Refining the Attack Behavior Model

The complete attack behavior model contains many execution paths that may not be relevant to our agility analysis. In this refinement step, we (1) identify the set of execution paths that are relevant to deterrence or deception and (2) eliminate the don't-care symbolic variables.

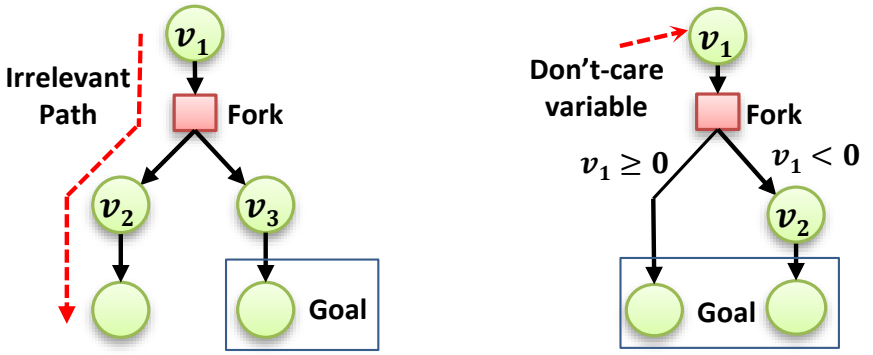
Identifying the Relevant Paths. Recall that deterrence and deception are not about blocking attacks, rather, it is about misleading and forcing them to follow particular paths that serve the desired goals. Hence, the selection of relevant execution paths from the attack behavior model depends on the deterrence or deception goal. For example, if our goal is distortion, the relevant paths are those that keep the malware misinformed about the environment to slow down the attack and force it to make additional environment checks. This is reflected in the paths that exhibit

aggressive interactions and queries with and about the environment. On the other hand, the relevant paths for depletion and discovery are those that lead the malware to interact with the remote master or adversaries, while paths in which the malware loses interest and abandons the system are more relevant to the deflection goal.

Definition 7 (Relevant Paths)**.** *A relevant path with respect to a particular deterrence or deception goal is an execution path that exhibits particular patterns of interactions with the environment that can be leveraged by the defender to achieve the goal.*

Regardless of which deterrence or deception goal is desired, it can be represented as a single call or a sequence of calls to system and library APIs leveraging existing tools that identify specific behaviors through patterns of call sequences, such as [185, 186, 187]. Then, the PoI nodes in our attack behavior model will be used to identify the execution paths that exhibit that particular sequence of calls. By pruning out all other paths that do not exhibit the desired sequence, we end up with a portion of the original behavior model that contains only the paths relevant to the desired goal(s). In Figure 4.4a, we show a simple example of an attack behavior model that has two paths, one leading to the desired goal and the other to the attack termination. In this case, the left path is considered irrelevant, and it will be eliminated. For a concrete real-world example, in order to deceive the FTP Credential Stealer malware in Section 4.8.3 with honey FTP passwords, the environment must not run OllyDbg because otherwise, the malware would follow an execution path irrelevant to the deception goal.

Eliminating the Don't-Care Variables. To clarify this step, we need first to define the execution path constraints. A path constraint is a logical expression that captures the conditions on the selected symbolic variables that need to be met in order for the execution to follow that particular path. Recall that we associate a set of symbolic variables to each PoI node p in the attack behavior model ($p \in P, \mu(p) = PoI$), which



(a) Relevant Path Selection. (b) Don't-care Elimination.

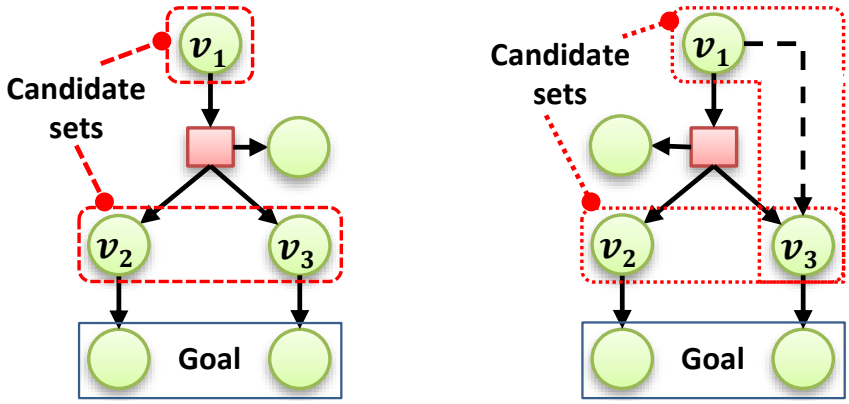
Figure 4.4: Attack Behavior Model Refinement.

correspond to its output arguments. Later in the execution, an expression will be generated for each branch at the following forking nodes in terms of the symbolic variables causing the fork. Those expressions are captured in the resulting edges of the fork nodes and mapped through the dependency function $\nu(.)$.

The constraint of an entire path in our attack behavior model is simply the conjunction of the logical expressions associated with all the edges that belong to the path. Formally, let $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, where $(\forall_{i\in[1,n]} : p_i \in P)$, represents a node path in the attack behavior model. Further, let $e_{i,j} \in E$ denote the edge between the nodes $p_i \in \mathcal{P}$ and $p_j \in \mathcal{P}$. The path constraint of the execution path represented by $\mathcal{P}$ can be computed as $\bigwedge_{i\in[1,n-1]} \mu(e_{i,i+1})$, where $\mu(e_{i,i+1})$ is the expression of the edge $e_{i,i+1}$. We define the don't-care symbolic variables as follows.

Definition 8 (Don't-Care Variables)**.** *A don't-care variable with respect to particular deterrence or deception goal is a symbolic variable that is part of one or multiple execution path constraints and its value is irrelevant to the desired goal.*

As Figure 4.4b illustrates, although there is a decision taken based on the symbolic variable v_1, the desired goal will be reached regardless of the variable's value. This makes v_1 a don't-care variable with respect to the desired goal. Consequently, it can be excluded from further deception analysis.



(a) Resilient Deception. (b) Consistent Deception.

Figure 4.5: Deception Parameters Selection.

After eliminating the irrelevant paths and the don't-care variables, we end up with refined path constraints for the relevant paths. Any parameter extracted based on this refined model complies with $\mathcal{C}1$ criteria.

4.7.2 Selecting Agility Parameters

In this step, we define a constraints optimization problem based on the refined attack behavior model to find an optimal set of agility parameters. Each PoI node in the refined attack behavior model is associated with a set of symbolic variables, which represent the output arguments of the malware interactions with the environment.

Although, the symbolic variables augment the attacker's perception about the environment, extracting the agility parameters out of them is not a trivial process due to the following. First, there is not necessarily a one-to-one mapping between the symbolic variables and the system parameters because the output of one interaction may be determined by multiple system parameters. Hence, we need to map the symbolic variables to the appropriate system parameters utilizing expert knowledge of the system and the system and library APIs. The documentation of the APIs can also be used to extract this mapping as it normally specifies the possible outputs of APIs and the cases in which each value is returned based on the system and the environment states. Second, multiple candidate sets of agility parameters may exist

in the model. As illustrated in Figure 4.5a, selecting either $\{v_1\}$ or $\{v_2, v_3\}$ satisfies $\mathcal{C}2$ criteria, but they might be associated with different costs. Third, the interdependence between system parameters mandates selecting additional parameters to satisfy $\mathcal{C}3$, which increases the cost of deterrence or deception. For example, in Figure 4.5b, although selecting $\{v_1\}$ is sufficient to satisfy $\mathcal{C}2$, we also need to select $\{v_3\}$ to satisfy $\mathcal{C}3$ because of the dependency of v_3 on v_1. To satisfy $\mathcal{C}4$, our selection must consider all possible candidate sets to find the most cost-effective one.

To formalize the problem of selecting the optimal set of agility parameters, let V be the set of system parameters and let S be the set of symbolic variables in the refined attack behavior model. We define the following mapping functions:

- $\theta(.) : S \rightarrow 2^V$ is the parameters assignment function that maps each symbolic variable in the path constraints to the corresponding system parameter(s).
- $\delta(.) : V \rightarrow \mathbb{Z}^*$ is the cost function that determines the cost of deterrence or deception through each system parameter. $\mathbb{Z}^*$ is the set of non-negative integers.

Based on these assumptions, the agility parameters selection is defined as the problem of selecting a set of system parameters, such that (1) at least one parameter is selected for each relevant path (to comply with $\mathcal{C}2$), (2) if a parameter is selected, all its dependents are also selected (to comply with $\mathcal{C}3$), and (3) the selected parameters incur the minimum cost on the defender (to comply with $\mathcal{C}4$).

To model this problem, we define the set of Boolean variables $\{d_1, d_2, ..., d_m\}$ with a variable for each system parameter in V. This set represents the decision variables of the constraints optimization problem, where d_i is set to 1 if the i-th parameter in V is selected for deterrence or deception and d_i is set to 0 otherwise. Then, we unfold the refined attack behavior model into the set $\mathcal{T}$ of paths, where each $t \in \mathcal{T}$ is a sequence of symbolic variables ($t \subseteq S$). The first constraint that at least one agility

parameter is selected for each path is expressed as follows:

$$\bigwedge_{t \in \mathcal{T}} \left(\bigvee_{s \in t} \bigvee_{i \in \theta(s)} d_i \right) \tag{4.1}$$

where $\theta(s)$ is the parameters assignment function that returns the indices of the system parameters associated with the symbolic variable s. To calculate the total deterrence or deception cost, we compute the value C that represents the cumulative cost of all the selected agility parameters.

$$C = \sum_{i \in [1,m]} (d_i \ ? \ \delta(v_i) \ : \ 0) \tag{4.2}$$

where v_i is the i-th element in the set V of system parameters and the notation $(\psi \ ? \ v_1 : v_2)$ represents the *if-then-else* construct that evaluates to the value v_1 if ψ is *true*, and to the value v_2 if ψ is *false*. We add another set of constraints to capture the dependency between the system parameters. If a parameter is selected for deterrence or deception, all the dependent parameters must also be selected. To capture this set of constraints, let $\epsilon(.) : V \rightarrow 2^V$ be a dependency function that maps each system parameter to a set of dependent parameters. Then, we represent the dependency constraints as follows:

$$\bigwedge_{v_i \in V} \left(d_i \rightarrow \bigwedge_{j \in \epsilon(v_i)} d_j \right) \tag{4.3}$$

After adding the constraints, we solve the constraints optimization problems using the Z3 solver. The optimization objective in this problem is to minimize the aggregate deterrence or deception cost denoted by the variable C in Equation 4.2. The result will be a set of system parameters that satisfies our four criteria to provide resilient, consistent, and cost-effective deterrence and deception.

4.8 Evaluation

We analyzed over 50 recent malware variants using *gExtractor* and extracted candidate agility parameters for each of them. The variants we analyzed represent the most common types of malware, including Cryptocurrency-mining malware, ransomware, worms, spyware, and Credential-stealing malware. To demonstrate that our systematic approach can indeed extract effective agility parameters, we selected six of the most prevalent malware, namely, *W32.Blaster* worm, *Bitcoin Miner*, *FTP Credential-Stealer*, and three instances of ransomware (*Cerber*, *Locky*, and *Gandcrab*). We discuss in detail the process of building the attack behavior model, extracting the agility parameters, and we suggest deterrence and deception schemes utilizing them.

4.8.1 Case Study I: W32.Blaster Worm

In this case study, we analyze a real worm called *W32.Blaster*. Although this worm is old (first discovered in 2003), we selected it since its source code is available [188], which provides the ground truth to validate the results we learn through our automated framework. In addition, it is using various attack techniques that are commonly used by old and recent malware, such as reconnaissance, Denial of Service (DoS), and malware propagation.

Malware Behavior. We constructed the behavior model of this malware using our framework and we show a simplified version in Figure 4.6. *W32.Blaster* consists of two modules: spreading and flooding. The spreading module replicates the worm in the vulnerable machines by exploiting a known vulnerability in Microsoft's DCOM RPC [189]. The flooding module conducts a SYN flood attack on the windows update server at specific times of the year. Before starting the malicious modules, Blaster goes through an initialization phase, which set some registry keys to ensure persistence and checks for infection markers. The spreading module scans blocks of 20 sequential IP addresses simultaneously. Once an active IP is found, the spreading continues as

Table 4.1: Major Intercepted APIs in the Blaster Worm

Module	**API**
ntdll.dll	NtCreateKey(KeyHandle, Access, Attributes, TitleIndex, Class, eOptions, Disposition)
ntdll.dll	NtSetValueKey(KeyHandle, ValueName, TitleIndex, Type, Data, DataSize)
ntdll.dll	NtCreateMutant(MutantHandle, DesiredAccess, ObjectAttributes, InitialOwner)
ntdll.dll	GetLastError()
wininet.dll	InternetGetConnectedState(lpdwFlags, dwReserved)
ws2_32.dll	WSAStartup(wVersionRequested, lpWSAData)
ws2_32.dll	sendto(socket, buf, len, flags, to, tolen)
ws2_32.dll	recvfrom(socket, buf, len, flags, from, fromlen)
ws2_32.dll	send(socket, buf, len, flags)
ws2_32.dll	bind(socket, name, namelen)
ws2_32.dll	select(nfds, readfds, writefds, exceptfds, timeout)
ws2_32.dll	connect(socket, name, namelen)
msvcr90.dll	fread(ptr, size, nmemb, stream)

follows. First, it initiates a connection to the IP address over the TCP port 135, which is the port used for RPC/DCOM. If the connection is established, the worm sends an exploit payload that, if successful, will establish a remote command shell process listening to the TCP port 4444 on the victim's machine. This allows the attacker to send and execute commands remotely. Second, *W32.Blaster* sends a command to the remote shell to use a Trivial File Transfer Protocol (TFTP) client that will download the worm binary to the remote victim. At the same time, *W32.Blaster* starts waiting for TFTP requests from the victim. Once a request is received, the binary file is sent to the victim. Finally, *W32.Blaster* sends a command to execute the worm replica on the victim's machine remotely.

Agility Parameters. We analyzed the behavior of *W32.Blaster* with respect to the goal of successful replication on the remote victim machine. We identified the critical parameters on which the goal depends using our framework. Table 4.1 shows the major interactions that we intercepted and in Table 4.2 we summarize their critical arguments along with the system parameters that can influence their values. We use *foo.ret* to denote the return value of the system call or user library API *foo*.

Deterrence Schemes. During the initialization phase, the Blaster worm creates a

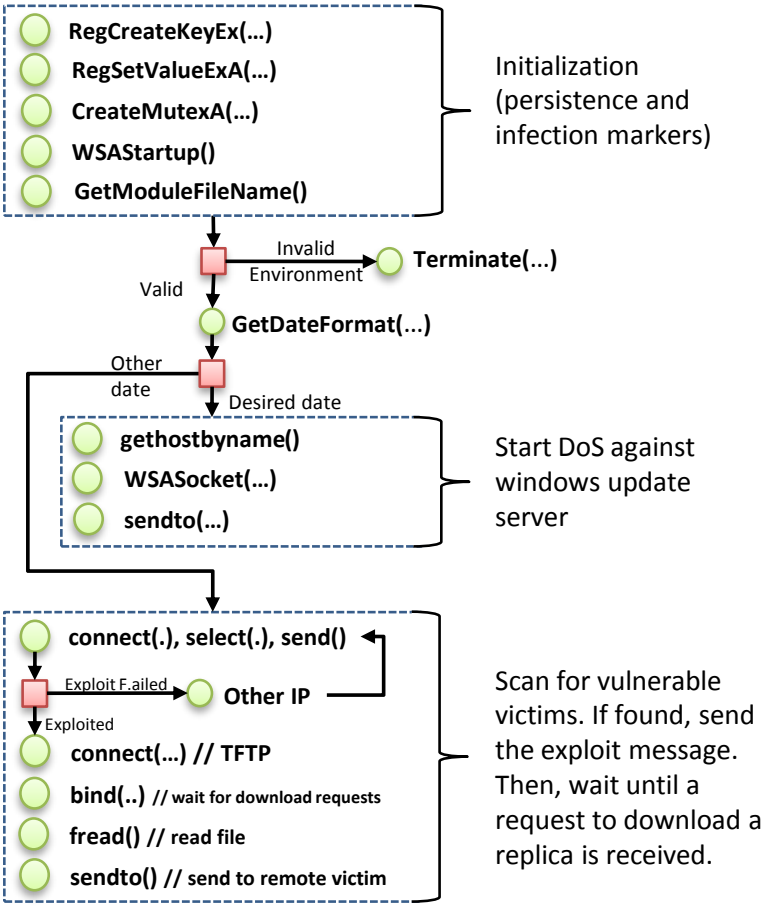


Figure 4.6: Simplified Behavior Model of the *W32.Blaster* Worm.

registry key/value, to ensure persistence, and creates a mutex as an infection marker. The name of the registry subkey *"SOFTWARE\Microsoft\Windows\CurrentVersion\Run"* is hardcoded in the worm. This registry subkey has a fixed meaning in Windows OS and is used to record all programs that should be launched automatically at boot time. Although changing this registry subkey (i.e., Windows will use a different subkey to locate auto-start programs) may break the persistence of this worm, the Blaster sample that we investigated does not check if this step is successful. This means that the worm will continue running regardless of the result of the registry key operation. Next, the malware invokes *NTCreateMutant* to create a mutex named *BILLY*, and if the result of the system call is equal to ERROR_ALREADY_EXIST, the malware will terminate its execution. Hence, the malware's goal depends on

Table 4.2: Critical Parameters of the Blaster Worm

Argument	**Meaning**	**System Parameter**
NTCreateMutant.ret	Mutant found or not?	Mutant Name
WSAStartup.ret	Socket library initialized?	Socket Library
InternetGetConnectedState.ret	Internet connected?	Connection State
select.ret	Remote party available?	IP Address, Port Number
send.ret	Session established?	Communication Sessions
connect.ret	Connection established?	IP Address, Port Number
bind.ret	TFTP port available?	Available Ports
recvfrom.ret	Remote party alive?	IP Address, Port Number
recvfrom.buf	Request content.	Communication Messages
fread.ret	Bytes read.	File Names, File Content
sendto.ret	Attack commands are sent?	IP Address, Port Number

whether a mutex named *BILLY* exists. The initialization phase continues by verifying the version of the socket API through calling *WSAStartup*. As shown in Table 4.2, the two arguments *NTCreateMutant.ret* and *WSAStartup.ret* are critical parameters in the initialization phase because their values can determine whether the execution continues or not.

After successful initialization, the interaction of Blaster with its environment is mostly related to the network communication with the victim. The worm calls multiple networking APIs, such as *connect*, *send* and *recvfrom*, to exploit the RPC/DCOM service on the victim's machine and deliver a replica of itself. All the parameters that are shown in Table 4.2 and related to networking are guaranteed to prevent the worm from reaching its goal if their values mismatch the ground truth in the environment. For example, the IP address, one of the well-studied agility parameters, is discovered using our analysis. Specifically, the malware sends an exploit payload to a target machine. Then the malware verifies if the exploit has been successful by trying to establish a *new* TCP connection to the target machine over port 4444. Between these two steps, the malware assumes that the target machine's IP address stays the same. Therefore, mutating the IP address of the target machine, even after the initial exploit, will make the worm unable to reach it and deliver the worm replica.

Deception Schemes. The rest of the parameters shown in Table 4.2, such as the *Mutant Name*, *Socket Library*, *Internet Connection State*, and the *Communication Messages*, are valid candidates for deception because the attacker decides whether to go forward or not based on their values. However, they are not candidates for mutation because the attacker does not use their values again in the following attack phases. On another hand, *Port Number* and *File Names* may be candidates for deterrence. The attacker first discovers if the IP Address/Port number are active, then she uses them to send the exploits and further commands. In the same way, the attacker first discovers the file name of the worm, and then she uses the file name to read its content.

4.8.2 Case Study II: Bitcoin Miner

We analyzed a recent bitcoin mining malware (MD5: efd1326e5289a9359195120fd6 c55290) that works in several stages. First, it drops and runs a Visual Basic (VB) script. Second, the script queries the Windows Management Instrumentation (WMI) service for the processor's information, such as the availability of GPU and the system architecture (32-bit or 64-bit), to download the right executable file for the target system from an external distribution website, *winxcheats.tk*. Third, the downloaded executable (*csrs.exe*) downloads yet another executable (*AudioHD.exe*) from *getsoed9.beget.tech*. The last program (*AudioHD.exe*) interacts with a bitcoin mining pool server at *xmr.pool.minergate.com* to perform the mining on behalf of an account, which is hard-coded in the executable.

Malware Behavior. Using *gExtractor*, we construct the behavior model of this malware (see the simplified version in Figure 4.7), which covers the malware execution stages. We use common patterns of API calls to recognize significant malware activity. For example, the use of APIs that create new processes (e.g., *ShellExecuteExA* and *WshShell.Run*), indicates the beginnings of consecutive malware stages. Moreover, interacting with a well-known bitcoin mining pool server through networking and

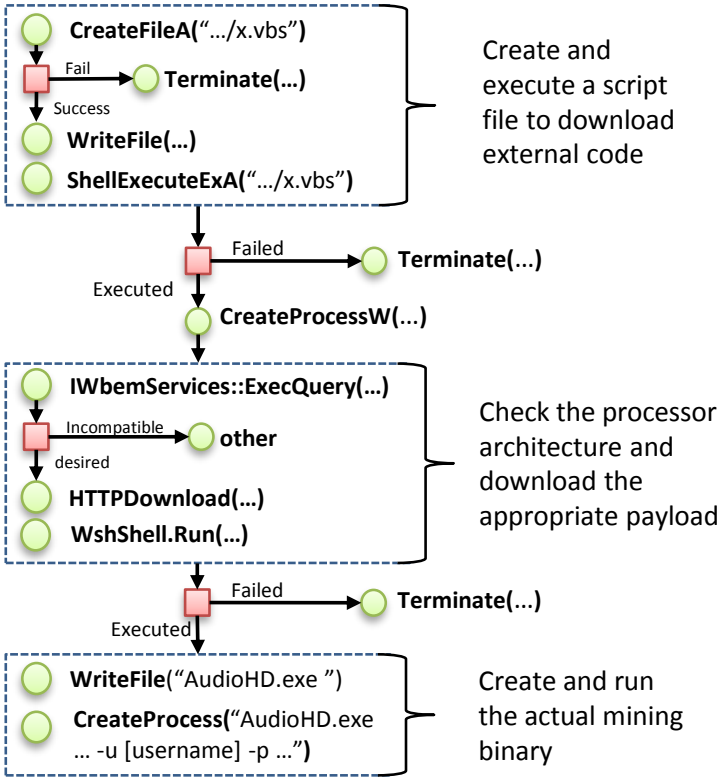


Figure 4.7: Simplified Behavior Model of the Bitcoin Miner

HTTP APIs reveals that one goal of this malware is to use the victim machine to perform bitcoin mining on behalf of the attacker. Therefore, we refine the malware behavior model by recognizing the relevant paths that lead to that goal and design deterrence and deception schemes around it. After mapping the symbolic variables of the relevant paths' constraints to the system parameters, our analysis reveals the following necessary conditions for successful mining:

1. The file *C:\Windows\system32\wscript.exe* exists.

2. Windows Script Host (*WSH*) engine is enabled to run Visual Basic scripts [190].

3. *WMI* service and *Microsoft Win32 WMI* provider are running.

4. *win32_processor* WMI class reports the correct processor information.

5. The distribution website (*http://winxcheats.tk*) is available and hosts the exe-

cutable file (under */miners/3/csrs.exe*).

6. The second distribution site (*getsoed9.beget.tech*) is available and hosts the second executable file (*AudioHD.exe*).

7. The bitcoin mining pool server (*xmr.pool.minergate.com*) is still running correctly.

8. The hard-coded account (*iden1930@mail.ru*) is authenticated successfully at the mining pool server.

9. The target system can run the file *AudioHD.exe* successfully.

To clarify how *gExtractor* facilitates the detection of such conditions, let us take condition 2 as an example. We mark the output parameter *"Buffer"* of the *RegQueryValueExW* API call, which is required to successfully complete the second stage of the malware, as symbolic. The API's input parameter, *hKey*, refers to the registry key *"HKLM\SOFTWARE\Microsoft\Windows Script Host\Settings\"* and the other input parameter *"ValueName"* is set to *"Enabled"*. Then we observe that *"Buffer"* is used in a conditional jump, and in one path the message *"Windows Script Host access is disabled on this machine"* is displayed before the process terminates, while in another path we do not see this message. Alternatively, we see multiple queries to the WMI service. The first path will be regarded as irrelevant and pruned out by *gExtractor* and we will only consider the later. Similarly, *gExtractor* can detect the dependence of this malware on the remaining conditions by tracking the decisions taken based on the associated symbolic variables and refining the behavior model.

Agility Parameters. We analyzed the refined bitcoin miner behavior model with respect to different deception goals: deflection, distortion, depletion, and discovery. We identified the major agility parameter that satisfies our criteria defined in Section 4.7.2 and can be utilized to achieve each goal. In the following, we discuss a

Table 4.3: Deception Schemes Against the Bitcoin Mining Malware

Parameter	**Goal**	**Deception Action**	**Estimated Cost**
wscript.exe	Deflection	Replace it with a version that rewrites the input VB script for better protection	No CC; High OC; High DC
WSH engine	Discovery	Enable its capability to run VB scripts	Low CC; No OC; No DC
WMI class	Distortion	Change the way that it handles requests (e.g., returning misinformation about processors)	No CC; Low OC if used on a honeypot; Medium DC
The resulting hash	Depletion	Corrupt the resulting hash	No CC; No OC; High DC

(CC: configuration cost, OC: operation cost, DC: development cost)

number of recommended deception schemes based on these parameters, and we provide a summary with the estimated deception costs in Table 4.3.

Deflection Schemes. For this purpose, we can enhance the designated script host *C:\Windows\system32\wscript.exe*. If the malicious VB script initiates a connection to a critical server, the enhanced *wscript.exe* can rewrite the VB script statement so that it connects to a honey server instead. This scheme could have high development cost because it requires a change to a Windows system utility, for which we do not have a source code. In terms of operation cost, this scheme can have a high cost because it can confuse benign applications that need to run VB scripts, even if this is on a honeypot. However, it has little configuration cost because the current Windows OS does not have a configuration option to replace *wscript.exe* with an alternative version.

Discovery Schemes. We can use the *Windows Script Host (WSH) engine* to construct a discovery strategy against malware that needs to run VB scripts. The WSH enables applications to run VB scripts and JScripts, and it provides a configuration option (via Windows registry) to enable/disable the VB script support. By enabling it, we can observe malware behavior through its VB scripts and have a better understanding of the malware. This strategy incurs only a low configuration cost.

Distortion Schemes. Through the *"win32_processor WMI class"* parameter, we can construct a distortion scheme that returns misinformation about the system's architecture in order to confuse the malware (or the attacker behind the malware) who queries the *win32_processor class* interface. This strategy requires a change to the implementation of the *win32_processor class* interface, so there can be some development cost, and it can have a low operation cost if it is used on a honeypot.

Depletion Schemes. The last parameter in Table 4.3 is the resulting hash, which the malware sends reliably to the mining pool server. We can create a depletion scheme by corrupting the results so that they become invalid. Deceiving the malware by sending excessive invalid results damages the attacker's reputation or cause financial losses (e.g., the mining pool server bans her account, freezes her mining wallet, or applies a penalty to her account). This scheme requires writing code to carry out the scheme, so it has some development costs. It has no operation cost because it modifies only the data of the malware.

We have experimentally confirmed the feasibility of depleting the attacker by corrupting the resulting hashes. To profit from mining on a victim machine, the attacker communicates with the mining pool server under the mining username to receive the credit. Therefore, at the mining time, the attacker username must be present on the victim machine. We leveraged this fundamental "vulnerability" of this malware (i.e., revealing the mining username) for an effective deception. Based on a study of multiple mining pools, they establish various penalty policies for participants who submit invalid hashes. Misbehaving users are often banned (for a particular period of time), and their wallets can even be locked. In some servers, such as *minergate*, the balance will be reduced to penalize this behavior.

In a parallel work by our group, we built a tool that deliberately sends invalid hashes

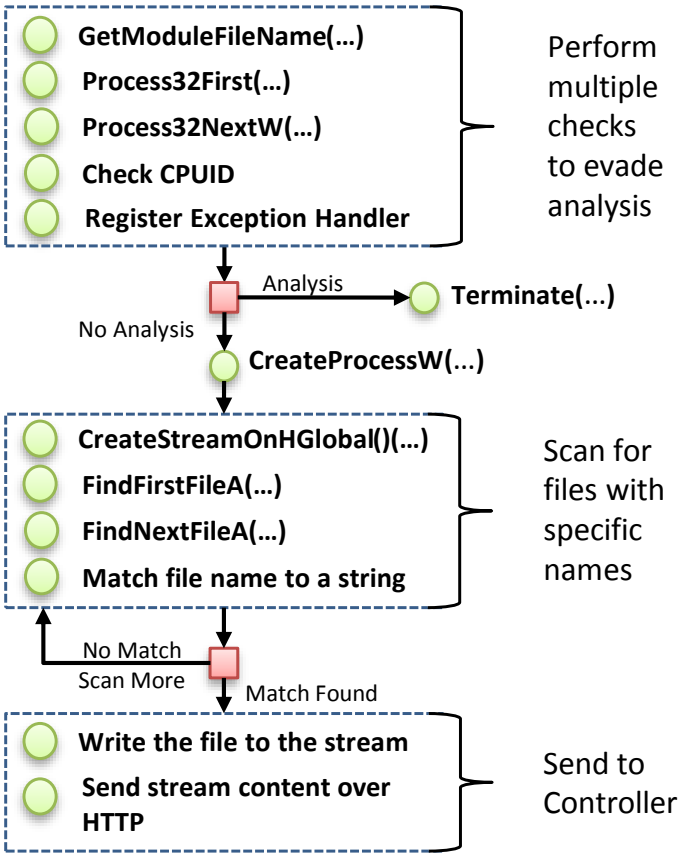


Figure 4.8: Simplified Model of the FTP Credential-Stealer

on behalf of a particular user to prove the effectiveness of this depletion scheme. As different mining pool servers may implement distinct protocols to authorize jobs and submit resulting hashes, we obtain the required settings and methods to communicate with the mining pools by analyzing the mining malware. Then, we submit a login request to the pool server. In response, the pool server returns a job and an ID that corresponds to the username. At that point, our tool will generate and send a random hashing result, which will most likely be recognized as an invalid hash by the pool server.

4.8.3 Case Study III: FTP Credential-Stealer

In this case study, we analyze a recent malware (MD5: 7572fb188134-d141eac175 1b19b79a70) that scans the victim system for sensitive information, such as FTP

login passwords and then sends the stolen information to a remote server.

Malware Behavior. This malware consists of two processes. The first process employs multiple methods to check whether the malware is being analyzed, then terminates immediately if the checking result is positive. If no signs of analysis are detected, the first process drops and launches another piece of malware, which collects sensitive information from the victim system and sends it to a remote server under the adversary's control. A simplified version of the behavior model of this malware, generated by *gExtractor*, is shown in Figure 4.8.

The first malware process is heavily obfuscated and employs multiple tricks to evade analysis and ensure a safe execution environment: (1) it tests whether the executable file's name (via the API *GetModuleFileNameA*) contains any of the strings *"sandbox", "malware", "virus", or "self"*; (2) it scans the list of running processes (via the APIs *CreateToolhelp32Snapshot*, *Process32FirstW*, and *Process32NextW*) for known dynamic analysis tools, such as *procmon.exe, procmon64.exe, procexp.exe, ollydbg.exe, and windbg.exe*; (3) it checks the *BeingDebugged* flag in its PEB (Process Environment Block) [191] at multiple places of its code section; (4) it checks whether it is running inside a virtual machine by matching the result of the *CPUID* instruction with *"KVMKVM", "XenVMM", "Microsoft Hv", and "pri hyperv"*; (5) it extracts the second malware binary from its resource section, decrypts it, and then uses process injection to launch it in a second process. If any sign of malware analysis is detected, the malware immediately terminates.

In the second process, the malware collects sensitive information from the Windows registry and the local file system and sends it to a remote site as follows. First, it searches certain Windows registry keys, which correspond to a specific list of FTP clients, for saved login credentials. For example, to steal information related to WinSCP, it searches for the key "Software\Martin Prikryl". If the key is found, it

Table 4.4: Defense Strategies against FTP Credential-Stealing Malware

Parameter	**Goal**	**Deception Action**	**Cost**
Malware file name	Discovery	Avoid naming the malware "sandbox.exe", "malware.exe","virus.exe", or "self.exe"	Low CC; no OC
Dynamic Analysis tools	Discovery	Rename the dynamic analysis tools	Low CC; no OC
CPUID result	Discovery	Deny that the true result is one of "KVMKVM","XenVMM", "Microsoft Hv", and "pri hyperv"	Low CC; High OC; High DC
WinSCP registry entries	Depletion	Plant encrypted and invalid FTP passwords to waste the energy of the attacker who tries to decrypt and use those invalid passwords	Low CC; Low OC; No DC
WinSCP registry entries	Discovery	Plant honey FTP passwords to entice the attacker to login to a honey FTP server	Med CC; Low OC; No DC
Registry entries of applications that maintain login credentials	Depletion	Plant encrypted and invalid login credentials	Low CC; High OC; High DC
Registry entries of applications that maintain login credentials	Discovery	Plant honey login credentials	Med CC; Low OC; No DC
Files that contain sensitive information	Depletion	Plant Honey files with seemingly sensitive information to waste the energy of the attacker who tries to act upon the content of the files	Low CC; Low OC; No DC
Files that contain sensitive information	Deflection Distortion Discovery	Depending on the meaning of the file content, plant crafted content that can help the defender achieve Deflection / Distortion / Discovery goals	Var CC; Low OC; No DC

(CC: configuration cost, OC: operation cost, DC: development cost)

recursively enumerates the subkeys with the names *"HostName", "UserName", "Password", "RemoteDirectory"*, and *"PortNumber"*, read their values, and stores them in a stream object for later exfiltration. Strings such as *"Software\Martin Prikryl"* and *"HostName"* are hard-coded in the malware. Second, it looks up files whose path contains particular patterns *(e.g., "WS_FTP", "LastSessionFile", "FTPRush", "Quick.dat", and "History.dat")*, and if any such file exists, it stores the file's path and content in the stream object. To optimize the search, it focuses on known folders, identified by their Constant Special Item ID List (CSIDL) values, such as the users' public documents, desktop, and local settings. It also searches the folders of installed applications discovered by their *"UninstallString"* registry values under the registry key *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall*.

After collecting the targeted information, the malware extracts the data from the stream object (via the API call sequence {*GetHGlobalFromStream*, *GlobalLock*}), then it constructs and sends a HTTP POST message to *"http://www.luxzar.com/drake/november/omg/hot/ gate.php"*. The HTTP communication is conducted by the API call sequence {*InternetCrackUrlA, ObtainUserAgentString, socket, connect, setsockettopt, send, closesocket*}.

Agility Parameters. We employ the methods discussed in Section 4.7 to the behavior model we obtain from the above analysis. We recognize a number of agility parameters that enable different deception schemes, as summarized in Table 4.4.

Discovery Schemes. Since the malware applies many checks to evade analysis, these checks can be used to inspire effective discovery schemes that encourage the malware to run normally. Specifically, we can rename common analysis tools if we must run them and modify the behavior of the *CPUID* instruction so that it gives an impression that the environment is not a virtual machine, which is commonly the

case for malware analysis. The cost of renaming common dynamic analysis tools is low. However, the cost of manipulating the result of the *CPUID* instruction can be high: it is cheap if the environment has a way to intercept *CPUID* instructions in software (e.g., on top of QEMU), but it is infeasible otherwise. Alternatively, the registry entries of the FTP clients, such as WinSCP, can be leveraged to lure the attacker to honeypots so we can learn more about its capabilities and intents. We can create honey FTP accounts, save the honey login credentials in WinSCP, and run the malware, so it delivers the honey login credentials to the attacker. The configuration cost of this kind of scheme is medium because it is necessary to set up the honey FTP server and deploy monitoring tools.

Depletion Schemes. The registry entries of the FTP clients can also be leveraged to feed the attacker fake login credential and deplete her resources and effort. For example, we can install WinSCP in the environment and save many sessions with fake values for the information targeted by the attacker (e.g., username and password) decreasing the likelihood of her landing on legitimate victims. An even better scheme is to create an encrypted version of an invalid password and save it in the Windows registry entry for WinSCP, which will give the attacker an additional burden to decrypt the password, thus further depleting the attacker's resources.

Deception Schemes using the File System. Similar to registry entries, files that contain sensitive information are useful parameters for multiple goals: depletion, deflection, distortion, and discovery. For example, we can plant honey files with seemingly sensitive but useless information to waste the energy of the attacker who tries to act upon the content of the honey files. Although the general idea is well known, the specific details as to which files should be planted can be greatly informed by analyzing the malware decisions. The cost of carrying out these kinds of strategies

can vary depending on the purposes of the files: it may require simple editing of a file on one hand, or development of tools to create the files on the other; the operation cost may also vary depending on the purpose of the files: if they are used only by attackers, the cost is low, but if they are used by benign users, the cost can be quite steep, since the honey content can confuse benign users.

4.8.4 Case Study IV: Cerber, Locky, and Gandcrab

Ransomware has moved from the 22nd most common variety of malware in the 2014 data breach investigations report to the fifth most common in 2017's report [2]. We analyzed three instances of this family using *gExtractor*: Cerber, Locky, and Gandcrab and we summarize the analysis in the following. Since these instances encrypt files on the compromised computers, one can expect that their functionality relies heavily on file-related interactions. However, they share some functionality with traditional other types of malware. First, during the initialization, they access the system registry to set up the keys that guarantee persistence, and they use mutant-based infection markers. Second, they communicate with a C&C server for key sharing and reporting. However, the communication is limited to information sharing, and the server commands do not normally change the primary goal of those malware.

Agility Parameters. In Table 4.5, we highlight the major interactions with the file system, and we map them to system parameters. Note that some interactions depend on multiple parameters. For example, *GetFileSize* will fail if the file no longer exists and if it does, it will return the file size. Hence, it depends on both the existence of the file and its size. The persistence registry keys and the mutant infection markers, the common folders (e.g., temp and system directory), and the parameters in Table 4.5 (e.g., files names, sizes, and attributes) were all part of the relevant paths that lead to the goal and can be candidates for deterrence or deception. In addition, after refining the behavior models, we observed a difference between *Cerber* and the *Locky*

Table 4.5: Selected File-Related Interactions

Interaction	**System Parameter(s)**
GetSystemDirectory	System Directory Path
SearchPath	File Existence
FindFirstFile	File Existence
NtQueryInformationFile	File Existence/Info
GetFileAttributes	File Existence/Attributes
GetFileSize	File Existence/Size
NtReadFile	File Content
NtWriteFile	File Content

that confirms others' manual observations. Although *Cerber* called the *sendto* API to communicate with the C&C server, the encryption process starts regardless of the success of the communication. This means that the success of the *sendto* call, manifested as a symbolic variable, was not part of the conditions to reach the goal and it was pruned out. However, successful communication was a condition to start encryption in the *Locky* instance we evaluated.

Depletion Schemes. We developed a depletion scheme against the adversary behind ransomware, specifically Gandcrab (MD5: 8a45b0941ec2af89bfd9ed3-3dae2053f). The malware sends encrypted information about the victim to the C&C server, then receives a response message derived from the message sent, which implies that the C&C server has to process the message from the malware. Therefore, we can overwhelm the C&C server by sending a substantial number of messages to it in a brief period. We have experimentally verified that (1) it is possible to replay the same message many times while the C&C server responds to each individual message; (2) if we intentionally send a corrupted encrypted message, the C&C server responds with an error, which implies that it tries to decrypt the message but fails. Both confirm the feasibility of our deception scheme.

4.8.5 Challenges and Future Work

Through our case studies, we recognized a few technical challenges with respect to our approach. First, it is non-trivial to build a general agility parameter extraction technique due to inherent limitations of symbolic execution. For example, to avoid following back edges in a loop, we have to supply the exact addresses of the source and destination instructions, which is, unfortunately, malware specific. One solution would be to recognize back edges automatically. Second, the naive use of symbolic execution cannot effectively discover interesting malware dependency on the environment because the execution can slip into paths leading to other than the desired goals, such as getting stuck in loops. We plan to develop new plugins that would guide the symbolic execution engine towards more meaningful paths leveraging existing approaches that were previously proposed to address similar challenges in dynamic taint analysis and mixed concrete and symbolic execution [192, 193].

CHAPTER 5: SUMMARY AND FUTURE WORK

In this dissertation, we address key challenges to improve the effectiveness and deployment of several resilience techniques: isolation, diversity, adaptive response, coordinated defenses, deception, and deterrence (or dynamic positioning). Our research has three primary objectives: (1) enabling effective isolation and diversity composition to improve the cyber resilience significantly, (2) developing provably safe and efficient refinement of adaptive response policies, and (3) automated extraction of agility parameters from malware binary code to construct effective deception and deterrence plans. We achieve these objectives in three research thrusts discussed in the previous chapters in detail. In this chapter, we summarize our contributions, and we outline the significant findings and evaluation results in each research thrust. Alongside, we propose future work as new directions or extensions based on our research in this dissertation.

5.1 Automated Synthesis of Isolation and Diversity Configuration Composition

In Chapter 2, we presented two major contributions towards effective composition of resilience techniques. We presented a formal framework that models the specification and quantifies the effectiveness of isolation and diversity in the context of network connectivity, and synthesizes a cost-effective resilient configuration that integrates both techniques. Our model quantifies the effectiveness of isolation and diversity decisions in terms of cyber risk reduction and the optimal isolation-diversity composition under connectivity and budget constraints. Since resilience is about managing attacks consequences rather than only preventing them, our model considers all possible attack paths that attackers use as a stepping-stone to reach their targets.

Our framework synthesizes the appropriate software program diversity and isolation countermeasure configurations to counter attacks in each step in the attack path. Our second contribution is the development of network decomposition algorithms to improve the scalability of our synthesis approach significantly.

Since we used formal synthesis and we model all possible isolation and diversity decisions with respect to all possible attack paths, the obtained solution is correct-by-construction [194, 195] as it inherently guarantees the cyber risk and budget constraints for any multi-step attack. Hence, we focus on evaluating the scalability of our approach. We have shown that without using our decomposition and model reduction techniques, we can synthesize resilient configuration for networks of up to 600 connected services, considering attacks of 8 steps (i.e., intermediate hops), with a polynomial growth rate of the processing time with respect to the network size. However, using the network decomposition approach, we show that our synthesis can support networks of several thousands of nodes.

Future Work: First, further research should be undertaken to explore how we can effectively integrate more resilience techniques into one unified synthesis framework. We believe that the challenge is in modeling the interdependence between them and how they can influence each other. Second, we also plan to investigate efficient optimization solutions to transform our solution from constraints satisfaction to constraint optimization. Third, currently, the objective of our work is to find a configuration that brings the risk below a certain threshold. It would be more practical to reduce the risk to the minimum, instead. This might be challenging due to the complexity and the scale of current networks. Thus, further research is needed to develop new techniques for scalability optimization under these new constraints.

5.2 Provably Safe and Efficient Course of Action Orchestration for Active Cyber Defense Policies

In Chapter 3, we presented a specification of Active Cyber Defense (ACD) policy that initiates a course of investigation and configuration actions (called CoA) to respond to security events and incidents. However, initiating such CoAs can lead to unsafe orchestration due to accessing shared cyber resources including objects such as flows and files and actuators such as firewalls and servers. We identified and formalized the sources of potential conflicts between actions that cause the ACD to fail, and we designed a formal framework that can schedule a safe CoA orchestration to avoid these conflicts. Our safe orchestration framework will ensure that (1) actions that operate on shared resources are not executed simultaneously, (2) interdependent actions are scheduled for execution such that the pre-condition of the executed action is satisfied by the dynamic system state as a result of the post-condition of the previously executed actions, (3) the executed CoA do not violate the mission requirements. Otherwise, actions are executed concurrently whenever it is safe to minimize the total execution time. To satisfy the cyber mission requirements, we develop a model checking framework to verify that configuration updates by the CoAs do not violate the network mission in terms of reachability, QoS, and security requirements.

We evaluated two major aspects of our framework. First, we assessed the performance of our orchestrator in terms of the processing time required to synthesize the optimal execution workflow. The results show that we can guarantee the safety and efficiency of up to 500 actions within an average overhead of 60 seconds. Second, we evaluated the performance of our model checker in verifying the satisfaction of network mission requirement given a snapshot of the network data plane configuration. Our verification takes less than a minute to verify the mission requirements for a network of 1000 nodes and the required time increases in polynomial time for more extensive networks.

The aim of the present work in this direction is to examine the feasibility of automated orchestration of ACD CoAs. However, to deploy this in a real production environment, further research should be done to explore the complete set of cyber actions and system dynamic attributes that are used in practice. This study is crucial to identify additional constraints required for efficient CoA orchestration. Our evaluation also suggests high computation complexity with respect to the number of control variables. We plan to investigate model reduction heuristics or alternative models to make our orchestration more scalable.

5.3 Automated Extraction of Agility Parameters for Cyber Deterrence and Deception Planning

In Chapter 4, we presented the first analytic framework towards automated creation of deterrence and deception techniques schemes based on symbolic malware binary code execution and automated reasoning of attack behaviors and decision-making process. We have implemented a framework that leverages a powerful symbolic execution, S^2E that provide a customizable environment for running and monitoring malware. We developed plugins and scripts and instrumented S^2E to (1) intercept system and library calls and mark relevant information as symbolic, (2) guide the path exploration of the symbolic execution, and (3) collect the appropriate logs and execution traces to construct the complete behavior model for the malware. We then analyze the malware behavior model and identify the system parameters that can influence the decisions of the malware and steer its execution towards the desired deception and deterrence goals. Since multiple parameters may be identified as candidates for deterrence and deception, we formalized a set of properties and formal constraints to select the optimal set of parameters that can deliver effective deterrence and deception. We analyzed many recent malware and demonstrated through four detailed case studies how our agility-oriented analysis can lead to effective deception and deterrence schemes against major malware types: worms, cryptocurrency mining

malware, credential-stealing malware, and ransomware. We recommended multiple deception and deterrence techniques for each of these malware families based on our findings.

From our experience in developing this agility-oriented symbolic execution framework, we have found various potentially challenging problems for future research. First, we will run an extensive evaluation of thousands of malware instances and identify the most common agility parameters. This can be useful in designing proactive deception mechanisms that are active all the time and can be effective against the majority of current and new malware. In addition, we plan to extend the set of intercepted APIs. Although we currently intercept the most commonly used API, the behavior analysis cannot be perforce without considering all possible APIs. We have also experienced some technical challenges dealing with different implementation and evasion techniques used by malware developer. Further research needs to examine more closely these techniques and address them using special scripts and plugins.

REFERENCES

[1] Verizon, "2016 data breach investigations report." http://www.verizonenterprise.com/resources/reports/rp_DBIR_2016_Report_en_xg.pdf, 2016.

[2] Verizon, "2017 data breach investigations report." http://www.verizonenterprise.com/resources/reports/rp_DBIR_2017_Report_en_xg.pdf, 2017.

[3] S. Jackson, *Architecting resilient systems: Accident avoidance and survival and recovery from disruptions*, vol. 66. John Wiley & Sons, 2009.

[4] Y. Yuan, F. Sun, and H. Liu, "Resilient control of cyber-physical systems against intelligent attacker: a hierarchal stackelberg game approach," *International Journal of Systems Science*, vol. 47, no. 9, pp. 2067–2077, 2016.

[5] P. Uday and K. Marais, "Designing resilient systems-of-systems: A survey of metrics, methods, and challenges," *Systems Engineering*, vol. 18, no. 5, pp. 491–510, 2015.

[6] M. Zhu and S. Martínez, "On the performance analysis of resilient networked control systems under replay attacks," *IEEE Transactions on Automatic Control*, vol. 59, no. 3, pp. 804–808, 2014.

[7] B. Cornu, L. Seinturier, and M. Monperrus, "Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions," *Information and Software Technology*, vol. 57, pp. 66–76, 2015.

[8] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

[9] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, *et al.*, "Software engineering meets control theory," in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 71–82, IEEE Press, 2015.

[10] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 389–400, ACM, 2014.

[11] C. G. Chittister and Y. Y. Haimes, "The role of modeling in the resilience of cyberinfrastructure systems and preparedness for cyber intrusions," *Journal of Homeland Security and Emergency Management*, vol. 8, no. 1, 2011.

[12] H. Goldman, "Building secure, resilient architectures for cyber mission assurance," *The MITRE Corporation*, 2010.

[13] Z. A. Collier, D. DiMase, S. Walters, M. M. Tehranipoor, J. H. Lambert, and I. Linkov, "Cybersecurity standards: Managing risk and creating resilience," *Computer*, vol. 47, pp. 70–76, Sept 2014.

[14] H. Tran, E. Campos-Nanez, P. Fomin, and J. Wasek, "Cyber resilience recovery model to combat zero-day malware attacks," *computers & security*, vol. 61, pp. 19–31, 2016.

[15] I. Linkov and J. M. Palma-Oliveira, "An introduction to resilience for critical infrastructures," in *Resilience and Risk*, pp. 3–17, Springer, 2017.

[16] P. E. Roege, Z. A. Collier, V. Chevardin, P. Chouinard, M.-V. Florin, J. H. Lambert, K. Nielsen, M. Nogal, and B. Todorovic, "Bridging the gap from cyber security to resilience," in *Resilience and Risk*, pp. 383–414, Springer, 2017.

[17] G. Pescaroli, R. Wicks, G. Giacomello, and D. Alexander, "Increasing resilience to cascading events: The m. or. d. or. scenario," *Safety Science*, 2018.

[18] D. Bodeau and R. Graubart, "Cyber resilience metrics: Key observations," 2016.

[19] W. A. Conklin, D. Shoemaker, and A. Kohnke, "Cyber resilience: Rethinking cybersecurity strategy to build a cyber resilient architecture," in *ICMLG2017 5th International Conference on Management Leadership and Governance*, p. 105, Academic Conferences and publishing limited, 2017.

[20] W. Leonard, "Resilient cyber-secure systems and system of systems: Implications for the department of defense," in *Disciplinary Convergence in Systems Engineering Research*, pp. 145–156, Springer, 2018.

[21] A. Melin, E. Ferragut, J. Laska, D. Fugate, and R. Kisner, "A mathematical framework for the analysis of cyber-resilient control systems," in *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, pp. 13–18, Aug 2013.

[22] C. Rieger, D. Gertman, and M. McQueen, "Resilient control systems: Next generation design research," in *Human System Interactions, 2009. HSI '09. 2nd Conference on*, pp. 632–636, May 2009.

[23] D. Bodeau and R. Graubart, "Cyber resiliency engineering framework," *MTR110237, MITRECorporation*, 2011.

[24] H. Goldman, R. McQuaid, and J. Picciotto, "Cyber resilience for mission assurance," in *Technologies for Homeland Security (HST), 2011 IEEE International Conference on*, pp. 236–241, IEEE, 2011.

[25] M. A. Rahman and E. Al-Shaer, "A formal framework for network security design synthesis," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pp. 560–570, IEEE, 2013.

[26] M. Sahinoglu, "Quantitative risk assessment for dependent vulnerabilities," in *Reliability and Maintainability Symposium, 2006. RAMS '06. Annual*, pp. 82–85, Jan 2006.

[27] M. Sahinoglu, "Security meter: a practical decision-tree model to quantify risk," *Security Privacy, IEEE*, vol. 3, pp. 18–24, May 2005.

[28] M. N. Alsaleh, S. Al-Haj, and E. Al-Shaer, "Objective metrics for firewall security: A holistic view," in *Communications and Network Security (CNS), 2013 IEEE Conference on*, pp. 470–477, IEEE, 2013.

[29] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao, "Heterogeneous networking: A new survivability paradigm," in *Proceedings of the 2001 Workshop on New Security Paradigms*, NSPW '01, (New York, NY, USA), pp. 33–39, ACM, 2001.

[30] A. Miu, H. Balakrishnan, and C. E. Koksal, "Improving loss resilience with multi-radio diversity in wireless networks," in *Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, MobiCom '05, (New York, NY, USA), pp. 16–30, ACM, 2005.

[31] Y. Yang, S. Zhu, and G. Cao, "Improving sensor network immunity under worm attacks: A software diversity approach," in *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '08, (New York, NY, USA), pp. 149–158, ACM, 2008.

[32] M. G. Bailey, "Malware resistant networking using system diversity," in *Proceedings of the 6th Conference on Information Technology Education*, SIGITE '05, (New York, NY, USA), pp. 191–197, ACM, 2005.

[33] C. Hall, R. Anderson, R. Clayton, E. Ouzounis, and P. Trimintzios, "Resilience of the internet interconnection ecosystem," in *Economics of Information Security and Privacy III*, pp. 119–148, Springer, 2013.

[34] M. N. Alsaleh and E. Al-Shaer, "Towards automated verification of active cyber defense strategies on software defined networks," in *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, pp. 23–29, ACM, 2016.

[35] A. Shameli-Sendi and M. Dagenais, "Arito: Cyber-attack response system using accurate risk impact tolerance," *International journal of information security*, vol. 13, no. 4, pp. 367–390, 2014.

[36] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 43–48, ACM, 2012.

[37] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control,"

[38] IBM, "IBM Resilient." https://www.resilientsystems.com/.

[39] AlienVault, "AlienVault USM Anywhere." https://www.alienvault.com/products/usm-anywhere/.

[40] Phantom, "Phantom Security Automation and Orchestration Platform." https://www.phantom.us/.

[41] V. Casola, A. De Benedictis, and M. Albanese, "A multi-layer moving target defense approach for protecting resource-constrained distributed devices," in *Integration of Reusable Systems*, pp. 299–324, Springer, 2014.

[42] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 942–953, ACM, 2014.

[43] A. Kaur, "Dynamic honeypot construction," 2013.

[44] S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeyproxy: Design and implementation of next-generation honeynet via sdn," in *IEEE Conference on Communications and Network Security (CNS)*, 2017.

[45] M. A. McQueen and W. F. Boyer, "Deception used for cyber defense of control systems," in *Human System Interactions, 2009. HSI'09. 2nd Conference on*, pp. 624–631, IEEE, 2009.

[46] J. Sun, K. Sun, and Q. Li, "Cybermoat: Camouflaging critical server infrastructures with large scale decoy farms," in *Communications and Network Security (CNS), 2017 IEEE Conference on*, pp. 1–9, IEEE, 2017.

[47] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving target defense: creating asymmetric uncertainty for cyber threats*, vol. 54. Springer Science & Business Media, 2011.

[48] S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense II*. Springer, 2013.

[49] N. Soule, B. Simidchieva, F. Yaman, R. Watro, J. Loyall, M. Atighetchi, M. Carvalho, D. Last, D. Myers, and C. B. Flatley, "Quantifying & minimizing attack surfaces containing moving target defenses,"

[50] Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang, "Incentive compatible moving target defense against vm-colocation attacks in clouds," in *Information Security and Privacy Research* (D. Gritzalis, S. Furnell, and M. Theoharidou, eds.), vol. 376 of *IFIP Advances in Information and Communication Technology*, pp. 388–399, Springer Berlin Heidelberg, 2012.

[51] S. Wagner, E. V. D. Berg, J. Giacopelli, A. Ghetie, J. Burns, M. Tauil, S. Sen, M. Wang, M. Chiang, T. Lan, *et al.*, "Autonomous, collaborative control for resilient cyber defense (accord)," in *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*, pp. 39–46, IEEE, 2012.

[52] F. Xing and W. Wang, "Analyzing resilience to node misbehaviors in wireless multi-hop networks," in *Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE*, pp. 3489–3494, March 2007.

[53] D. Rosenkrantz, S. Goel, S. Ravi, and J. Gangolly, "Resilience metrics for service-oriented networks: A service allocation approach," *Services Computing, IEEE Transactions on*, vol. 2, pp. 183–196, July 2009.

[54] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 517–529, ACM, 2015.

[55] A. Martínez-Álvarez, F. Restrepo-Calle, S. Cuenca-Asensi, L. M. Reyneri, A. Lindoso, and L. Entrena, "A hardware-software approach for on-line soft error mitigation in interrupt-driven applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 4, pp. 502–508, 2016.

[56] S. Fukumoto and M. Ohara, "Software rejuvenation schemes for time warp-based pdes," in *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, pp. 313–314, IEEE, 2015.

[57] S.-T. Shen, H.-Y. Lin, and W.-G. Tzeng, "An effective integrity check scheme for secure erasure code-based storage systems," *IEEE Transactions on Reliability*, vol. 64, no. 3, pp. 840–851, 2015.

[58] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures.," in *NDSS*, vol. 15, pp. 8–11, 2015.

[59] S. Lal, S. Ravidas, I. Oliver, and T. Taleb, "Assuring virtual network function image integrity and host sealing in telco cloue," in *Communications (ICC), 2017 IEEE International Conference on*, pp. 1–6, IEEE, 2017.

[60] G. Russo, I. Rego, J. Perelman, and P. P. Barros, "A tale of loss of privilege, resilience and change: the impact of the economic crisis on physicians and

medical services in portugal," *Health Policy*, vol. 120, no. 9, pp. 1079–1086, 2016.

[61] A. Blankstein and M. J. Freedman, "Automating isolation and least privilege in web services," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 133–148, IEEE, 2014.

[62] S. Barnum, "Standardizing cyber threat intelligence information with the structured threat information expression (stix)," *MITRE Corporation*, vol. 11, pp. 1–22, 2012.

[63] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, ACM, 2001.

[64] R. Nieuwenhuis and A. Oliveras, "On sat modulo theories and optimization problems," in *Theory and Applications of Satisfiability Testing-SAT 2006*, pp. 156–169, Springer, 2006.

[65] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, 2011.

[66] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, (New York, NY, USA), pp. 121–126, ACM, 2012.

[67] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pp. 37–44, ACM, 2010.

[68] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *ICNP*, pp. 123–132, 2009.

[69] M. N. Alsaleh, E. Al-Shaer, and A. El-Atawy, "Towards a unified modeling and verification of network and system security configurations," in *Automated Security Management*, pp. 3–19, Springer, 2013.

[70] N. Bjørner and L. De Moura, "Z310: Applications, enablers, challenges and directions," in *Sixth International Workshop on Constraints in Formal Verification*, 2009.

[71] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*, pp. 7–12, IEEE, 2015.

[72] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[73] A. Biewer, B. Andres, J. Gladigau, T. Schaub, and C. Haubelt, "A symbolic system synthesis approach for hard real-time systems based on coordinated smt-solving," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 357–362, EDA Consortium, 2015.

[74] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 470–487, 2015.

[75] L. De Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Formal Methods: Foundations and Applications*, pp. 23–36, Springer, 2009.

[76] Microsoft, *Z3: An Efficient Theorem Prover*, 2012. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[77] S. International, *Yices: An SMT Solver*, 2012. `http://yices.csl.sri.com/`.

[78] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Computer aided verification*, pp. 171–177, Springer, 2011.

[79] H. D. Macedo and T. Touili, "Mining malware specifications through static reachability analysis," in *Computer Security–ESORICS 2013*, pp. 517–535, Springer, 2013.

[80] J. Wei, F. Zhu, and Y. Shinjo, "Static analysis based invariant detection for commodity operating systems," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*, pp. 287–296, IEEE, 2011.

[81] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 424–438, 2010.

[82] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Behavior abstraction in malware analysis," in *Runtime Verification*, pp. 168–182, Springer, 2010.

[83] M. Gaudesi, A. Marcelli, E. Sanchez, G. Squillero, and A. Tonda, "Malware obfuscation through evolutionary packers," in *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, pp. 757–758, ACM, 2015.

[84] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[85] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in Computer Virology*, vol. 7, no. 4, pp. 247–258, 2011.

[86] V. P. Nair, H. Jain, Y. K. Golecha, M. S. Gaur, and V. Laxmi, "Medusa: Metamorphic malware dynamic analysis usingsignature from api," in *Proceedings of the 3rd International Conference on Security of Information and Networks*, pp. 263–269, ACM, 2010.

[87] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[88] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 2, 2012.

[89] FIRST, "Common vulnerability scoring system v3.0: Specification document," 2015.

[90] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pp. 164–175, IEEE, 2017.

[91] S. Abraham and S. Nair, "Exploitability analysis using predictive cybersecurity framework," in *Cybernetics (CYBCONF), 2015 IEEE 2nd International Conference on*, pp. 317–323, IEEE, 2015.

[92] A. Younis, Y. K. Malaiya, and I. Ray, "Assessing vulnerability exploitability risk using software properties," *Software Quality Journal*, vol. 24, no. 1, pp. 159–202, 2016.

[93] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[94] C. Theisen, "Reusing stack traces: automated attack surface approximation," in *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 859–862, ACM, 2016.

[95] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese, "Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 5, pp. 1071–1086, 2016.

[96] R. Ierusalimschy, *Programming in lua*. Roberto Ierusalimschy, 2006.

[97] M. A. Rahman and E. Al-Shaer, "Automated synthesis of distributed network access controls: A formal framework with refinement," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 416–430, 2017.

[98] B. Zhang and E. Al-Shaer, "On synthesizing distributed firewall configurations considering risk, usability and cost constraints," in *Proceedings of the 7th International Conference on Network and Services Management*, pp. 28–36, International Federation for Information Processing, 2011.

[99] H. Goldman, R. McQuaid, and J. Picciotto, "Cyber resilience for mission assurance," in *Technologies for Homeland Security (HST), 2011 IEEE International Conference on*, pp. 236–241, IEEE, 2011.

[100] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Os diversity for intrusion tolerance: Myth or reality?," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pp. 383–394, IEEE, 2011.

[101] M. A. Rahman, A. Farooq, A. Datta, and E. Al-Shaer, "Automated synthesis of resiliency configurations for cyber networks," in *Communications and Network Security (CNS), 2016 IEEE Conference on*, pp. 243–251, IEEE, 2016.

[102] S. Neti, A. Somayaji, and M. E. Locasto, "Software diversity: Security, entropy and game theory.," in *HotSec*, 2012.

[103] A. Newell, D. Obenshain, T. Tantillo, C. Nita-Rotaru, and Y. Amir, "Increasing network resiliency by optimally assigning diverse variants to routing nodes," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 6, pp. 602–614, 2015.

[104] A. J. O'Donnell and H. Sethu, "On achieving software diversity for improved network security using distributed coloring algorithms," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 121–131, ACM, 2004.

[105] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz, "Large-scale automated software diversity - program evolution redux," *IEEE Transactions on Dependable and Secure Computing*, no. 1, pp. 1–1, 2017.

[106] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[107] A. Keromytis, M. Polychronakis, and V. Pappas, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*, pp. 601–615, IEEE, 2012.

[108] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 763–780, IEEE, 2015.

[109] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity.," in *NDSS*, vol. 26, pp. 27–30, 2015.

[110] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pp. 193–202, IEEE, 2001.

[111] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 157–168, ACM, 2012.

[112] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits.," in *USENIX Security Symposium*, pp. 17–17, 2005.

[113] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*, pp. 77–98, Springer, 2011.

[114] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 319–328, ACM, 2012.

[115] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pp. 339–348, IEEE, 2006.

[116] M. Frigault, L. Wang, A. Singhal, and S. Jajodia, "Measuring network security using dynamic bayesian network," in *Proceedings of the 4th ACM workshop on Quality of protection*, pp. 23–30, ACM, 2008.

[117] L. Wang, A. Singhal, and S. Jajodia, "Measuring the overall security of network configurations using attack graphs," in *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 98–112, Springer, 2007.

[118] A. Saidane, V. Nicomette, and Y. Deswarte, "The design of a generic intrusion-tolerant architecture for web servers," *IEEE Transactions on dependable and secure computing*, vol. 6, no. 1, pp. 45–58, 2009.

[119] A. Teixeira, I. Shames, H. Sandberg, and K. H. Johansson, "Distributed fault detection and isolation resilient to network model uncertainties," *IEEE transactions on cybernetics*, vol. 44, no. 11, pp. 2024–2037, 2014.

[120] W. Zeng and M.-Y. Chow, "Resilient distributed control in the presence of misbehaving agents in networked control systems," *IEEE transactions on cybernetics*, vol. 44, no. 11, pp. 2038–2049, 2014.

[121] A. Farraj, E. Hammad, and D. Kundur, "A cyber-enabled stabilizing control scheme for resilient smart grid systems," *IEEE Transactions on Smart Grid*, vol. 7, no. 4, pp. 1856–1865, 2016.

[122] M. N. Alsaleh, E. Al-Shaer, and G. Husari, "Roi-driven cyber risk mitigation using host compliance and network configuration," *Journal of Network and Systems Management*, vol. 25, no. 4, pp. 759–783, 2017.

[123] A. Roy, D. S. Kim, and K. S. Trivedi, "Attack countermeasure trees (act): towards unifying the constructs of attack and defense trees," *Security and Communication Networks*, vol. 5, no. 8, pp. 929–943, 2012.

[124] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "Dag-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer science review*, vol. 13, pp. 1–38, 2014.

[125] M. N. Alsaleh, E. Al-Shaer, and Q. Duan, "Verifying the enforcement and effectiveness of network lateral movement resistance techniques," in *Proceedings of the 15th International Conference on Security and Cryptography*, 2018.

[126] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[127] P. Larsen, S. Brunthaler, L. Davi, A.-R. Sadeghi, and M. Franz, "Automated software diversity," *Synthesis Lectures on Information Security, Privacy, & Trust*, vol. 10, no. 2, pp. 1–88, 2015.

[128] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 268–279, ACM, 2015.

[129] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *Computer Security – ESORICS 2015* (G. Pernul, P. Y A Ryan, and E. Weippl, eds.), (Cham), pp. 69–89, Springer International Publishing, 2015.

[130] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 24:1–24:26, Jan. 2013.

[131] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection.," in *USENIX Security Symposium*, vol. 2014, 2014.

[132] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in

Proceedings of the 6th USENIX conference on Offensive Technologies, pp. 7–7, USENIX Association, 2012.

[133] W. M.-A. Tomasik Joanna, "Internet topology on as-level: model, generation methods and tool," in *29th IEEE International Performance Computing and Communications Conference (IPCCC'10)*, (Albuquerque, NM, USA), December 2010.

[134] "Center for applied internet data analysis." http://www.caida.org/research/topology/, 2010.

[135] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, pp. 291–307, Feb 1970.

[136] ServiceNow, "Security Incident Response Orchestration by ServiceNow." https://docs.servicenow.com/bundle/kingston-security-management/page/product/security-incident-response-orchestration/concept/c_SecIncRespOrchestration.html/.

[137] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *Practical Aspects of Declarative Languages*, pp. 235–249, Springer, 2011.

[138] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," *Technical Reprot of USENIX*, 2013.

[139] P. Hudak, "Functional reactive programming," in *Programming Languages and Systems*, pp. 1–1, Springer, 1999.

[140] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, pp. 279–291, ACM, 2011.

[141] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks.," in *NSDI*, pp. 113–126, 2012.

[142] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in openflow," in *Communications (ICC), 2013 IEEE International Conference on*, pp. 1974–1979, June 2013.

[143] N. Bjorner and K. Jayaraman, "Network verification: Calculus and solvers," in *Science and Technology Conference (Modern Networking Technologies)(MoNeTeC), 2014 International*, pp. 1–4, IEEE, 2014.

[144] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[145] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis.," in *NSDI*, pp. 99–111, 2013.

[146] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "Flowguard: Building robust firewalls for software-defined networks," 2014.

[147] K. Chatterjee and T. A. Henzinger, "Assume-guarantee synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 261–275, Springer, 2007.

[148] K. Chatterjee and V. Raman, "Assume-guarantee synthesis for digital contract signing," *Formal Aspects of Computing*, vol. 26, no. 4, pp. 825–859, 2014.

[149] B. S. Networks, "Project floodlight." http://www.projectfloodlight.org/.

[150] F. Gillani, E. Al-Shaer, S. Lo, Q. Duan, M. Ammar, and E. Zegura, "Agile virtualized infrastructure to proactively defend against cyber attacks," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 729–737, IEEE, 2015.

[151] P. Qin, B. Dai, B. Huang, and G. Xu, "Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data," *arXiv preprint arXiv:1403.2800*, 2014.

[152] K. Lab, "Kaspersky security bulletin. overall statistics for 2017." https://securelist.com/ksb-overall-statistics-2017/83453/, 2017.

[153] K. E. Heckman, F. J. Stech, R. K. Thomas, B. Schmoker, and A. W. Tsow, *Cyber denial, deception and counter deception*. Springer, 2015.

[154] E. Al-Shaer and M. A. Rahman, "Attribution, temptation, and expectation: A formal framework for defense-by-deception in cyberwarfare," in *Cyber Warfare*, pp. 57–80, Springer, 2015.

[155] H. Jafarian, Q. Duan, and E. Al-Shaer, "Effective address mutation approach for disrupting reconnaissance attacks," *To appear in IEEE Transactions on Information Forensics and Security*, 2016.

[156] S. F. H. Gillani, E. Al-Shaer, S. L. ?, Q. Duan, and M. A. ?and Ellen Zegura, "Agile virtualized infrastructure to proactively defend against cyber attacks," in *Infocom*, 2015.

[157] Q. Zhu and T. Başar, "Game-theoretic approach to feedback-driven multi-stage moving target defense," in *Decision and Game Theory for Security*, pp. 246–263, Springer, 2013.

[158] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "An effective address mutation approach for disrupting reconnaissance attacks," *IEEE Transactions on Information Forensics and Security*, vol. 10, pp. 2562–2577, Dec 2015.

[159] P. Team, "PaX address space layout randomization (ASLR)." `http://pax.grsecurity.net/docs/aslr.txt`, 2015. [Online; accessed 10-Feburary-2017].

[160] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pp. 260–269, Oct 2003.

[161] O. E. David and N. S. Netanyahu, "Deepsign: Deep learning for automatic malware signature generation and classification," in *Neural Networks (IJCNN), 2015 International Joint Conference on*, pp. 1–8, IEEE, 2015.

[162] H. D. Macedo and T. Touili, "Mining malware specifications through static reachability analysis," in *Computer Security–ESORICS 2013*, pp. 517–535, Springer, 2013.

[163] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proc. of USENIX Security'09*, 2009.

[164] P. R. C. Song and W. Lee, "Impeding automated malware analysis with environment-sensitive malware," in *Proc. of HotSec'12*, 2012.

[165] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," in *Proc of NDSS'10*, 2010.

[166] M. L, C. K., and M.Paolo, "Detecting Environment-Sensitive Malware," in *Proc. of RAID'11*, 2011.

[167] Z. Xu, L. Chen, G. Gu, and C. Kruegel, "PeerPress: Utilizing enemies' p2p strength against them," in *Proc.of CCS'12*, 2012.

[168] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Goldeneye: Efficiently and effectively unveiling malware's targeted environment," in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, September 2014.

[169] J. Wilhelm and T. Chiueh, "A forced sampled execution approach to kernel rootkit identification.," in *Proc. of RAID'07*, 2007.

[170] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 2014 USENIX Security Symposium*, (San Diego, CA), August 2014.

[171] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Analysis and Defense* (W. Lee, C. Wang, and D. Dagon, eds.), vol. 36, pp. 65–88, Springer, 2008.

[172] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proc. of S&P'07*, 2007.

[173] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *Proc. of CCS'11*, 2011.

[174] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *Proc. of S&P'12*, 2012.

[175] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Krugel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proc. of S&P'10*, 2010.

[176] G. Portokalidis and A. D. Keromytis, "Global isr: Toward a comprehensive defense against unauthorized code execution," in *Moving Target Defense*, pp. 49–76, Springer, 2011.

[177] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*, pp. 77–98, Springer, 2011.

[178] Q. Zhu, A. Clark, R. Poovendran, and T. Basar, "Deceptive routing games," in *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pp. 2704–2711, IEEE, 2012.

[179] S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, eds., *Moving Target Defense II - Application of Game Theory and Adversarial Modeling*, vol. 100 of *Advances in Information Security*. Springer, 2013.

[180] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Formal approach for route agility against persistent attackers," in *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pp. 237–254, 2013.

[181] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Computer Networks*, vol. 51, no. 12, pp. 3471–3490, 2007.

[182] "VirusSign." `http://www.virussign.com/`.

[183] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pp. 421–430, IEEE, 2007.

[184] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, pp. 190–200, ACM, 2005.

[185] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, (New York, NY, USA), pp. 5–14, ACM, 2007.

[186] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware detection using assembly and api call sequences," *J. Comput. Virol.*, vol. 7, pp. 107–119, May 2011.

[187] Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu, *CBM: Free, Automatic Malware Analysis Framework Using API Call Sequences*, pp. 225–236. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[188] "Blaster worm source." `https://gist.github.com/yorickdewid/a9fb98da3c367b360e36`.

[189] "Microsoft Security Bulletin MS03-026." https://technet.microsoft.com/en-us/library/security/ms03-026.aspx.

[190] W. Maples, "Disable windows scripting host (wsh),"

[191] N. Falliere, "Windows anti-debug reference." `https://www.symantec.com/connect/articles/windows-anti-debug-reference`. [Online; accessed 04-February-2018].

[192] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 225–236, ACM, 2009.

[193] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: dynamic taint analysis with targeted control-flow propagation.," in *NDSS*, 2011.

[194] N. Ge, A. Dieumegard, E. Jenn, and L. Voisin, "Correct-by-construction specification to verified code," *Journal of Software: Evolution and Process*, 2017.

[195] I. Poernomo and J. Terrell, "Correct-by-construction model transformations from partially ordered specifications in coq," in *International Conference on Formal Engineering Methods*, pp. 56–73, Springer, 2010.

APPENDIX A: Intercepted APIs

Table A.1 shows the complete list of the APIs we intercept and mark their output arguments as symbolic. The output arguments have different structures and formats. Hence, we carefully mark the appropriate registers and memory blocks based on complete understanding of the APIs and their arguments to avoid any access violations.

Table A.1: The complete list of APIs Intercepted by *gExtractor*

API	API
kernel32.dll	ntdll.dll
GetComputerNameA	NtQueryAttributesFile
GetComputerNameW	NtQueryFullAttributesFile
GetTimeZoneInformation	NtOpenFile
GetDiskFreeSpaceW	NtReadFile
GetDiskFreeSpaceExW	NtWriteFile
GetSystemTime	NtQuerySystemInformation
GetSystemTimeAsFileTime	NtQueryMultipleValueKey
GetTickCount	version.dll
GetFileAttributesExW	GetFileVersionInfoW
SearchPathW	GetFileVersionInfoExW
GetSystemDirectoryW	GetFileVersionInfoSizeExW
SetFileTime	GetFileVersionInfoSizeW
GetTempPathW	user32.dll
GetFileType	GetSystemMetrics
CreateDirectoryW	RegisterHotKey
GetSystemDirectoryA	EnumWindows
SetFileInformationByHandle	FindWindowExA
GetFileInformationByHandleEx	GetKeyboardState
CopyFileW	UnhookWindowsHookEx
SetFilePointer	GetKeyState
CopyFileA	GetForegroundWindow
GetSystemWindowsDirectoryW	LoadStringA
SetFilePointerEx	DrawTextExW
CopyFileExW	FindWindowW
SetFileAttributesW	LoadStringW
CreateDirectoryExW	FindWindowExW
GetFileSize	FindWindowA
GetSystemWindowsDirectoryA	DrawTextExA
DeleteFileWGetFileInformationByHandle	GetAsyncKeyState
GetFileAttributesW	SetWindowsHookExA
RemoveDirectoryW	SetWindowsHookExW
FindFirstFileExA	

API	API
MoveFileWithProgressW	netapi32.dll
kernel32.dll	CreateServiceW
SetEndOfFile	EnumServicesStatusA
RemoveDirectoryA	LookupPrivilegeValueW
FindFirstFileExW	NetUserGetLocalGroups
GetFileSizeEx	NetUserGetInfo
GetSystemInfo	NetShareEnum
GetNativeSystemInfo	GetUserNameW
SetErrorMode	GetUserNameA
crypt32.dll	LookupAccountSidW
CertControlStore	EnumServicesStatusW
CertOpenSystemStoreA	StartServiceW
CertOpenStore	OpenServiceA
CertCreateCertificateContext	CreateServiceA
CertOpenSystemStoreW	OpenSCManagerW
srvcli.dll	OpenServiceW
NetShareEnum	ControlService
wininet.dll	StartServiceA
InternetGetConnectedState	DeleteService
InternetReadFile	OpenSCManagerA
InternetOpen	RegEnumKeyExA
InternetConnect	RegQueryInfoKeyA
HttpOpenRequest	RegQueryValueExW
HttpSendRequest	RegCreateKeyExW
InternetQueryOption	RegDeleteKeyA
InternetSetOption	RegEnumValueW
HttpQueryInfo	RegCloseKey
InternetQueryDataAvailable	RegCreateKeyExA
ws2_32.dll	RegSetValueExW
WSAStartup	RegQueryInfoKeyW
sendto	RegQueryValueExA
recvfrom	RegEnumKeyExW
send	RegOpenKeyExW
bind	RegSetValueExA
select	RegDeleteValueW
connect	RegEnumValueA
secur32.dll	RegEnumKeyW
GetUserNameExW	RegDeleteKeyW
GetUserNameExA	RegOpenKeyExA
	RegDeleteValueA