ACHIEVING NETWORK RESILIENCY USING SOUND THEORETICAL AND PRACTICAL METHODS

BY

MOHAMMAD A. NOUREDDINE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair
Professor Klara Nahrstedt
Professor Tamer Başar
Associate Professor Matthew Caesar
Assistant Professor Kassem Fawaz, University of Wisconsin-Madison

ABSTRACT

Computer networks have revolutionized the life of every citizen in our modern interconnected society. The impact of networked systems spans every aspect of our lives, from financial transactions to healthcare and critical services, making these systems an attractive target for malicious entities that aim to make financial or political profit. Specifically, the past decade has witnessed an astounding increase in the number and complexity of sophisticated and targeted attacks, known as advanced persistent threats (APT). Those attacks led to a paradigm shift in the security and reliability communities' perspective on system design; researchers and government agencies accepted the inevitability of incidents and malicious attacks, and marshaled their efforts into the design of resilient systems.

Rather than focusing solely on preventing failures and attacks, resilient systems are able to maintain an acceptable level of operation in the presence of such incidents, and then recover gracefully into normal operation. Alongside prevention, resilient system design focuses on incident detection as well as timely response. Unfortunately, the resiliency efforts of research and industry experts have been hindered by an apparent schism between theory and practice, which allows attackers to maintain the upper hand advantage. This lack of compatibility between the theory and practice of system design is attributed to the following challenges. First, theoreticians often make impractical and unjustifiable assumptions that allow for mathematical tractability while sacrificing accuracy. Second, the security and reliability communities often lack clear definitions of success criteria when comparing different system models and designs. Third, system designers often make implicit or unstated assumptions to favor practicality and ease of design. Finally, resilient systems are tested in private and isolated environments where validation and reproducibility of the results are not publicly accessible.

In this thesis, we set about showing that the proper synergy between theoretical analysis and practical design can enhance the resiliency of networked systems. We illustrate the benefits of this synergy by presenting resiliency approaches that target the inter- and intra-networking levels. At the inter-networking level, we present CPuzzle as a means to protect the transport control protocol (TCP) connection establishment channel from state-exhaustion distributed denial of service attacks (DDoS). CPuzzle leverages client puzzles to limit the rate at which misbehaving users can establish TCP connections. We modeled the problem of determining the puzzle difficulty as a Stackleberg game and solve for the equilibrium strategy that balances the users' utilizes against CPuzzle's resilience capabili-

ties. Furthermore, to handle volumetric DDoS attacks, we extend CPuzzle and implement Midgard, a cooperative approach that involves end-users in the process of tolerating and neutralizing DDoS attacks. Midgard is a middlebox that resides at the edge of an Internet service provider's network and uses client puzzles at the IP level to allocate bandwidth to its users.

At the intra-networking level, we present sShield, a game-theoretic network response engine that manipulates a network's connectivity in response to an attacker who is moving laterally to compromise a high-value asset. To implement such decision making algorithms, we leverage the recent advances in software-defined networking (SDN) to collect logs and security alerts about the network and implement response actions. However, the programmability offered by SDN comes with an increased chance for design-time bugs that can have drastic consequences on the reliability and security of a networked system. We therefore introduce BiFrost, an open-source tool that aims to verify safety and security properties about data-plane programs. BiFrost translates data-plane programs into functionally equivalent sequential circuits, and then uses well-established hardware reduction, abstraction, and verification techniques to establish correctness proofs about data-plane programs.

By focusing on those four key efforts, CPuzzle, Midgard, sShield, and BiFrost, we believe that this work illustrates the benefits that the synergy between theory and practice can bring into the world of resilient system design. This thesis is an attempt to pave the way for further cooperation and coordination between theoreticians and practitioners, in the hope of designing resilient networked systems.

To those who have paved the way for us to prosper.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Professor William H. Sanders, for his constant support throughout the past six years. Bill has given me the chance to grow as an independent researcher and has provided me with the opportunity to sharpen my teaching and research skills in a supportive environment.

I am also very grateful to the members of my Ph.D. Committee: Professors Klara Nahrstedt, Tamer Başar, Matthew Caesar, and Kassem Fawaz for their invaluable comments and discussions. Specifically, I would like to express my indebtedness to Professor Tamer Başar for comments and discussions about various aspects of game theory and control theory. In addition, I would like to thank Professor Matthew Caesar for the lengthy discussions we've shared about various network paradigms and designs, as well as general life advice.

I would like to acknowledge Professor Adam Bates for allowing me to collaborate with several members of the secure and transparent systems (STS) laboratory, and for providing me with guidance and comments about data provenance for the past two years. In addition, I really enjoyed working with Adam as a teaching assistant (TA) for the Operating Systems class. Being a TA with Adam is one of the main reasons why I chose to pursue a career in teaching.

During my Ph.D., I was fortunate to have had the company of the performability engineering (PERFORM) research group, Ahmed Fawaz, Uttam Thakore, Atul Bohara, Carmen Cheh, Brett Feddersen, Michael Rausch, Ben Ujcich, Varun Krishna, Ken Keefe, and Ron Wright. I would like to single out Ahmed Fawaz for being a close friend and for helping make my move to Illinois all the more easy and enjoyable. Also, I would like to thank Uttam Thakore for all the lengthy discussions we've shared about politics and the different aspects of the culinary world. Furthermore, I would like to thank Wajih Ul-Hassan and Pubali Datta from the STS lab for their collaboration and discussions when I decided to take a leap into the world of data provenance. Finally, I would like to thank Jenny Applequist for her editorial comments on all of my papers and this thesis. Jenny has been very patient and has tolerated my sloppy writing throughout the past six years. This thesis would have read very differently if it wasn't for her.

I am forever grateful and indebted to my closest friends: Dana Joulani, Samah Karim, Hussein Sibai, Ahmed Fawaz, and Amin Jaber. Having the love and support of a close-knit group of friends is a privilege that I have enjoyed throughout my life. They always provided me with the space to escape my work, to rant about my frustrations, and to eat my feelings

at various restaurants.

Most importantly, none of my work would have been possible without the generous and wonderful support of my family: my mother Sonia, my father Ali, my brother Houssein, and my sister-in-law Dalia. I was fortunate to have been born in a household that fostered education and curiosity. My parents have made huge sacrifices to provide my brother and me with the chances to pursue higher education. I am forever in their debt. Additionally, my brother Houssein has been my closest companion throughout the past six years. Despite the eight hours time difference, Houssein always found the time to hear me out and support me when I most needed it. I will never forget the times our conversations kept my brother up until 4 in the morning on a week night. Finally, in the past two years, I was blessed to enjoy the presence of my nephew Sharif and my niece Yassma. Looking at their pictures and videos provided me with the drive to keep going and make them proud. My family is the ultimate manifestation of unconditional love and support. Everything I have achieved and will achieve, I owe it to them.

This work is based upon work supported in part by the Department of Energy under Award Number DE-OE0000780, in part by the Office of Naval Research (ONR) MURI Grant N00014-16-1-2710, and in part by U.S. Army Research Laboratory (ARL) Cooperative Agreement W911NF-17-2-0196. The views and opinions of the authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Computer networks play an essential role in today's societies. They have revolutionized every citizen's daily life. From financial transactions to healthcare, elections, social networks, and critical services, networked devices have become an essential component of societal and economic well-being [1]. Thus, it is critical to protect the continued development and progress of our societies by ensuring the security, reliability, and availability of our networking infrastructure, at both the software and hardware levels.

From a security perspective, our increased dependence on networked systems has created ample space for malicious entities to enact attacks that have drastic consequences. In fact, 2018 witnessed the largest-ever recorded *Distributed Denial of Service* (DDoS) attack, targeting `Github`; it reached a peak bandwidth of 1.35 Tbps [2]. Similarly, in 2016, three college students contributed to a nationwide Internet outage by creating the `Mirai` botnet; access to many major websites (such as Amazon, Netflix, and Visa) was disrupted due to an attack on the Dyn domain name system (DNS) provider [3].

Furthermore, the massive amounts of data that networked companies collect about their users are a very lucrative target. Hackers can compromise such companies' internal networks to gather critical consumer data, such as Social Security numbers [4], credit card numbers [5, 6], and healthcare records [7]. At a larger and more critical scale, attackers illegally acquired social-networking information to influence the outcomes of the 2016 U.S. presidential elections [8, 9]. Those incidents highlight the ever-increasing scale and implications of the security vulnerabilities of our networked systems; indeed, criminal cyber-activity is estimated to have cost the global economy an astounding $600 billion to date [10].

Similarly, from a reliability perspective, the cost of benign downtime in today's data centers is estimated to be $7,900 per minute [11]. Although it enjoys a long and rich history [12], reliability analysis today is facing the challenges of the increased complexity and programmability of today's systems. More specifically, the introduction of *software-defined networking* (SDN) and programmable data-planes has provided network designs with flexible programmability at the expense of higher chances of benign failures [13]. Therefore, it is of the utmost importance to incorporate reliability analysis that uses appropriate modeling and verification techniques as part of the design and implementation of computer networks.

1.1 CYBER RESILIENCY

Faced with the challenges described above, security and reliability engineers and researchers have accepted the inevitability of incidents and malicious attacks. The question

is no longer that of securing only the perimeters of our networked systems; it is more of designing systems that are cyber resilient. Cyber resiliency is a networked system's ability to maintain an acceptable level of operation and proper service in the presence of failures and targeted attacks, and to then recover gracefully into normal operation [14, 15, 16]. The definitions of "acceptable operation" and "proper functionality" are system-specific and are often determined by the services provided by the system and its users' expectations. For example, an e-shopping website can tolerate failed connection attempts or network delays on the order of seconds. On the other hand, an online gaming service provider has a significantly lower tolerance to network delays since, it must provide a near-real-time service to its clients. Nevertheless, resiliency is concerned with the ability to provide service continuously, even if at an acceptable level of degradation, in the presence of failures and attacks.

Achieving cyber resiliency rests on three important functions: (1) prevention, (2) detection, and (3) response [17]. Most of our early efforts in system security focused on prevention. However, the increasing complexity of attacks, and the large uncertainties in the The Internet moved the community towards pouring more efforts into detection and response [18]. Cyber-resilient design builds on the effective and successful tools developed in the realm of fault tolerance. However, it must go beyond addressing benign and predictable failures to account for dynamic attackers that launch targeted attacks. The presence of such attackers significantly increases the uncertainty that designers have about their systems and their inputs, and thus creates greater challenges. *It is our hypothesis in this thesis that cyber resiliency can be enhanced by integrating theoretical and model-based analysis of systems with practical and secure engineering.*

1.2 STATE OF THE ART

To address the hard challenges faced by our networked systems, government agencies have set national agendas to promote cyber-resilient networking design, especially for critical services and infrastructure such as the interconnected power grid [14]. For example, in 2013, then-President Barack Obama issued an executive order setting the U.S. policy to "enhance the security and resilience of the Nation's critical infrastructure and to maintain a cyber environment that encourages efficiency, innovation, and economic prosperity while promoting safety, security, business confidentiality, privacy, and civil liberties" [19].

The academic and industrial communities responded with increased efforts that proposed secure-by-design and resilient systems. Driven by the successes in the field of cryptography [20], computer theoreticians attempted to model and analyze systems and prototypes and measure their relative security and resiliency properties such, as availability, integrity,

confidentiality, reliability, and safety [21, 22, 23, 24, 25, 26, 27, 28]. On the other hand, network and systems engineers ventured to propose and implement secure and resilient designs that can reduce the increasing and often unpredictable waves of failures and cyber attacks [29, 30, 31, 32, 33, 34, 35].

Unfortunately, however, the efforts of the researchers and industry experts have been hindered by an apparent schism between theory and practice, which allows attackers to maintain the upper hand advantage. For example, the security community is deeply fractured between fierce proponents [36, 37] and fierce opponents [38] of security metrics. Similarly, despite enormous governmental efforts, we are yet to see a widely accepted definition of what a *science* of security is, let alone definitions of the requirements for sound scientific security experiments [20, 39, 40].

We attribute the apparent lack of compatibility between theory and practice to the following challenges:

- *Impractical and unjustifiable assumptions:* Theoretical computer scientists often sacrifice practicality for mathematical tractability. Systems in general, and networked systems in particular, often do not exhibit many of the properties needed by mathematically solvable frameworks. For example, in control theory, it is a standard assumption to consider a system to be described by a set of linear differential equations. However, actual systems seldom exhibit this linearity. Similarly, when attempting to quantify the impact of security attacks [27], modelers often place probability values on attack steps that are not reasonably justifiable (e.g., the probability of success of a certain attack step, or the probability that an attacker will adopt such a step). That makes the mathematical model easier to reason about, yet does not create an accurate map of reality.

- *Undefined success criteria:* As mentioned earlier, resiliency and security do not have well-defined success metrics. Questions such as "Is the system secure?" or "How secure is the system?" often do not have quantifiable answers. Therefore, it is challenging to model and design a system when the success metrics of the system are not well-defined.

- *Unstated and unclear assumptions:* In contrast to theoreticians, system designers often sacrifice clarity for practicality and ease of design. Assumptions made at design time often remain unstated which hinders the ability of future researchers to build accurate and practical mathematical models of the systems.

- *Irreproducible experiments:* Finally, resilient and secure systems are often tested in private environments so that validation and reproducibility of the results are not possible.

That leaves the community unable to understand and evaluate the appropriateness of a given design or solution.

Therefore, the resiliency and security of our networked systems, and, in turn, the stability of our societies, rests on our ability to integrate sound theoretical modeling and analysis with practical design and engineering. Bridging the gap between theory and practice is a goal we set out to achieve in this dissertation.

1.3 CONTRIBUTIONS

In this thesis, we set about showing that the proper integration of theoretical analysis and practical design can enhance the resiliency of networked systems. For that purpose, we focus on two essential properties of networked system designs: (1) *theoretical soundness* and (2) *practical realizability* [1]. We provide a formal definition of these two properties and discuss them in detail in this thesis. We will, however, restrict ourselves in this chapter to informal definitions.

We define a design to be *theoretically sound* if it can be modeled by a mathematical model that accurately captures its behavior, and if all inferences made about said mathematical model are *valid* with respect to a set of assumptions. Also, we define a design to be *practically realizable* if it can be built in practice while having (1) input parameters that are quantifiable through observations and data analysis, and (2) well-defined and quantifiable metrics of success and proper service.

The goal of practical realizability forces modelers and designers to think carefully about the assumptions they make in their analyses. The assumptions must be at an appropriate level of abstraction such that they can achieve some level of mathematical tractability without sacrificing faithfulness to the system's actual nature. That balancing act must encompass a careful consideration of the success metrics by which a design is deemed to be achieving proper service.

Complementary to practicality, theoretical soundness compels system designers to clearly state all of their designs' implicit assumptions as well as the expected input and output relationships. Such assumptions and relationships form the ground upon which analytical reasoning about a model is built and guarantees about performance and reliability are proven. Finally, the combination of soundness and practicality cannot be achieved without thorough testing. That requires reproducible experiments that will allow multiple designers, modelers, and analysts to validate a set of obtained metrics.

[1] Not to be confused with the notion of "realizability" in control theory.

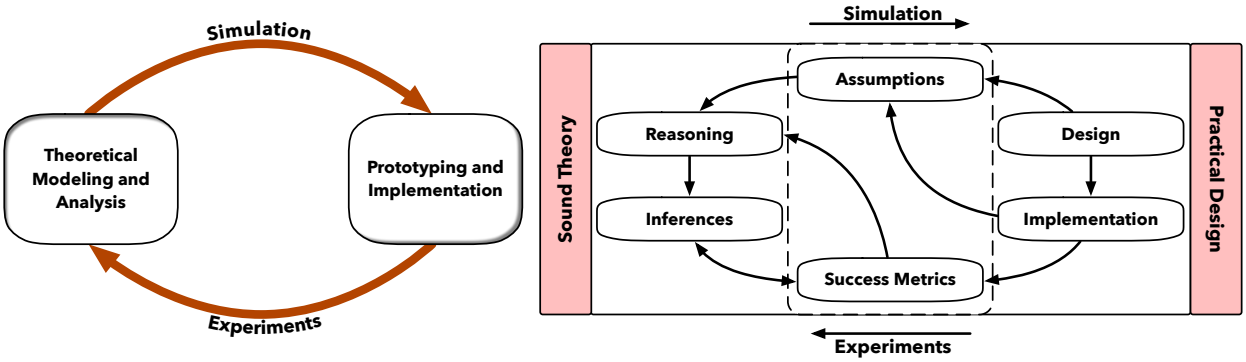


Figure 1.1: Design approaches using traditional reasoning (left) and our integrative reasoning (right).

Therefore, in this work, we set out to validate the following thesis statement:

> *Cyber resiliency of intra- and inter-networks can be enhanced through the integration of theoretically sound and practically realizable networking designs across multiple network layers.*

Figure 1.1 contrasts the traditional system design approach (left) with the approach we adopt in this thesis (right). Traditionally, theoretical modeling and analysis and implementation were seen as separate, non-interleaving processes performed by often disjunct groups of designers. At the end of the modeling phase, *simulations* were conducted to assess the model's properties, which would then be used to drive the prototyping and implementation phase. On the other hand, once a prototype has been built, *experiments* are conducted to evaluate the design, and that could lead to another modeling phase.

In this thesis, we show that theoretical modeling and practical design are interleaved processes that cannot be conducted separately, especially when dealing with designs for security and reliability. Both modelers and practitioners must work from a shared set of *assumptions* and *success metrics*. Those assumptions and success metrics must be driven by the constraints on the implementation, rather than a focus on mathematical tractability. Inferences on the success metrics, made during the theoretical modeling phase via simulation and analytical reasoning, will drive the realizable implementation of the design. Using experiments, the designers revisit their assumptions and possibly add new success metrics, thus driving a new and interleaved phase of modeling and analysis. This integration between theoretical modeling and practical design should be the basis for secure and reliable networking design.

Table 1.1: Summary of thesis contributions. *IER* and *IAR* refer to inter-networking resiliency and intra-networking resiliency, respectively. `GT` refers to game-theoretic techniques, while `CT` refers to control-theoretic techniques.

Solution	**IER**	**IAR**	**Theoretical Tools**	**Deployment**	**Target Property**
CPuzzle	✓	–	GT: Stackelberg game	kernel patch	availability
Midgard	✓	–	CT	kernel patch, middlebox	availability
sShield	–	✓	GT: zero-sum game	SDN	integrity, confidentiality
BiFrost	✓	✓	formal methods	open-source tool	safety, reliability

1.3.1 Summary of Contributions

At the core of this dissertation is an attempt to understand how the integration of sound theory and practical implementation can lead to secure and reliable networking designs. Our investigation is driven by assumptions and success metrics that can act as the gateway between theoretical modeling and practical design. To inform our exploration, we pose the following research questions:

1. What does it mean for a model to be *theoretically sound*?
2. What does it mean for a design to be *practically realizable*?
3. Where do current networking designs fail in providing availability, confidentiality, and reliability?
4. How can the integration of theoretical and practical reasoning be used to improve the state of current networking design?

Table 1.1 summarizes our approach to validating our thesis statement and answering the above research questions. We address the challenges of designing a resilient networking infrastructure by combining the results of four system designs. At the inter-networking level, **CPuzzle** [41] enables network resilience to state-exhaustion DDoS attacks at the transport layer. **Midgard** extends CPuzzle to combat volumetric attacks at the network

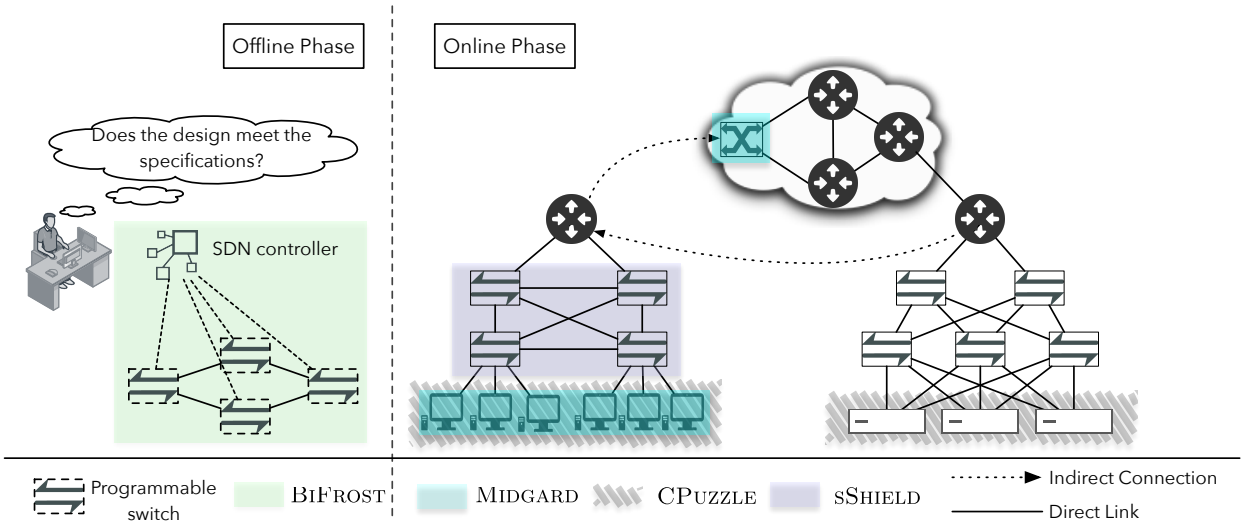


Figure 1.2: Global view of this thesis' contributions.

layer. At the intra-networking level, **sShield** [42] dynamically protects a high-value asset from potential attackers moving laterally in the network. Finally, **BiFrost** [43] is a design-time tool that allows for scalable verification of data-plane programs specified using the P4 language.

Figure 1.2 illustrates the different parts of a network where our proposal approaches are to be deployed. We consider, without loss of generality, an internetworking scenario consisting of several clients connecting to a remote server and requesting a particular service. CPuzzle, Midgard, and sShield are all *online* solutions that get triggered during attacks, while BiFrost is an *offline* analysis tool. CPuzzle is a Linux kernel patch that resides at both the client and server ends, *i.e.* at both ends of a communication. Midgard comprises of two components: (1) a Linux kernel patch applied to the client machines and (2) a middlebox (the Midgard box) residing at the ingress of the server's upstream *Internet Service Provider* (ISP). Internally, sShield is a game-theoretic dynamic topology modifier that leverages the presence of SDN switches at a local network. Finally, network administrators can use BiFrost to verify properties about their SDN deployments that use programmable switches.

CPuzzle

CPuzzle is a theoretical and practical framework for employing client puzzles as a means for tolerating TCP state-exhaustion DDoS attacks. A TCP state-exhaustion attack attempts to consume entries in a victim server's TCP queues, namely the `listen` and `accept` queues. When those queues overflow, the server stops accepting new connections, and benign clients

are thus denied service. Our work in CPuzzle is grounded in the observation that defending against such attacks is no longer solely the responsibility of the network. Rather, it is up to the edge devices (the clients and the victim server) to participate in achieving resilience.

To that end, we developed a theoretical framework that allows a victim server to ask its benign users and the malicious bots to solve computational puzzles. Such puzzles serve as virtual payment for slots in the TCP queues and achieve the goal of rate-limiting malicious actors. We model the problem of selecting the puzzle difficulty as a Stackelberg game [44] in which the server is the leader, and the users and bots are the followers. The model captures the trade-off that the server must consider in balancing the user's computational load against its ability to rate-limit the attackers.

Based on the obtained solution, we designed and implemented a prototype of the CPuzzle framework in the Linux kernel networking stack. Our design favors efficiency and stability at both ends of the communication channel. To evaluate our approach, we deployed CPuzzle on a testbed network comprising 30 machines on the DETER [45] academic testbed. Our evaluation shows that CPuzzle can effectively enhance our infrastructure's ability to tolerate TCP state-exhaustion attacks while balancing that ability against the computational cost for benign users. Our results also highlight how CPuzzle increases the cost to attackers: rate-limiting the attackers forces them to purchase or compromise more botnet machines. We are providing open-source access to the kernel patch as well as all scripts needed to reproduce our experiments on the DETER testbed.

Midgard

We propose Midgard to complement CPuzzle in combating multi-vector DDoS attacks that target the consumption of network bandwidth as well as computational resources. Midgard is an in-network DDoS resilience mechanism that combines the benefits of cloud-based defense solutions with those achieved by puzzle-based rate-limiting. To that end, we suggest the addition of IP puzzles that are distributed by a middlebox (the Midgard box) residing at the edge of a victim server's ISP. When an attack is detected, the Midgard box monitors and seamlessly distributes puzzles to all users and monitors the *round-trip time* (RTT) taken by every request. This RTT value presents a reflection of each user's computational capabilities, allowing the Midgard box to achieve the desired rate-limiting while adapting the puzzle difficulty to each user's prowess as well as ts previous behavior.

In designing Midgard, we leveraged the sound theoretical tools of *control theory* to design a network controller that allocates bandwidth resources to each user based on her estimated computational prowess, her puzzle difficulty, and her behavior. The controller

aims to achieve *fairness* among benign users while thwarting the effects of the malicious actors. We implemented MIDGARD by using `eBPF` and `AF_XDP` sockets in the Linux kernel to allow for a combination of fast packet processing in the kernel space and puzzle generation and verification in user space. We also developed a patch for the IP networking stack of the kernel to support IP puzzles. To evaluate our approach, we deployed MIDGARD on the `DETER` testbed and launched bandwidth exhaustion attacks against a victim server. Our experiments showed that MIDGARD can effectively rate-limit misbehaving users by forcing them to solve harder and harder puzzles.

BiFrost

To address the challenges introduced by the flexibility of programmable data-planes, we propose and have developed BIFROST, a tool for scalable and efficient circuit-level verification of programmable data-planes. BIFROST exploits the fundamental feature of a data-plane program; namely that it describes a bounded hardware pipeline. We therefore leverage previous work on software verification that uses circuit-level approaches [46] to translate a P4 program into an equivalent sequential circuit expressed as an *And-Inverter-Graph* (AIG). That translation introduces two main benefits: (1) the ability to make use of mature, well-established, and scalable circuit verification techniques, and (2) the ability to verify equivalence between multiple compilations of the same program. The latter advantage is especially beneficial when one is faced with a heterogeneous networking environment composed of switches from multiple vendors. We implemented BIFROST using C++ and evaluated it on real-world data-plane programs written in the P4 language. Using BIFROST, we detected header validity bugs in two open-source programs. In such cases, an attacker, using a well-crafted packet, can force a programmable switch to read from invalid memory locations, thus introducing nondeterministic behavior that can lead to reliability and security incidents. In addition, we evaluated BIFROST on a large set of data-plane programs and showed that we could prove properties about those programs in < 3 minutes. We plan to release BIFROST as an open-source tool that can be used by data-plane developers to verify their designs.

sShield

Intrusion detection systems (IDSes) provide network administrators with a plethora of monitoring information, but that information must often be processed manually to enable decisions on response actions and thwart attacks. The gap between detection time and

response time, which may be months long, may allow attackers to move freely in the network and achieve their goals.

sShield is a game-theoretic network response engine that takes practical actions in response to an attacker that is moving laterally in an enterprise network. The engine receives monitoring information from IDSes in the form of a network services graph, which is a graph data structure that represents vulnerable services running between hosts, augmented with a labeling function that highlights services likely to have been compromised. We formulate the decision-making problem as a defense-based zero-sum matrix game that the engine analyzes to select appropriate response actions by solving for saddle-point strategies. Given the response engine's knowledge of the network and the location of sensitive components (e.g., database servers), its goal is to keep the suspicious actors as far away from the sensitive components as possible. The engine is not guaranteed to neutralize threats, if any, but can provide network administrators with enough time to analyze suspicious movement and take appropriate neutralization actions. The decision engine makes use of the monitoring information to decide which nodes' disconnections from the network would slow down the attacker's movements and allow administrators to take response actions.

The rest of this thesis is organized as follows. In Chapter 2, we present our definitions of resiliency, theoretical soundness, and practical realizability. We then turn to inter-networking resiliency and present CPuzzle in Chapter 3 and Midgard in Chapter 4. Subsequently, we discuss our approaches to intra-networking resiliency by presenting BiFrost in Chapter 5 and sShield in Chapter 6. Finally, we conclude and present our notes for the future in Chapter 7.

CHAPTER 2: THE ROAD TO CYBER RESILIENCE

The rapid increase in technological advancements over the past decade has brought a comparable increase in the arsenal of attack vectors that malicious actors can leverage to launch devastating attacks. The ubiquity of connected devices, ranging from home appliances to smartphones and desktops, offers adversaries a variety of lucrative targets that are relatively low-cost and high-reward. Just as connectivity has become an inevitable part of our lives, cyber threats assert themselves as an undeniable fact of the connected lifestyle. The myth of the "fully secure system" has been decisively debunked, and systems engineering is moving into the era of resilient computing.

In this chapter, we motivate and define cyber resilience by drawing an analogy from the field of fault-tolerant computing. We highlight some of the recent technology failures in adopting the resilience design methodology and the often drastic consequences that have resulted from doing so. Finally, we discuss how our proposed approach, namely the integration of theoretical soundness and practical realizability, can be a cornerstone to the building of resilient networks.

2.1 A LOOK THROUGH FAULT-TOLERANT DESIGN

The notion of system resiliency enjoys a rich history that has grown from the need to develop dependable and reliable systems that can withstand hardware software failures [47, 48]. As early as the construction of the first networks, designers and engineers realized that failures are bound to happen and that systems must be able to continue to provide a certain level of service even if such failures occur. That realization led to the development of the rich fields of dependability and fault tolerance, which introduced a variety of techniques, both theoretical and practical, to allow networks and systems to withstand benign failures and continue to operate acceptably.

Although not labeled initially as "resilient" systems, fault-tolerant designs gained vast popularity and enjoyed high levels of success. Many mission-critical systems, such as airplane navigation systems, airspace controllers, and telephone and data networks, enjoy a high level of reliability enabled by years of fault tolerant design work. Furthermore, even in this era of high interconnectedness, cloud computing offers unprecedented levels of reliability and availability due to efficient and well-designed replication and reliability strategies. For example, most current cloud computing providers, such as Google, Microsoft, and Amazon, offer their customers service-level agreements that guarantee 99.999% availability [49, 50, 51].

That translates to just a few minutes of downtime every year while providing continuous service across the world. Such designs leverage years of fault tolerant design research in areas such as georeplication and data duplication (*e.g.* RAID) to provide continuous service at guaranteed latencies and prevent unexpected data loss. Such techniques illustrate an important design paradigm: the fact that faults and failures are not always preventable. Therefore, designers must account for their presence and design for both prevention and tolerance.

We attribute the success of fault tolerance techniques in enabling designs that can withstand benign faults and failures to the following characteristics:

Strong ties to physical phenomena. The study of hardware faults and failures in current networking designs builds upon years of studies of the natural phenomena that underly the operations of such designs (for example, the switching of transistors and the representation of data through magnetic waves). Thus designers can gain a better understanding of the operations of their designs by leveraging the literature of the physical sciences.

Predictability. The fact that the operation and failure of systems can often be traced to underlying natural phenomena presents designers with an attractive property: predictability. Fault-tolerant designs benefit from studies that estimate the lifetimes of their components. Such studies allow designers to build statistical models of the operation and failure behavior of the systems they are building, and thus predict when failures may occur and account for their presence in the designs.

Synergy between theoretical studies and practical designs. The predictability of system failures has allowed designers to develop theoretical models that can effectively act as *mediators* between the practical designs and the theoretical studies and simulations that are to be performed on the models. Most models are based on assumptions tied to well-studied physical phenomena, which enables a rich understanding of the system's behavior and makes it possible to leverage theoretical models to design fault-tolerant systems. The fact that assumptions are based on real and measurable parameters allows models to reflect of the practical reality, and that allows designers to transition easily from a theoretical design to a practical prototype with tangible fault-tolerance guarantees.

As an example of that synergy, the designed of the TCP congestion avoidance and control algorithm [52] leveraged sound control-theoretic methodologies to design an estimator for a packet's round-trip time. The algorithm uses the estimated value to control the TCP socket's sending rate in response to the network's inferred congestion state. By using real-time feedback from the network (*i.e.* measurements of a packet's travel time), the algorithm can effectively adapt to changes and congestion events in the network. The design of this algo-

rithm shows that sound theory, coupled with quantifiable input parameters and measurable success metrics, can lead to the design, and later refinement, of a robust algorithm.

2.2 FROM FAULT TOLERANCE TO RESILIENCY

Traditional approaches to security and resiliency have focused on building systems that can build and harden perimeter defenses to prevent malicious attacks. However, as the number of successful high-impact attacks has increased, security engineers have realized that prevention alone is not enough; attacks change and evolve at a rapid rate that traditional perimeter defenses cannot keep up with. For example, although initially considered the holy grail for protecting personal computers, antivirus software eventually proved unable to effectively keep pace with the rapid rate of malware change, leaving it helpless against targeted and sophisticated malware [53, 54, 55]. That led the senior vice president of information security at Symantec, the inventors of commercial antimalware software, to declare antivirus approaches "dead" [56].

The success of dependable and fault tolerant design methodologies has led researchers, government officials, and industry leaders to reexamine their cybersecurity approaches. Thus, the notion of resiliency has surfaced as a fundamental design principle in the fight against malicious cyber entities. At its essence, *resiliency* is a system's ability to detect failures and attacks, maintain acceptable and proper operation in the presence of those failures and attacks, and recover a normal state of operation in a timely and graceful fashion.

In conjunction with fault tolerance, researchers turned to the human immune system for inspiration and analogies to drive reasoning about the resiliency of networked systems against known and unknown threats. Fault tolerance and the human immune system present beneficial starting points for reasoning about the resiliency posture of networked systems. However, the presence of targeted and adaptive adversaries introduces several fundamental differences that require us to reconsider such analogies to avoid making erroneous decisions. System resiliency possesses several crucial characteristics that make it fundamentally different from fault tolerance or human immunology. Specifically, we consider the following properties.

Human engineering and the presence of software components. Modern computing systems are the result of the engineering and composition of numerous hardware and software components. The interactions among those components are rarely governed by predictable rules. The presence of software programs written in general-purpose, Turing-complete programming languages makes the process of modeling the different behaviors of networked

systems impractical and often computationally intractable. For example, it is feasible to predict when a magnetic hard disk drive will fail by studying the physical properties of the mechanical phenomena that govern its operations. On the other hand, buffer overflows are hard to predict since they result from design-time bugs that cause undefined behavior. Such behavior can then be exploited by an attacker to trick the memory manager and the instruction decoder into executing malicious instructions or reading confidential data blocks; the execution of such an exploit relies on the interactions among several software components that lead the CPU to execute malicious instructions. Therefore, modeling and prediction of buffer overflow exploits, let alone detecting them, does not enjoy the same stochastic predictability properties enjoyed by the hardware fault tolerance approaches.

Targeted and resourceful adversaries. In contrast to benign failures, adversaries have well-defined targets and may enjoy access to numerous resources to support their attacks. From leaked password databases to repositories of working exploits to unpatched systems, attackers can build a vast arsenal of attack methods that they can employ at relatively low cost. The past decade witnessed the emergence of *advanced persistent threats* (APT) that involve powerful, well-funded, and determined adversaries targeting critical infrastructure and government agencies (*e.g.* the Stuxnet worm, the Ukrainian power grid attack, and the Target and Equifax data breaches). It is much more challenging to predict the behavior of such a strategic and sophisticated attacker than to predict a benign fault, and it may be impossible.

Insider threats. In addition to external threats, resilient systems must consider the possible threats emanating from malicious internal actors, such disgruntled employees. Such actors are especially difficult to model and analyze when one is designing resilient systems. Insiders enjoy low-cost access to critical resources that can form the basis for launching severe and impactful attacks.

Rapidly adaptive adversaries. Although the human immune system has evolved robust mechanisms for combatting intruding organisms, such as microbes and viruses, it cannot fight them off entirely when they change their genetic makeup rapidly. Viral infections such as the flu and the common cold can still escape the human immune system's grasp. Similarly, intelligent adversaries can rapidly change their attack tactics in response to added security measures, sometimes even leveraging new security designs to uncover new flaws and gain more attack vectors. Security measures that do not take into consideration adaptive adversaries might increase the protected system's attack surface, ironically providing the adversaries with more weapons to add to their arsenal.

Those challenges make the process of designing resilient networked systems very difficult

and require increased cooperation between theoreticians and practitioners. The adaptive and intelligent nature of the adversaries poses several challenges for theoretical modeling and analysis; however, it does not render the process futile. By working closely with engineering teams, theoreticians can learn to make reasonable assumptions and perform worst-case analyses that allow them to make design decisions that can translate into practical implementations. Similarly, engineers and practitioners must carefully state and document their assumptions and provide feedback to the modeling teams to make design decisions that favor the system's resiliency.

In light of the challenges and requirements, the goal of this dissertation is to illustrate how the synergy between sound theory and realizable design can lead to improved network resiliency. We start by defining sound theory and realizable designs, and then present CPuzzle, Midgard, BiFrost, and sShield as evidence of the power of synergizing theory and practice.

2.3 DEFINITIONS

To formally define theoretical soundness and practical realizability, we draw on notions from the theory of formal systems [57] to reason about models and their implementations.

Definition 2.1 (System Model)**.** A *system model* M is a tuple $(V, S, R, L, \mathcal{T})$ such that

$$\begin{cases} V & \text{is a set of variables,} \\ S & \text{is a set of states,} \\ L : S \times V \times \mathcal{T} \to \mathcal{R} & \text{is a labeling function,} \\ R : S \times \mathcal{T} \times \{\epsilon, L\} \to S \times \mathtt{T} & \text{is a transition function, and,} \\ \mathcal{T} = \tau_0, \tau_1, \ldots & \text{is a monotonically increasing sequence of time values.} \end{cases}$$

V is the set of variables whose behavior the modeler is interested in capturing. To do so, the modeler defines a set of states S where each state $s \in S$ captures the values of the variables in V at that state. L is a labeling function that represents the relationships among the variables of the model at different states. For example, L can be a logical formula or a probability distribution over V. $\mathtt{T}$ represents a sequence of time instants, which can be either discrete or continuous, as defined in [58]. Finally, R is a transition function that defines the possible transitions between the states of the model. We do not place any restrictions on the types of the transitions, *i.e.*, they can be deterministic, stochastic, or nondeterministic (that last of which is represented by ϵ).

Definition 2.2 (Realization)**.** In a networking context, a *realization* or *implementation* of a model M is an interpretation, through software or hardware or both, of the model's semantics that assigns values to its set of variables V and defines semantics for S, L, and R.

Note that a realization D of a model M can define a set of variables $V' \subseteq V$ that subsumes M's variables. In that case, M is referred to as an *abstraction* of D, and R can be thought of as the projection of D's possible transitions onto V.

For example, RFC 793 [59] is a model of the Internet's Transmission Control Protocol (TCP), and different operating systems, such as Linux, Windows, and OpenBSD, provide various implementations or realizations of that RFC. In addition to defining the semantics of the RFC, those implementations include several other components, such as memory managers, thread managers, and network device drivers.

Definition 2.3 (Soundness)**.** Given a model M, a set of assumptions $\mathcal{A} : V \times S \rightarrow \{\texttt{true}, \texttt{false}\}$ and a set of inference rules Γ, the system (M, A, Γ) is said to be *sound* if and only if starting from $\mathcal{A}$, any valid formula $f : V \times S \rightarrow \{\texttt{true}, \texttt{false}\}$ proven by following Γ over M is valid for any realization D of M.

In other words, soundness ensures that any property or metric that can be obtained by reasoning about a model M over a set of assumptions $\mathcal{A}$ will be valid for any realization or implementation of the model.

Definition 2.4 (Practical Realizability)**.** Given a model M and a set of assumptions $\mathcal{A}$, a realization D of M is said to be *practically realizable* if D can be feasibly implemented and D does not violate any of the assumptions $a \in \mathcal{A}$.

Definitions 2.3 and 2.4 rule out any theoretical model that makes unreasonable assumptions about the real systems it must represent. At the same time, the combination of soundness and practical realizability forces engineers to make clear and explicit assumptions that modelers can take into consideration when conducting their analytic studies. As previously stated, our goal in this dissertation is to show that sound modeling and practically realizable design can lead to the enhancement of the resiliency posture of our networked systems when faced with both formidable adversaries and benign faults.

CHAPTER 3: CPUZZLE: CLIENT PUZZLES FOR STATE-EXHAUSTION DDOS ATTACKS RESILIENCY.

3.1 INTRODUCTION

In recent years, the scale and complexity of Distributed Denial of Service (DDoS) attacks have grown significantly. The introduction of DDoS-for-hire services has substantially decreased the cost of launching complex, multi-vectored attacks aimed at saturating the bandwidth as well as the state of a victim server [60, 61]. Common mitigations for large-scale DDoS attacks are focused around cloud-based protection-as-a-service providers, such as CloudFlare. When under attack, a victim's traffic is redirected to massively over-provisioned servers, where proprietary traffic-filtering techniques are applied and only traffic deemed benign is forwarded to the victim. The relative success of such over-provisioning techniques in absorbing *volumetric* attacks has pushed attackers to expand their arsenal of attacks to span multiple layers of the OSI network stack [62]. In fact, 39.8% of the attacks launched through the Mirai botnet were aimed at TCP state exhaustion, while 32.8% were volumetric [63]; the source code contained more than 10 vectors in its arsenal of attacks [64].

State exhaustion attacks are particularly challenging as they target *stateful* devices such as firewalls, load balancers, and application servers. Attackers can disguise them as benign traffic by leveraging a large number of machines that can use their authentic IP addresses [60], and can bypass cloud-based protection services [65, 66], capabilities, and filtering techniques [29, 30, 31, 32, 67, 68] by sending slower and non-spoofed traffic. This situation is further exacerbated by the imbalance between the cost of launching a multi-vectored DDoS attack and the cost of mitigating one. Launching an attack incurs an average cost of $66 per attack and can cause damage to the victim of around $500 per minute [69]. Furthermore, such attacks cannot be mitigated by employing geo-distributed *Content Delivery Networks* (CDN) as these CDNs still need to provide stateful services (such as TCP and HTTP) to the original victim's users; cloud protection services in fact recommend that defenses against state exhaustion attacks should reside at the victim's premises [65].

In this chapter, we revisit the application of client puzzles as a mechanism for resisting state exhaustion DDoS attacks. Client puzzles are a promising technique that alleviates the cost imbalance between the attacker and the defender with only software-level modifications at the end hosts and no changes to the Internet infrastructure [70, 71]. In essence, client puzzles attempt to hinder the malicious actors' ability to flood the server by forcing all clients, benign and malicious, to solve computational puzzles for each request they make.

While TCP client puzzles are a promising technique for resisting state exhaustion attacks,

they have not seen their way into adoption because of (1) the lack of guidelines on how to set the difficulty, and (2) the lack of publicly available implementations and performance studies [70, 72, 73]. A TCP client puzzle's difficulty determines the computational burden placed on the server and clients. Current standards [73, 74] suggest using a fixed difficulty level for all clients in order to maintain a stateless protocol, they however do not provide any sound ways for selecting the appropriate difficulty level. Difficulty selection becomes even more challenging when the victim serves clients with a mixture of power-endowed devices. Additionally, the few existing implementations of TCP client puzzles [34, 35, 72, 75] are outdated and are not publicly available, further hindering the community's ability to evaluate their effectiveness and adopt them.

In this chapter, we make the following contributions to address the shortcomings of TCP client puzzles research. First, we introduce a theory for determining an appropriate TCP puzzle difficulty based on the game-theoretic Stackelberg interaction between the defender and the clients [76, 77, 78] (Section 3.3 and Section 3.4). Using the theory we established, we provide a practical method for selecting the TCP puzzles difficulty based on the defender's capabilities and the expected computational prowess of the clients.

Then, we describe how we designed, implemented and evaluated an extension to TCP to support client puzzles using our practical difficulty-setting method. We incorporate puzzles into the TCP handshake and otherwise do not interfere with the operation of the protocol. We efficiently encode the challenges and their solutions into the *options* of the TCP header, resulting in low packet-size overhead to the protocol packets and no significant changes the TCP header. Then, we implement TCP puzzles as part of the Linux kernel TCP stack (Section 3.5). Our patch is publicly available at `https://github.com/nouredd2/puzzles-utils`.

We evaluated the performance of our TCP puzzles against a range of attacks through reproducible experiments performed using the DETER testbed (Section Section 3.6). Our results show the effectiveness of TCP puzzles in boosting tolerance against state exhaustion attacks. If a server using client puzzles with our game-theory-based difficulty setting method, it can tolerate both SYN and connection floods that would bring down an unprotected server or one that relies solely on SYN cookies [79]. Finally, we present a preliminary study of *puzzles-based queueing* (3.6.5) as a technique to provide fair treatment to a mixture of power-endowed devices.

3.2 BACKGROUND AND RELATED WORK

We first start by reviewing the TCP three-way handshake and TCP state exhaustion attacks. We then present client puzzles and their current limitations. For the remainder of

this chapter, we use the terms *puzzles* and *challenges* interchangeably.

3.2.1 TCP primer and SYN flood attacks

In current TCP implementations, a client initiates a TCP connection by sending a SYN packet to the server. Upon receiving the SYN packet, the server saves state for this new incoming connection request in a data structure, often referred to as the *Transmission Control Block* (TCB), and then sends a SYN-ACK packet and waits for the client to acknowledge receipt of this packet. A half-open connection is one for which the client's ACK packet has not yet been received; those new connection sockets are queued into a `listen` queue. The number of elements in this queue is upper-bounded by an implementation parameter, called the *backlog*, that bounds the server's memory usage to avoid exhaustion of the system's resources. Once a connection has been established, the server moves it into the `accept` queue. A socket is removed from the accept queue once the server's application accepts it for processing. On the other hand, a half-open connection socket is removed from the queue if it expires before it receives an acknowledgment from the client [59]. Once the server's listen or accept queue overflows, it either (1) no longer accepts incoming connections, or (2) drops old connection sockets from the appropriate queue.

TCP SYN flood attacks aim to overflow a victim server's `listen` queue by overwhelming it with half-open connection requests. The attack forces the server to drop new incoming connections, thus denying service to new clients [80]. A variant of the TCP SYN flood attack is a TCP connection flood in which an attacker attempts to overflow the server's `accept` queue for the same purpose of denying legitimate clients the opportunity to connect to the server. In a connection flood, the attacker completes the three-way handshake instead of leaving the connections half-open.

Among the server-based mitigations for SYN flood attacks, the SYN cache and TCP SYN cookies are the most common [79, 80, 81]. The SYN cache reduces the amount of memory needed to store state for a half-open connection by delaying the allocation of the full TCB state until the connection has been established. Servers that implement SYN caches instead maintain a hash table for half-open connections that contains partial state information and provides fast lookup and insertion functions. SYN cookies, on the other hand, operate by eliminating the source of the vulnerability in TCP implementations: the state reserved for half-open connections in the TCB. When SYN cookies are enabled, the server encodes a new TCP connection's parameters as a cookie in the packet's initial sequence number, and refrains from allocating state for a new connection until the cookie is again received from the client and validated.

The SYN cache aims to contain TCP SYN attacks by reducing the amount of state maintained on the server for half-open connections. Although efficient against a single attacker (or a small botnet), SYN caches do no provide protection against larger botnets for which the attack rate can easily exceed the space allocated for the cache. Once the cache is full, the server will default to the same behavior it performs when its backlog limit is reached, defeating the purpose of the cache. Although SYN cookies eliminate the key target of the SYN flood attack (the TCP backlog), they do not provide protection against large botnets. Attackers in control of a large number of zombie machines with valid (non-spoofed) IP addresses can, without added effort, overload the server's `listen` queue with valid TCP requests at a rate that surpasses the server's ability to accept them. Because they only tackle the problem only on the server end, SYN cookies are not a mechanism for stripping the malicious actors of their ability to conduct exhaustion attacks; further, it is not clear how SYN cookies can be generalized to serve as protection schemes for different types of state exhaustion attacks [72].

3.2.2 Client puzzles

Cryptographic client puzzles have been proposed to counter an asymmetry in today's Internet: clients can request substantial server resources at relatively little cost. Client puzzles alleviate this asymmetry by forcing clients to commit compute power as payment for requested resources.

Client puzzles have previously been proposed as a mechanism to combat junk mail [82], website metering [83], protecting the network IP and TCP channels [35, 71, 72, 84], protecting the TLS connection setup [34, 73], protecting key exchange [74], and protecting the capabilities-granting channel [30]. In addition, client puzzles are at the heart of the mining process of today's cryptocurrencies [85, 86]. Upon receiving a SYN packet, the server computes a puzzle challenge, sends it to the client, and does not commit any resources. After receiving the challenge, the client will employ its computational resources to solve the challenge and send the solution to the server. The server will then commit resources to the client only if the solution is correct.

Despite its promise, several challenges face the adoption of client puzzles as a practical defense measure against state exhaustion attacks. First, there is a shortage of implementations that allow for the comparison and the evaluation of different types of challenge creation and verification mechanisms. In this work, we implement clients puzzles in the Linux kernel and provide access to our implementation as a kernel patch.

Second, an important advantage of client puzzles is the ability to influence the clients by

setting an appropriate puzzle difficulty. However, there are no concrete and theoretically backed recommendations for selecting the appropriate difficulty, especially when faced with a mixture of power-endowed devices. Previous approaches [34, 71, 73, 74] suggest using a fixed, victim determined, puzzle difficulty for all clients in an effort to maintain a stateless protocol and to avoid creating a new state-exhaustion attack vector. These approaches do not however provide concrete methods to select the difficulty level in a manner that reflects the victim server's load and its clients' computational prowess. In our work, we present a game-theoretic formulation of the difficulty selection problem incorporating the server's provisioning as well as the computational profile of its clients.

Furthermore, a fixed puzzle difficulty might lead to fairness concerns as low-powered devices have to solve the same puzzles as higher-powered ones. RFC 8019 [74] suggests providing per-IP puzzles, however, it does not specify how the difficulties should be computed nor how to avoid increasing the attack surface. The work in [72] attempts to alleviate this problem by requiring clients to place bids on the server's resources by solving increasingly difficult puzzles. In addition to violating the TCP protocol by adding more round trips, this mechanism can be exploited to target clients since it moves the puzzle initiation process from the server to the client. In our thesis, we present a prototype *puzzle-based queuing* approach that rewards slow sending devices with higher access priority to the server's resources. It maintains negligible state and falls back to a fixed difficulty if that state is exhausted.

Laurie and Clayton [87] present an economic analysis to argue against the use of proof-of-work mechanisms to combat email spam. We agree with the authors that computational puzzles do not possess "magical" properties that make them practical in every situation. We however argue that state exhaustion attacks do not have the same nature as spam emails. First, unlike spam, state exhaustion attacks do not depend on the involvement of human users to click on malicious links. Second, DDoS attacks have a lower cost barrier as they are launched from compromised botnet machines and not from specialized attacker hardware. We believe that our theoretical and experimental results showcase the merits of proof-of-work mechanisms in tolerating SYN and connection floods. In fact, our work complements the security analysis performed in [88, 89, 90] with the required protocol engineering and design, allowing for an improved understanding of client puzzles.

3.3 THE GAME-THEORETIC MODEL

In this section, we introduce our game-theoretic model for computing the puzzle difficulties that balance the clients' computational load as well as the server's provisioning. We first present our threat model and assumptions and then discuss our game-theoretic model.

3.3.1 Assumptions and threat model

In our work , we make the following assumptions.

Assumption 1. Common state exhaustion attacks, specifically TCP connection floods as well as higher-layer attacks, require the presence of a two-way communication channel between the attacker bots and the victim server. That is evident from the nature of the state exhaustion attacks as well as their ability to circumvent scrubbing and filtering techniques by sending lower volumes of traffic [60]. The implication is that during a single-vector state exhaustion attack, the victim server is able to receive packets from, and send packets to, its legitimate users as well as the attackers' machines. In the presence of a multi-vectored attack, we assume the presence of volumetric attack mitigation techniques (such as cloud-based protection-as-a-service); client puzzles will complement those techniques to provide DDoS defenses against hybrid attacks.

Assumption 2. We assume that the attackers can control a large number of zombie machines that form botnets to coordinate large-scale attacks aimed at depleting a target server's resources. However, we assume that the attacker's army of bots comprises commodity machines (e.g., workstations, mobile phones, and IoT devices) but not clusters of servers with large computing resources. Such clusters are part of enterprise solutions and therefore employ better protective mechanisms than commodity machines, so are harder to compromise. We further assume that the attackers can capture and replay packets, but are not able to change their content; protection against integrity attacks is beyond the scope of our thesis.

The above assumptions are similar to the ones made in [71] and [72]. Moreover, client puzzles do not require the end-server to differentiate between malicious and benign traffic. In fact, the low-volume nature of state exhaustion attacks and the requirement for quick and effective protective mechanisms can impede the accuracy of anomaly detection mechanisms.

3.3.2 Difficulty Selection as a Stackelberg game

We formalize the problem of selecting the puzzle difficulty similar to a network pricing problem [76, 77, 78]. We model the problem as a Stackelberg game between the service provider and the service users. The service provider is the leader who sets the difficulty of the puzzles that the clients must solve to receive service. The users are the followers who then choose their request rates to optimize their local utility.

Our model rests on the assumption that all clients are selfish agents seeking to optimize their local utilities; we do not specifically posit a model for malicious bots. This assumption is rooted in the following observations. First, before the attack starts, the server does not

have the means to distinguish between benign clients and malicious bots. Second, TCP by default treats every connection request it receives as a benign request, and thus sends an ACK packet back without checking whether the request came from a benign user or a compromised bot. Third, positing a specific attacker model would require the estimation of attacker preferences and utilities, which the server has no means of measuring. This could create a schism between the model and its application in the real world. We therefore treat every request as if it is benign, and capture the presence of a large botnet by obtaining the asymptotic solution for our model.

Let x_i be user i's request rate, for $i \in \{1, 2, \ldots, N\}$, where N is the total number of users in the system. Consequently, $x_{-i} = \sum_{j \neq i}^{N} x_j$ is the total request rate of all the other users. Our model captures the puzzle's difficulty by using the expected number of hash operations needed to find and verify its solution. Let p_i be user i's puzzle; $\ell(p_i)$ is then the expected number of hash operations that user i has to perform to find a solution to p_i. Let $S(\bar{x} = \sum_i x_i)$ be the expected service time for a user's request. User i's utility can be written as

$$u_i\left(x_i, x_{-i}, p_i\right) = w_i \log(1 + x_i) - \ell\left(p_i\right) x_i - S(\bar{x}) \tag{3.1}$$

w_i is a user-specific parameter that models the user's valuation of the provider's service. In other words, w_i represents the amount of work user i is willing to pay per request. $\log\left(1 + x_i\right)$ represents the user's expected benefit when making decisions under risk or uncertainty [76, 91]. The utility function can be interpreted as the difference between the user's expected benefit and the amount of work she has to expend to solve a puzzle per request added to the expected service delay she incurs. Each user, being a rational and selfish agent, will choose a request rate that optimizes her local utility. That will lead to the users adopting the Nash Equilibrium (or simply, equilibrium) rates x_i^* for $i \in \{1, 2, \ldots, N\}$ such that

$$u_i\left(x_i^*, x_{-i}^*, p_i\right) \geq u_i\left(x_i, x_{-i}^*, p_i\right), \ \forall x_i > 0, \forall i \tag{3.2}$$

The service provider's problem is to find a puzzle difficulty such that (1) it can effectively reduce the impact of state exhaustion attacks and (2) minimize the amount of work the server does to generate and verify puzzles. Let $\mathcal{P}$ be the space of all possible cryptographic puzzles and $g(p_i)$ and $d(p_i)$ be the expected numbers of hash operations that the provider needs to perform to generate and verify a solution to puzzle p_i, respectively. We model the provider's problem as finding the puzzles $\mathbf{p}^* = \{p_i^* \in \mathcal{P},\ i \in \{1, 2, \ldots, N\}\}$ such that

$$\mathbf{p}^* = \underset{\mathbf{p} \in \mathcal{P}^N}{\arg\max} \sum_{i=1}^{N} \left(\ell(p_i) - \left(g(p_i) + d(p_i)\right)\right) x_i^* \tag{3.3}$$

Equation (3.3) captures the provider's goal of maximizing the amount of work that the clients have to perform to obtain service under attack while minimizing the amount of work it must perform to generate puzzles and verify solutions. This formulation, in fact, captures the trade-off between the puzzle's complexity and the expected work that the provider needs to perform to generate and verify puzzles. The tuple

$$(\mathbf{x}^* :=< x_1^*, x_2^*, \ldots, x_N^* >, \mathbf{p}^* :=< p_1^*, p_2^*, \ldots, p_N^* >) \tag{3.4}$$

represents the solution to the full Stackelberg game.

3.4 APPLICATION TO THE JUELS PUZZLE SCHEME

We now show how the framework we introduced in Section 3.3 can be applied to the puzzles protocol presented in [71]. We first describe the puzzles protocol from [71] and then show the solutions we obtain using our framework. For our modeling and analysis, we assume that the server issues puzzles with the same difficulty for all of its clients, i.e., $\ell(p_i) = \ell(p_j)\ \forall i, j \in \{1, 2, \ldots, N\}$. This assumption ensures a stateless protocol, follows the IETF TLS puzzles draft [73], and is recommended in previous work [71].

A puzzle in this scheme is a bitstring of length l bits having $m < l$ bits of difficulty. The puzzle-issuing server starts by creating the hash $y = h\,(s, T, \text{packet-level data})$, where s is a secret key; T is a timestamp; the packet-level data are a concatenation of the source and destination IP addresses and ports; and h is a collision-resistant hash function. The server challenges a client to provide k solutions to a puzzle P formed by the first l bits of y.

Upon receiving P, the client computes, by brute force, k solutions $\{s_1, \ldots, s_k\}$ such that for $1 \leq i \leq k$, $|s_i| = l$ and the first m bits of $h(P, i, s_i)$ match the first m bits of P, where h is the same hash function that the server used. The client then sends the solutions back to the server, which in turn, verifies their validity and subsequently accepts the request.

The solution

Since obtaining a single solution of length m bits is best done by brute force, it requires a maximum of 2^m and an average of 2^{m-1} hashing operations. Since each puzzle requires k solutions, solving a puzzle then requires an average of $k \times 2^{m-1}$ hashing operations. Therefore, for each user $i \in \{1, 2, \ldots, N\}$, $\ell(p_i) = k \times 2^{m-1}$.

To capture the expected service time for the users, called $S(\bar{x})$, we abstract the server's operation with an $M/M/1$ queue with a service rate μ. We argue that this abstraction is

enough for our purpose, since the attacks in which we are interested target the TCP stack and are independent of the application that the server is running; they are affected only by the application's ability to remove established connections from the accept queue. The service rate μ can be obtained by running stress tests on the application provider's infrastructure and can capture different service optimizations such as replications and caching. Subsequently, we express the expected service time as $S(\bar{x}) = \frac{1}{\mu - \bar{x}}$, when $\bar{x} < \mu$. This condition assumes that the server is well-provisioned to handle the users' load under regular conditions. We therefore rewrite Equation (3.1) as

$$u_i\left(x_i, x_{-i}, p_i\right) = w_i \log(1 + x_i) - k \times 2^{m-1} x_i - \frac{1}{\mu - \bar{x}} \tag{3.5}$$

We now turn to the provider's formulation. We represent the space of all possible puzzles as the set of tuples (k, m) where $k \in \mathbb{N}$ is the number of solutions requested and $m \in \mathbb{N}$ is the number of bits of difficulty in each. Therefore we write $\mathcal{P} = \{(k, m), k, m \in \mathbb{N}\}$. As previously discussed, every challenge can be generated using only one hash operation; therefore, we write $g(p_i) = 1, \ \forall i$.

When the server receives a solution, it generates a hash from the received packet's header and then verifies each of the k solutions until it finds a violating one or deems the puzzle correctly solved. If the server chooses which of the k solutions to verify uniformly at random, it then needs an average of $\frac{k}{2}$ hashing operations. Therefore, we can write $d(p_i) = 1 + \frac{k}{2}, \ \forall i$.

Since we assume that the service provider issues puzzles with the same difficulty for all users, we henceforth write $p = (k, m) = p_i, \forall i$. We can then rewrite Equation (3.3) as

$$p^* = \arg\max_{p \in \mathcal{P}} \sum_{i=1}^{N} \left(k \times 2^{m-1} - 2 - \frac{k}{2} \right) x_i^*(p) \tag{3.6}$$

Let w_{av} be the average client valuation of the server's service and α be the server's asymptotic service rate per user, under normal operation.

Theorem 3.1. *The Nash equilibrium is achieved at $p^* = (k^*, m^*)$ such that:*

$$\ell(p^*) = k^* \times 2^{m^* - 1} = \frac{w_{av}}{(\alpha + 1)} \tag{3.7}$$

Proof. In order to analytically solve for the equilibrium solution of the game, we follow an approach similar to that in [76]. We start by noting that the Nash Equilibrium solution of the users' game is not affected if we add the quantity

$$\sum_{j \neq i} \left(w_j \log(1 + x_j) - k \times 2^{m-1} x_j \right) \tag{3.8}$$

to each users' utility function. Therefore we can now build a strategically equivalent game where each user's utility function is

$$H\left(x_1, \ldots, x_N, p\right) = \sum_{i=1}^{N} w_i \log(1 + x_i) - k \times 2^{m-1} \bar{x} - \frac{1}{\mu - \bar{x}} \tag{3.9}$$

Now looking at the Hessian matrix of H we get

$$\begin{aligned} \mathbf{H}_{ii} = \frac{\partial^2 H}{\partial x_i^2} = -\frac{w_i}{(1 + x_i)^2} - \frac{2}{(\mu - \bar{x})^2} < 0, \quad \forall i \\ \mathbf{H}_{ij} = \frac{\partial^2 H}{\partial x_i x_j} = -\frac{2}{(\mu - \bar{x})^3} < 0, \quad \forall i, j, \ i \neq j \end{aligned} \tag{3.10}$$

Therefore $\mathbf{H}$ is negative-definite and thus H is strictly concave for $0 \leq \bar{x} < \mu$. Additionally, since $H \to -\infty$ as $\bar{x} \to \mu$, we can conclude the that optimization problem

$$\max_{x_i \geq 0 : \forall i, \bar{x} < \mu} H\left(x_1, x_2, \ldots, x_N, p\right) \tag{3.11}$$

admits a unique solution $\mathbf{x}^* = \{x_1^*, \ldots, x_N^*\}$ in the interval $0 \leq \bar{x} < \mu$ which corresponds to the Nash Equilibrium strategies to the users' game as defined in Equation (3.5). We obtain the solution strategies by solving the first order condition of H where for $i \in \{1, \ldots, N\}$

$$\frac{\partial H}{\partial x_i}\left(x_1^*, \ldots, x_N^*, p\right) = 0 \tag{3.12}$$

which translates to

$$\frac{w_i}{1 + x_i^*} - k \times 2^{m-1} - \frac{1}{\mu - \bar{x}^*} = 0, \ \forall i \tag{3.13}$$

Let $y_i = 1 + x_i$, $\bar{y} = \sum_{i=1}^{N} y_i = N + \bar{x}$, and $\bar{w} = \sum_{i=1}^{N} w_j$, from which we obtain

$$\frac{w_i}{y_i} = \frac{w_j}{y_j}, \quad \forall i, j \in \{1, \ldots, N\} \tag{3.14}$$

or equivalently

$$y_i = \frac{w_i}{w_j} y_j, \quad \forall i, j \in \{1, \ldots, N\} \tag{3.15}$$

We can then rewrite $\bar{y}$ as

$$\bar{y} = \sum_{i=1}^{N} y_i = \sum_{i=1}^{N} \frac{w_i}{w_j} y_j = \frac{\bar{w}}{w_j} y_j \tag{3.16}$$

and thus we can express (3.13) in terms of $\bar{y}$ as

$$\tilde{L}(\bar{y}) = \frac{\bar{w}}{\bar{y}} - k \times 2^{m-1} - \frac{1}{(\mu + N - \bar{y})^2} = 0 \tag{3.17}$$

We can thus turn our attention to solving Equation (3.17) for $N \leq \bar{y} < \mu + N$. Since $\frac{\partial \tilde{L}}{\partial \bar{y}} = -\frac{\bar{w}}{\bar{y}^2} - \frac{2}{(\mu+N-\bar{y})^2} < 0$, $\tilde{L}$ is strictly decreasing. Additionally, $\tilde{L}(\bar{y}) \to -\infty$ as $\bar{y} \to \mu + N$. We therefore need $\tilde{L}(N)$ to be non-negative so that $\tilde{L}(\bar{y})$ would admit a solution in the interval $N \leq \bar{y} < \mu + N$, which translates to

$$\tilde{L}(N) = \frac{\bar{w}}{N} - k \times 2^{m-1} - \frac{1}{\mu^2} > 0 \tag{3.18}$$

or equivalently

$$k \times 2^{m-1} < \frac{\bar{w}}{N} - \frac{1}{\mu^2} := \hat{r} \tag{3.19}$$

We can see $\hat{r}$ as the maximum possible difficulty that the service provider can select while guaranteeing that a solution for the clients' game exists. We also notice that if the provider had infinite resource, i.e., $\mu \to \infty$, $\hat{r} \to \frac{\bar{w}}{N}$ which suggests that a client should not be charged a price higher than the average user valuation of the provider's services.

Furthermore, it is beneficial for the service provider to ensure that all clients participate in the game, i.e., that $x_i > 0$ for all $i \in \{1, 2, \ldots, N\}$. This therefore translates to the conditions on $\bar{y}$

$$\bar{y} > \frac{\bar{w}}{w_i} \quad \forall i \tag{3.20}$$

Now let $\bar{y}(k, m)$ be a solution to Equation (3.17) that satisfies condition (3.20) and where (k, m) satisfy condition (3.19), and let $\bar{x}(k, m)$ be the corresponding value of $\bar{x}$. We turn to the provider's problem of finding the optimal pricing $p^* = (k^*, m^*)$ that maximizes

$$I(p) := \left(k \times 2^{m-1} - 2 - \frac{k}{2} \right) \bar{x}(k, m) \tag{3.21}$$

In order to obtain an analytical solution to the optimization problem in Equation (3.21) we make use of the following approximation. We solve for the pricing $\tilde{p}(\tilde{k}, \tilde{m})$ that maximizes

$$\tilde{I}(p) := \left(k \times 2^{m-1} \right) \bar{x}(k, m) \tag{3.22}$$

Lemma 3.1. *$|I(p^*) - \tilde{I}(\tilde{p})| < c$ for some constant $c > 0$, where p^* and $\tilde{p}$ are the solutions that maximize I and $\tilde{I}$, respectively.*

Proof. Let $p^* = (k^*, m^*)$ and $\tilde{p} = (\tilde{k}, \tilde{m})$ be the prices that maximize $I(p)$ and $\tilde{I}(p)$, respectively. We therefore have that

$$\tilde{k} \times 2^{\tilde{m}-1}\bar{x}(\tilde{k}, \tilde{m}) \geq k \times 2^{m-1}\bar{x}(k, m),\ \forall k, m \tag{3.23}$$

Let $p' = (k', m')$ be a price with minimum $0 < k' \leq \tilde{k}$ such that $k' \times 2^{m'-1} = \tilde{k} \times 2^{\tilde{m}-1}$ and $I(k', m') \geq I(\tilde{k}, \tilde{m})$. We can therefore write

$$I(p') \geq I(k, m) - (\frac{k'}{2} + 2)\bar{x}(k', m'),\ \forall k, m \tag{3.24}$$

and since $I(p^*) \geq I(p)\ \forall p$ we can therefore conclude that

$$|I(p^*) - I(p')| \leq (\frac{k'}{2} + 2)\bar{x}(k', m') < \left((\frac{k'}{2} + 2)\mu\right) := c \tag{3.25}$$

and since $\bar{x}$ only depends on $k \times 2^{m-1}$ and not on the individual values of k and m, p' also maximizes Equation (3.22) and thus solving for p' brings us within a constant c of p^*, the maximum of I. QED.

We can now proceed with finding a solution for Equation (3.22) following the approach presented in [76]. By using the one-to-one correspondance between $k \times 2^{m-1}$ and $\bar{y}$ (and thus $\bar{x}$) presented in Equation (3.17), we can substitute $\bar{y}$ in (3.22) and then compute p^* using the solution to the obtained equation. We then write the equivalent problem as finding $\bar{y}^*$ such that

$$\bar{y}^* = \underset{N<\bar{y}<N+\mu}{\arg\max} \left(\frac{\bar{w}}{\bar{y}} - \frac{1}{(\mu + N - \bar{y})^2}\right)(\bar{y} - N) \tag{3.26}$$

We define $G(\bar{y}) := \left(\frac{\bar{w}}{\bar{y}} - \frac{1}{(\mu+N-\bar{y})^2}\right)(\bar{y} - N)$. It is easy to see that $\frac{\partial^2 G}{\partial \bar{y}^2} < 0$ and thus $G(\bar{y})$ is strictly concave. Additionally, $G(N + \mu) \to -\infty$, we can thus conclude that $G(\bar{y})$ admits a unique maximum in the open interval $(N, N+\mu)$. We can then solve the first order condition

$$\frac{\partial G(\bar{y})}{\partial \bar{y}} := \frac{\bar{w}N}{\bar{y}^2} - \frac{\mu + \bar{y} - N}{(\mu + N - \bar{y})^3} = 0 \tag{3.27}$$

Obtaining a closed form solution for $\bar{y}^*$ is not possible for finite N. Therefore we solve Equation (3.27) asymptotically (i.e., as $N \to \infty$) as proposed in [76]. For that, we make the following assumptions. (1) We assume that the average user preference $w_{av}(N) = \frac{\bar{w}}{N}$ has a

well defined limit w_{av} as $N \to \infty$. (2) We assume that as the number of users grows larger, the service provider can always service a fraction of its users, even if that fraction is small. In other words, we assume that $\lim_{N\to\infty} \frac{\mu}{N} = \alpha$ for some $\alpha > 0$. For convenience, we rewrite Equation (3.27) in terms of $x_{av}(N) = \frac{\bar{x}}{N}$ and $w_{av}(N)$ as $N \to \infty$ as

$$\frac{w_{av}}{(1 + x_{av}(N))^2} = \frac{\alpha + x_{av}(N)}{(\alpha - x_{av}(N))^3 N^2} \tag{3.28}$$

Equation (3.28) possesses a solution for $x_{av}(N)$ iff,

$$\lim_{N\to\infty} (\alpha - x_{av}(N))^3 N^2 = \gamma \tag{3.29}$$

for some $\gamma > 0$. We thus substitute back in Equation (3.17) and obtain the solution

$$k^* \times 2^{m^*-1} \sim \frac{w_{av}}{\alpha + 1} + \frac{2\alpha - 1}{\gamma^{\frac{2}{3}} N^{\frac{2}{3}}} \tag{3.30}$$

where $f \sim g$ denotes the fact that $\lim_{N\to\infty} \frac{f}{g} = 1$.

Since we are considering the asymptotic solution, we restrict our attention to the first order term of the solution in Equation (3.30) and thus obtain our desired form

$$k^* \times 2^{m^*-1} = \frac{w_{av}}{\alpha + 1} \tag{3.31}$$

In fact, as shown in [76], Equation (3.31) corresponds to the solution of the same problem when ignoring the service delay at the server. Since SYN and connection flood attacks target the TCP protocol and not the application layer service, it is convenient for the purposes of this chapter to only consider the first order term of Equation (3.30), thus completing the proof. QED.

Analysis

The equilibrium difficulty we obtained in Theorem 3.1 illustrates an important design tradeoff between the server's provisioning and the difficulty of the puzzles that the clients should solve when the server is under attack. A well-provisioned server, i.e., one for which $\alpha > 1$, will be able to absorb a larger fraction of the attack and subsequently ask its clients to solve less complex challenges. In that case, the clients help the server tolerate the attack and commit fewer resources than they are willing to — the average number of hashes they would need to perform to solve a challenge is less than w_{av} — so the client achieves high

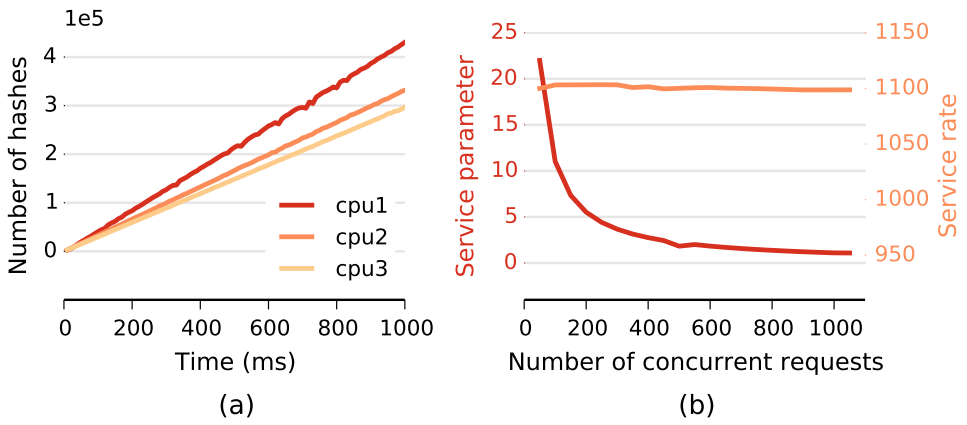


Figure 3.1: Profiles of (a) client (w_{av}) and (b) server (α).

utility. On the contrary, a server that is not able to handle all of its clients' regular load, i.e., one for which $\alpha < 1$, would require its clients to solve harder puzzles ($p^* \simeq w_{av}$) and thus achieve lower utility levels. Therefore, to tolerate an attack, the server asks its clients to commit more resources, risking the dropout of more clients as the intensity of the attack increases. Those clients with $w_i < w_{av}$ would consider it best for them to drop out, since it would be too costly as a function of the resources committed to obtain a connection.

We further note that our model and solution are agnostic to the application that is run by the server. That, in fact, is consistent with TCP being a transport-layer protocol that is independent of the type of application running on top of it. All our model requires is an estimate of the server's capacity to handle large loads (i.e., the parameter α), which can be obtained by running appropriate stress tests. Server replication and load balancing are then captured in our model through an increase in the value of α (given the same load).

Finally, we note that our result is not affected by the presence of long-lived TCP connections (for example, if HTTP/1.1 [92] is being used). The puzzles protect the TCP connection establishment channel and allow users to connect to the server in the presence of malicious attacks. The lifetime of the established connection is not affected by the presence or absence of puzzles; in the case of HTTP/1.1, the goal of the challenges is to allow clients to establish the TCP connection upon which the HTTP session persists.

Obtaining model parameters

The model parameters, w_{av} and α, relate to the performance capabilities of the server and the clients. We first describe an experimental procedure for obtaining the model parameters. Then we discuss how we applied the procedure to an experimental setup to demonstrate the

Nash strategy.

First, w_{av} is the number of hashes we assume the client is willing to perform to complete the TCP handshake. It represents the level of acceptable service degradation as each TCP connection will take longer to finish. To find w_{av}, we assume that 400 ms is adequate time to establish a TCP three-way handshake for a legitimate client when the server is under attack. Usability studies show that a 400 ms delay does not interrupt the user's flow of thoughts [93]. Using that assumption, we find the number of hashes a machine can perform in 400 ms by profiling the machines. w_{av} is the average value obtained during the experiments.

Second, α is the service parameter of the server. It is directly related to the processing power of the server. To obtain the parameter, we start by stress testing a server. The stress test varies the rate of requests per second and records the time it takes to get service for each rate. We compute α as the ratio of the service rate over the number of concurrent requests.

Finally, after obtaining w_{av} and α, we calculate the equilibrium difficulty parameters (k^*, m^*) by using Equation (3.7). The choice of those parameters exposes a trade-off between the number of hashes the server needs to verify a solution and the probability that an adversary can guess a solution. Choosing a very small k will increase the attacker's ability to guess a solution, and selecting a large k will increase the solution verification time. On the other hand, if lower values of k are selected, the challenge difficulty m would increase allowing the server to offset its lack of computational resources by asking its clients to solve harder challenges.

Example

In the following, we present an example of computing the Nash equilibrium difficulty for a server serving a variety of machines with varying processing powers. Starting with the client, we obtain w_{av} by profiling the number of SHA-256 operations per second. Figure 3.1(a) shows the profile of three CPU types: (1) `cpu1` is an Intel Xeon E3-1260L quad-core processor running at 2.4 GHz, (2) `cpu2` is an Intel Xeon X3210 quad-core processor running at 2.13 GHz, and (3) `cpu3` is an Intel Xeon processor running at 3GHz.

The average number of hashes that can be performed over the three types of CPUs is $w_{av} = 140630$. Although the CPUs we profiled are not an exhaustive representative set of the processing powers of a typical clientele, in all of our experiments, we leveled the playing field by providing all the attackers with similar or better computational powers.

Then we estimate the server's α parameter. We deployed an Apache2 web server on a dual Intel Xeon hexa-core processor running at 2.2 GHz with 24 GB of RAM. We then used the Apache benchmarking tool `ab` [94] to profile the performance of the server under regular

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31			
Opcode 0xfc	Length	k	m
ℓ	Preimage $\cdots$		
$\cdots$ preimage	Padding (NOP)		

Figure 3.2: TCP Options block for a SYN challenge.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31		
Opcode 0xfd	Length	MSS value
Wscale	Solution $\cdots$	
$\cdots$ solution	Padding (NOP)	

Figure 3.3: TCP Options block for a SYN solution with $k = 1$.

and high loads. Figure 3.1(b) shows the service rate μ and the service parameter α of our server as the number of concurrent requests attempted by `ab` increases. Our server was able to maintain a constant service rate under high load ($\mu \simeq 1100$ requests/s), and thus the parameter α converged to a value of 1.1 as the load increased. Thus for our example, with $w_{av} = 140630$ and $\alpha = 1.1$, the TCP puzzle difficulty is set at $(k^* = 2, m^* = 17)$.

3.5 IMPLEMENTATION

We implemented the TCP challenges in the TCP stack of the Linux 4.13.0 kernel. The puzzles are turned off by default and are only enabled when the socket's queue is full. We designed our implementation in such a way that the challenges take precedence over the SYN cookies once the queue is full; we do, however, support SYN cookies as a backup option. We provided support for dynamic tuning of the parameters of the challenges (k, m) through the kernel's `sysctl` interface.

We generate the challenge's pre-image by hashing a string containing (1) a server's secret key, generated once at the start of a socket's lifetime, (2) the server's current timestamp, (3) the SYN packet's source and destination IP addresses, and (4) the packet's source and destination port numbers. We used the Linux kernel's SHA256 hashing function, since it provides the necessary pre-image resistance guarantees [71].

To avoid breaking the TCP definition, we inject the challenges and solution into the options field of the TCP SYN-ACK and ACK packets. Figure 3.2 shows the format of the TCP option we implemented to transmit a challenge in the SYN-ACK packet. We chose an unused opcode (0xfc) to represent a challenge option. The `Length` field indicates the length of each option block in bytes, including the opcode and the field itself. We allocate one byte

each for the number of solutions k, the difficulty of the puzzle m (in bits), and the pre-image and solution length l. Next, we insert the challenge's pre-image. Finally, following the TCP stack requirement, each option block must be 32 bits aligned, we, therefore, insert 0 to 3 NOP fields to ensure alignment.

Figure 3.3 shows the format of the TCP option used by a client to send a solution. Much as in the challenge option, we made use of unallocated opcode (0xfd). Since the server keeps no state about the client after receiving the first SYN packet, the client safely assumes that the server has ignored its previously announced *Maximum Segment Size* (MSS) and *Window Scaling* (Wscale) parameters. We then resend the MSS and Wscale values within the solution, write down each of the k solutions, and perform alignment to 32 bits.

The benefits of adding the MSS and Wscale parameters to the solution option block are twofold. First, it means that the challenge protocol will be self-contained; implementation of the TCP stack usually ignores all options other than timestamps in any packet other than the SYN and SYN-ACK packets. Therefore, support for the challenge protocol does not require changes to legacy options parsing. The addition also provides us with the benefit of reducing the space needed to resend the options in the ACK packet. For example, sending the MSS values as a separate option would require 4 bytes, while we need only 2 in the case of the self-contained solution option. Second, we encode the MSS value by using 16 bits (as defined in the specification of TCP), instead of the 3 bits provided by SYN cookies. In addition, when SYN cookies are in place, the client and the server cannot agree on the window parameters, which reduces the performance of the TCP connection.

Also, modern TCP implementations support the exchange of timestamps as options in the TCP header. Our implementation makes use of the timestamps, whenever available, to generate, solve, and verify challenges. However, if the timestamp option is not enabled (for example, it was disabled by the client or the server), our implementation embeds the timestamp used in the generation of the challenge (an additional 4 bytes) in both the challenge and the solution packets.

Furthermore, when the server's accept queue overflows, its default behavior is to reject new connections, even if the protection mechanism is in place. However, for our purposes, since the goal of the puzzle protection mechanism is to throttle the rate of all clients (both benign and malicious), we modified the listening TCP socket's implementation to send a challenge when the protection is in effect, even if the accept queue overflows. When the server receives an ACK packet while under attack, it first checks whether the queue is full and performs the verification procedure only if there is room to accept the connection. If the queue is full, the server will ignore the ACK packet. In such a case, the user (whether benign or malicious) assumes that the connection has been established and will begin sending

application-level packets thus causing the server to reply with a reset (RST) packet to signal that the connection was not established. This implementation choice achieves the goal of deceiving the malicious users into thinking that they have established a connection when they have not; the malicious agents that do not send application-level packets will not receive a RST packet to indicate that the server has dropped the connection.

Finally, to combat replay attacks, we make use of the timestamp in the solution to check whether a challenge has expired. This stateless mechanism hinders an attacker's ability to replay solution packets, since tampering with the timestamp will cause the solution verification to fail. The timeout interval can be tuned through the kernel's `sysctl` interface.

3.6 EVALUATION

Using our modified Linux kernel, we evaluated the performance of the TCP puzzles in safeguarding a server TCP connection establishment channel from state exhaustion attacks.

We performed the experiments using the DETER [95] cybersecurity testbed. In the spirit of moving towards a "science of security" through reproducible experiments [20], we provide all of our experiment scripts and datasets online at `https://github.com/nouredd2/puzzles-utils`.

The goal of our experiments is to evaluate (1) the effectiveness of TCP puzzles in protecting against state exhaustion attacks, (2) the impact of TCP puzzles on service quality, and (3) the ability of the Nash equilibrium puzzle difficulty to balance the client solution and the server verification load as well as its ability to effectively rate-limit attackers. Our victim server runs an HTTP application that accepts "`gettext/size`" requests and returns messages that contain `size` random bytes. The clients run an HTTP client that requests text from the server at a prespecified rate.

In a real-world deployment, service would be provided by a farm of servers, but our scenario uses only one server and a smaller set of clients. In larger systems, since a load balancer forwards TCP connection requests to individual servers, an attack has to ensure that its wave of requests reaches all of the servers to effectively deny service. Therefore, adding more servers allows a service provider to tolerate bigger attacks by large botnets. Our results show that a server using TCP puzzles as a means for state exhaustion DDoS protection can tolerate a larger botnet than an unprotected server can. We hence argue that when all the servers in a farm employ our protection, the system will be able to tolerate a larger botnet that is proportional to the improved tolerance of a single server. Our experiment scenario thus studies the protection offered to a single server; the results are to scale when more load-balanced puzzles-equipped servers are deployed.

We consider two types of attackers. The first uses randomized source IP addresses to target the server's `listen` queue with a flood of half-open TCP connections (using `hping3`). The second type uses real IP addresses to flood the server with established connections (using `nping`) in an attempt to fill its `accept` queue and prevent new, legitimate connections from being established. Unless otherwise stated, we use the following experiment parameters. The set of clients contains 15 machines requesting $10,000$ bytes of data at exponentially distributed time intervals, with rate $r_c = 20$ requests per second. The botnet consists of 10 machines running an attack at a constant rate $r_a = 500$ requests per second, amounting to an overall attack rate of $5,000$ packets per second (pps). All of the malicious machines are equipped with a computational power equal to, or greater than, that of the clients' machines. Except in Experiment 4, all of the machines in our setup were equipped with our modified kernel.

Finally, except for Experiment 5, all the experiments used the same network topology with well-provisioned link bandwidths so as to avoid link saturation. The backbone consisted of three routers fully connected with 1 Gbps links. The server connected to the network with a 1 Gbps link, while all the other hosts connected to the network with 100 Mbps links. All of our agents ran on physical machines with Ubuntu 16.04 LTS along with our patched Linux 4.13.0 kernel. We deployed the packet-monitoring software, `tcpdump`, on all of the machines, and used the captures to measure the throughput at the server, the throughput at each host, and the number of dropped TCP connections. We report here on the throughput since it represents a direct assessment of the impact of puzzles on our application; nevertheless, we acknowledge that different applications will require different metrics.

3.6.1 Experiment 1: SYN and connection flood protection

In the first scenario, we started a distributed SYN flood attack. Without protection, the SYN flood filled the `listen` queue with half-open TCP connections, leading the server to drop new incoming connections. We measured the throughput at a client and the server for three settings: (1) no protection (control settings), (2) TCP SYN cookies, and (3) TCP client puzzles. Figure 3.4 shows the throughput measured during the experiment. The attack duration, shown by the shaded region, was initiated at $t = 120$ and concluded at $t = 480$. The throughput's behavior for both the server and client was consistent; we therefore restrict our analysis to the server's case. For the control setting, the server's throughput dropped to zero as soon as the attack started and returned to full capacity 30 seconds after the attack ended. On the other hand, SYN cookies were effective at rendering the attack ineffective and ensuring a constant throughput at the server throughout the attack. By storing partial

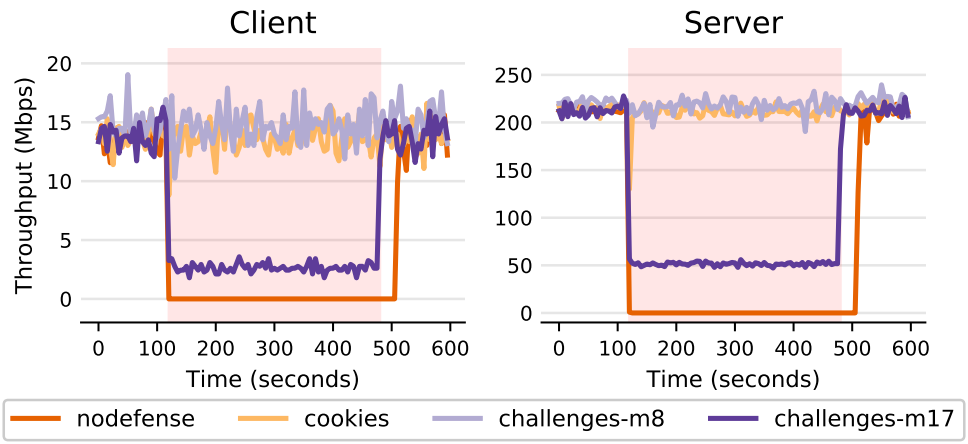


Figure 3.4: Throughput at a client and server during SYN flood.

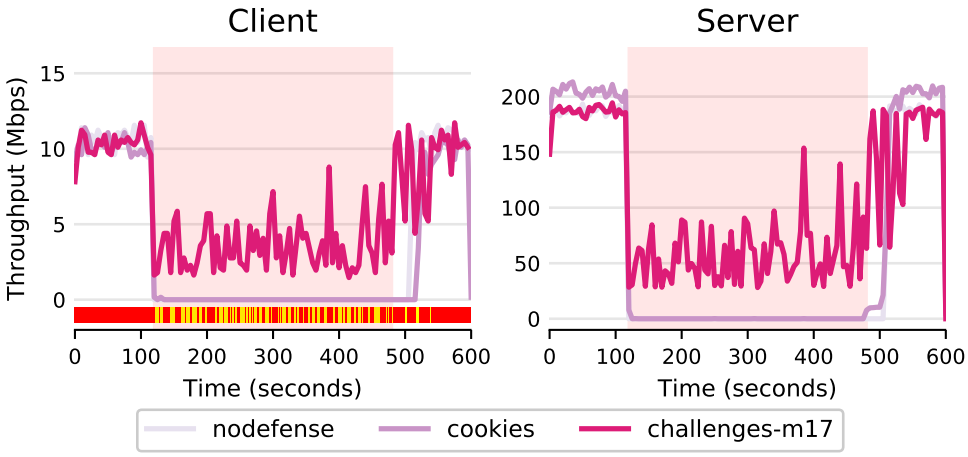


Figure 3.5: Client and server throughput during a connection flood.

state of the connection in the TCP sequence number instead of in the `listen` queue, SYN cookies provide protection against this type of attack. Finally, when low difficulty puzzles are enabled, $(k, m) = (1, 8)$, the throughput is unaffected during the attack. Similarly to SYN cookies, the puzzles enabled reconstruction of a connection's state with no use of the `listen` queue. However, when we used the Nash equilibrium difficulty, $(k, m) = (2, 17)$, the throughput was reduced to 50 Mbps during the attack. This reduction occurred because the equilibrium strategy is more aggressive than the easier setting; in this scenario, easy puzzles were enough to alleviate the attack as the botnet was not completing the connection.

For the second scenario, we used the attacker nodes to launch a distributed connection flood attack. We measured the same metrics as in the first scenario for three cases: no protection, SYN cookies, and TCP puzzles at Nash difficulties. The TCP puzzles at a difficulty

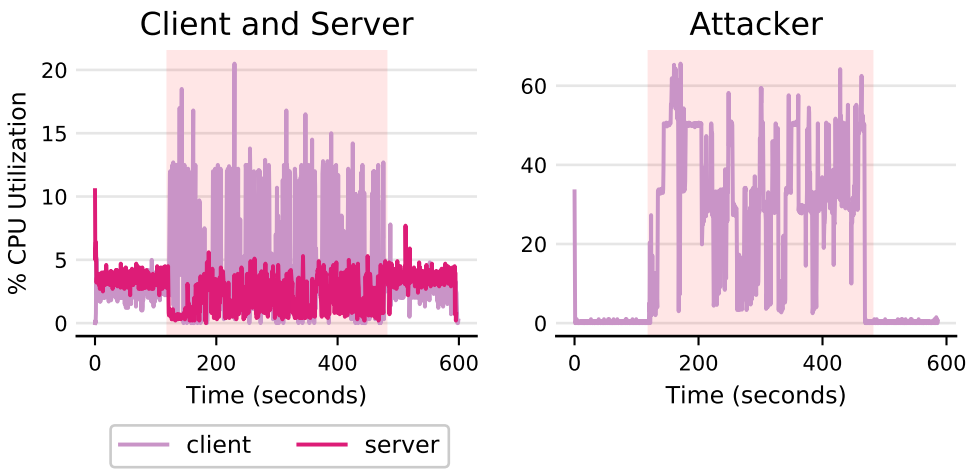


Figure 3.6: CPU utilization during a connection flood attack.

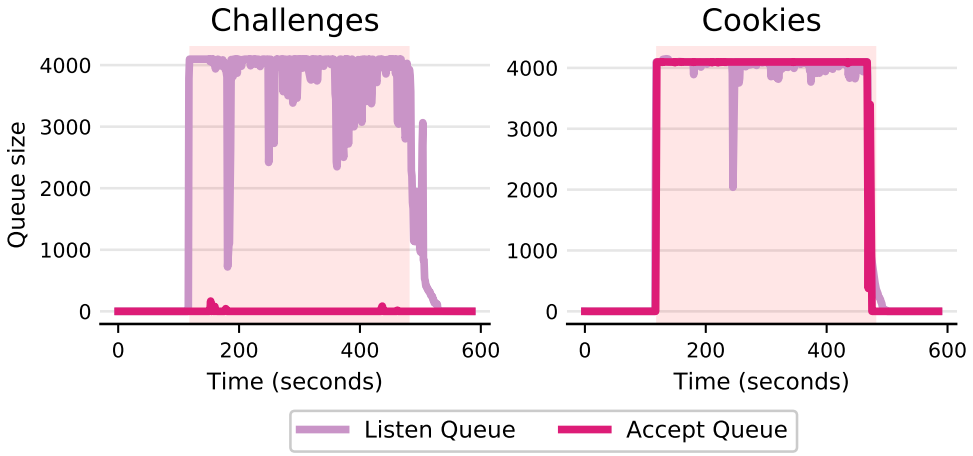


Figure 3.7: Queue sizes during a connection flood attack.

of 8 bits were ineffective at protecting the server's state. For readability, we elected not to show these results in this plot since we will revisit various difficulty settings in Section 3.6.2.

Figure 3.5 shows the throughput of a client and the server during the experiment. We used the sparkline in the client plot to mark when the server sent a SYN-ACK packet with a challenge (bright tick) or without a challenge (dark tick). The results show that SYN cookies are ineffective during a connection flood; the server's throughput drops to 0, as it would if no protection were in place. In both those cases, the server needs 30 seconds to detect the end of the flood and fully recover. On the other hand, TCP puzzles at Nash difficulties provide tolerance of the flood attack. The throughputs of the client and the server were about 40% of their respective nominal rates. It is interesting to note that the throughput periodically spiked during the attack phase. This occurs because not all the requests of

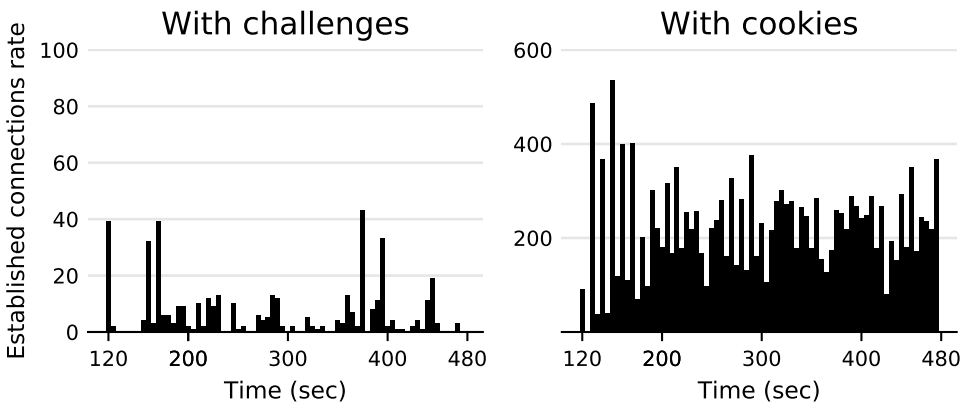


Figure 3.8: Effective attack rate during a connection flood attack.

the clients required a puzzle, as shown by the dark ticks in the sparkline during the attack phase. The performance improvement was due to the opportunistic nature of the protection controller; that is, when the `listen` queue was not full, connection requests were answered without a challenge, allowing a host to take advantage of the resource instantly. We also note that easy puzzles were unable to affect the attacker bots' connectivity rates and thus provided no better protection than SYN cookies did.

In addition, we measured the impact of the TCP challenges on the CPU utilization of the client, server, and attacker machines. Figure 3.6 shows that the impact on the server of generating and verifying the puzzles was negligible; the server's CPU utilization stayed below 5% and did not exceed its nominal (under regular load) value. In accordance with the nature of computational puzzles, the CPU utilization at the clients' machines increased during the attack, but still remained well under 20%, with an average of 10%. The attacker machines, on the other hand, witnessed a spike in CPU utilization during the period of the attack, reaching a maximum of 60%. These results show that our equilibrium difficulty setting achieved our desired goals of (1) putting minimal overhead on the server in generating and verifying puzzles, (2) inducing tolerable nuisance for the clients, and (3) effectively rate-limiting the attackers' established request rate and increasing their computational burden. In fact, the sudden increase in the CPU utilization at the botnet machines can alert the owners of these machines to the presence of malware.

We further studied the impact of the TCP cookies and puzzles on the server's `listen` and `accept` queues during a connection flood attack. Figure 3.7 shows that when SYN cookies were the only defensive mechanism in place, both queues were fully saturated, which explains the zero throughput observed by the benign clients. On the other hand, with TCP challenges in place, the `accept` queue was almost always empty, which was a direct result

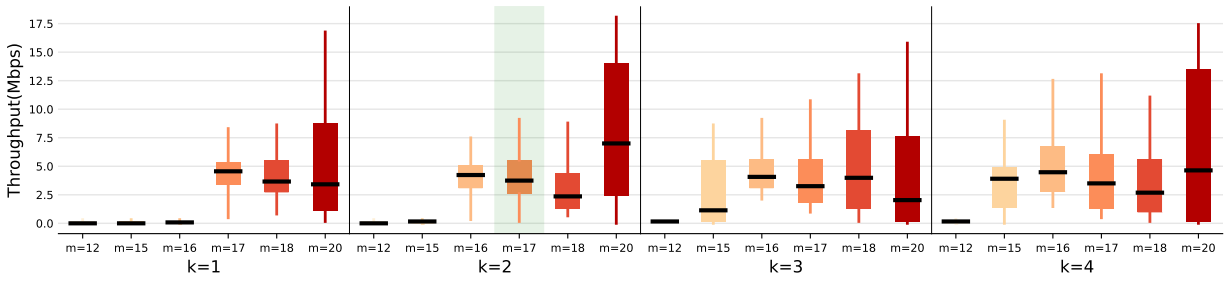


Figure 3.9: Box plot of the client throughput for different puzzle difficulties.

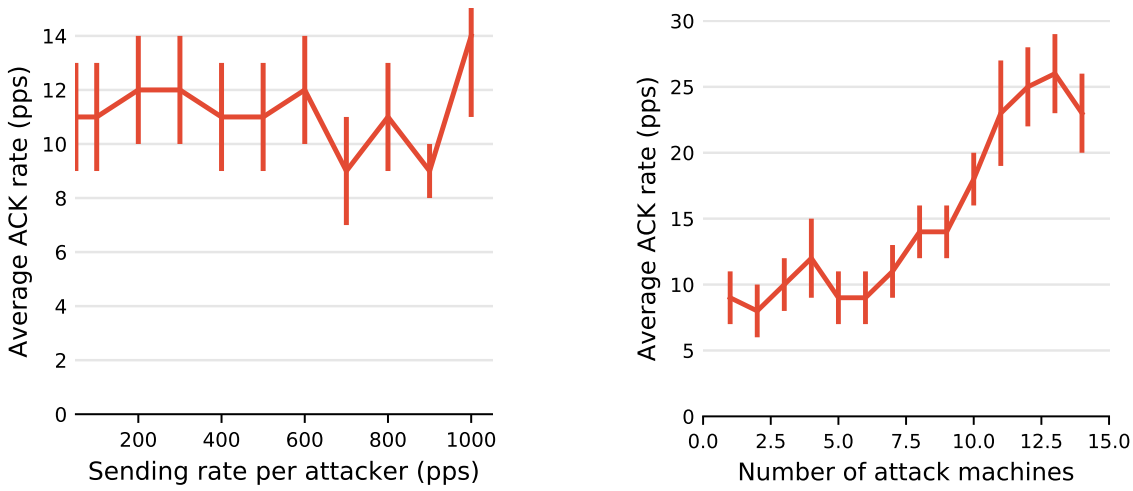
of the puzzles' ability to rate-limit every user, whether benign or malicious, to an average of 2 requests per second. In addition, the `listen` queue, although mostly saturated, showed frequent openings that are consistent with the opportunistic nature of our implementation, as indicated by the sparklines in Figure 3.4.

Finally, we showed that TCP puzzles (at Nash difficulty) throttled the attacker's rate of established connections. We measured the effective completed connection rate of all attackers as seen by the server during the connection flood. The measurements, shown in Figure 3.8, reveal that the attack rate was not affected by TCP cookies, achieving an average rate of 225 connections per second (cps), whereas puzzles severely limited the attackers' rate down to an average of 4 cps, a reduction by a factor of 37.

3.6.2 Experiment 2: Nash equilibrium strategy

In this experiment, we showed that the Nash equilibrium difficulty provides the optimal balance between the clients' throughput and the attack tolerance during an attack. We selected the Nash equilibrium based on the capabilities of the clients and the server's defense requirements.

Figure 3.9 shows the average and standard deviation of the throughput of a client during an attack. In general, for any k, if $m < 12$, the ease of solving the challenges did not affect the attackers' rate, so a denial of service occurred. The Nash equilibrium strategy resulted in the most stable throughput, with an average of 3.90 Mbps and low variability. Even though some of the other settings had a higher average throughput, their throughput was highly unstable, reaching zero at many times. Further, we note that when the difficulty was set to $(k = 2, m = 16)$, the throughput achieved a slightly better average with comparable variability. However, the Nash difficulty setting provided the rate that balanced the acceptable cost a client was willing to pay and the server's ability to tolerate state exhaustion attacks by throttling the attackers' rates. In fact, at the Nash difficulty, the puzzles mechanism reduced the attackers'



(a) Impact on flooding rate

(b) Impact on botnet size

Figure 3.10: Impact of the puzzles on the attack as the size of the botnet and the flooding rate are varied

average SYN sending rate from 2250 pps for $(k = 2, m = 16)$ to 1668 pps, and the average connection establishment rate from 30 cps to 22 cps.

3.6.3 Experiment 3: Botnet effectiveness

In the third experiment, we showed that TCP puzzles increased the server's tolerance to a botnet and required attackers to increase their botnet's size to deny service. We varied the botnet's size and attack rate and measured the cumulative attack rate as seen by the server, referred to as the *connection completion rate*. The connection completion rate is the effective attack rate that actually impacts the server. In the first scenario, we set the number of nodes in the botnet to 5 and varied the sending rate of each node between 100 and 1000 pps. Figure 3.10a shows the rate of completed connections as the rate of each attacker machine was varied. The results show that the TCP puzzles were capable of rate-limiting the effective attack rate. As the per-node attack rate increased, the effective attack rate was limited to 11 cps in all cases.

In the second scenario, we varied the number of machines in the botnet while setting the cumulative attack rate to $5,000$ pps; each machine's rate was set at $5,000/(\text{size of botnet})$. Figure 3.10b shows the measured effective attack rate as the number of machines was varied. The results show that attackers had to increase the size of their botnets to increase their effective attack rates. The effective attack rate, although it linearly increased with the increase in the number of attack machines, only peaked at 25 cps. In contrast to the near-

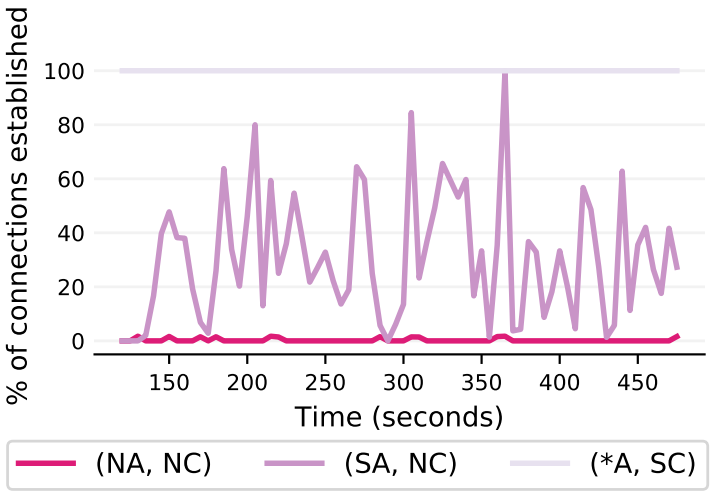


Figure 3.11: Percentage of established connections when TCP puzzles adoption is not complete

constant rate in the first scenario, the effective attack rate increased in this scenario since more machines were enlisted in the botnet. However, this increase did not reflect the increase in resources being committed to the botnet. At this rate of increase, a botnet has to commit 500 machines to reach an effective attack rate of 5000 cps.

In conclusion, the attacker cannot increase the effective attack rate by increasing the individual rates; she has to increase the number of machines in the botnet. TCP puzzles at the Nash equilibrium difficulty significantly increased the cost of a state exhaustion attack.

3.6.4 Experiment 4: Adoption of TCP puzzles

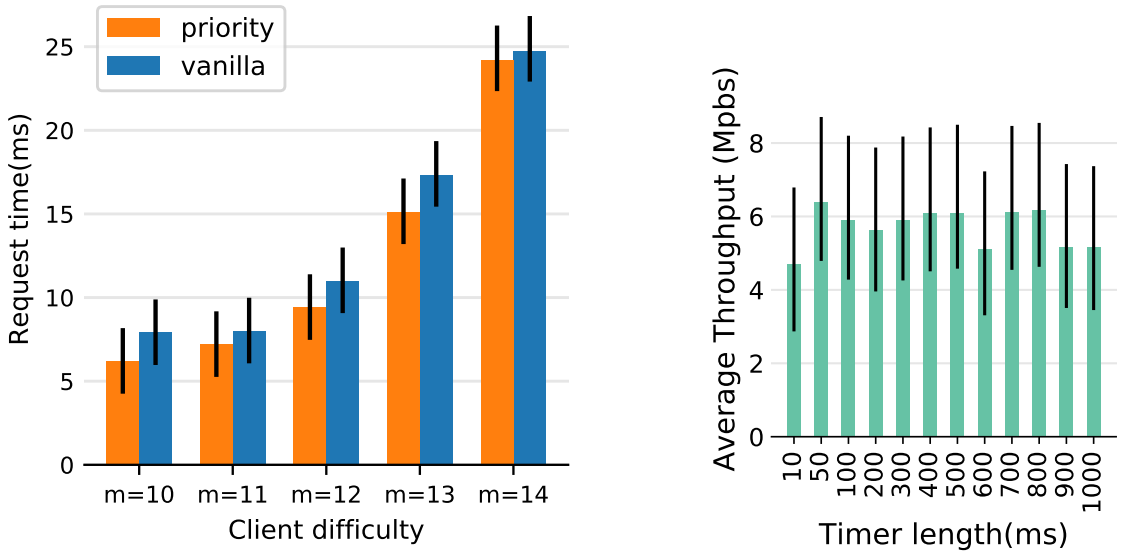
In this experiment, we showed that a client solving the TCP puzzles is almost always able to connect to the server regardless of whether the attacker elects to solve or ignore the puzzles or select a combination thereof. On the other hand, a client that does not solve puzzles gets erratic service when the attacker is solving the puzzles and almost no service when the attacker floods the server without solving any puzzles. In this experiment, we used machines that were not patched to support TCP puzzles; we tested four scenarios in which (1) neither the attacker and the clients solved puzzles (NA,NC); (2) the attacker solved puzzles while the clients did not (SA,NC); (3) both the clients and the attacker solved puzzles; and (4) the clients solved puzzles and the attacker did not. We group scenarios (3) and (4) together and label them (*A, SC). Figure 3.11 shows the percentage of completed connections for all the proposed scenarios. We observe that a client solving puzzles is not denied service

regardless of the attacker's type; this happens because the attacker, being rate-limited when solving puzzles and having its requests ignored when not solving, is not able to fill the `accept` queue of the server. On the other hand, a non-solving client faced with a solving attacker experiences a highly variable percentage of completed connections, reaching 0 at some instances. The reason is the opportunistic nature of the puzzles controller (as observed in Experiment 2); the rate-limiting impact on the attacker machines can empty slots in the server's queues, thus providing openings in which the non-solving client can establish connections. However, when faced with an attacker that does not solve the challenges, the non-solving clients are denied service, because the attacker's vast resources beat the clients' requests for the resources freed by the puzzles controller. We note that the service promises provided by our implementation to noncompliant clients are similar, and sometimes better than, those provided by network capabilities [67].

3.6.5 Experiment 5: Puzzle-based Queueing

In this experiment, we present a prototype solution that addresses the challenge that arises from treating every user as potentially malicious, namely the possibility that more powerful attackers will be able to control the majority of the server's resources. To that end, we modified the queueing behavior of the kernel's TCP stack from a *First In First Out* (FIFO) discipline to a *priority-based* behavior. Our design rests on the observation that a benign user would have an average of $\hat{n}_i = \frac{2^{m_i-1}}{\beta_i}$ seconds between their requests, where β_i is that user's hashing rate per second. So a user that can send more than one request in a single $\hat{n}_i$ interval is likely to be attempting to launch a connection flood attack.

We therefore allow each user i to solve puzzles at any difficulty $m_i > m^*$, and then tag that user's request with a weight $\eta_i = \lfloor \frac{m_i}{x_i} \rfloor$, where x_i is the user's request rate in a given time interval ΔT. Computing the weights for each request requires the server to keep state about incoming requests, which might also be a target of a state exhaustion attack. However, our design safeguards against this vulnerability in the following manner. First, for each user i, we need only $\lceil \log_2(m_{\max}) \rceil$ bits to keep track of x_i, where $m_{\max}$ is the maximum possible puzzle difficulty bits (32 bits in our case). This is valid since when $x_i > m_{\max}$, the weight is always 0, and thus we do not need to keep incrementing x_i. We thus achieve a significant reduction in state compared to that for a half-open TCP request. Second, we associate each entry in our state with a timer that expires after ΔT microseconds and deletes that entry. If we receive another request from the same user before the timer expires, we reset the timer for another ΔT microseconds. Thus, our state will be used for only potentially malicious users that are sending requests at a rate greater than $\frac{1}{\Delta T}$. The server can control the timer



(a) Performance evaluation

(b) Throughput at a benign client

Figure 3.12: Evaluation of our prototype puzzle-based queueing

interval ΔT through the kernel's `sysctl` interface. Finally, if the server's state table is full, it assigns the same weight to all requests and falls back to the same case as in Experiment 2.

We first evaluated the performance impact of performing priority queueing for every incoming request. We compared the average service time per request for 5 clients that were simultaneously connecting to the server, each solving at a difficulty of 10, 11, 12, 13, and 14 bits, respectively. We differentiate between two cases: (1) the *vanilla* case in which we used the FIFO queue, and (2) the priority queueing implementation. Figure 3.12a shows the average service time for a single request in each case and indicates that, with 95% confidence, our implementation incurs no performance penalty.

We then evaluated the effectiveness of our design when it faced a mixture of users solving at different difficulties. We fixed the server's minimum acceptable difficulty at $(k = 2, m = 15)$ and used 8 attacker machines and only 2 client machines (thus giving the majority of the hashing power to the attackers). The client machines solved at the minimum allowable difficulty, while the attackers used difficulties ranging from 15 to 19 bits. We varied the state table timer (ΔT) from 10 ms to 1000 ms and measured the throughput at the client.

Figure 3.12b shows the throughput at a benign client as the timer interval ΔT was increased. We notice that in all cases, the client was able to connect to the server and reach a throughput of at least 4.5 Mbps. That is a significant improvement over the scenario presented in Experiment 2, in which for $(k = 2, m = 15)$ the clients were not able to connect to

the server (Figure 3.9). We also note that the length of ΔT had little impact on the observed client throughput. However, when $\Delta T = 10$ ms, the client saw its lowest throughput. The reason was that the client was attempting to connect to the server at a rate of 20 packets per second (i.e., a new connection every 50 ms). Therefore, for $\Delta T < 50$ ms, the client's timer was always renewed, and thus the weight assigned to it quickly reached 0 and its requests were treated as malicious.

3.7 LIMITATIONS AND DISCUSSION

In this section, we discuss the challenges facing the adoption of client puzzles and provide an analysis of their limitations.

Software adoption: As showcased by our experiments, there is a great benefit for servers to adopt client puzzles as a mechanism for tolerating state exhaustion attacks. By rate-limiting users and protecting the server's queues, client puzzles present service providers with a chance to provide continuous service during state exhaustion attacks. Our implementation has several features that make it easy to adopt. First, a server can easily support client puzzles by simply patching its kernel. Second, our patch does not introduce any changes to the normal operation of the server and sends challenges only when the queues overflow; the TCP stack remains intact otherwise. Finally, our patch is compatible with earlier versions of the Linux kernel provided they support cryptographic operations.

While servers are incentivized to adopt TCP puzzles by the increased tolerance of attacks, clients, on the other hand, benefit from the promise of receiving service even during attacks. As shown in Experiment 4, users that enable support for client puzzles are always able to connect to the server. The users that choose not to adopt the challenges still receive full service under regular load. However, during an attack, those users will be in contention with the nonsolving attackers for the spots are freed up by the server's opportunistic challenges controller. That scenario is no worse than the case when no challenges are applied. Note that our theoretical formulation validates this observation; a user that does not adopt challenges is similar to one that values the server's services at $w = 0$.

Replay attacks: Since the server does not retain state about an incoming connection before receiving a valid challenge solution, an attacker might capture legitimate clients' solutions and replay them to overflow the server's `accept` queue. We note, however, that for a replayed solution to be validated, the attacker must retain the packet's parameters (IP addresses, port numbers, and timestamps). Therefore, a replayed solution can only be used to occupy one slot in the server's queue at a time. In addition, our implementation ensures that puzzles expire after a set timeout interval. The timeout interval limits an attacker's ability to carry

out a replay flood effectively, and thus our implementation is resistant to such attacks.

Fairness and power considerations: Finally, client puzzles research faces an important challenge arising from the presence of a nonuniform mix of power-limited (e.g., mobile phones, IoT devices) and power-endowed (e.g., GPU-enabled desktops) benign devices. Motivated by the work in [70], in Experiment 5, we built a prototype design that is intended to combine puzzle difficulty and request rates to assign weights to clients' requests. Our preliminary results are promising, and we plan to explore this further in the future.

3.8 CONCLUSION

In this chapter, we presented a theoretical formulation and implementation of client puzzles as a means for tolerating state exhaustion attacks. We addressed the challenge of selecting puzzle difficulties by modeling the problem as a Stackelberg game in which the server is the leader and the clients are the followers. We obtained the equilibrium solution that illustrates a tradeoff between the clients' valuation of the requested services and the server's service capacity. We then provided our implementation of the puzzles as a Linux kernel patch and evaluated its performance on the DETER testbed. Our results show that client puzzles are an effective mechanism that can be added to our arsenal of defenses against DDoS attacks.

CHAPTER 4: MIDGARD: CROSS-LAYER DEFENSE TO VOLUMETRIC DISTRIBUTED DENIAL OF SERVICE ATTACKS

4.1 INTRODUCTION

In recent years, the scale and complexity of *Distributed Denial of Service* (DDoS) attacks have grown significantly. The introduction of DDoS-for-hire services has greatly reduced the cost of launching targeted attacks to a low of $10 per-hour [96]. This is further exacerbated by the increased adoption of poorly-secured, hard-to-patch, *Internet of Things* (IoT) devices, rendering it easy for malicious actors to amass botnets of millions of devices [62]. The direct exposure of IoT devices to the Internet has also allowed attackers to diversify their arsenal of attacks and launch multi-vectored attacks that include bandwidth-exhaustion, state-exhaustion, and application layer attacks [60].

DDoS-for-hire and IoT devices have greatly contributed to lowering the barrier of entry for DDoS actors, thus creating a large imbalance between the cost of launching an attack and that of sustaining one [69]. This imbalance is further brought to light by attackers that employ reflection attacks such as DNS or NTP reflections. In fact, using poorly configured `memcached` servers, attackers were able to reach an unprecedented 51,200x amplification factor [97]. It is therefore of paramount importance that we address this imbalance and reclaim the defenders' advantages.

Common DDoS mitigation techniques nowadays rely on the services of cloud- and CDN-providers, such as Cloudflare or Akamai, that massively over-provision data centers in an attempt to absorb attack traffic and perform filtering and scrubbing techniques. Although widely adopted by industrial settings, such approaches carry the risk of turning the DDoS landscape into an unsustainable bandwidth war between attackers and cloud providers, furthering the stress on the network infrastructure, especially as we enter the era of unprecedented Tbps attack traffic levels [98]. They further do not provide any solutions to the cost imbalance problem; *the burden of defending DDoS attacks is falling increasingly on the shoulders of the victim and the network, while the barrier of entry for DDoS is getting lower.*

Capability-based approaches [29, 99, 100] attempt to authorize network traffic that traverse congested links by requiring senders to obtain specific authorization tokens from a destination server. The server can then request specific rate limiting mechanisms or bandwidth allocation schemes. It is the job of the network (i.e., the routers) to implement the mechanisms and validate the capabilities. In addition, capability-based approaches require routers to reserve a fraction of their bandwidth for capability request packets; that capability request channel can also be subject to denial of service (or denial of capabilities)

attacks [30]. Furthermore, both CDN- and capability-based approaches do not provide any protection against attacks that target links other than the victim's direct upstream link, such as the Crossfire [101] and the Coremelt [102] attacks.

In this chapter, we present MIDGARD, an end-to-end solution that combines (1) *routing defenses*, (2) *network capabilities*, (3) *client puzzles*, and (4) *cloud elasticity and deployability* to provide resistance to large-scale, multi-vectored DDoS attacks. MIDGARD is comprised of traffic management and policing boxes and routers that reside in a victim's upstream ISP, and uses client puzzles as network capabilities to influence routing and traffic policing decisions on the path to the victim. We implement MIDGARD on a custom box running general purpose Linux and the elasticity and provisioning of cloud networks to provide a deployable solution. We implemented MIDGARD using the Linux kernel's `AF_XDP` sockets that allow us to maintain near line-rate processing speeds. We evaluated our implementation using a set of experiments on the DETER testbed. Our results show that the use of client puzzles to drive traffic policing decisions improves the network's ability to sustain volumetric DDoS attacks, and restore the balance between the cost of launching and defending such attacks.

4.2 PROBLEM STATEMENT

MIDGARD is designed as an easy-to-deploy end-to-end solution to bandwidth- and state-exhaustion DDoS attacks occurring at Layers 3 and 4 of the networking stack. *Dealing with application layer attacks is beyond* MIDGARD*'s scope as it requires application level information and control over the actual victim's resources.* We designed MIDGARD to bring about the advantages of (1) capabilities for rate limiting and traffic policing, (2) client puzzles for balancing the cost of attacks and defenses, and (3) cloud deployments for initial attack containment. In what follows, we identify a set of desirable properties that a DDoS solution must provide for it to be effective.

Cost Balance and Fairness. Solutions such as traffic filtering, capabilities, and cloud-powered overlays and CDN, put the burden of providing resistance of large scale attacks solely on the victim's shoulders (and the cloud provider, though the victim must bear the monetary cost). However, motivated by the quick increase in the scale and diversity of attacks, the barrier of entry for attackers is at an all-time low, especially with the advent of DDoS-for-hire services. This *cost imbalance* has greatly tipped the scale in favor of the attackers and has effectively provided them with an easy pass to an ever increasing scale of DDoS attacks.

MIDGARD addresses this cost imbalance problem by leveraging client puzzles at the net-

work level, and essentially forcing attackers to provide "payment" for their share of the traversed links' bandwidth. This also carries the additional benefit of alerting users of compromised bot machines of the possibility of security breaches.

However, client puzzles suffer from two major disadvantages. First, the puzzles do not provide fairness guarantees to low-powered devices as well as devices that have a generally low traffic sending rate. Second, similar to the 51% attack in cryptocurrencies, attackers with large botnets that can control a larger aggregate hashing power than the legitimate clients will end up controlling the largest fraction of the bandwidth, and thus achieve their goals of the denial of services; we refer to such a case as the *majority-hashing* attack. The *per-computation* fairness model introduced in [30] and [103] (1) disfranchises low-powered IoT and mobile devices, (2) does not reward legitimate clients that send lower volumes of traffic, and (3) provides little resistance to the majority-hashing attack. MIDGARD addresses these issues with client puzzles by combining traffic policing and capability-base rate limiting with client puzzles at the victim's upstream ISP level.

Deployability. MIDGARD's design introduces the following deployability challenges. First, since MIDGARD employs client puzzles, it will require software updates at the affected clients. This is an essential part of our end-to-end solution paradigm. Second, adopting client puzzles requires MIDGARD to provide a puzzle distribution mechanism. [30] and [103] suggest using DNS servers and ICMP messages as puzzle distribution schemes, respectively. We discuss the shortcoming of these approaches in Section 4.4.1 and discuss how MIDGARD leverages the two ways nature of Internet communication (from the point of view of the client) to distribute puzzles without the need for trusted third parties or ICMP messages.

Finally, MIDGARD's capabilities-based policing and rate-limiting requires upgrades to the network hardware, as it needs enforcement at the network level (routers and switches). We leverage recent advances in SDN and programmable networking to provide an incrementally deployable adoption scheme that minimizes the need for hardware upgrades. Additionally, unlike previous capabilities-based and client puzzles approaches [29, 32, 99, 103], MIDGARD only requires updates to the networking equipment at the victim's upstream ISP, and does not require implementation across administrative domains. In fact, we believe that MIDGARD can be provided as a subscription-based service for the clients of an ISP, thus alleviating the one-time cost of its deployment.

Resistance to Cloud Bypass Attacks It is possible for attackers to circumvent current cloud-based DDoS solutions by exposing the victim's IP address and sending attack traffic directly to it (rather than going through DNS) [104, 105, 106, 107]. We discuss how MIDGARD provides resistance to such circumvention attempts in Section 4.4.2.

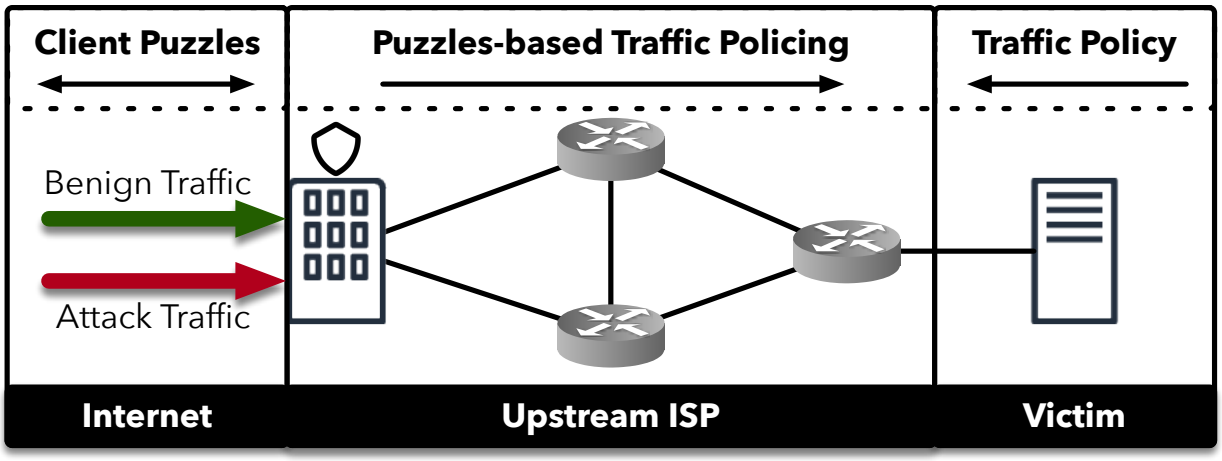


Figure 4.1: Overview of MIDGARD. The MIDGARD server distributes client puzzles and creates capabilities that the victim's ISP uses to police ingress traffic.

4.3 MIDGARD OVERVIEW

We show a high-level overview of MIDGARD in Figure 4.1. The MIDGARD server (or box) resides at the edge of the victim's ISP. We envision that MIDGARD's protection mechanism can be provided as a subscription to interested clients, similar to Amazon's AWS Shield [108]. Deployment of MIDGARD within a single ISP provides the benefits of network-level protection without the need for agreements that span multiple authoritative domains such as those necessary for other mechanisms [29]. Furthermore, we leverage the recent advances in SDNs and programmable networking hardware to provide cost effective deployment and maintenance of MIDGARD within the ISP's network.

When under a DDoS attack, all traffic destined to the victim is redirected to the overly provisioned MIDGARD server, in a way similar to that performed by cloud-protection providers and adopted in [104]. However, unlike cloud protection services, MIDGARD only uses the over-provisioned cloud to absorb the initial impact of the attack. It does not perform any traffic scrubbing and rather attempts to stop the attack at the source. To that end, the MIDGARD server acts as a *puzzles distribution service* and handles the generation, distribution, and verification of the cryptographic challenges. When a sender's (whether malicious or benign) puzzle verification is successful, the server will then create a cryptographic capability based on the state of the downstream links to the victim, the sender's puzzle difficulty, and the sender's past behavior.

Unlike the work in [104], beyond dropping packets with incorrect puzzle solutions, the MIDGARD server does not perform any direct routing decisions. It will create the capability nonce that the downstream ISP routers will then use to enforce the traffic policies. This

reduces the workload of the MIDGARD server while also reducing the stress on the routers' processors and memory, and requiring no inter-router communication, as compared with prior capability-based approaches [29].

Finally, unlike previous capability and cloud-based approaches, by using client puzzles, MIDGARD gets the added benefit of protecting MIDGARD's internal maintained state from state-exhaustion attacks, thus proving resistance to multi-vectored DDoS and reducing the attack surface introduced by the added mechanisms. In fact, in MIDGARD's setup, client puzzles serve a dual purpose. First, as originally intended in [71], the puzzles protect MIDGARD's state from state-exhaustion attacks since every sender will need to solve a puzzle to occupy an entry in the server's memory. Second, MIDGARD translates the received puzzle solutions into capabilities that downstream routers will use to perform traffic policing, thus eliminating the need for a capability requests channel and efficiently dealing with denial of capability attacks.

4.4 MIDGARD ARCHITECTURE

MIDGARD aims to provide a synergy between the most efficient DDoS protection mechanisms in the goal of providing strong resistance to large scale, multi-vectored, DDoS attacks. The MIDGARD box is composed of 3 agents: (1) *the puzzle generation and distribution agent*, (2) *the puzzle to capabilities translation agent*, and (3) *the network state estimation agent*. On the other hand, routers in the victim's upstream ISP network run a *capabilities enforcement and policing agent*. In what follows, we discuss the design and architecture of each of these agents.

4.4.1 Puzzle Generation and Distribution

Once an attack is detected, the MIDGARD server will start asking each sender to solve a cryptographically hard puzzle in order to gain a "right of entry" into the victim's upstream ISP network. Previous puzzle based defenses often made it the responsibility of the sender (*i.e.* the client) to initiate the puzzles protocol. Portcullis [30] requires users to send DNS queries to specific domains to obtain a valid puzzle nonce. Users will then keep solving harder and harder puzzles until they are able to obtain a valid capability. Mirage [109] redirects senders to specific puzzle servers that will handle the dissemination of puzzles, and Congestion Puzzles [103] requires users to continuously probe the network for congestion and receive puzzles from routers in the form of special- purpose ICMP message.

Such approaches violate our deployability requirement for the following reasons. First, they require the cooperation and approval from authoritative DNS server to store and deliver puzzles. Second, they require the presence of dedicated servers or CDNs thus increasing the cost and trust requirements of deployment. Third, ICMP message are ill-equipped to handle puzzle distribution since (1) clients should not be expected to explicitly probe the network for congestion information and (2) more importantly, senders behind NATs or firewalls will not be able to receive such messages; typical firewall configurations often block ingress ICMP messages or only allow ingress traffic that matches previously recorded outgoing traffic.

In this work, we adopt a different strategy. We believe that any puzzle-based protocol should always be initiated by the destination when it detects the presence of an attack. It is therefore the responsibility of the MIDGARD server to generate and distribute puzzles to the senders. By doing that, MIDGARD requires no support from DNS authorities and remains within the same trust boundary as the victim server and its upstream ISP. Additionally, MIDGARD leverages the experimental capabilities of IP's *Experimental Congestion Notification* (ECN) [110, 111][1] and IP header options to signal congestion and distribute puzzles, thus eliminating the dependency on ICMP messages and the requirements for clients to send probe messages.

When congestion is detected at the downstream links to the victim, the MIDGARD server start generating puzzles and capabilities. After receiving a packet from sender s_i destined to the victim's IP address IP_v, the server will generate a puzzle nonce x_i such that

$$x_i = H\left(T, IP_v, K\right) \tag{4.1}$$

where T is the current server's timestamp, K is a secret only known to the MIDGARD server, and H is a publicly known preimage resistance cryptographic hash function, such as SHA3. Similar to the approach in [30], we specifically do not include s_i's IP address to avoid the problems introduced by devices behind NATs and attackers with spoofed IP addresses. The server will subsequently send a packet back to s_i with the ECN bits of its IP header set to 01 and the nonce x_i and the timestamp T included in the packet's IP options. The server will also forward s_i's packet to the downstream routers to be served as legacy traffic, *i.e.*, it will receive a best-effort treatment by the downstream routers.

In order to be able to deliver its puzzle packets and avoid such packets being dropped by sender-side firewalls, the MIDGARD server will peak into the sender s_i's transport layer protocol and send the puzzle in a packet carrying a header for the same protocol and destined

[1]Restrictions on experimentation with ECN were relaxed in RFC [111] which provides us with a deployable option for puzzle distribution.

to the same port numbers. Achieving this is trivial for the case of TCP: the server can simply send a TCP RST packet containing the puzzle nonce, to which the sender will respond with a new connection handshake request (TCP SYN) along with the puzzle solution. Alternatively, to avoid the overhead of initiating a new connection, the server can send the puzzle nonce to s_i through a TCP KeepAlive message. In the case of UDP, since the protocol contains no state information, MIDGARD will simply send a payload-free UDP packet to the same originating port but with the IP ECN bits set and the nonce in the IP options. This will indicate to s_i that the destination is under attack and that it should start solving puzzles. In the cases of UDP and TCP KeepAlive messages, since parsing and processing of IP headers will happen prior to transport headers, the puzzles agent at the sender can safely ignore these packets and flag to the application the need for solving puzzles for the next packets.

The Client's Agent. Upon receiving a packet with the ECN bits set and with a puzzle nonce, the sender s_i will generate a random client nonce r, select a puzzle difficulty d it is ready to commit to, and generate the solution y such that

$$H\left(y||x_i||d||r\right) = \underbrace{b_0 b_1 b_2 \ldots b_{d-1}}_{0 \text{ bits}} b_d b_{d+1} \ldots b_{m-1} \tag{4.2}$$

where H is the same publicly known hash function that the server used to generate x_i and m is the bit length of H's output. Assuming H is chosen well, the sender's best option is to find y by brute force, and is expected to perform an average of 2^{d-1} hash operations to find the solution [71, 112]. s_i will finally attach r, d, T, and y to its next packet destined to IP_v and set the ECN bits of the IP packet to 02.

Upon receiving such a packet, the MIDGARD server will first regenerate x_i (according to Equation (4.1)) and check that the first d bits of the outcome of Equation (4.2) are 0. If the puzzle solution is correct, the server will generate a capability nonce c_i for s_i and create a mapping in its table from s_i's nonce r to its capability c_i. It will include c_i in the IP header options of s_i's packet and set the ECN bits of the IP packet to 11 informing the downstream routers that congestion has occurred that they should use the capability to perform traffic policing.

Dealing with puzzle reuse. As shown in Figure 4.2, an attacker that can sniff a sender's packets (either by compromising the sender itself, or by capturing packets on the fly) can observe the sender's nonce r and solution y and use them to send its own traffic, thus sending packets on the original sender's dime. Although acknowledged in previous research [30, 103], little attempts have been done to alleviate this problem; in [103], the authors dismiss this challenge by assuming that the attackers have limited access to the sender's packets, while

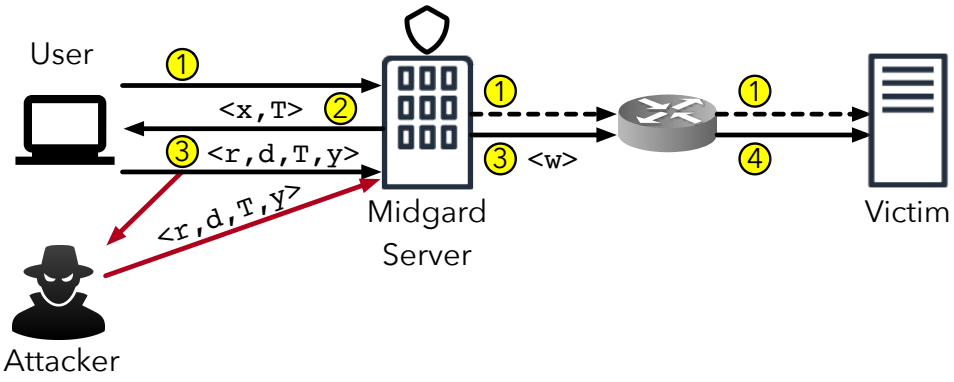


Figure 4.2: Puzzle distribution mechanism. Upon receiving the first packet from the user, the MIDGARD server forwards that packet in best-effort and sends a nonce and a timestamp to the user. The user then solves the puzzle at a certain difficulty and adds the solution to every packet it sends to the victim server. Upon receiving such packets, the MIDGARD server validates the solution and forwards the packet to the victim server while tagging it with the computed user's capability. However, in such a case, an attacker that can sniff the user's packets can steal the user's solution and send packets on its behalf.

in [30], the authors suggest caching puzzle solutions at routers (by using Bloom filters) to detect puzzle reuse, which then increase the memory load on the routers and opens up a new vector of attack. We note that public key cryptography is not an adequate solution in this case since key exchange and signature time can quickly become bottlenecks in our protocol, thus defeating the entire purpose of MIDGARD.

In this work, we adopt a different strategy. We treat puzzle solutions similarly to one-time pads, *i.e.* each puzzle once attached to a packet will henceforth be discarded. To that end, we leverage recent results [90] showing that puzzles defined as in Equation (4.2) have *strong difficulty guarantees*: namely that solving n such puzzles costs about n times the cost of solving a single one. Therefore, instead of solving a single puzzle of difficulty d, the sender s_i will solve $n = 2^k$ *sub-puzzles* each having a difficulty $d' = d - k$. Using the strong puzzle notion, this will guarantee that s_i will perform the same amount of work, on average, as it would have had it solved a single puzzle of difficulty d, since the expected number of hash operations it must perform is

$$n \times 2^{d'-1} = 2^k \times 2^{d-k-1} = 2^{d-1} \tag{4.3}$$

Formally, upon receiving a puzzle nonce x_i, s_i will generate n sub-puzzles $\{y_1, \ldots y_n\}$ such that

$$H\left(y_j||j||x_i||n||d'||r\right)) = \underbrace{b_0 b_1 \ldots b_{d'-1}}_{0 \text{ bits}} b_{d'} d_{d'+1} \ldots b_{m-1} \tag{4.4}$$

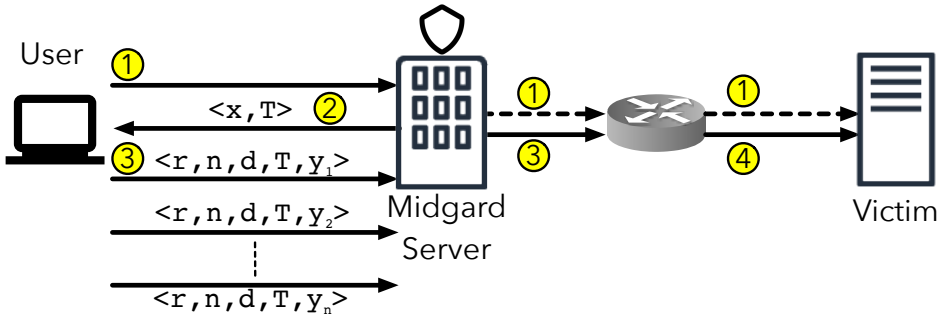


Figure 4.3: Countering puzzle reuse by malicious attackers. Unlike Figure 4.2, instead of solving a single puzzle and tagging each packet, the user solves n sub-puzzles of lower difficulty and tags each packet with a different solution. Each solution allows the user to send only one packet before being discarded, therefore preventing attackers from reusing puzzles even if they can sniff the user's packets.

for $j \in \{1, 2, \ldots, n\}$. The sender can then attach $< r, n, d', T, j, y_j >$ to every packet it sends to the victim server. The sub-puzzles in this case will then serve the dual purpose of rate limiting the senders as well as allowing s_i to specify that it wishes to send n packets using the same puzzle nonce, and thus using the same capability (note however that the MIDGARD server can only guarantee that s_i will be able to send $\leq n$ packets). An attacker that can capture $y_1, y_2, \ldots, y_k$ for a certain $k < n$ cannot reuse them to send any packets. Also, since puzzles are strongly difficult, the attacker cannot infer anything about the next solution y_{k+1}. Finally, at the MIDGARD end, the server can keep track of the next sub-puzzle to expect by keeping a counter of the packets, p, sent by s_i (as it will need to do that for capability enforcement). It can discard any solution with $j < p$. We illustrate how the MIDGARD protocol defeats puzzle reuse in Figure 4.3.

4.4.2 Traffic Policing

In order to provide a fair allocation of resources, the MIDGARD service leverages the fact that a legitimate (and a conforming malicious) user will send n packets while solving a sub-puzzle for each of them. Since the timestamp t_0 at which the puzzle was generated is always echoed in the sub-puzzle solutions (and cannot be manipulated on a correct solution), the service can estimate how long did it take the user to produce a solution for a sub-puzzle of a certain difficulty as well as the rate at which that user is sending her packets. We illustrate the estimation process in Figure 4.4.

Let h_i be user i's hashing rate and $X_j(d)$ be a random variable representing the time

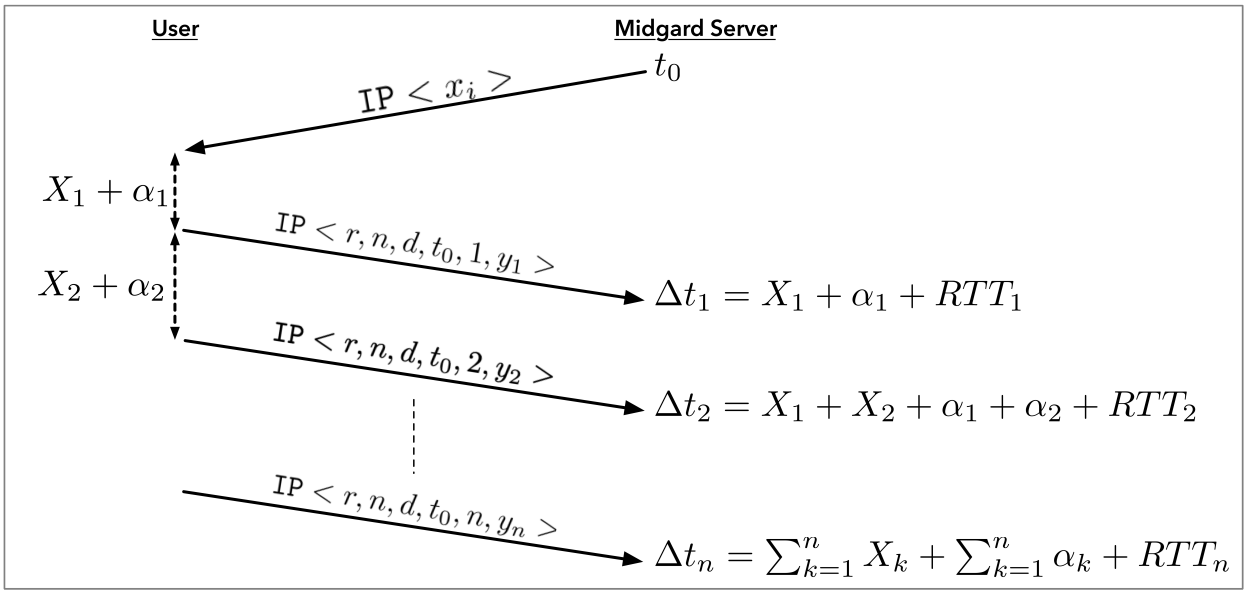


Figure 4.4: Hashing and flooding rate estimation process.

needed by client i to produce a solution for sub-puzzle j at a difficulty d. For ease of notation, we will drop the (d) indexing since the puzzle difficulty is always known to both the user and the MIDGARD service. From the puzzle definition in [71] and the results in [90], we know that the X_j's are i.i.d $\sim$ Uniform$(0, \frac{2^d}{h_i}]$ random variables with $E[X_j] = \frac{2^{d-1}}{h_i}$, $\forall j$. Additionally, let $\alpha_j \geq 0$ be a random variable representing the time interval that the user holds off on sending packet j after having solved its corresponding sub-puzzle. In other words, a user, after sending packet j, will wait $X_{j+1} + \alpha_{j+1}$ before sending packet $j+1$. We note that the MIDGARD service does not know the distribution of the α_j random variables as they are dependent on the application and the behavior of the users. In addition, malicious users would want to get $\alpha_j \searrow 0$, $\forall j$.

Finally, let $\widehat{RTT}_j$ be the round trip time taken between the time the MIDGARD service first sent the puzzle nonce x_i and the time it received the solution y_j. We note that $\widehat{RTT}_1 = RTT_i$ where RTT_i is the round trip time between the MIDGARD service and the user, which can be estimated using TCP packets [113, 114] or through dedicated services [115, 116]. This model illustrates an important feature of client puzzles, namely that they impose physical constraints on any bot machine to flood the network. Since the random variables X_j depend on the computational prowess of the machine performing the puzzle computation, for a fixed difficulty d, the attacker's flooding rate is bound by the computational limitations of her botnet machines.

Estimation. Let t_j be the time when the MIDGARD service receives a packet containing

a solution y_j, we can thus compute

$$\Delta T_j = t_j - t_0 = \sum_{k=1}^{j} X_k + RTT_j + \sum_{k=1}^{j} \alpha_k, \text{ for } 1 \leq j \leq n \tag{4.5}$$

Then by taking pairwise subtractions of observed ΔT_j values, we can build the following sequence

$$\begin{cases} \delta_1 := \Delta T_1 = X_1 + \alpha_1 + RTT_i, \\ \delta_j := \Delta T_j - \Delta T_{j-1} = X_j + \alpha_j \quad + \underbrace{\left(\widehat{RTT}_j - \widehat{RTT}_{j-1}\right)}_{\epsilon} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } j > 1 \end{cases} \tag{4.6}$$

We note that $\widehat{RTT}_j - \widehat{RTT}_{j-1} = \epsilon$ is the instantaneous packet delay variation from the user to the MIDGARD service (often referred to as *jitter*).

Equation (4.6) translates to the MIDGARD services obtaining a series of observations $\{\delta_1, \delta_2, \ldots, \delta_n\}$ that it can use to estimate the user's computational power as well as their sending rates. Since at every observation, the system of variables to be estimated is always *under-determined*, MIDGARD cannot estimate both X_j and α_j. Instead, we use the observations to estimate the sum $X_j + \alpha_j$.

We implemented MIDGARD in a modular way that allows ISPs to implement different estimators given the knowledge they have about the network and their users. For the purposes of this chapter, we describe one of those possible estimators.

Our MIDGARD estimator draws on the approaches to estimate a TCP packet's round-trip-time to infer network congestion [52]. Upon receiving its first observation δ_1 from user i, the MIDGARD server computes its first estimate $\hat{\lambda}_1$ by simply subtracting its previously known from δ_1. Subsequently, MIDGARD uses the low-pass filter to adjust its estimates using the low-pass filter update rule as shown in Equation (4.7).

$$\begin{cases} \hat{\lambda}_1 = \delta_1 - RTT_i \\ \hat{\lambda}_j = \beta \times \hat{\lambda}_{j-1} + (1 - \beta) \times (\delta_j - \delta_{j-1}), \text{ for } j > 1,\ 0 < \beta < 1 \end{cases} \tag{4.7}$$

By using the low-pass filter with $\beta > 0.5$, the MIDGARD estimator is less sensitive to fluctuations in the clients' computation times $(X_{j+1} + \alpha_{j+1} - (X_j + \alpha_j))$ by placing more emphasis on the prior estimates. This is particularly important since the X_j random variables are uniformly distributed over the range $(0, \frac{2^d}{h_i}]$, for a difficulty d and hashing rate h_i. Considering the worst-case scenario where an attacker attempts to flood the victim server (*i.e.* $\alpha_j = 0,\ \forall j \geq 0$), $X_{j+1} - X_j$ has a triangular distribution over the interval $[-\frac{2^d}{h_i}, \frac{2^d}{h_i}]$.

This fact can introduce higher variability in the observed differences $\delta_{j+1} - \delta_j$ and cause the estimator to fluctuate. The low-pass filter counters this behavior by placing more emphasis on the previously computed estimates and lesser emphasis on the differences in observations.

Computing and enforcing traffic policies. As discussed earlier, MIDGARD allows the server seeking protection to choose their own puzzle-based traffic policing mechanisms. In other words, given the estimated values for $\hat{\lambda}^i_j$, the ISP customers can choose the mechanism that assigns capabilities and allocates the appropriate share of bandwidth for each user. This approach is similar to that presented in [104] aiming to enforce destination-based policies at cloud-based protection providers.

In this chapter, we provide an illustrative policy that assigns each user a share of the bandwidth that is inversely proportional to its sending rate. Specifically, the MIDGARD server assigns a weight ω^i_j for each user i's packet j, and then instructs the downstream routers to perform *weighted-fair-queueing* based on the assigned weights.

Let ω^k be the weight that the MIDGARD server assigned to user k's last packet. Upon receiving packet j from user i, MIDGARD updates i's weight to

$$\omega^i_j = \frac{\hat{\lambda}^i_j}{\sum_k \lambda^k} \tag{4.8}$$

To enforce the traffic policy, the MIDGARD service will stamp all downstream packets pertaining to user i with its current computed weight ω^i_j. If user i behaves in a regular predictable way, then its weights $\{\omega^i_1, \ldots, \omega^i_n\}$ will not change much and due to the design of MIDGARD's estimator, will minimally be affected by path instability.

The routers downstream of MIDGARD will parse the packet weight and perform *weighted-fair-queueing* on all packets destined to the victim IP address. For those packets that do not contain weights, but still have their ECN bits set, the downstream routers will treat those packets as best-effort packets.

Finally, we note that throughout the period where the attack is taking place, we configure the ISP's edge routers to drop any packets destined to the victim's IP address that arrive on an ingress port. Ignoring packets those packets *serves the dual purpose of preventing cloud-bypass attacks as well as preventing an attacker from assigning false weights to its traffic.* Note that unlike in previous work [29, 30, 103, 104], since the MIDGARD service and the downstream routers reside within the same trust boundary, we do not need explicit authentication on and thus do not require a key management infrastructure. We believe that the benefits of MIDGARD's ISP-level deployment alleviates many of the challenges facing the deployments of previous DDoS defense mechanisms.

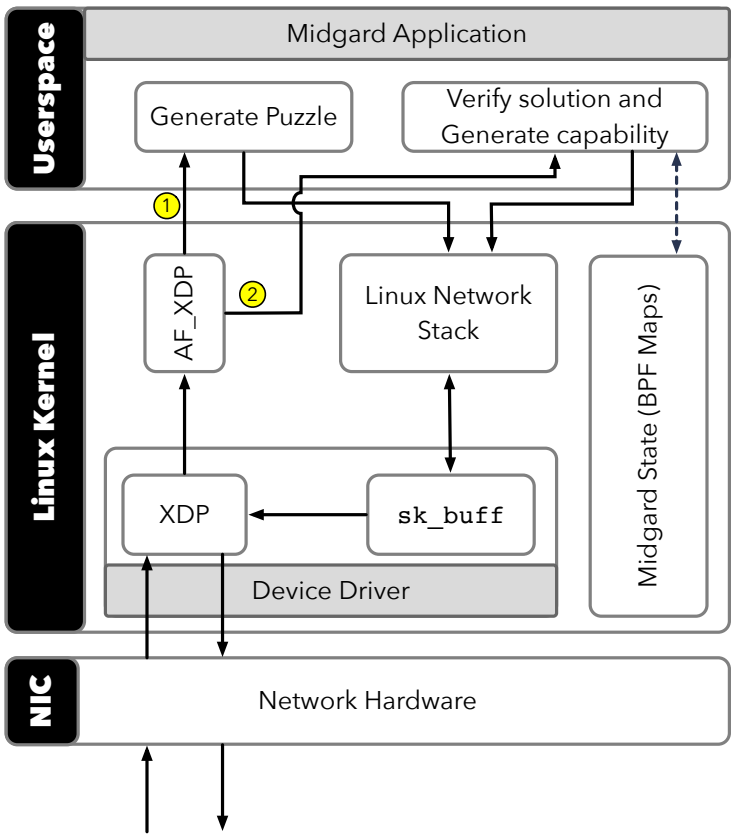


Figure 4.5: Workflow implementation of MIDGARD using `AF_XDP`. Dashed arrows correspond to data communication between the user application and the kernel maps. This workflow shows only the lifetime of the packets that are destined to the victim. After passing through the network hardware, packets bypass the Linux kernel's networking stack and are directly passed to userspace. Packets that follow the path labeled with (1) are packets that do not contain nonces and thus require MIDGARD to generate a new puzzle. On the other hand, packets following the (2) path contain nonces and thus will undergo the verification and capability generation procedures.

4.5 IMPLEMENTATION

We implemented MIDGARD using the capabilities provided by the Linux `eXpress Data Path` (XDP) and its associated userspace `AF_XDP` sockets. XDP allows for safe, fast, and programmable packet processing by integrating a limited in-kernel virtual machine (namely the *extended BSD packet filter*, `eBPF`) with userspace applications without sacrificing security and isolation [117]. By leveraging `XDP`, the MIDGARD proxy implementation bypasses the kernel networking stack and allows for zero-copy packet processing by the MIDGARD application. This optimization, in turn, reduces the possibility that MIDGARD's would become a bottleneck in the DDoS defense mechanism.

Figure 4.5 shows the path of a packet as it passes through the MIDGARD server. For

brevity, we only show the path taken by packets destined to the victim server. After the network hardware captures the packet, it passes it to the Linux kernel for processing. The `XDP` module sits within the networking device driver and bypasses the need for creating an `sk_buff` structure as is typically performed. Instead, the module passes the packet's memory area ownership to the userspace MIDGARD application through the use of the `AF_XDP` interface. Packets that do not contain a client nonce and a puzzle solution follow path ① where the application will generate a puzzle nonce, send it back to the client, and forward the packet in a best-effort manner through the Linux network stack using raw sockets. On the other hand, the application verifies packets that contain solutions (path ②) and generates capabilities for the packet's associated client nonce. The application then forwards the packet to the destination using raw sockets through the Linux network stack. In Section 4.6, we show that our implementation adds little overhead to the packet processing pipeline.

Maintaining State at the Server. For each client the solves puzzles, the MIDGARD server must maintain some metadata in order to process the requests and generate the capabilities. However, we note that the puzzles protocol itself is stateless, *i.e.* the server does not need to maintain the created nonce to validate a solution; it can recreate the original nonce from the solution itself and then validate it in two hashing operations.

As described in Section 4.4, to detect reused puzzles, the MIDGARD server maintains an 8 bit counter that is reset every 256 packets. Any packet received with a puzzle index that is less than the current counter would be discarded. However, when implementing this set up in practice, we noticed that packets might arrive out-of-order due to network delays. Therefore, using a monotonically increasing counter risks dropping packets that contain valid solutions. To address this challenge, we modified our approach as follows. Instead of an 8 bits counter, the MIDGARD server maintains a circular 64-bit vector ℓ for each sender s_i. For any puzzle solution y_j, the bit ℓ_j is set if and only if y_j has been received and validated. We show the details of our modified implementation in Algorithm 4.1. To allow for packets to arrive out-of-order, we split the 64-bit vector into four buckets, each of width 16 bits. Upon receiving solution y_j, the MIDGARD server drops the packet if bit j is set. Otherwise, we first set bit j, then clear the bucket $(\frac{j}{16}+1)\%4$ and invalidate the bucket $(\frac{j}{16}+2)\%4$. By doing this, the server guarantees that only the 16 packets from the bucket $(\frac{j}{16}-1)\%4$ to arrive out-of-order, while earlier packets are dropped. On average, this approach guarantees that a packet can be delayed by up to $8 \times RTT$, which is enough to account for lost or delayed packets under TCP. Unlike previous approaches [30], our implementation does not store any solutions and only requires tracking a 64 bit value for each sender.

Overall, for each user, the MIDGARD server maintains 336 metadata bits composed of

Algorithm 4.1: Checking for solution reuse in MIDGARD.

```
1  count ← bitvec < 64 > ()
2  Func CheckForReusedSolutions
       Inputs  : packet p, solution y_j
       Outputs: {ACCEPT, DROP}
3      mask ← 1 << j
4      if count ∧ mask then
5          DropPacket(p)
6          return DROP
7      else
           /* clear our the next bucket                              */
8          bucket ← (j/16 + 1) %4
9          mask ← ∼(0xFFFF << (bucket ∗ 16))
10         count ← count ∧ mask
           /* disable the bucket after                               */
11         bucket ← (j/16 + 2) %4
12         mask ← (0xFFFF << (bucket ∗ 16))
13         count ← count | mask
           /* set the matched entry                                  */
14         mask ← 1 << j
15         count ← count | mask
16     return ACCEPT
```

the following fields: 32 bits nonce, 32 bits timestamp to record the time the last puzzle was generated, 64 bits to track the number of packets sent by this user, 64 bits to track the number of bytes sent by this user, 64 bits for the puzzle reuse counter, 8 bits for the puzzle difficulty, 8 bits for the number of sub-puzzles the user committed to, and finally a 64 bit pointer to a timer function. To account for users dropping out or merely ending their sessions, the MIDGARD server maintains a timer for each user. That timer is reset every time a new valid puzzle solution is received. One the timer expires, all metadata corresponding to that user is deleted, and the memory can be reused for new users.

We believe that 336 bits are an acceptable state to maintain for each user for the following reasons. First, since we implement MIDGARD on a dedicated server running Linux, we benefit from access to large amounts of free memory (in the order of GB to TB). Second, even the MIDGARD is managing 1 million simultaneous active users, the total memory needed is $10^6 \times 336 = 420\texttt{MB}$, which can be easily supported by any standard Linux machine, let alone a dedicated server.

Client-side Implementation. MIDGARD requires clients to parse the puzzle packets and to generate solutions for each IP packets sent subsequently. Therefore, we wrote a kernel patch to the Linux 4.18 kernel that allows any general-purpose machine to parse IP options and recognize puzzle nonces sent from a server. If the kernel detects a puzzle nonce, it

sets a flag in the corresponding socket's data structure and saves the nonce along with its corresponding timestamp. Anytime an application wants to send an IP packet, the kernel checks the puzzle flag in the socket and creates a new solution if the nonce has not yet expired. If the nonce has expired, the kernel sends does append any options, signaling to the MIDGARD server that it is requesting a new nonce. In our implementation, we allow the users to change the number of puzzles to solve as well as the difficulty of every puzzle through the use of the kernel's `sysctl` interface.

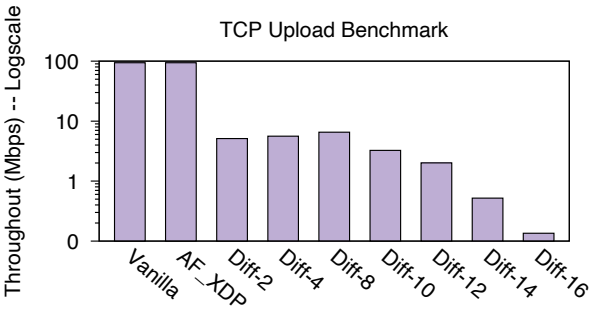
Deployability. We designed MIDGARD to be easily deployable within an ISP network. With the advent of SDN and *Network Function Virtualization* (NFV), ISPs can easily deploy a MIDGARD server as a virtual function that can be spun anytime a DDoS attack is detected against a protected victim. In addition, an NFV deployment of MIDGARD allows ISPs to easily scale up against larger attacks by spawning additional virtual appliances, as described in [33]. ISP customers that subscribe to the MIDGARD protection service need not make any changes to their software or infrastructure. They should only provide custom traffic policing policies that the ISP can deploy on the MIDGARD servers.

On the other hand, clients that wish to communicate with a server that uses the MIDGARD protection service must be able to support client puzzles at the IP layer. To that end, the server can distribute our simple Linux kernel patch to clients that wish to add this functionality. Alternatively, to accommodate users behind firewalls, legacy devices, and low-powered IoT devices, the ISP can designate puzzle *supernodes* to act as proxies on behalf of such clients, similar to those deployed by Skype [118], PlanetLab [119], and in [120]. In such a case, instead of communicating directly with the victim server, clients would be redirected to a puzzle supernode that will solve puzzles on their behalf and tag their packets with the corresponding solutions. However, this mechanism comes at the cost of requiring authentication to make sure that bot machines do not use supernodes as part of their DDoS attack. We plan to explore the challenges of using puzzle proxies in our future work. Finally, we note that by using the IP options, our implementation does not cause packet drops along the paths between clients and the server since routers typically ignore such option fields.

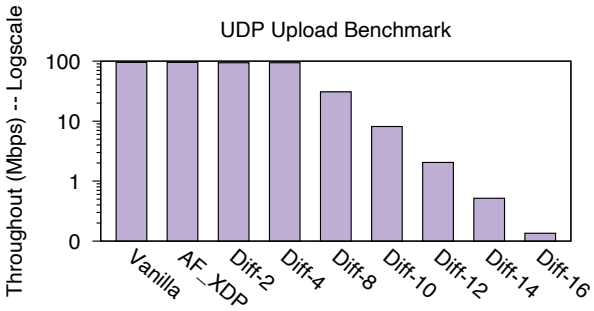
4.6 EXPERIMENTS

We now turn to evaluate MIDGARD and measure its impact on the resiliency of our networks against DDoS attacks. Therefore, in this section, we set out to answer the following research questions.

RQ 4.1 What is the overhead of implementing MIDGARD on the operation of a network?



(a) Throughput at a client running a TCP upload stream.



(b) Throughput at a client running a UDP upload stream.

Figure 4.6: Performance evaluation of TCP and UDP upload streams. Vanilla refers to the default Linux routing implementation, `AF_XDP` refers to the MIDGARD server running without any puzzles, and `Diff-XX` refers to the client solving puzzles of difficulty `XX`. Our results show that MIDGARD does not incur a performance penalty when compared to the default Linux kernel routing implementation. MIDGARD is more UDP- friendly as the puzzle expirations cause TCP streams to shrink their congestion window to accommodate the resulting packet retransmissions.

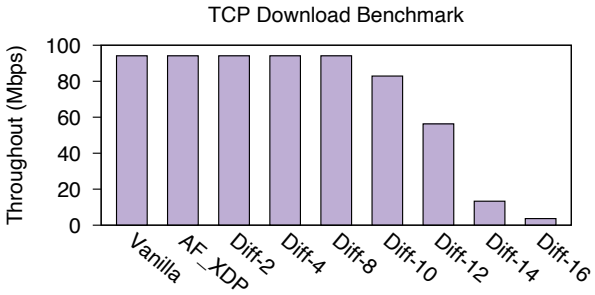
RQ 4.2 What is the overhead of running the MIDGARD kernel patch on the users of a service that employs the MIDGARD protection?

RQ 4.3 How does MIDGARD distribute bandwidth when under attack?

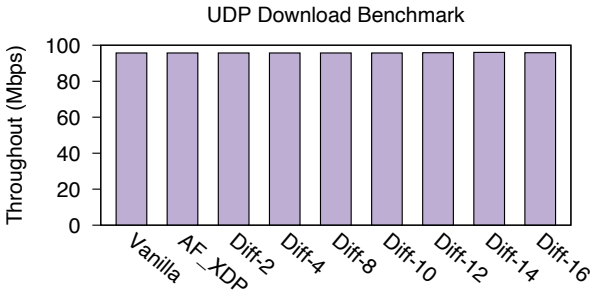
To answer these questions, we implemented the MIDGARD service on an Ubuntu Linux server running Ubuntu 18.04 with 16 processors and 16 GB of RAM. Since it is unethical and impractical to launch massive DDoS attack in thewild, we ran all of our experiments in an isolated environment on the `DETER` testbed. To capture the intended behavior of a typical MIDGARD implementation, we set the bandwidth of the MIDGARD server's ingress to 1Gbps. At the same time, the bottleneck link at the victim has a much lower bandwidth of 10Mbps. This setup makes it easy to overload the victim's ingress without affecting the links driving traffic into the MIDGARD server, as would be the caste with over-provisioned cloud providers and ISPs.

4.6.1 Performance Benchmarks

To answer **RQ 4.1** and **RQ 4.2**, we ran performance benchmarks using the `iperf` [121] network performance measurement tool. Figures 4.6a and 4.6b show the performance results of running TCP and UDP upload benchmarks, respectively. Under this benchmark, a client attempts to send as much traffic as possible to the victim server using the appropriate



(a) Throughput at a client running a TCP download stream.



(b) Throughput at a client running a UDP download stream.

Figure 4.7: Performance evaluation of TCP and UDP download streams. Vanilla refers to the default Linux routing implementation, `AF_XDP` refers to the MIDGARD server running without any puzzles, and `Diff-XX` refers to the client solving puzzles of difficulty `XX`. Unlike the upload benchmarks, TCP streams are not affected by the presence of the puzzles when the client is solving at a low difficulty (< 10 bits). When the puzzles become more difficult, the client's acknowledgment packets will be delayed, thus causing a drop in throughput.

protocol. Similarly, Figures 4.7a and 4.7b show the performance results of running TCP and UDP download benchmarks, respectively. Under this benchmark, a client attempts to download as much data as possible from the victim server using the appropriate protocol.

First, in all of our benchmarks, running the MIDGARD server without any puzzle requests incurs no performance penalty. In such cases, the MIDGARD server bypasses the Linux kernel networking stack, performs a table lookup for the victim server's IP address, and then forwards the packet as is on the corresponding interface. The fact that MIDGARD bypasses the kernel's networking stack and uses zero-copy to transfer the packet to user space allows it to offset the performance overhead to the table lookups. Therefore, our implementation highlights the benefits of using `AF_XDP` sockets for running network applications at line-rate.

Second, we note that our current MIDGARD implementation is not friendly to TCP streams. Specifically, our TCP upload benchmark results (Figure 4.6a) show a sever drop in throughout at the client, even at small puzzle difficulties. Upon further investigation, we discovered that MIDGARD's puzzle expiration and redistribution mechanism introduces a periodic small time window in which packets would be dropped. For that duration, the MIDGARD server has marked the puzzle as expired, but the new nonce has not been updated at the client yet. Therefore, packets generated from the client are still using the previous nonce, and thus fail the server's validation checks. Therefore, TCP will detect those dropped packets and interpret them as a sign of network congestion, thus reducing its congestion window and causing the observed drop in throughput. One possible remediation for this behavior is for the MIDGARD server, at each nonce renewal, to save the previous

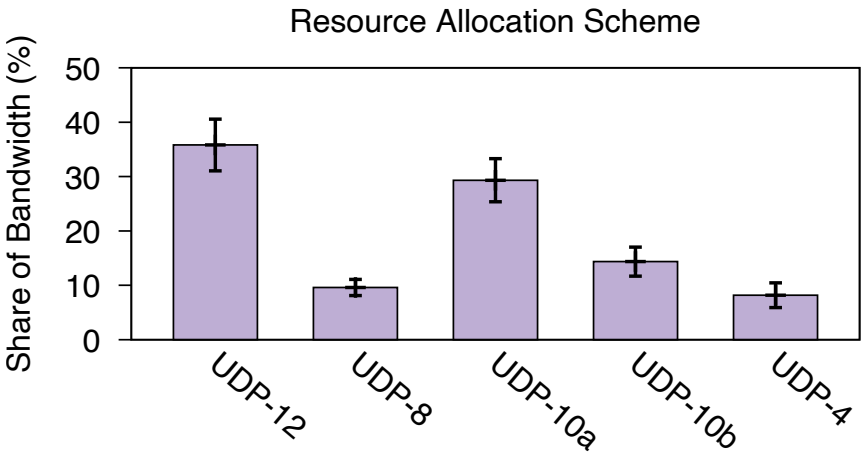


Figure 4.8: Bandwidth allocation results for the MIDGARD puzzle-based weighted fair queueing scheme. `UDP-XX` refers to a client attempting to send UDP traffic to the victim server, at its maximum bandwidth possible, while solving puzzles of difficulty `XX` bits. The results show that the MIDGARD server allocates more bandwidth to clients that solve harder puzzles (thus sending traffic at a lower rate).

nonce for a short time period. During this period, the server will attempt to validate puzzle solutions using both the current and the previous nonces, thus avoiding dropping TCP packets while the client updates its nonce. However, this will introduce an additional memory overhead since the server must maintain two nonces (*i.e.* two timestamps) for each client. We plan to explore this tradeoff in our future work.

Finally, unliked TCP, UDP streams dot not suffer from the same performance drop since they do not perform any congestion control. As shown in Figure 4.6b, no drop in throughput is observed at the client when the puzzle difficulty is low (< 8 bits). When the difficulty of the puzzles increases, the client will need to wait larger and larger amounts of time before sending each packet, effectively rate-limiting a flooding client, which is the intended behavior of the client puzzles.

4.6.2 Bandwidth Allocation

To answer **RQ 4.3**, we turn to studying the bandwidth allocation scheme of MIDGARD's puzzle-based weighted fair-queueing scheme. Figure 4.8 shows the bandwidth allocation scheme of MIDGARD's mechanism when faced with 5 clients that are attempting to flood the victim server with UDP traffic. Each client attempts to send as much traffic as possible while solving puzzles at different difficulties. Our results show that clients solving harder puzzles (`UDP-12, UDP-10a,` and `UDP-10b`) occupy the larger share of the bandwidth, while clients solving easier puzzles (thus sending at higher rates) end up with the smaller share of the

bandwidth ($< 10\%$). Our results show that, by combining client puzzles with simple weighted fair queueing, the MIDGARD server can reward users that are investing more resources into sending more difficulty puzzles with a higher share of the bandwidth. On the other, users that are solving easier puzzles, or alternatively are using more powerful machines to flood the victim server, end up with a lower share of the bandwidth thus limiting the impact of their attack. This naive bandwidth allocation scheme showcases that the MIDGARD server achieves a notion of puzzle-fairness by encouraging users to invest their computational resources into solving harder puzzles and benefit from a higher share of the bandwidth.

4.7 CONCLUSION

In this chapter, we presented MIDGARD, a DDoS resiliency service that combines the benefits of cloud elasticity and provisioning, client puzzles, and network capabilities to enhance network resiliency. MIDGARD is a service that resides at the edge of a victim server's ISP and serves to absorb the attack traffic and rate-limit misbehaving users. When under attack, MIDGARD asks each user to solve a computational puzzle with a user-chosen difficulty, and then uses each packet's travel time to estimate that user's computation prowess. Based on its estimates, the MIDGARD then assigns each packet a capability that indicates the share of bandwidth that each user can use. We designed MIDGARD as a flexible and extensible service that allows customers to implement their traffic estimation and bandwidth allocation policies at the edge of their upstream ISP. We implemented MIDGARD using the `AF_XDP` sockets in the Linux kernel that allow us to keep the overhead of our implementation to a minimum. We deployed MIDGARD on an experiment network on the `DETER` testbed and evaluate it under different attack scenarios. Our results show that MIDGARD can effectively absorb volumetric attacks and rate-limit attackers while keeping its performance overhead to a minimum. We envision that MIDGARD would be implemented as a service provided by ISPs to their customers to defend against DDoS attacks.

CHAPTER 5: BIFROST: CIRCUIT-LEVEL VERIFICATION OF DATA PLANE PROGRAMS

5.1 INTRODUCTION

Programmable data planes allow for a significant shift in the paradigms of modern network designs. Data plane-specific programming languages, such as P4 [122], allow network designers to design, develop, modify, and test packet-forwarding protocols and pipelines in a hardware-agnostic manner. This has led to an increase in the flexibility and programmability of network design.

Data plane descriptor languages are more limited in their functionality and semantics than other general purpose languages, therefore one might assume that they are more resistant to security risks. However, languages such as P4 also expose a whole new set of potential bugs in their implementations. For example, data planes programs that are compiled and programmed onto target devices require a mechanism for the CPU and the networking hardware to communicate. This is often implemented using a specialized packet header that the packet parser can recognize and forward accordingly. Such packets contain important metadata information, such as the value for the ingress port, that are to be written onto the switch's metadata registers. In current implementations of the popular `switch.p4` program, such packet headers are not sanitized. Consequently, an attacker can craft a CPU header, prepend it onto its packets, and overwrite each of their packets' ingress post in order to bypass *Access Control Lists* (ACLs) and perform privilege escalation [123].

On a larger scale, undiscovered bugs in networked systems can have drastic impacts on performance and security [124, 125, 126]; experts estimate the cost of downtime in data centers to be $7,900 per minute [11]. Data plane programmability may worsen this problem by introducing new classes of bugs largely prevented by traditional fixed-function switch hardware, such as overwriting header fields[1] and invalid headers due to read-before-write[2] [127]. With our ever-increasing dependence on networked systems across all sectors of our economy and lives, it becomes increasingly important to ensure that any increase in function does not lead to any sacrifices in security and reliability.

Current approaches to testing programmable data planes perform exhaustive testing using simulation and automatic packet generation software [128]. However, due to computational and time limitations, complete coverage of all execution paths can never be achieved using testing alone. Additionally, such testing approaches can confuse bugs that occur in

[1] https://github.com/p4lang/switch/issues/97
[2] https://github.com/p4lang/switch/pull/102

the data-plane program with those that occur due to hardware-specific compilation. A complementary strategy is to perform static verification leveraging *satisfiability* (SAT) and *Satisfiability Modulo Theory* (SMT) solvers to verify annotated data-plane programs. Several approaches [128, 129, 130, 131, 132] use symbolic execution to perform verification. However, these techniques become language dependent; each data-plane programming description requires different software and verification techniques. More fundamentally, these existing techniques target Turing complete languages. However, data plane programs describe bounded and restricted hardware pipelines that are simpler to verify.

In this work, we explore the fundamental property of data plane programs – namely, that they describe restricted hardware pipelines – to achieve scalable verification. We present BiFrost: a tool for the formal verification of data plane programs using hardware verification techniques. BiFrost translates data plane programs into *functionally equivalent* sequential circuits and uses well-established sequential circuit abstraction and verification techniques to achieve expressive and scalable static verification. Specifically, the boundedness and the features of data plane programs make them ideal candidates for BiFrost's translation: they operate on bounded inputs, their loops are linearly bounded by the number of packet headers, and they do not use dynamic memory allocation. BiFrost specifically targets P4 programs, however, it can be easily extended and applied to different languages since it performs verification at the hardware level.

Our goal in designing BiFrost is to achieve scalable and effective verification of data plane programs to detect and fix design-time programming bugs. To achieve that goal, we adopt the following approach:

Bit-level Semantics. By translating data plane programs into functionally equivalent sequential circuits, BiFrost can reason about packets in terms of the individual bit fields. Rather than looking at high-level descriptions of packet headers and their fields, BiFrost can reason about bits that cross the boundaries of headers and their fields. Furthermore, depending on the target property it is trying to verify, BiFrost can identify individual bits that are deemed to be non-relevant for the verification tasks. For example, if a target property depends only on the lower 8 bits of an IP address, BiFrost can identify that the first 24 bits of the address are irrelevant and replace them with *don't care* values.

Support for Various Specification Types. To provide developers with the flexibility of defining various properties about their data plane programs, we design BiFrost to support *first order logic* (FOL) properties expressed at multiple program locations. In addition to built-in header validity checks, BiFrost supports `guarantee` and `assert` statements as well as program pre- and post-conditions.

Domain-specific Transformation and Abstraction. We leverage knowledge about the data plane programs' functioning and boundedness to generate sequential circuits that are optimized for scalable verification. For example, since parser loops are linearly bounded, we unroll all such loops and avoid generating complex circuitry that is hard to verify. In addition, we provide means to abstract the control plane tables, thus incurring possible false positives, in favor of proving properties for all possible values in such tables.

Sequential Verification. By targeting sequential verification, BiFrost unlocks a plethora of optimization and abstraction techniques that can aid the verifier in achieving scalable verification. Those techniques, both structural [133, 134, 135] and functional [136, 137, 138, 139, 140], often have no counter-parts in traditional software verification.

In this chapter, we make the following contributions. We design and build BiFrost, a tool for the scalable verification of data plane programs using sequential circuit verification techniques. First, we define the operational semantics of data plane programs in terms of sequential circuits (Section 5.5.1). We introduce the BiFrost transformations that generate a functionally equivalent sequential circuit to an input data plane program (Section 5.5.2). We introduce optimizations and abstraction techniques (Section 5.6) to aid in scaling to larger programs and more complex properties. We implement BiFrost (Section 5.7) and conduct an exhaustive evaluation of its scalability using 11 benchmark programs. Our experiments show that BiFrost can verify the header-validity of complex programs in < 3 minutes. We also showcase two case studies (Section 5.8) where we used BiFrost to (1) detect a header-validity bug in the `ecn.p4` tutorial program and (2) verify functional properties about the `calc.p4` program.

5.2 BACKGROUND ON PROGRAMMABLE DATA PLANES

Software Defined Networking (SDN) decouples traffic decisions, made in the control plane, from traffic forwarding, made in the data plane. A logically centralized controller makes traffic forwarding decisions (*e.g.*, computing routes, load balancing, enforcing security policies) and populates the tables in the data plane via the *southbound API*. Traditionally, forwarding devices have been regarded as fixed-function hardware pipelines that operate on a fixed set of packet headers. However, recent efforts have focused on extending the programmability into the data plane to support custom packet formats, new networking protocols, line-rate applications [141, 142, 143], and security countermeasures [144, 145, 146, 147].

Data plane programming languages, such as P4 [122, 148] and NPL [149], grant developers the ability to program packet forwarding devices using abstract declarative constructs

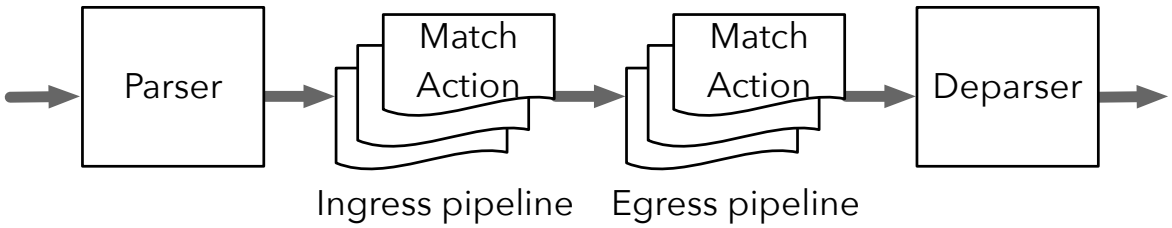


Figure 5.1: An abstract pipeline of a simple programmable switch.

that are later compiled onto a device's hardware. Figure 5.1 illustrates the abstract packet forwarding pipeline of a simple programmable switch. Incoming packets are first handled by a parser that operates on user-specified header formats. After extracting the headers and setting appropriate metadata fields, the parser forwards the packet into one or more match-action pipelines (the `ingress` and `egress` pipelines in Figure 5.1). During this stage, the switch queries a set of read-only match-action tables, populated at runtime by the control plane, that determine what actions are to be undertaken depending on the values contained in the packet's headers. Actions can manipulate packet header fields (such as performing MAC address swapping) or packet metadata (such as setting the packet's egress port or marking it to be dropped).

Finally, the deparser is responsible for reassembling an output packet by serializing the packet's headers and its payload according to a user-specified order. The packet is finally queued into the appropriate egress port's queue awaiting to be sent on the wire. Unfortunately, in an effort to favor efficient compilation, data plane programming languages lack notions of type safety, thus introducing several classes of undefined behaviors and design-time bugs [129, 150]. Such bugs can lead to runtime failures such as wrong forwarding decisions, forwarding loops, or device crashes. More importantly, malicious attackers can exploit these flaws to launch impactful attacks such as denial of service or privilege escalation [123].

Comparison to Active Networks. Active networking [151, 152, 153] is an early approach to incorporate programmability and flexibility into network architectures. It allows network operators, developers, and even end-users, to encapsulate program fragments into communication packets that forwarding devices can parse and execute locally. To that end, routers and switches would run a network operating system that would enable them to execute the code fragments and implement applications such as content caching, firewalls, and quality of service mechanisms.

However, unlike SDNs and programmable data planes, active networks did not see widespread adoption and were later dropped by the networking community. First, active network do not present a clear-cut separation between forwarding decisions (the control plane) and for-

warding behavior (the data plane); it is up to the developer to identify where each operation is implemented thus creating several compatibility and reliability challenges. Second, and more importantly, by allowing forwarding devices to include program fragments, active networks pause critical security challenges. Making sure that only authenticated users can executed such programs on routers and switches would require authentication mechanisms. Such mechanisms, which typically rely on asymmetric key cryptography, can become bottlenecks on the forwarding critical path. In addition, if active networking is to be achieve widespread adoption, it is unclear who would the authority responsible for maintaining keys and certificates.

Finally, programmable data planes have benefited for recent and fast-paced developments in *Application-Specific Integrated Circuits* (ASIC) that allow for programmability without sacrificing performance. By requiring forwarding devices to run an operating system, active networking introduces a performance penalty that most network developers are not willing to pay. On the other hand, by leveraging the power of ASICs, forwarding devices can be programmed while maintaining line-rate processing at every point in the network. For example, the recent Barefoot Tofino ASIC [154] is fully programmable while maintaining a processing rate of 6.5 Tbps.

5.3 A CASE FOR HARDWARE VERIFICATION

A key characteristic of hardware description programs is that they must account for the physical limitations of their target device. That is, they must be tailored to model hardware. A circuit's capabilities are often vastly simpler than general-purpose software and therefore have vastly smaller state spaces. Therefore, when we consider verification approaches that search the state space of a program exhaustively, hardware is a much more scalable candidate.

We use this fact to relate this approach to data plane verification. Data plane programs specify bounded forwarding pipelines, even simpler than standard hardware, that describe the lifetime of a network packet. That is, a packet's journey from a switch's ingress point to its commitment to a specific egress port, so long as it is not discarded in case of errors or policy violations. Thus, it is appropriate and efficient to limit the scope of verification to the specificities of a switch's forwarding behavior throughout the lifetime of a packet.

In this section, we argue that data plane programs are an excellent contender for hardware verification. We first provide an introduction to the techniques used in such verification efforts, then showcase the benefits of using hardware verification in this context. For the remainder of this text, we use the terms hardware verification and sequential circuit verification interchangeably.

5.3.1 Introduction to Hardware Verification

Hardware verification enjoys a mature and rich history of development and deployment in both industrial and academic settings [155, 156, 157, 158, 159]. Its motivation stems from the costly fact that post-production, circuits and silicon chips cannot be patched upon the detection of a design-time bug; they must be recalled and replaced. For example, the infamous Intel FDIV bug is reported to have cost the company $475 million in replacement and reputation costs [160]. It is therefore of paramount importance to catch design-time bugs before production.

Hardware verification techniques aim to provide formal guarantees that a circuit satisfies its design specifications and to discover potential bugs. *Model Checking* is popular approach to general verification that is rooted in hardware. This technique exhaustively explores, either explicitly or symbolically, a system's state space to examine whether any reachable states can lead to a violation of the user specifications [161]. If any such states are found, a concrete counterexample can be returned to the developer for debugging. However, this technique is plagued by the *state-space explosion* problem where the number of reachable states in a program is exponential in several parameters (number of inputs, width of variable types, etc.), thus hindering the scalability and practicality of the verification efforts.

To improve scalability, a common technique is to impose limits on the space of the program. Several approximations and abstractions (such as *counterexample-guided abstraction refinement [162]*) are often employed to prune unnecessary explorations. Another common approach is explicit user annotations. Additionally, *Bounded Model Checking* (BMC) is an orthogonal technique that can be employed to find bugs in traces up to a specified length k of the program's execution. BMC is particularly popular in the hardware sphere because the precise state space of the hardware description is known at the time of synthesis. That is, because there are physical limitations of hardware programs, language features such as loops must be unrolled at the time of synthesis and the finite-state representation of the program is fixed.

The accessibility of a hardware program's state space is a key characteristic in other verification techniques. In addition to BMC and symbolic execution, circuit-specific reduction and abstraction procedures, such as structural analysis [133, 134], retiming [136, 137, 138], rewriting [135] and register sweeping [139, 140], are often employed to improve scalability.

5.3.2 Why Hardware Verification?

Building on the successes of hardware verification in improving the design-time correctness of sequential circuits, we argue that formal hardware verification is better suited for data

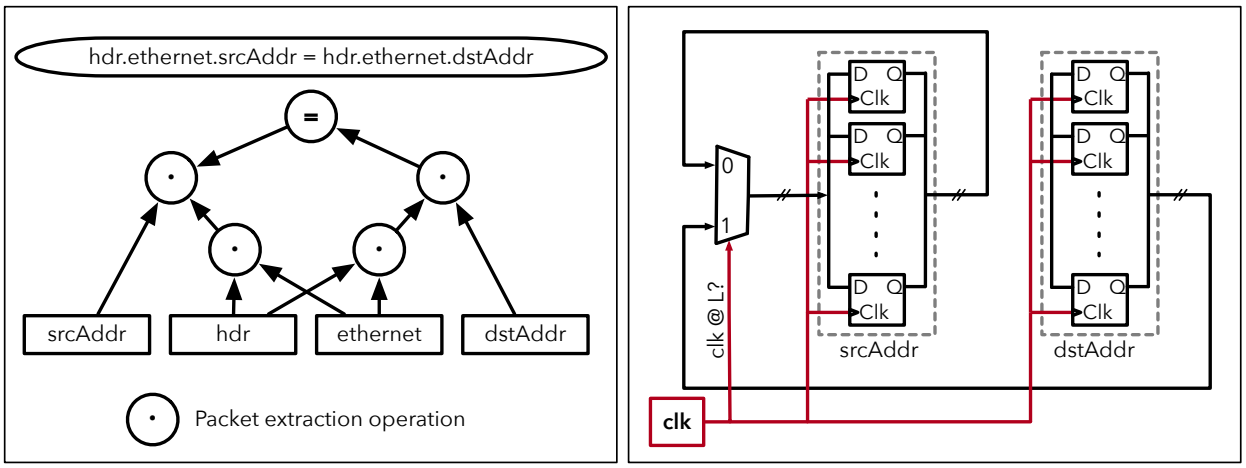


Figure 5.2: MAC address swap operation in BiFrost. Considered as a general purpose operation (left-hand side), the statement involves several operations for packet extraction and memory copy. Interpreted as a sequential circuit, the assignment operation can be done in a single clock cycle by simply wiring the `dstAddr` register as an input to `srcAddr` register.

plane verification than general-purpose software verification techniques. Hardware verification has made strides in exploring the large state space of its programs. It is more difficult for software verification techniques to seek the same extensive approach because their state space is exponentially more massive.

Specifically, as in [163], we identify two important properties that verification tools must achieve, namely *expressiveness* and *scalability*. In this chapter, we show that sequential circuit verification can be effectively used to achieve these properties in data plane programs.

Expressiveness. The expressiveness of a hardware verification formalism determines what constructs (*e.g.* data-types, formats, number representations, etc.) can be used in the verification process. This also determines the set of possible properties that the verification tool can reason about. For example, reasoning about timing and delays is necessary for sequential verification, while it is not a requirement for high-level programming such as `C` or `Python`.

Scalability. Verification techniques, both software and hardware, are plagued by several problems that hinder their scalability. First, determining the *satisfiability* of a Boolean formula (referred to as the SAT problem) is NP-complete. Second, more expressive languages are usually undecidable. Third, the number of states that a program or circuit can reach grows exponentially with the size of its inputs, and can also be infinite [164]. To combat those problems, verification tools often employ various abstractions and heuristics to verify larger and larger designs.

First, data plane programming languages (*e.g.* P4) are highly domain-specific. They pro-

```
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

Figure 5.3: Sample P4 code fragment for swapping MAC addresses.

vide a strict set of constructs that allow for the description of the packet processing pipeline in a packet forwarding device. Those constructs have therefore strict semantics that are tailored towards building a pipeline that is guaranteed to only execute a constant number of operations for each byte of an input packet [122]. Often, language features such as loops and pointers are omitted completely. Therefore, we argue that sequential circuit verification is more expressive than its software counterparts in capturing and reasoning about the semantics of data plane programs. For example, consider the code snippet shown in Figure 5.3 that performs MAC address swapping for an egress packet. Considered as a general purpose software program, this step involves multiple instructions to parse the content of the packet, extract the Ethernet header and the source and destination MAC addresses (e.g., `hdr.ethernet.srcAddr` and `hdr.ethernet.dstAddr`), and memory copy operations for swapping the addresses. This can complicate the verification process as the programs become larger and larger. On the other hand, leveraging the fact that a P4 program strictly describes a packet forwarding pipeline, we can consider the input packet to be simply a register, and then the process of parsing and swapping the MAC addresses as well as decrementing the `TTL` value can be done in a single shot (i.e., in a single clock cycle). Figure 5.2 illustrates how this process can be performed in a single clock cycle by considering the circuit properties of the P4 processing pipeline.

Second, sequential circuit verification of data plane programs allows for *bit-level* reasoning about the forwarding pipelines, allowing for several optimizations that have no counterpart in the software verification realm. For example, consider that a network operator is interested in proving a safety property about hosts that are in a specific subnet, say `192.168.100.0/24`. When performing verification, it is only the lower 8 bits of any packet's IP address that will have an impact on the safety property; the upper 24 bits will remain constant throughout. By reasoning at the bit-level, a hardware verifier can quickly realize that only the lower 4 bits of its inputs are important. It will then discard the upper 24 bits from the analysis, significantly reducing its search space compared to treating the IP address as a single variable, as a software verifier would. Therefore, bit-level analysis can greatly enhance the scalability of the verification process.

Finally, network operators often have networking devices from different vendors that use different compilations of the same data plane program. To ensure correct operation, they must then have guarantees that all of the devices are running a functionally equivalent version of the data plane program. By translating the original program into an equivalent circuit, hardware verifiers can allow operators to reason about the functional equivalence between different compilations (for different targets) of the same program. This allows developers to isolate unexpected behavior to the actual devices that the programs run on, rather than the logic of the program itself.

5.4 OVERVIEW OF BIFROST

Our key insight in this work is that *sequential synthesis, reduction, and verification techniques provide various avenues for scalable and expressive verification of data plane programs.* This insight rests on the observation that data plane programs, unlike general-purpose programs, describe bounded forwarding pipelines targeted for deployment on programmable hardware switches. Therefore, to avoid expensive network outages and security risks, sequential circuit verification is suitable for verifying data plane programs and uncovering design-time bugs.

BiFrost is composed of three main components, shown in Figure 5.4. Given a data plane program $\mathcal{P}$, BiFrost first employs several program transformation techniques (Section 5.5) to build a *functionally equivalent* sequential circuit $\mathcal{S}$. Functional equivalence guarantees that each and every execution path in $\mathcal{P}$ corresponds to a sequence of executions in $\mathcal{S}$ that generates the same output, and vice versa. Second, BiFrost accepts program specifications in the form of (1) guarantee statements, (2) assertions, (3) preconditions, and (4) postconditions, and generates verification conditions that are added as primary outputs in the circuit $\mathcal{S}$ (Section 5.6). In addition, BiFrost provides built-in support for *header-validity* checking to detect reads and writes to invalid header fields.

Finally, BiFrost passes the circuit $\mathcal{S}$, along with its primary outputs, to the sequential synthesis and verification tool `ABC` [165] to perform verification tasks. `ABC` is equipped with a wide variety of synthesis, reduction, and abstraction techniques that reduce the size of the circuit and constrain the verification search space. Subsequently, BiFrost asks `ABC` to check if there is any setting of $\mathcal{S}$'s inputs (i.e., an input packet) that will lead to the invalidation of any of $\mathcal{S}$'s primary outputs. This corresponds to a violation of the data plane program's specification. If no such setting is discovered, BiFrost marks the program as verified. On the other hand, BiFrost interprets `ABC`'s counterexample to generate a violating packet along with an execution trace that will allow developers to debug their implementations.

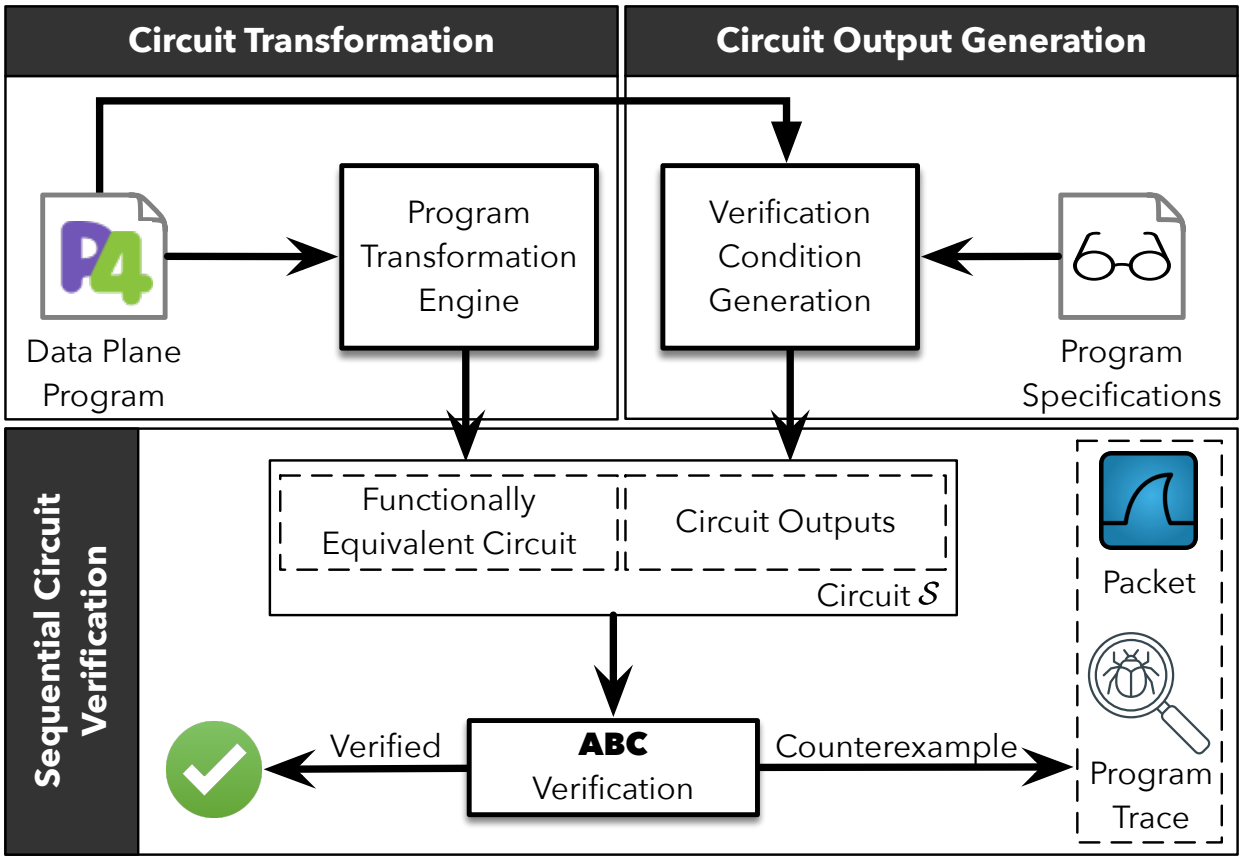


Figure 5.4: High Level Overview of BiFrost. Given a data plane program, BiFrost first translates it into a functionally equivalent sequential circuit. Then, it parses a set of program specifications, expressed in First Order Logic, to generate verification conditions. Those conditions are then translated into circuit components and connected as primary output to the generated circuit. Finally, BiFrost uses the `ABC` verifier to check if the specifications are satisfied, or to return a counter example, in the form of a network packet, for developers to debug their programs.

5.5 BIFROST SYSTEM DESIGN

In this section, we present BiFrost's main transformation of a P4 program into an equivalent sequential circuit. Our design of this system builds upon the specification provided in the official $P4_{16}$ specifications document [166] and are not specific to a certain target switch.

We first present definitions that we use to show that our transformation preserves P4's operational semantics and produces a sequential circuit that is functionally equivalent to the original program. We then elaborate to describe in detail how BiFrost implements and processes specific elements of a P4 program.

5.5.1 Definitions

We start by defining a sequential circuit and its semantics in the context of BiFrost. We then proceed to describe the transformations that, given a P4 program $\mathcal{P}$, produces a sequential circuit $\mathcal{S}$ that is functionally equivalent to $\mathcal{P}$.

Definition 5.1 (Sequential Circuit)**.** *A sequential circuit $\mathcal{S}$ is a tuple $(\mathcal{G}, \mathcal{R}, \texttt{clk}, \mathcal{I}, \mathcal{O}, \mathcal{E})$ where $\mathcal{G}$ is a set of logic gates and arithmetic blocks, $\mathcal{R}$ is a set of clocked registers,* `clk` *is a* monotonically increasing *clock, $\mathcal{I}$ is a set of inputs, and $\mathcal{O} \subseteq \mathcal{R}$ is a set of outputs. For brevity, we define $\mathcal{V} = \{\mathcal{G} \cup \mathcal{R} \cup \mathcal{I} \cup \mathcal{O}\}$ to be the set of* circuit components*. Therefore, $\mathcal{E} = \mathcal{V} \times \mathcal{V}$ is a set of edges that define the connectivity between the components of $\mathcal{S}$.*

A register $r^n \in \mathcal{R}$ is a set n of one-bit latches (or memory elements) such that the value held by each latch of r^n is updated every time the clock `clk` increments. For clarity, we abuse notation and drop the asterisk for registers, and use $|r| = n$ to represent the number of latches in a given register r. The set of edges $\mathcal{E}$ captures the logic connectivity of the different components of $\mathcal{S}$. In other words, $\mathcal{S}$ is a directed graph that represents the input-output relationships between the different components $\mathcal{V}$ of $\mathcal{S}$.

Definition 5.2 (Fanins and fanouts)**.** *Given a component $v \in \mathcal{V}$, we define the* fanins *of v as the set of components $\omega \in \mathcal{V}$ that have directed edges to v. More formally,* $\texttt{fanins}\,(v) = \{\omega \in \mathcal{V} \mid (\omega, v) \in \mathcal{E}\}$.

Conversely, the fanout of v is the set of components $u \in \mathcal{V}$ such that v has a directed edge to u. Formally, $\texttt{fanouts}\,(v) = \{u \in \mathcal{V} \mid (v, u) \in \mathcal{E}\}$

Definition 5.3 (Well formed circuits)**.** *A sequential circuit $\mathcal{S} = (\mathcal{G}, \mathcal{R}, \texttt{clk}, \mathcal{I}, \mathcal{O}, \mathcal{E})$ is said to be well-formed iff*

$$\forall(r \in \mathcal{R}, i \in \mathcal{I}, o \in \mathcal{O}).\ (|\texttt{fanins}\,(r)| = 1) \wedge (|\texttt{fanins}\,(i)| = 0) \wedge (|\texttt{fanouts}\,(o)| = 0) \quad (5.1)$$

Well-formed sequential circuits according to Definition (5.3) can be used to represent the functionality of any finite (or infinite) state program. In this work, we restrict our attention to well-formed sequential circuits.

Semantics of sequential circuits. We now present the semantics of sequential circuits to define *functional equivalence*. We then show how BiFrost's transformations create sequential circuits that are functionally equivalent to their P4 counterparts. The following definitions apply to a sequential circuit $\mathcal{S} = (\mathcal{G}, \mathcal{R}, \texttt{clk}, \mathcal{I}, \mathcal{O}, \mathcal{E})$.

Definition 5.4 (Valuation)**.** *Let* $\mathbb{B}$ *bet the set of Boolean values* $\{0,1\}$ *and* $\mathbb{N}$ *be the set of natural numbers. A valuation* ℓ *is a function* $\mathcal{V} \times \mathbb{N}^+ \to \mathbb{B}^{\mathbb{N}}$ *where*

$$\ell(v,c) = \begin{cases} b_0\ b_1\ \ldots\ b_{|v|} & \text{if } v \in \mathcal{R} \cup \mathcal{I} \cup \mathcal{O}, \\ \mathtt{G}\left(\ell(u,c) \mid u \in \mathtt{fanins}\left(v\right)\right) & \text{if } v \in \mathcal{G} \end{cases} \tag{5.2}$$

where $c \in \mathbb{N}$ *is the value of the circuit's clock,* $b_i \in \mathbb{B}$ *for* $i \geq 0$ *is the value of a single bit and* G *is the combinatorial operation represented by the gate* $v \in \mathcal{G}$*.*

As an example, the valuation of a conjunction gate g with fanins r_1 and r_2, at clock cycle c, is $\ell(g,c) = \ell(r_1,c) \wedge \ell(r_2,c)$.

Definition 5.5 (State)**.** *At a specific clock cycle* $c \in \mathbb{N}^+$*, the state of* $\mathcal{S}$ *is the set* $\sigma_c = \left\{\ell(r_1,c), \ell(r_2,c), \ldots, \ell(r_{|\mathcal{R}|},c)\right\}$ *of all valuations over the circuit's registers. For brevity, we write* $\sigma_c(r) = \ell(r,c)$*.*

Definition 5.6 (Trace)**.** *A trace* $\mathtt{tr}\left(\mathcal{S},k\right)$ *over* $\mathcal{S}$ *and of length* $k \in \mathbb{N}^+$ *is a sequence of states* $\langle \sigma_0, \sigma_1, \ldots, \sigma_{k-1} \rangle$*, such that,* $\forall r \in \mathcal{R}$*,*

$$\ell(r,c) = \begin{cases} \mathbf{r_0} \in \mathbb{B}^{|r|} & \text{if } c = 0, \\ \ell\left(\mathtt{fanins}\left(r\right), c-1\right) & \text{if } c \geq 1 \end{cases} \tag{5.3}$$

where $\mathbf{r_0}$ *represents the initial values of the latches of* r*.*

Similarly, given a P4 program $\mathcal{P}$ that operates on an input packet p and a match-action table $\mathcal{T}$, with syntax and semantics as defined in [148, 166], a state of $\mathcal{P}$ represents the values held by the header fields and internal structures (*e.g.*, `metadata`) of $\mathcal{P}$ at a given instant k. Thus a trace $\mathtt{tr}\left(\mathcal{P},k\right)$ is a sequence of states that capture the values of $\mathcal{P}$'s variable for $0 \leq j < k$.

Definition 5.7 (Functional equivalence)**.** *Given a P4 program* $\mathcal{P}$ *with a set of headers and internal variable* $\mathcal{X}$*, and a sequential circuit* $\mathcal{S} = (\mathcal{G}, \mathcal{R}, \mathtt{clk}, \mathcal{I}, \mathcal{O}, \mathcal{E})$ *are* functionally equivalent*, denoted as* $\mathcal{P} \approx \mathcal{S}$*, iff*

$$\begin{cases} \exists\, \mathtt{f} : \mathcal{X} \to \mathcal{R} \text{ s.t. } \mathtt{f} \text{ is injective,} & \text{and,} \\ \forall\left(\text{packet } p,\ \text{table } \mathcal{T}, k > 0, \mathtt{tr}_1 := \mathtt{tr}\left(\mathcal{P},k\right)\right). \\ \ \exists k_2 \geq k,\ \mathtt{tr}_2 := \mathtt{tr}\left(\mathcal{S},k_2\right)\ \text{ s.t. } \forall(0 \leq i < j \leq k, x \in \mathcal{X}). \\ \quad \exists\ m,n \text{ s.t. } 0 \leq m < n \leq k_2\ \wedge \\ \qquad \sigma_{1i}(x) = \sigma_{2m}(f(x)) \wedge \sigma_{1j}(x) = \sigma_{2n}(f(x)) \end{cases} \tag{5.4}$$

In other words, functional equivalence indicates that the circuit $\mathcal{S}$ captures the functional behavior of the P4 program $\mathcal{P}$; given the same set of inputs $\mathcal{P}$ and $\mathcal{T}$, there exists a mapping of $\mathcal{P}$'s variables $\mathcal{X}$ to registers $\mathcal{R}$ in $\mathcal{S}$, such that the execution of $\mathcal{P}$ on $\mathcal{X}$ yields the same states as the execution of $\mathcal{S}$ on $\mathcal{R}$. Therefore, both $\mathcal{P}$ and $\mathcal{S}$ would produce the same output packet, even if their execution paths are different.

5.5.2 Transformation

We now turn our attention to defining BiFrost's transformation Φ, such that given an input P4 program $\mathcal{P}$, $\mathcal{S} = \Phi(P)$ is a sequential circuit such that $\mathcal{P} \approx \mathcal{S}$. In what follows, we characterize a P4 program $\mathcal{P}$, accepting an input packet p, by its match-action tables $\mathcal{T}$ and its set of variables $\mathcal{X} = \{\mathcal{H} \cup \mathcal{M}\}$, where $\mathcal{H}$ and $\mathcal{M}$ are the set of header fields and standard metadata in $\mathcal{P}$, respectively.

Variables and Packets. In the context of the sequential circuit $\mathcal{S}$, all variables and their types are captured by collections of fixed number of one-bit latches (*i.e.* registers). The interpretation of these bits differ only in the context of the operations involving them (*e.g.* one's complement addition vs two's complement addition).

We start BiFrost's transformation by constructing the set $\mathcal{R}$. We create a register $r := \Phi(x)$ for each variable $x \in \mathcal{X}$, such that $|r| = |x|$ (*i.e.*, r is composed of $|r|$ one-bit latches). For each header $h \in \mathcal{H}$, we also create a one-bit register, denoted by $\Phi(h)$.`valid`, capturing the validity of h's extraction from the input packet p. To represent $\mathcal{P}$'s input packet p, we create a register β such that $|\beta| = \sum_{h \in \mathcal{H}} |h|$. For verification purposes, we restrict our attention to the headers of p since the p's data payload does not affect the behavior of the switch.

Next, to capture $\mathcal{P}$'s control flow (*i.e.*, the path of execution taken by $p \in \mathcal{P}$), we create a *control register*, `ctrl`, that captures the *execution state* of the circuit $\mathcal{S}$. The number of latches in `ctrl` is determined in a post-processing step after accounting for all of $\mathcal{S}$'s unique execution states. Formally, we write $\mathcal{R} := \Phi(\mathcal{X}) \cup \{\Phi(h).\texttt{valid}, \in \mathcal{H}\} \cup \{\texttt{ctrl}\}$ and $\mathcal{I} := \{\beta\}$.

Finally, the set of gate $\mathcal{G}$ that we use is the standard set of logic gates in addition to some function blocks such as adders, subtractors, and multipliers. Formally, $\mathcal{G} := \{\wedge, \vee, \neg, +, -, *, et.c\}$. We note, however, that in BiFrost' final AIG circuit, $\mathcal{G}_{AIG} = \{\wedge, \neg\}$ contains only the conjunction and negation operations, from which all other gates can be constructed.

Statements and Functions. For each statement s of $\mathcal{P}$, BiFrost assigns a unique identifying label $l(s)$ indicating when s should be executed. When the `ctrl` register's value

matches the label of a statement s, that statement is executed and its impacts on the circuit's state is observed in the following clock cycle.

Algorithm 5.1 shows BiFrost's transformations of P4 statements. The function Mux creates a multiplexer gate m in the sequential circuit $\mathcal{S}$ with semantics $\ell\left(m(e, v_0, v_1)\right) = \ell(e)\ ?\ \ell(v_1)\ :\ \ell(v_0)$. The function AdjustRegisterEdges$(r, m, \mathcal{E})$ removes r's current fanin and replaces it with an edge from m to r. BiFrost leverages this operation to continuously update the graph $\mathcal{G}$'s structure to reflect multiple assignments for every register. This effectively creates a cascade of multiplexers that determine a register's value based on the value of the `ctrl` register.

For an assignment statement s, BiFrost creates a multiplexer that transfers the value of s's right hand expression into the target register r when the `ctrl` register reaches $l\left(s\right)$. In addition, BiFrost updates the next value of the `ctrl` register to the label of the next statement following s (denoted by s.`next`). On the other hand, if s is a conditional statement, it only affects the control flow of the program $\mathcal{P}$ and does not affect the values of the $\mathcal{S}$'s registers. Therefore, BiFrost creates two multiplexers that assign `ctrl`'s next value depending on the value of s's condition.

Finally, for each function $f \in \mathcal{P}$ (we treat P4 `actions` as functions as well), BiFrost creates a set of registers for f's arguments as well as a special register for its return address. When encountering a function call s to f, BiFrost assigns s's argument values to f's arguments (lines 23 – 25), saves the label of s's next statement (*i.e.*) to f's return address register (line 27), and then transfers the control to f's first statement (line 26). We note that this transformation allows us instantiate only one circuit block for each function, and then reuse that same block with different argument values for each function call. This allows us to avoid inlining the function body for each function, thus avoiding expensive replications of circuit blocks. The fact that sequential circuits allow us to reuse memory elements is at the core of BiFrost's expressiveness and succinct representation of data plane programs.

Parsers. A parser in P4 programs represent a finite state machine that define the behavior of a programmable switch upon receiving an input packet. Every state in a parser defines a set of operations to perform on the packet's headers (*e.g.*`extract`) and the transition into the next based on some condition over those headers' fields. Therefore, we can view the parser as nested conditional statements where the next parser state to visit is determined by the current state as well as the values of the variables involved in that state (which often are packet header fields, but can also be local variables).

By interpreting parsers as nested conditional statements, BiFrost assigns each state α in the parser a unique label and then uses the same conditional statement transformation

Algorithm 5.1: Statement handling in BiFrost

```
1  Func AdjustRegisterEdges
       Inputs : r ∈ R, v ∈ V, E
2      ω ← fanins(r)
3      E ← E \ {(ω, r)}
       /* fanins(r) ← {v}                                              */
4      E ← E ∪ {(v, r)}
5  Func HandleStatement
       Inputs : Statement s, Circuit S
6      if s is assignment then
7          r ← Φ(s.target)
8          v ← Φ(s.expression)
9          m_1 ← Mux(ctrl = l(s), v, fanins(r))
10         AdjustRegisterEdges(r, m_1, E)
11         m_2 ← Mux(ctrl = l(s), l(s.next), fanins(ctrl))
12         AdjustRegisterEdges(ctrl, m_2, E)
13     else if s is conditional then
14         e ← Φ(s.condition)
15         m_1 ← Mux(ctrl = l(s) ∧ e, l(s.true), fanins(ctrl))
16         m_2 ← Mux(ctrl = l(s) ∧ ¬e, l(s.false), m_1)
17         AdjustRegisterEdges(ctrl, m_2, E)
18         HandleStatement(s.true, S)
19         HandleStatement(s.false, S)
20     else if s is function call then
21         f ← s.function
22         r ← f.return_register
23         foreach a ∈ Φ(f.args) do
24             m_a ← Mux(ctrl = l(s), s.args[a], fanins(a))
25             AdjustRegisterEdges(a, m_a, E)
26         m_1 ← Mux(ctrl = l(s), l(f), fanins(ctrl))
27         m_2 ← Mux(ctrl = l(s), l(s.next), fanins(r))
28         AdjustRegisterEdges(ctrl, m_1, E)
29         AdjustRegisterEdges(r, m_2, E)
30     else if s is return call then
31         f ← s.function
32         r ← f.return_register
33         m_1 ← Mux(ctrl = l(s), r, fanins(ctrl))
34         AdjustRegisterEdges(ctrl, m_1, E)
```

in Algorithm 5.1 to generate the equivalent circuit block.

Packet Extraction and Lookahead. BiFrost translates packet extraction and lookahead operations by simply connecting wires (*i.e.* edges in $\mathcal{E}$) from a subset of the input packet's bits to the appropriate header fields. The subset of the packet's bits that must be accessed is determined by the order and size of each header parsed in each previously visited states. Therefore, packet extraction operations will depend on runtime information of the program's execution.

Algorithm 5.2: BiFrost's static analysis algorithm.

```
1  Func AnalyzeParserState
       Inputs  : Parser State α, Transition Key key, i
       Outputs: Extraction Indices Map M
2      foreach Operation op ∈ α.operations do
3          if op is extract then
4              M[op] ← M[op] ∪ {(key, i)}
5              i ← i + op.hdr_size
6          else if op is lookahead then
7              M[op] ← M[op] ∪ {(key, i)}
8      foreach (α, α', k) ∈ α.transitions do
9          k' ← key ∧ k
10         AnalyzeParserState(α', k', i)
   /* Initial call for handling the parser                                  */
11 α_0 ← parser's initial state
12 M ← AnalyzeParserState(α_0, True, 0)
```

To address this challenge, BiFrost performs a static analysis of each state in the P4 program's parser and determines the subset of input bits that must be read at each extraction operation. Algorithm 5.2 presents BiFrost's packet extraction analysis algorithm. For each parser state α, BiFrost maps each `extract` or `lookahead` operation with a pair (`key`, `index`) where `index` represents the index of the first bit to extract in the input packet when the condition `key` is satisfied. Additionally, for each extraction operation in α, the index i is updated based on the size of the header to extract (Note that `lookahead` operations do not update the index, as defined by the P4 specification). Then, for each transition (α, α', k), BiFrost propagates the indices to the operations of α' by creating they update key condition `key` $\wedge\, k$. Finally, when generating the sequential circuit $\mathcal{S}$, BiFrost creates a cascade of multiplexer that determine the subset of the input packet's bits that will mapped to each header register based on the keys captured in Algorithm 5.2.

Finally, we note that BiFrost currently only supports fixed-width packet headers where the length of each packet is predetermined at compile-time We acknowledge that this poses a limitation on the developers' ability to parse variable size options (such IPv4 options fields). However, after surveying 11 publicly available P4 programs, we found none that use variable-sized headers. Nevertheless, we plan to provide support for such parsing abilities in future version of BiFrost.

Match-Action Tables. Data plane programs only describe the data forwarding behavior of a networking device, the control plane provides that device with the contents of its forwarding table to drive the functioning of match-action pipelines. For the matching part of the table, BiFrost flattens all the matching conditions into an equivalent combinatorial

Algorithm 5.3: Least Prefix Match Transformation

Func AssignEntryLabels

Inputs : Table Entries T, Key k, $\mathcal{S}$

$T' \leftarrow \text{Sort}(T)$

$entries \leftarrow []$

foreach $i, (e, a, \_) \in \text{Enumerate}(T')$ **do**

  $network\_addr \leftarrow k \;\&\; e.\texttt{mask}$

  $entries[i].\texttt{label} \leftarrow (network\_addr = e.\texttt{addr}) \wedge \left(\bigwedge_{j=0}^{i-1} \neg\, entry[j].\texttt{label} \right)$

/* $\mathcal{D}$ is the runtime data for each table entry. */

foreach $i, (\_, a, \mathcal{D}) \in \text{Enumerate}(T')$ **do**

  foreach $v \in a.\texttt{parameters}$ **do**

    $\lambda \leftarrow (\texttt{ctrl} = l\,(a)) \wedge (entries[i].\texttt{label})$

    $m \leftarrow \texttt{Mux}(\lambda, \mathcal{D}(v), \texttt{fanins}\,(v))$

    AdjustRegisterEdges(v, m, $\mathcal{E}$)

  $\texttt{loc} \leftarrow \left(\bigcup_{j=0}^{|entries|} entries[j].\texttt{label} \right) \wedge \texttt{ctrl} = l\,(T)$

  $m \leftarrow \texttt{Mux}(\texttt{loc}, l\,(a), \texttt{fanins}\,(\texttt{ctrl}))$

  AdjustRegisterEdges(ctrl, m, $\mathcal{E}$)

circuit (*i.e.*, one without any clocked registers) depending the type of the matching procedure (`exact`, `lpm`, or `ternary`). Algorithm 5.3 shows an example of how this is performed for least-prefix-match tables. BiFrost identifies the logic clause that would lead to the selection of each action defined by the table, and then transfers control to that specified action if its clause is satisfied. For cases when multiple matches are possible, BiFrost sorted the list of possible actions by order of priority and then for each lower priority match entry, adds a clause specifying that all higher priority match entries must be false.

After selecting the appropriate action based on its match entry's clause, BiFrost transfers control to the start of each action, and assigns the action parameters to the set of values defined in the match-action table. This is followed by normal execution of the all the statements in the action using the values specified by the appropriate table entry.

Checksums. BiFrost provides circuit blocks to perform IPv4 checksum calculation and verification operations as defined in RFC 1071 [167]. Since circuit area is not our primary design goal, we save on clock cycle and circuit complexity by implementing those blocks as *combinatorial blocks*, *i.e.* the entire calculation or verification task completes within one clock cycle. We achieve this property by flattening the checksum algorithm into a series of one's complement full adders operating on pairs of 16 bits registers; this is the counterpart of loop unrolling in compiler design techniques [168].

```
<Property> → Assert(<Expression>) |
             Guard (<Expression>) |
             <Expression>.

<Expression> → ∀ (<VariableList>) <Expression>  |
               ∃ (<VariableList>) <Expression>  |
               (<Expression>)                   |
               <Expression> <Op> <Expression>   |
               ¬ <Expression>                   |
               Variable                         |
               ϵ.

<VariableList> → Variable | ϵ |
                 Variable, <VariableList>.

<Op> → {+, −, ∗, ==, ∧, ∨, ⟹, etc}.
```

Figure 5.5: BiFrost's grammar for defining properties.

Functional Equivalence. BiFrost' transformation Φ on a program $\mathcal{P}$ maps each P4 statement and parser state to a unique control location captured by the value of the circuit $\mathcal{S}$'s `ctrl` register. As shown in Algorithm 5.1, Φ preserves the semantics of each statement s by operating on the circuit registers $\mathcal{R}$ that correspond to any affected header or variable in $\mathcal{P}$. Therefore, it is clear that, by construction, any trace $tr_1 := \texttt{tr}(\mathcal{P}, k)$ of length k on $\mathcal{P}$ corresponds to a trace $tr_2 := \texttt{tr}(\mathcal{S}, k')$ on $\mathcal{S}$ such that $k' \geq k$ and ever state $\sigma_1 \in tr_1$ has an equivalent state $\sigma_2 \in tr_2$, thus proving the functional equivalence between $\mathcal{P}$ and its corresponding circuit $\mathcal{S} = \Phi(\mathcal{P})$.

5.6 VERIFICATION METHODOLOGY

5.6.1 Specifying Program Properties

BiFrost accepts data plane program specifications in the form of *First Order Logic* (FOL) formulae provided by developers as annotations to the input program. Figure 5.5 shows BiFrost' property specification grammar; properties are either assertions (`Assert`) or guards (`Guard`) tied to a specific statement in the data plan program. Assertions are properties that must hold when the specific control location is reached during execution, while guards provide constraints on the inputs to the program or specific actions and functions. In addition, BiFrost accepts properties in the form of pre and post conditions on the program as defined in Hoare triples [169]. Contrary to assertion and guards that are statement specific, pre and post conditions are global to the program. In other words, a pre

condition provides constraints on the input packet of the data plane program, while the post condition specifies assertions that must hold at the end of the processing pipeline.

When applying the circuit transformation Φ, BiFrost translates each property into its equivalent combinatorial circuit and creates the verification conditions `guard` $\implies$ `assertion` and `precondition` $\implies$ `postcondition` as circuit outputs. When verification starts, the verifier will try to find input assignments (*i.e.*, an input packet) that will cause the verification conditions to be violated. If no such assignments can be found, the properties are deemed to be satisfied.

Header Validity. In addition to developer-provided specifications, BiFrost provides built-in support for header validity checks. During the transformation, for each statement s that reads or write to a program header h, BiFrost adds the verification condition `ctrl` $= l(s) \implies$ `IsValid`(h) where `IsValid` corresponds to h's validity bit being set to 1. Then, at the end of the transformation, BiFrost creates the conjunction of all such validity conditions and add them as an output to the generated circuit $\mathcal{S}$, to be checked for violations during the verification phase.

5.6.2 Control Plane Abstraction

Data plane programs specify the forwarding behavior of a switch, while the control plane controls the content of the switch's match-action tables. In that regard, the transformation we introduced in Section 5.5 operates on a single snapshot of the target switch's tables. Therefore, any properties proven by BiFrost are only proven with regard to that specific snapshot of tables.

To make the verification process more general, BiFrost provides avenues for abstracting the control plane tables, thus proving properties for the programmable switch regardless of the content of its tables. To achieve that goal, BiFrost provides developers with the option to abstract the match-action tables by replacing them with free input variables that non-deterministically choose actions and runtime data. During the transformation, when BiFrost encounters a table lookup, it uses a set of free inputs to non-deterministically choose an action from that table's allowed actions. It then assigns that actions' parameters to free inputs of the same bit width. In such a case, when BiFrost proves a property, that property will hold on the data plane program regardless of the contents of its tables, *i.e.*, for any function of the control plane.

The abstraction of the control plane tables introduces two important challenges. First, by adding more free inputs, the abstraction greatly increases the verifier's search space, which can often make the verification process computationally intractable. We address this issue in

BiFrost by (1) allowing developers to provide `Guard` conditions on the control plane tables to constrain the search space, and (2) performing bounded model checking on the length of the switch's pipeline (as discussed in the following subsection).

Second, abstracting the match-action tables violates the functional equivalence property of BiFrost's transformation and produces a circuit $\mathcal{S}$ that *over-approximates* the input data plane program $\mathcal{P}$. This might lead the verifier to find false positive counterexamples; such counterexamples, although they violate the program's properties, are not possible program traces and thus must not be considered. To address this challenge, BiFrost provides the developer with a detailed trace of each generated counterexample (showing the content of the input packet as well as every header and variable throughout the pipeline), thus allowing them to sort through the true positives and the false positives. However, we plan to explore techniques such as *Counter Example Guided Abstraction Refinement* (CEGAR) [162] to use false positives for adding constraints to the abstracted match-action tables.

5.6.3 Bounded Model Checking

To provide higher-levels of scalability, and to address the challenges brought-upon by the abstraction of the match-action tables, we leverage the boundedness of the programmable switches to perform *bounded model checking* (BMC) with provable pipeline length guarantees.

Based on the observation that data plane programs are linear in the number of headers in incoming packets, during BiFrost's transformation, we estimate an upper bound on the number of clock cycles needed to process a single packet, from the moment it enters the pipeline until it is written onto the appropriate output port. If the program's parser is acyclical, the upper bound, `B`, is simply equal to the largest control flow label generated during the transformation. On the other hand, if the parser contain a cycle of length l, then we estimate the upper bound as $\mathtt{B} + k \times n \times l$ where n is the number of headers in the program, and $k > 1$ is a constant multiplier.

Next, BiFrost proves that the estimated bound `B` holds over any possible program execution. It first abstracts the program's match-action tables. Then, it adds a simple counter `C` that is incremented every clock cycle, and creates the post condition $\Gamma := \mathtt{C} \leq \mathtt{B}$. Finally, it launches a verification process to prove that Γ is always satisfied, *i.e.*, for all input packets and for all match-action tables, the switch's pipelines fully processes the packet in less than `B` clock cycles. The verification of this boundedness property is more efficient than the verification of program-specific properties since it is restricted to the transitions made during execution, and not the actual content of the registers. For programs with cyclical parsers, BiFrost starts with $k = 1$ and then keeps increasing k as long as it finds counterexamples to

the Γ. Since the program is linear in n, we are guaranteed to find a k such that boundedness condition holds.

Once the bound `B` is proven, BiFrost can then perform BMC on the original specification with the clock cycle bound set to `B`. If no counter example is generated, the verifier asserts that the no input assignment can violate the program's properties in `B` clock cycles. Coupled with the previous boundedness proof, this is enough to guarantee the program's properties hold for any execution possible execution path.

5.7 IMPLEMENTATION

We implemented BiFrost in C++ using 13 869 lines of code. Our implementation is modular and extensible, allowing developers to provide support for different data plane programming languages and transformations for designer-specific `extern` blocks. We plan to release BiFrost as an open-source tool that developers can easily use to verify their implementations.

The current version of BiFrost accepts the input P4 program in its `JSON` format, as generated by the `p4c` reference compiler. We extend the `JSON` format to support the specification of FOL properties. To support modular and extensible transformations, BiFrost first translates the input program into an intermediate, high-level, circuit representation, and exposes its API to future developers. Finally, BiFrost uses the `ABC` API to generate the `AIG` and perform the verification tasks. BiFrost comes pre-equipped with several reduction, abstraction and verification techniques, but also provides a built-in interactive shell that allows developers to interact with the `ABC` verifier.

To validate the correctness of our implementation, we generated input and output packets by simulating each of our test P4 programs using the P4 behavioral model version 2 (`bmv2`). We then use the input packet to simulate our generated `AIG` and verifies that it outputs the correct packet at its egress.

5.7.1 Handling Parser Loops

If the input P4 program employs header stacks, parser loops may arise to extract all the headers from the packet into the stack. Although such loops are always bounded by the maximum number of headers in the stack, the exact number of iterations undertaken by the parser is not known until the packet is received. The last extracted header in the stack can be later accessed using the stack's `last` operator.

To keep our transformations simple and efficient, instead of introducing complex circuitry to track loop counters, we choose to unroll parser loops to allow for the extraction of all of the possible headers in the stack. Subsequently, the stack's `last` operator is simply a constant value obtained from the corresponding unrolled parser state, and can be later one manipulated using `push` and `pop` operations. Although unrolling increases the sizes of the sequential circuit, it favors simplicity and thus aids in scaling verification efforts.

5.7.2 Register Arrays Reads and Writes

P4 supports register arrays that can be accessed using index values computed during the program's execution. To support such arrays in BiFrost, we implement a very simply index-based memory manager that allows reads from and writes to arrays of registers. When reading the value of a certain register `a[i]`, BiFrost creates a hierarchy of multiplexers that compare the value of `i` to the possible index values allowable for the register array `a`. Since all values are mutually-exclusive, only one value is returned, which corresponds to the contents of the register `a[i]`. Similarly, we translate each write to a register element into writes to all the elements in the array, out of which only one will be activated based on the runtime value of the index. If the index is out of bound, all accesses will fail and an out of bound error is produced.

5.7.3 Counterexamples

When the verification procedures in `ABC` generate a counterexample, BiFrost lifts that counterexample back into the original program and generates (1) a `pcap` file containing the violating packet (or subset thereof) and (2) a trace that shows the state observed during the processing of that packet. To that extent, we maintain a reverse lookup map between the circuit's control locations and their corresponding P4 program locations (such as table lookups, parser states, and action primitives). Note that the counterexample might include only a subset of the program's variables since several variables, or their individuals bits, can be removed during the reduction and abstraction phase.

5.8 EVALUATION

In this section, we set out to evaluate BiFrost and answer the following research questions:

RQ 5.1 How effective is BiFrost in proving data plane program properties and generating counterexamples?

RQ 5.2 What are the benefits of using sequential circuit verification when verifying data plane program properties?

RQ 5.3 How scalable is BiFrost when attempting to verify large P4 programs?

5.8.1 Case Study I: Header Validity Bug in `ecn.p4`

We first illustrate the benefits of employing data plane program verification. Using BiFrost, we uncovered a read from an invalid header bug in one of the P4 tutorial examples `ecn.p4`[3]. The goal of the exercise is to develop a programmable switch that performs layer 3 forwarding while providing support for *Explicit Congestion Notification* (ECN) bits to signify that the switch has encountered congestion in its queues, without the need for dropping packets. When a packet arrives, the program parses its Ethernet header and then parses an IP header if the Ethernet type is set to 0x0800. In its ingress pipeline, the switch performs an IP address lookup in its match-action tables to determine the packet's egress port, decrement its TTL value, and assign its MAC source and destination addresses. This pipeline is guarded against invalid header accesses by only performing the lookup if the packet's IPv4 header is valid.

At the egress pipeline, the switch reads the packet's ECN bits and checks its queue depth to determine if it should notify the receiver that it has encountered congestion. However, in this case, the developers did not guard against non-IP headers; the switch attempts to read the ECN bits of the packet even if its IPv4 header was not previously set to valid during the parsing stage. The switch's behavior at this stage is not defined by the P4 specification [148], it is rather determined by the target architecture, thus introducing vulnerabilities that malicious entities can exploit [123].

Using BiFrost, we were able to discover and locate this bug in the `ecn.p4` program, as shown in Figure 5.6. BiFrost generated a counterexample trace showing that when execution reached control location `0x0E` (which corresponds to the start of the `MyEgress` pipeline), the header validity property was violated. Lifting this counterexample back into the program's source code shows that the conditional statement at the start of the pipeline accesses the `ecn` field of the ipv4 while the header is invalid.

[3]We confirmed this issue with the tutorial developers. The example solution was provided for illustrative purposes so no corrective action was taken.

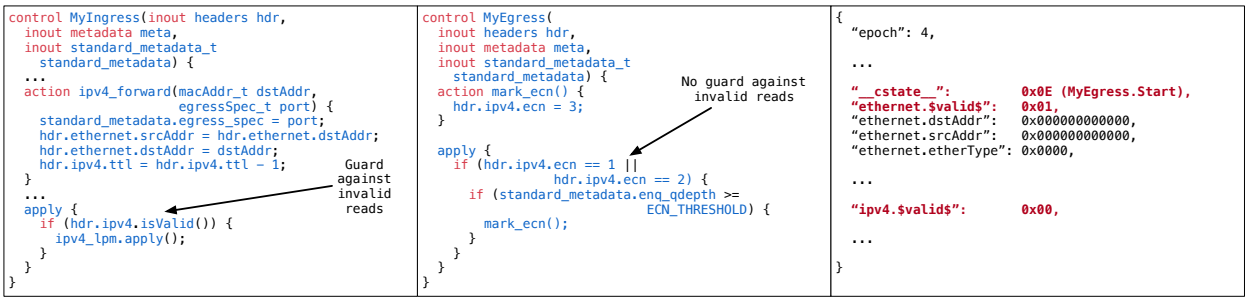


Figure 5.6: Bug discovered in `ecn.p4`. The leftmost code snippet shows the ingress pipeline where invalid header reads are guarded against using an `if` statement. The center code snippet shows the egress pipeline where the ecn bits of the ipv4 header of the packet are read even if that packet is invalid. The rightmost snippet shows an except of the counterexample generated by BiFrost when attempting to validate the program's header validity. It shows that the validity property was invalidated when execution reached the start at of the egress pipeline, where the Ethernet header was valid while the IPv4 header was not.

5.8.2 Case Study II: Verifying Functional Properties in `calc.p4`

To answer **RQ 5.2**, we use the simple calculator example, `calc.p4`, provided by the P4 language tutorials[4], to illustrate the efficacy of sequential circuit reduction, abstraction, and verification techniques in proving functional program properties. `calc.p4` implements a simple calculator that, after parsing a packet's Ethernet header, parses a custom header that contains two 32-bit operands, an 8-bit operation field, and a 32-bit result. Depending on the type of the operation field, the switch will perform a certain arithmetic or logic operation on the two operands and store the result in the corresponding header field. It will then send the packet back to the user on the same port from which it was received.

To verify that the program was implemented correctly, we annotate it with pre and post conditions to capture its correct behavior. For example, for the addition operation, we verify the property

$$\begin{aligned} &\forall(\text{packet headers } h).\\ &\underbrace{(h.\texttt{eth.valid} \wedge h.\texttt{calc.valid} \wedge h.\texttt{calc.op} = \texttt{0x2b})}_{\text{pre condition}} \implies \\ &\underbrace{(h.\texttt{calc.res} = h.\texttt{calc.operand\_a} + h.\texttt{calc.operand\_b})}_{\text{post condition}} \end{aligned} \tag{5.5}$$

where h represents the set of packet headers extracted by the parser, $h.\texttt{eth}$ and $h.\texttt{calc}$ are the packet's Ethernet and custom calculator packet headers, respectively. The clause $h.\texttt{calc.op} = \texttt{0x2b}$ makes sure the packet is intended to request an addition operation, while

[4] `https://github.com/p4lang/tutorials/blob/master/exercises/calc/solution/calc.p4`

Table 5.1: Verification results for `calc.p4`. PI and Lat. stand for primary inputs and latches, respectively. BMC represents bounded model checking and PDR represents property directed reachability.

$\mathcal{P}$	AIG size (Original)			AIG size (Abstracted)			Method	Time (sec.)
	PIs	**Lat.**	**Gates**	**PIs**	**Lat.**	**Gates**		
calc	240	1589	4650	144	346	1227	PDR	Timeout
calc	240	1589	4650	144	346	1227	BMC	23.35
calc-8	168	941	3330	72	202	819	PDR	57.43

the post condition ensures that after processing the packet, the result indeed includes the addition of the two operands.

We set out to fully verify property (5.5) for all of its input space (14 bytes for the Ethernet header and 16 bytes for the `calc` header). As show in Table 5.1, BiFrost generates an AIG with 240 free inputs (PIs) that represent the 30 bytes header space of the input packet. After performing sequential circuit reduction techniques, the number of free inputs is reduced to 144, capturing only the Ethernet header's type field (16 bits) and the `calc` header (16 bytes). In other words, BiFrost was able to eliminate the source and destination MAC addresses since they have no impact on property (5.5).

However, when first attempting to verify the property using property directed reachability [170, 171], after 5 minutes, the `ABC` verifier produced an undecided result; it did not find a counterexample within the allocated time limit, but was not able to prove the property. This is a reflection of the still very large input space (2^{144} values after reduction) of the program, and the relative complexity of property (5.5). Effectively, verifying property (5.5) amounts to verifying a full *Arithmetic Logic Unit* (ALU) over two 32-bit operands. To validate our intuition, we reduced the size of the operands from 32 bits to 8 bits and were able to fully verify the property in 1 minute (row 3 of Table 5.1).

Nevertheless, we would still like to a conclusive verification result for `calc.p4`. To that end, we leveraged our circuit bounding technique to perform BMC. During the transformation, BiFrost estimated that processing a single packet through the switch's pipeline takes no more than 69 clock cycles. We then launched a separate verification procedures and formally proved that execution always completes within that bound, which the `ABC` verifier quickly verifies. Finally, instead of using PDR, we ran BMC with a bound of 69 clock cycles and were able to verify property (5.5) in 24.5 seconds. What the BMC result asserts is that for all input values, the property is always satisfied within the first 69 clock cycles of execution. Coupled with BiFrost's verified circuit bound, this guarantees that the program satisfies its specification.

5.8.3 Scalability Benchmarks

Finally, to answer **RQ 5.3**, we evaluate BiFrost on 11 open source P4 programs. These programs represent a range of data plane applications such as simple forwarding, multicasting, quality of service mechanisms, and a data plane-only firewall. For each program, we verified the header validity property for all possible control plane settings by abstracting the contents of the tables and replacing them with free inputs. As in [129], we chose the header validity property since it generalizes to any data plane program. However, we note that unlike [129], BiFrost does not require any inputs from the control place. Our verification results hold for any control place configuration. We conducted our experiments on a server-class machine with 32 Intel(R) Xeon(R) CPU E5-2450 v2 @ 2.50GHz processors and 64GB of memory space.

Table 5.2 shows the results of our experiments. We report on the size of the generated sequential circuit before and after abstraction in terms of the number of primary inputs, latches, and logic gates. We also breakdown the performance profiles in terms of transformation, abstraction, and verification times. We first note that BiFrost can verify header validity for most programs in less than a second. Some of those programs have a relatively large input space (such as 38 bytes for the headers in `basic_tunnel`). `firewall.p4` is the program with the largest running time ($>$ 2 minutes). Although we believe this is acceptable for verification, we attribute the runtime increase for `firewall.p4` to the fact that it implements a complex firewall and maintains two Bloom filters, each with 4096 elements, that use `crc16` and `crc32` as hashing functions. This is exemplified by the large increase in the size of the generated circuit compared to the other benchmarks. The presence of such data structures can complicate the generate circuit and this increase the verification time.

Second, we note that over all the experiments, BiFrost achieves $>$ 80% reduction in the size of the generated sequential circuit. This further emphasize the power of sequential reduction and abstraction techniques in eliminating the parts of the circuit that do not contribute to the property being verified. Furthermore, for `resubmit.p4` and `multicast.p4`, the `ABC` verifier was able to flatten the circuit into a combinatorial circuit and prove that the header validity property is a tautology, as showcased by the elimination of all circuit components during the abstraction phase. This is due to the fact that these two programs only include Ethernet headers, thus making it simple for `ABC` to completely flatten the circuit into one combinatorial formula.

Finally, we used BiFrost to uncover another header validity bug in `vpc.p4`[5]. `VPC` implements a virtual private cloud in programmable switches. After parsing an Ethernet header,

[5] `https://github.com/joncastro/p4vpc`

Table 5.2: Performance results for BiFrost on several open source P4 benchmarks. LOC stands for Lines of Code. Out. represents the verification outcome where ✓stands for verified and ✗ stands for counterexample generated. PIs and Lat. stand for primary inputs (i.e., free variables) and latches, respectively. Trans., Abs., and Ver. represent the transformation, abstraction, and verification times, respectively.

$\mathcal{P}$	LOC	Parser States	Header length (bytes)	Out.	AIG size						Time (s)		
					Original			After abstraction			Trans.	Abs.	Ver.
					PIs	Lat.	Gates	PIs	Lat.	Gates			
basic	121	3	34	✓	331	890	3075	18	46	156	0.061	0.076	0.060
basic_tunnel	153	4	38	✓	403	997	5029	38	87	345	0.063	0.095	0.079
copy_to_cpu	95	4	16	✓	131	520	1421	67	211	371	0.059	0.091	0.066
calc	139	4	30	✓	243	1588	4561	43	120	603	0.065	0.118	0.068
qos	188	3	34	✓	567	891	3674	34	65	364	0.061	0.094	0.185
ecn	134	3	34	✗	331	909	3181	20	49	196	0.060	0.082	0.012
multicast	89	2	14	✓	123	394	648	0	0	0	0.056	0.060	0.001
firewall	210	4	54	✓	494	9643	63 372	126	8495	44 335	0.216	3.293	134.047
paxos-leader	205	5	90	✓	787	2233	15 370	247	454	1355	0.076	0.324	0.535
vpc	272	7	88	✗	10 263	2532	49 634	558	257	6657	0.141	1.189	0.077
resubmit	89	3	14	✓	129	452	790	0	0	0	0.054	0.064	0.001
Average Reduction								85.629	82.428	85.498			

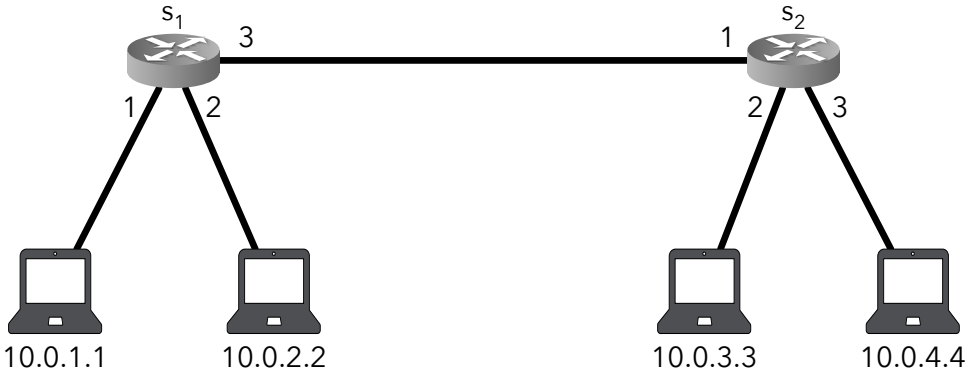


Figure 5.7: A sample topology comprised of four hosts connected to two programmable switches. s_1 and s_2 are programmable switches. Numbers on the links represent each switch's port number.

the switch reads an ARP header, `arp_rarp`. Subsequently, if `arp_arp.protoType` is `0x0800`, the switch parses another ARP header called `arp_rarp_ipv4`. However, in its `ingress` pipeline, after checking `arp_rarp`'s validity, the switch assumes that `arp_rarp_ipv4` is present, without checking that `arp_arp.protoType` is indeed `0x0800`. In such a case, an attacker can craft a packet that will lead the switch to read from an invalid header, thus introducing an undefined behavior. We confirmed this bug by tracing BiFrost's counterexample into the original program.

Overall, we believe that our results showcase that BiFrost can scale to real world data plane programs, and can leverage the power of sequential synthesis and verification to achieve scalable and efficient verification for different properties.

5.9 GENERALIZATION TO NETWORK PROPERTIES

BiFrost currently only support properties that express correctness and safety specifications about a single programmable switch. In other words, those properties only relate to the lifetime of a packet once it enters a switch through the ingress port until it leaves that switch at the output port. However, network verification, and specifically when dealing with security properties, requires an overall view of the network that allows developers to express reachability and safety properties about a deployed network. In this section, we discuss how BiFrost can be extended to support such verification tasks.

Consider the simple network topology shown in Figure 5.7. The network is composed of four hosts connected to two programmable switches, forming four different subnetworks. Assume in this case, that the network developer set up a firewall program that blocks

`ssh` traffic from hosts on the subnetwork `10.0.1.0/24` to the host `10.0.4.4`. To verify the correctness of her deployment, the developer would like to verify that the data plane programs deployed on the switches correctly block `ssh` traffic to the target host. Formally, the developer can express this property as ∀packet `p`,

$$(\texttt{p.ip.srcAddr} \in \texttt{10.0.1.0/24} \wedge \texttt{p.ip.dstAddr} = \texttt{10.0.4.4} \wedge \texttt{p.tcp.dstPost} = 22) \implies \texttt{isDropped}\,(p) \tag{5.6}$$

To verify this property using BiFrost, we first translate data plane programs that the developer implemented into their equivalent sequential circuits. However, unlike the single-switch verification tasks, each circuit will have different inputs and outputs that correspond to the number of different input and output ports available. Then, given the network topology, we can connect the corresponding circuit inputs and outputs in accordance to the connectivity profiles between the switches. Since we are interested in security properties, we do not place restrictions on the packets originating from every host. In other words, we allow packets on ingress ports to take on any values, even if they are not consistent with the network topology. This allows us to capture the case where a compromised host masquerades as a different host and send traffic on its behalf.

Finally, we project the property that the developer would like to verify onto the generated sequential circuit by adding appropriate primary outputs at each switch. For example, the firewall correctness property we previously discussed can be translated into two properties at each switch as follows. At switch s_1, ∀packet `p`,

$$(\texttt{p.ip.isValid()} \wedge \texttt{p.ip.dstAddr} = \texttt{10.0.4.4}) \implies (\texttt{p.metadata.eport} = 3) \tag{5.7}$$

and at s_2,

$$\forall \text{packet } \texttt{p}.\,(\texttt{p.ip.isValid()} \wedge (\texttt{10.0.1.0} \leq \texttt{p.ip.srcAddr} \leq \texttt{10.0.1.255})) \implies ((\texttt{p.tcp.isValid()} \wedge \texttt{p.tcp.dstPort} = 2) \implies \texttt{isDropped(p)}) \tag{5.8}$$

The first property at switch s_1 ensures that the switch is forwarding packets destined to `10.0.4.4` on the correct output port. The second property ensures that switch s_2 drops all packets originating from `10.0.1.0/24`, destined to `10.0.4.4`, and containing a valid TCP header with a destination port of 22. Note that in this case, we did not restrict that packets on switch s_2 to be received on the ingress port 1 since an attacker that has compromised host `10.0.3.3` can send packets destined to `10.0.4.4` with a spoofed source IP address of `10.0.1.1`. Therefore, the switch s_2 should drop all packets that satisfy the criterion, no

matter the ingress port the switch receives them on. We plan to extend BiFrost to support such transformations and properties in a future release.

5.10 RELATED WORK

Network Verification. Recently, a number of tools have been proposed to verify software-defined networks. Anteater [172] performs static analysis on the contents of the forwarding tables of data plane devices to detect violations of certain operator-provided invariants. It translates network invariants into SAT formulas and checks them using a SAT solver. HSA [173] statically checks failure conditions for software-defined networks in a protocol-independent way. It analyzes equivalence classes of related packets to perform reachability analysis, loop detection and slice isolation. These works perform static verification by analyzing snapshots of the programmable data plane against specified network properties. While these tools can be used to detect violations of static properties across devices, they cannot be used to detect issues in the packet processing procedures within the switches.

P4 Packet Generation. Several approaches generate packets in various ways to trigger bugs at runtime in P4 programs. `p4rl` [174] proposed a test packet generation tool that uses reinforcement learning to decide on which fuzzing actions to take when generating test cases. `p4pktgen` [128] uses symbolic execution to automatically generate test cases from specifications of P4 programs. The generated packets contain test packets, table entries, and expected paths. Suriya et al [175] provide complementary work to `p4pktgen` to localize bugs for P4 program at runtime. However, the test generation based approaches can be expensive when the search space grows. Static verification can be seen as a complementary approach to exhaustive testing.

P4 Verification. There are also works that perform static verification of P4 programs using symbolic execution. `p4v` [129] performs automatic verification of P4 data planes using Dijkstra's algorithm (GCL) [176] and the Z3 theorem prover [177] to check properties and compute counter examples. Vera [130] translates P4 programs into an equivalent SEFL representation and uses Symnet [178] to perform symbolic execution on the generated model. It explores fixed snapshots of the control plane and uses symbolic table entries to check specified policies. Neves et al [179] present a tool that takes as input a data plane program manually annotated with assertions. It translates the program and its assertions into a C program and checks its correctness with symbolic execution. Unlike BiFrost, these approaches treat data plane programs similarly to general-purpose programs and heavily rely on symbolic execution. BiFrost's BMC approach to verify large programs has no

counterpart in any other general-purpose approach.

Model Checking. Traditional verification approaches use symbolic execution for model checking such as SymNet [178] and NuSMV [180]. They translate program invariants into boolean expressions and explore all possible code paths to perform model checking. The closest work to BiFrost is that in [181] where the authors propose to translate a restricted subset of general-purpose programs into sequential circuits. BiFrost builds upon the approach presented in [181] but uses several domain-specific transformations and optimizations targeted for data plane programs. The boundedness of data plane programs makes them better candidates for such a transformation than general purpose programs.

5.11 CONCLUSION

In this chapter, we presented BiFrost, a tool for the scalable verification of data plane programs using sequential circuit synthesis and verification. BiFrost exploits the boundedness of data plane programs to generate efficient and functionally equivalent circuits. BiFrost then uses the `ABC` verifier to reduce the size of those circuit and perform sequential verification for different types of properties. Our results show that BiFrost can scale to real-world programs by leveraging the power of sequential reduction and verification, and that BiFrost is able to prove complex properties and detect bugs in open source programs.

In the future, we plan to explore the following research paths. First, BiFrost currently supports the verification of the forwarding path of a single programmable switch. However, many network-wide properties depend on the interactions between different switches. Therefore, we would like to extend BiFrost to support the verification of network-wide properties that span the lifetime of a packet across several interconnected switches. Second, one of the promises of programmable data planes is flexibility; a network operator can deploy a program on switches from different vendors and expect them to behave in the same way. However, each vendor provides its own proprietary compiler and there is not guarantee that all switches will behave similarly. We plan to use our sequential circuit transformation to provide operators with the means to verify the functional equivalence of the different implementations. Using sequential equivalence checking, BiFrost's generated circuit can serve as an oracle that can be used to attest for the functional equivalence between the different compilations.

CHAPTER 6: SSHIELD: A GAME-THEORETIC APPROACH TO RESPOND TO ATTACKER LATERAL MOVEMENT

6.1 INTRODUCTION

In the wake of the increasing number of targeted and complex network attacks, namely Advanced Persistent Threats (APTs), organizations need to build more resilient systems. *Resiliency* is a system's ability to maintain an acceptable level of operation in light of abnormal, and possibly malicious, activities. The key feature of resilient systems is their ability to react quickly and effectively to different types of activities. There has been an ever-increasing amount of work on detecting network intrusions; Intrusion Detection Systems (IDSs) are widely deployed as the first layer of defense against malicious opponents [182]. However, once alarms have been raised, it may take a network administrator anywhere from weeks to months to effectively analyze and respond to them. This delay creates a gap between the intrusion detection time and the intrusion response time, thus allowing attackers a sometimes large time gap in which they can move freely around the network and inflict higher levels of damage.

An important phase of the life cycle of an APT is lateral movement, in which attackers attempt to move laterally through the network, escalating their privileges and gaining deeper access to different zones or subnets [183]. As today's networks are segregated by levels of sensitivity, lateral movement is a crucial part of any successful targeted attack. An attacker's lateral movement is typically characterized by a set of causally related chains of communications between hosts and components in the network. This creates a challenge for detection mechanisms since attacker lateral movement is usually indistinguishable from administrator tasks. It is up to the network administrator to decide whether a suspicious chain of communication is malicious or benign. This gap between the detection of a suspicious chain and the administrator's decision and response allows attackers to move deeper into the network and thus inflict more damage. It is therefore essential to design response modules that can quickly respond to suspicious communication chains, giving network administrators enough time to make appropriate decisions.

Intrusion Response Systems (IRSs) combine intrusion detection with network response. They aim to reduce the dangerous time gap between detection time and response time. Static rule-based IRSs choose response actions by matching detected attack steps with a set of rules. Adaptive IRSs attempt to dynamically improve their performance using success/failure evaluation of their previous response actions, as well as IDS confidence metrics [184, 185]. However, faced with the sophisticated nature of APTs, IRSs are still unable

to prevent network attacks effectively. Rule-based systems can be easily overcome by adaptive attackers. Adaptive systems are still not mature enough to catch up with the increased complexity of APTs.

In this chapter, we present a game-theoretic network response engine that takes effective actions in response to an attacker that is moving laterally in an enterprise network. The engine receives monitoring information from IDSs in the form of a network services graph, which is a graph data structure representing vulnerable services running between hosts, augmented with a labeling function that highlights services that are likely to have been compromised. We formulate the decision-making problem as a defense-based zero-sum matrix game that the engine analyzes to select appropriate response actions by solving for saddle-point strategies. Given the response engine's knowledge of the network and the location of sensitive components (e.g., database servers), its goal is to keep the suspicious actors as far away from the sensitive components as possible. The engine is not guaranteed to neutralize threats, if any, but can provide network administrators with enough time to analyze suspicious movement and take appropriate neutralization actions. The decision engine will make use of the monitoring information to decide which nodes' disconnection from the network would slow down the attacker's movements.

An important feature of our approach is that, unlike most IRSs, it makes very few pregame assumptions about the attacker's strategy; we only place a bound on the number of actions that an attacker can make within a time period, thus allowing us to model the problem as a zero-sum game. By not assuming an attacker model beforehand, our engine can avoid cases in which the attacker deviates from the model and uses its knowledge to trick the engine and cancel the effectiveness of its actions. We show that our engine is effectively able to increase the number of attack steps needed by an attacker to compromise a sensitive part of the network by at least 50%. Additionally, in most cases, the engine was able to deny the attacker access to the sensitive nodes for the entire period of the simulation.

The rest of this chapter is organized as follows. We describe the motivation behind our work in Section 6.2. We then present an overview of our approach and threat model in Section 6.3. Section 6.4 formally presents the response engine and the algorithms we use. We discuss implementation and results in Section 6.5. We review past literature in Section 6.6, present challenges and future directions in Section 6.7, and conclude Section 6.8.

6.2 MOTIVATION

The life cycle of an APT consists of the following steps [183, 186, 187]. The first is intelligence gathering and reconnaissance, which is followed by the establishment of an entry

point into the target system. Subsequently, the attacker establishes a connection to one or more command and control (C&C) servers, and uses these connections to control the remainder of the operation. Following C&C establishment is *lateral movement*, wherein the attacker gathers user credential and authentication information and moves laterally in the network in order to reach a designated target. The last step includes performance of specific actions on the targets, such as data exfiltration or even physical damage [188].

Lateral movement allows attackers to achieve persistence in the target network and gain higher privileges by using different tools and techniques [183]. In a number of recent security breaches, the examination of network logs has shown that attackers were able to persist and move laterally in the victim network, staying undetected for long periods of time. For example, in the attack against the Saudi Arabian Oil Company, the attackers were able to spread the malware to infect 30,000 personal machines on the company's network through the use of available file-sharing services [189]. In the Ukraine power grid breach, attackers used stolen credentials to move laterally through the network and gain access to Supervisory Control and Data Acquisition (SCADA) dispatch workstations and servers. The attackers had enough privileges to cause more damage to the public power grid infrastructure [190]. Furthermore, through the use of USB sticks and exploitation of zero-day vulnerabilities in the Windows operating system, the Stuxnet malware was able to move between different workstations in an Iranian nuclear facility until it reached the target centrifuge controllers [188].

Early detection of lateral movement is an essential step towards thwarting APTs. However, without timely response, attackers can use the time gap between detection and administrator response to exfiltrate large amounts of data or inflict severe damage to the victim's infrastructure. It took network administrators two weeks to effectively neutralize threats and restore full operation to the Saudi Arabian Oil Company's network [189]. Furthermore, attackers attempt to hide their lateral movement through the use of legal network services such as file sharing (mainly Windows SMB), remote desktop tools, secure shell (SSH) and administrator utilities (such as the Windows Management Instrumentation) [183]. This stealthy approach makes it harder for network administrators to decide whether the traffic they are observing is malicious lateral movement or benign user or administrative traffic.

In this work, we present a game-theoretic approach for autonomous network response to potentially malicious lateral movement. The response actions taken by our engine aim to protect sensitive network infrastructure by keeping the attacker away from it for as long as possible, thus giving network administrators enough time to assess the observed alerts and take effective corrective actions to neutralize the threats.

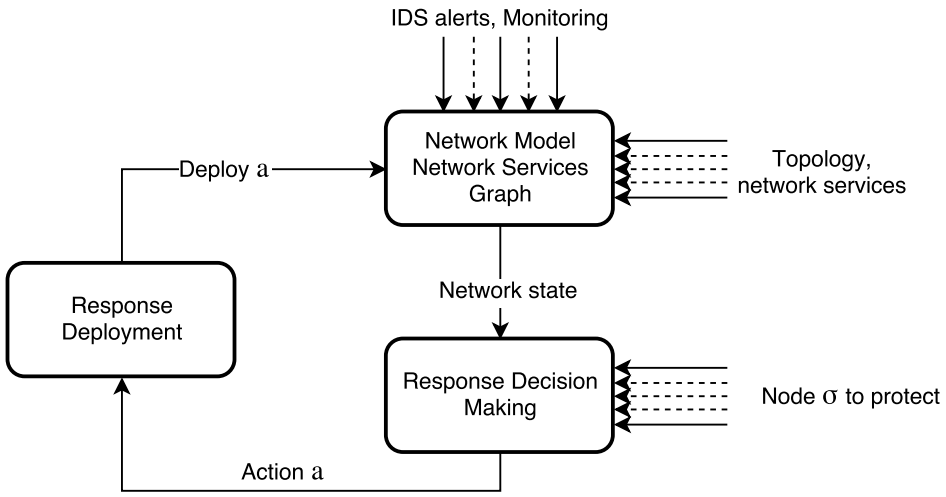


Figure 6.1: Our defender model. The defense module uses IDS alerts and monitoring data along with observed attacker steps to build a network model. Trying to protect a sensitive node σ, it builds a zero-sum game and solves for the saddle-point strategies in order to select an appropriate response action a. The *Response Deployment* module is then responsible for the implementation of a in the network.

6.3 OVERVIEW

We assume, in our framework, the presence of network level IDSs (such as Snort [191] and Bro [192]) that can provide the response engine with the necessary monitoring information. The response engine maintains the state of the network in the form of a network services graph, a graph data structure that represents the active services between nodes in the network. It then uses IDS information to define a labeling function over the graph that marks suspicious nodes and communications used for a possible compromise. Using the labels, the engine observes chains of communications between likely compromised nodes. Such chains are considered suspicious and require the engine to take immediate response actions. The engine considers all suspicious chains as hostile; its goal is to prevent any attackers from reaching specified sensitive nodes in the network, typically database servers or physical controllers.

From the observed states, the response engine can identify compromised nodes and possible target nodes for the attacker. It will take response actions that disconnect services from target nodes so that it prevents the attacker from reaching the sensitive node. This step can provide the network administrators with enough time to assess the IDS alerts and take appropriate actions to neutralize any threats. Figures 6.1 and 6.2 illustrate high-level diagrams of our response engine and a sample observed network state with 10 nodes, respectively.

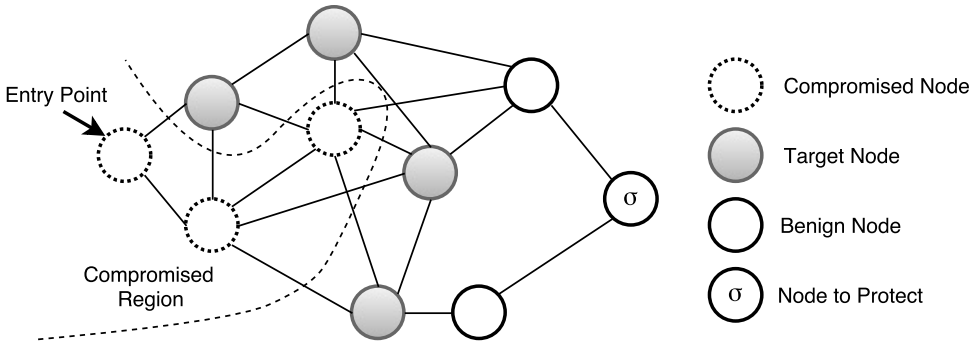


Figure 6.2: An illustration of our game model. The attacker has compromised 3 nodes in the network, and has four potential targets to compromise next. The defender, seeing the three compromised nodes, has to decide where the attacker is going to move next and disconnect services from the node, thus slowing down the attack.

Our threat model allows for the presence of a sophisticated attacker that has already established an entry point in an enterprise network, typically using spear phishing and social engineering, and aims to move laterally deeper into the network. Starting from a compromised node, the attacker identifies a set of possible target nodes for the next move. We assume that the attacker compromises one node at a time in order to avoid detection. We argue that this assumption is reasonable since attackers typically want to use legitimate administrator tools to hide their lateral movement activities [193]. Therefore, unlike computer worms that propagate widely and rapidly [194], lateral movement tends to be targeted, slow and careful. We will explore more sophisticated types of attackers with multi-move abilities in our future work.

Figure 6.2 illustrates an example network services graph with ten nodes, where an attacker has established a point of entry and already compromised three nodes. We highlight the target nodes that the attacker can choose to compromise next. We assume no prior knowledge of the strategy by which the attacker will choose the next node to compromise. Building our response engine on the assumption of like-minded attackers would lead to a false sense of security, since attackers with different motives would be able to overcome the responses of our engine, or possibly use them to their own advantage. Therefore, we formulate a defense-based game that attempts to protect a sensitive node in the network, regardless of the goals that the attacker is trying to achieve.

6.4 THE RESPONSE ENGINE

In this section, we formally introduce our response decision-making problem and its formulation as a zero-sum game. We provide formal definitions for the network state, attack

and response actions, and attack and response strategies, and then present how we build and solve the matrix game. We formulate the response engine's decision-making process as a complete information zero-sum game, in which the players are the engine and a potentially malicious attacker. We assume that both players take actions simultaneously, i.e., no player observes the action of the other before making its own move. In what follows, without loss of generality, we use the term attacker to refer to a suspicious chain of lateral movement communications. The response engine treats all communication chains as malicious and takes response actions accordingly. We use the terms *defender* and *response engine* interchangeably.

6.4.1 Definitions

Definition 6.1 (Network services graph)**.** A network services graph (NSG) is an undirected graph $G =< V, E >$ where V is the set of physical or logical nodes (workstations, printers, virtual machines, etc.) in the network and $E = V \times V$ is a set of edges.

An edge $e = (v_1, v_2) \in E$ represents the existence of an active network service, such as file sharing, SSH, or remote desktop connectivity, between nodes v_1 and v_2 in the network.
For any $v \in V$, we define a $\texttt{neighborhood}(v)$ as the set

$$\texttt{neighborhood}(v) = \{u \in V \mid \exists (u, v) \in E\} \tag{6.1}$$

Definition 6.2 (Alert labeling function)**.** Given an NSG $G =< V, E >$, we define an Alert Labeling Function (ALF) as a labeling function ℓ over the nodes V and edges E of G such that

$$\text{For } v \in V,\ \ell(v) = \begin{cases} \texttt{True} & \text{iff} \quad v \text{ is deemed compromised,} \\ \texttt{False} & \text{otherwise.} \end{cases} \tag{6.2}$$

$$\text{For } e = (u, v) \in E,\ \ell(e) = \begin{cases} \texttt{True} & \text{iff} \quad \ell(u) = \texttt{True} \wedge \ell(v) = \texttt{True}, \\ \texttt{False} & \text{otherwise.} \end{cases} \tag{6.3}$$

A *suspicious chain* is then a sequence of nodes $\{v_1, v_2, \ldots, v_k\}$ such that

$$\begin{cases} v_1, v_2, \ldots, v_k \in V, \\ (v_i, v_{i+1}) \in E \quad \forall i \in \{1, \ldots, k-1\}, \text{ and} \\ \ell(v_i) = \texttt{True} \quad \forall i \in \{1, \ldots, k\} \end{cases} \tag{6.4}$$

We assume that an ALF is obtained from monitoring information provided by IDSs such

as Snort [191] and Bro [192]. A suspicious chain can be either a malicious attacker moving laterally in the network, or a benign legal administrative task. The goal of our response engine is to slow the spread of the chain and keep it away from the sensitive infrastructure of the network, thus giving network administrators enough time to assess whether the chain is suspicious or not, and take appropriate corrective actions when needed.

Definition 6.3 (Network state)**.** We define the state of the network as a tuple

$$s = (G_s =< V_s, E_s >, \ell_s) \tag{6.5}$$

where G_s is an NSG and ℓ_s is its corresponding ALF. We use $\mathcal{S}$ to refer to the set of all possible network states.

For a given network state s, we define the set of vulnerable nodes $\mathcal{V}_s$ as

$$\mathcal{V}_s = \left\{ u \mid \left(u \in \bigcup_{v \in V_s \wedge \ell_s(v) = \texttt{True}} \texttt{neighborhood}(v) \right) \wedge \ell_s(u) = \texttt{False} \right\} \tag{6.6}$$

Definition 6.4 (Attack action)**.** Given a network state $s \in S$, an attack action a_e is a function over the ALFs, in which a player uses the service provided by edge $e = (v, v')$ such that $\ell_s(v) = \texttt{True}$ and $v' \in \mathcal{V}_s$, in order to compromise node v'. Formally we write

$$a_e(\ell_s) = \ell' \text{ such that } \ell'(v') = \texttt{True} \wedge \ell'(e) = \texttt{True} \tag{6.7}$$

For a network state s, the set of possible attack actions $\mathcal{A}_s$ is defined as

$$\mathcal{A}_s = \{a_e \mid e = (u, v) \in E_s \wedge \ell_s(u) = \texttt{True} \wedge v \in \mathcal{V}_s\} \tag{6.8}$$

Definition 6.5 (Response action)**.** Given a network state s, a response action d_v is a function over the NSG edges, in which a player selects a node $v \in \mathcal{V}_s$, and disconnects available services on all edges $e = (u, v) \in E_s$ such that $\ell_s(u) = \texttt{True}$. Formally, we write

$$d_v(E_s) = E' \text{ such that } E' = E_s \setminus \{(u, v) \in E_s \mid \ell_s(u) = \texttt{True}\} \tag{6.9}$$

For a network state s, we define the set of all possible response actions $\mathcal{D}_s$ as

$$\mathcal{D}_s = \{d_v \mid v \in \mathcal{V}_s\} \tag{6.10}$$

Definition 6.6 (Response strategy)**.** Given a network state s with a set of response actions

$\mathcal{D}_s$, a strategy $\mathbf{p}_r : \mathcal{D}_s \longrightarrow [0,1]^{|\mathcal{D}_s|}$ where $\sum_{d_v \in \mathcal{D}_s} \mathbf{p}_r(d_v) = 1$ is a probability distribution over the space of available response actions.

A response strategy $\mathbf{p}_r$ is a *pure response strategy* iff

$$\exists\ d_v \in \mathcal{D}_s \text{ such that } \mathbf{p}_r(d_v) = 1 \wedge (\forall d_{v'} \neq d_v,\ \mathbf{p}_r(d_{v'}) = 0) \tag{6.11}$$

A response strategy that is not pure is a *mixed response strategy.* Given a network state s, after solving a zero sum game, the response engine samples its response action according to the computed response strategy.

Definition 6.7 (Attack strategy)**.** Given a network state s and a set of attack actions $\mathcal{A}_s$, an attack strategy $\mathbf{p}_a : \mathcal{A}_s \longrightarrow [0,1]^{|\mathcal{A}|}$ where $\sum_{a_e \in \mathcal{A}_s} \mathbf{p}_a(a_e) = 1$ is a probability distribution over the space of available attack actions $\mathcal{A}_s$.

Definition 6.8 (Network next state)**.** Given a network state s, a response action $d_v \in \mathcal{D}_s$ for $v \in V_s$, and an attack action $a_e \in \mathcal{A}_s$ for $e = (u, w) \in E_s$, using Equations (6.7) and (6.9), we define the network next state (nns) as a function $\mathcal{S} \times \mathcal{D}_s \times \mathcal{A}_s \longrightarrow \mathcal{S}$ where

$$nns(s, d_v, a_e) = s' \text{ where } \begin{cases} (G_{s'} =< V_s, d_v(E_s) >, \ell_s) & \text{iff } v = w, \\ (G_{s'} =< V_s, d_v(E_s) >, a_e(\ell_s)) & \text{otherwise} \end{cases} \tag{6.12}$$

6.4.2 Formulation as a zero-sum game

The goal of our response engine is to keep an attacker, if any, as far away from a network's sensitive node (database server, SCADA controller, etc.) as possible. In the following, we assume that the engine is configured to keep the attacker at least `threshold` nodes away from a database server σ containing sensitive company data. The choices of `threshold` and σ are determined by the network administrators prior to the launch of the response engine.

Figure 6.3 shows the steps taken by our response engine at each time epoch $t_0 < t_1 < t_2 < \ldots < t$. In every step, the defender constructs a zero-sum defense-based matrix game and solves it for the saddle-point response strategy from which it samples an action to deploy. Assume that in a network state s, the response engine chooses to deploy action $d_v \in \mathcal{D}_s$ for $v \in V_s$, and the attacker chooses to deploy action $a_e \in \mathcal{A}_s$ for $e = (u, w) \in E_s$. In other words, the defender disconnects services from node v in the network while the attacker compromises node w starting from the already compromised node u. If $v = w$, then the attacker's efforts were in vain and the response engine was able to guess correctly where the attacker would move next. However, when $v \neq w$, the attacker would have successfully

1: **for each** time epoch $t_0 < t_1 < t_2 < \ldots$ **do**
2: (1) Obtain network state $s = (G_s, \ell_s)$.
3: (2) Compute the sets of possible attack and response actions $\mathcal{A}_s$ and $\mathcal{D}_s$
4: (3) Compute the payoff matrix $M_s = \texttt{BUILD\_GAME}(\mathcal{A}_s, \mathcal{D}_s, \texttt{threshold}, \sigma)$
5: (4) Compute the equilibrium response strategy $\hat{\mathbf{p}}_r$
6: (6) Sample response action $d_v \in \mathcal{D}_s$ from $\hat{\mathbf{p}}_r$
7: **end for**

Figure 6.3: The steps taken by our response engine at each time epoch. The engine first obtains the state of the network from the available monitors, and uses it to compute the sets of possible attack and response actions $\mathcal{A}_s$ and $\mathcal{D}_s$. It then builds the zero-sum game matrix M_s using Algorithm 6.1, and solves for the equilibrium response strategy $\hat{p}_s$. It finally samples a response action d_v from $\hat{p}_s$ that it deploys in the network.

compromised the node w. Note that this is not necessarily a loss, since by disconnecting services from certain nodes on the path, the response engine might be redirecting the attacker away from the target server σ. Furthermore, by carefully selecting nodes to disconnect, the engine can redirect the attacker into parts of the network where the attacker can no longer reach the target server σ, and thus cannot win the game. The attacker wins the game when it is able to reach a node within one hop of target server σ. The game ends when (1) the attacker reaches σ; (2) either player runs out of moves to play; or (3) the attacker can no longer reach σ.

Let $\texttt{sp}(u, \sigma)$ be the length of the shortest path (in number of edges) in G_s from node u to the target server σ. We define the payoffs for the defender in terms of how far the compromised nodes are from the target server σ. A positive payoff indicates that the attacker is more than `threshold` edges away from σ. A negative payoff indicates that the attacker is getting closer to σ, an undesirable situation for our engine. Therefore, we define the payoff for the defender when the attacker compromises node w as $\texttt{sp}(w, \sigma) - \texttt{threshold}$. If $\texttt{sp}(w, \sigma) > \texttt{threshold}$ then the attacker is at least $\texttt{sp}(w, \sigma) - \texttt{threshold}$ edges away from the defender's predefined dangerous zone. Otherwise, attacker is $\texttt{threshold} - \texttt{sp}(w, \sigma)$ edges into the defender's dangerous zone. Moreover, when the defender disconnects a node w that the attacker wanted to compromise, two cases might arise. First, if $\texttt{sp}(w, \sigma) = \infty$, i.e., w cannot reach σ, then it is desirable for the defender to lead the attacker into w, and thus the engine assigns d_w a payoff of 0 so that it wouldn't consider disconnecting w. Otherwise, when $\texttt{sp}(w, \sigma) < \infty$, by disconnecting the services of w, the defender would have canceled the effect of the attacker's action, and thus considers it a win with payoff $\texttt{sp}(w, \sigma) < \infty$.

Algorithm 6.1 illustrates how our response engine builds the zero-sum matrix game. For each network state s, the algorithm takes as input the set of response actions $\mathcal{D}_s$, the set

of attack actions $\mathcal{A}_s$, the defender's `threshold`, and the target server to protect σ. The algorithm then proceeds by iterating over all possible combinations of attack and response actions and computes the defender's payoffs according to Equation (6.13). It then returns the computed game payoff matrix M_s with dimensions $|\mathcal{D}_s| \times |\mathcal{A}_s|$.

Formally, for player actions $d_v \in \mathcal{D}_s$ and $a_e \in \mathcal{A}_s$ where $v \in V_s$ and $e = (u, w) \in E_s$, we define the response engine's utility as

$$u_d(d_v, a_e) = \begin{cases} 0 & \text{iff } v = w \wedge \texttt{sp}(w, \sigma) = \infty \\ \texttt{sp}(w, \sigma) & \text{iff } v = w \wedge \texttt{sp}(w, \sigma) < \infty \\ \texttt{sp}(w, \sigma) - \texttt{threshold} & \text{iff } v \neq w \end{cases} \tag{6.13}$$

Since the game is zero-sum, the utility of the attacker is $u_a(a_e, d_v) = -u_d(d_v, a_e)$.

For a response strategy $\mathbf{p}_r$ over $\mathcal{D}_s$ and an attack strategy $\mathbf{p}_a$ over $\mathcal{A}_s$, the response engine's expected utility is defined as

$$U_d(\mathbf{p}_r, \mathbf{p}_a) = \sum_{d_v \in \mathcal{D}_s} \sum_{a_e \in \mathcal{A}_s} \mathbf{p}_r(d_v) u_d(d_v, a_e) \mathbf{p}_a(a_e) \tag{6.14}$$

Similarly, the attacker's expected payoff is $U_a(\mathbf{p}_a, \mathbf{p}_r) = -U_d(\mathbf{p}_r, \mathbf{p}_a)$.

In step 4 of Figure 6.3, the response engine computes the saddle-point response strategy $\hat{\mathbf{p}}_r$ from which it samples the response action to deploy. $\hat{\mathbf{p}}_r$ is the best response strategy that the engine could adopt for the worst-case attacker. Formally, for saddle-point strategies $\hat{\mathbf{p}}_r$ and $\hat{\mathbf{p}}_a$,

$$\begin{aligned} &U_d(\hat{\mathbf{p}}_r, \hat{\mathbf{p}}_a) \geq U_d(\mathbf{p}_r, \hat{\mathbf{p}}_a) \text{ for all } \mathbf{p}_r, \text{ and} \\ &U_a(\hat{\mathbf{p}}_a, \hat{\mathbf{p}}_r) \leq U_a(\mathbf{p}_a, \hat{\mathbf{p}}_r) \text{ for all } \mathbf{p}_a \end{aligned} \tag{6.15}$$

Finally, the engine chooses an action $d_v \in \mathcal{D}_s$ according to the distribution $\hat{\mathbf{p}}_r$ and deploys it in the network. In this chapter, we assume that response actions are deployed instantaneously and successfully at all times; response action deployment challenges are beyond the scope of this work.

6.5 IMPLEMENTATION AND RESULTS

We implemented a custom Python simulator in order to evaluate the performance of our proposed response engine. We use Python iGraph [195] to represent NSGs, and implement ALFs as features on the graphs' vertices. Since the payoffs for the response engine's actions are highly dependent on the structure of the NSG, we use three different graph topol-

Algorithm 6.1: Algorithm M_s = `BUILD_GAME` $(\mathcal{D}_s, \mathcal{A}_s, \texttt{threshold}, \sigma)$

Inputs: $\mathcal{D}_s$, $\mathcal{A}_s$, `threshold`, σ
Outputs: Zero-sum game payoff matrix M_s
for each response action $d_v \in \mathcal{D}_s$ **do**
  for each attack action $a_e \in \mathcal{A}_s$ **do**
    let $e \leftarrow (u, w)$
    if $v = w$ **then**
      if $\texttt{sp}(w, \sigma) = \infty$ **then**
        $M_s(v, w) \leftarrow 0$
      else
        $M_s(v, w) \leftarrow \texttt{sp}(w, \sigma)$
      end if
    else
      $M_s(v, w) \leftarrow \texttt{sp}(w, \sigma) - \texttt{threshold}$
    end if
  end for
end for

ogy generation algorithms to generate the initial graphs. The Waxman [196] and Albert-Barabási [197] algorithms are widely used to model interconnected networks, especially for the evaluation of different routing approaches. In addition, we generate random geometric graphs, as they are widely used for modeling social networks as well as studying the spread of epidemics and computer worms [198, 199]. Because of the lack of publicly available data sets capturing lateral movement, we assume that the Waxman and Albert-Barabási models provide us with an appropriate representation of the structural characteristics of interconnected networks.

We use the geometric graph models in order to evaluate the performance of our engine in highly connected networks. We pick the initial attacker point of entry in the graph ω and the location of the database server σ such that $sp(\omega, \sigma) = d$, where d is the diameter of the computed graph. This is a reasonable assumption, since in APTs, attackers usually gain initial access to the target network by targeting employees with limited technical knowledge (such as customer service representatives) through social engineering campaigns, and then escalate their privileges while moving laterally in the network.

We implement our response engine as a direct translation of Figure 6.3 and Algorithm 6.1, and we use the Gambit [200] Python game theory API in order to solve for the saddle-point strategies at each step of the simulation. We use the NumPy [201] Python API to sample response and attack actions from the computed saddle-point distributions. As stated earlier, we assume that attack and response actions are instantaneous and always successful, and

thus implement the actions and their effects on the network as described in the network next-state function in Equation (6.12).

We evaluate the performance of our response engine by computing the average percentage increase in the number of attack steps (i.e., compromises) needed by an adversary to reach the target server σ. We compute the average increase with respect to the shortest path that the attacker could have adopted in the absence of the response engine. Formally, let k be the number of attack steps needed to reach σ and d be the diameter of the NSG; then, the percentage increase in attack steps is $\frac{k-d}{d} \times 100$. If the attacker is unable to reach the target server, we set the number of attack steps k to the maximum allowed number of rounds of play in the simulation, which is 40 in our simulations.

In addition, we report on the average attacker distance from the server σ as well as the minimum distance that the attacker was able to reach. As discussed earlier, we measure the distance in terms of the number of attack steps needed to compromise the server. A minimum distance of 1 means that the attacker was able to successfully reach σ. We also report and compare the average achieved payoff for the defender while playing the game. We ran our simulations on a Macbook Pro laptop running OSX El Capitan, with 2.2 GHz Intel Core i7 processors and 16 GB of RAM. We start by describing our results for various defender `threshold` values for an NSG with 100 nodes, and then fix the `threshold` value and vary the number of nodes in the NSG. Finally, we report on performance metrics in terms of the time needed to perform the computation for various NSG sizes.

6.5.1 Evaluation of `threshold` values

We start by evaluating the performance of our response engine for various values of the threshold above which we would like to keep the attacker away from the sensitive node σ. We used each graph generation algorithm to generate 10 random NSGs, simulated the game for `threshold` $\in \{1, 2, 3, 4, 5, 6\}$, and then computed the average values of the metrics over the ten runs.

Table 6.1 shows the structural characteristics in terms of the number of vertices, average number of edges, diameter, and maximum degree of the graphs generated by each algorithm. All of the graphs we generated are connected, with the geometric graphs showing the largest levels of edge connectivity, giving attackers more space to move in the network. The Waxman and Barabási generators have lower levels of edge connectivity, making them more representative of network services topologies than the geometric graphs are.

Figure 6.4a shows the average percentage increase in attacker steps needed to reach the target (or reach the simulation limit) for the various values of `threshold`. The results show

Table 6.1: Characteristics of generated NSGs (averages)

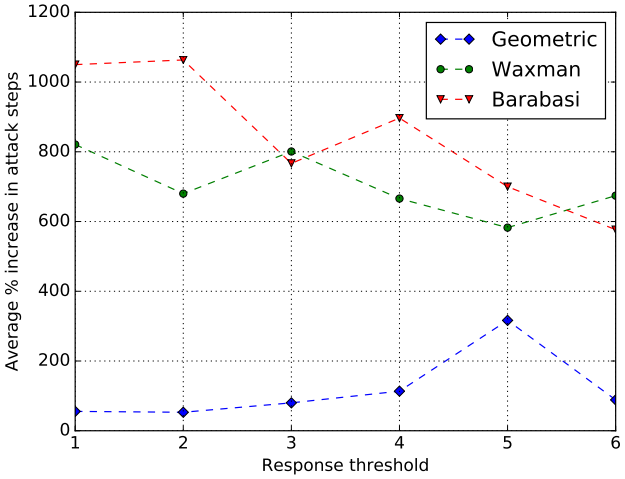
NSG Generator	$\|V\|$	$\|E\|$	Diameter	Max Degree
Barabási	100	294	4	50.2
Waxman	100	336.6	4.9	13.7
Geometric	100	1059.8	5.2	34.5

that in all cases, our engine was able to increase the number of steps needed by the attacker by at least 50%. Considering only the Waxman and Barabási graphs, the engine was able to increase the number of steps needed by the attacker by at least 600%. This is a promising result that shows the effectiveness of our engine, especially in enterprise networks. Further, the results show that smaller values for `threshold` achieve a greater average increase in attacker steps. This is further confirmed by the average defender payoff curves shown in Figure 6.4b, in which smaller values of `threshold` achieve greater payoffs. In fact, this result is a direct implication of our definition of the payoff matrix values in Equation (6.13). The smaller the values of `threshold`, the more the engine has room to take actions that have a high payoff, and the more effective its strategies are in keeping the attacker away from the server.

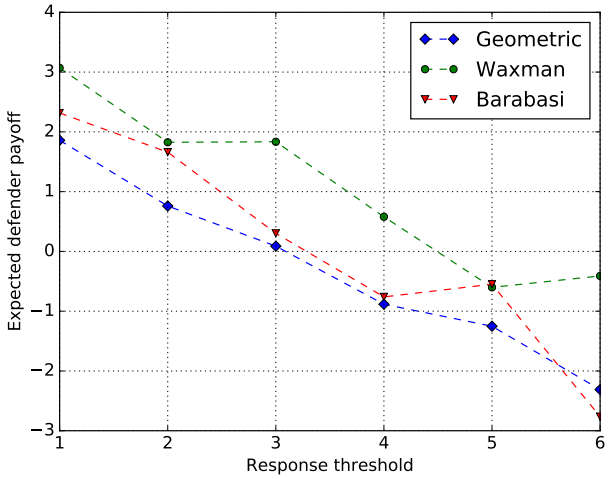
Figures 6.4c and 6.4d show the average distance between the attacker and the server, and the minimum distance reached by the attacker, respectively. For the Waxman and Barabási graphs, the results show that our engine keeps the attacker, on average, at a distance close to the graph's diameter, thus keeping the attacker from penetrating deeper into the network. For both types of graphs, Figure 6.4d confirms that the attacker was unable to reach the target server (average minimum distance ≥ 1).

In the case of the geometric graphs, Figure 6.4d shows that the attacker was almost always able to reach the target server. We attribute this attacker success to the high edge connectivity in the geometric graphs. Although our engine is able to delay attackers, because of the high connectivity of the graph, they may find alternative ways to reach the server. Nevertheless, our response engine was always able to cause at least a 50% increase in the number of attack steps needed to reach the server.

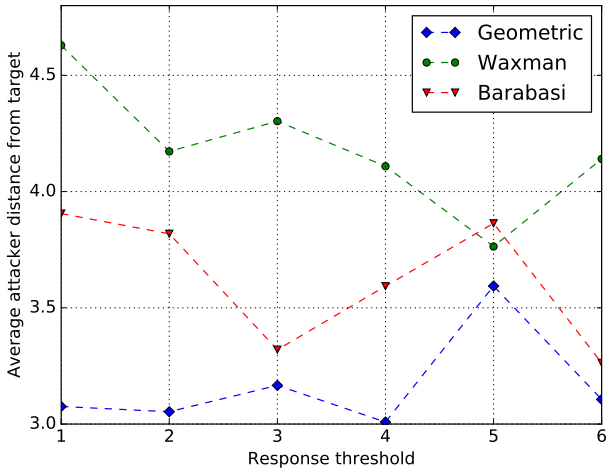
In summary, the results show that our response engine is able to effectively delay, and on average prevent, an attacker that is moving laterally in the network from reaching the target database server. It was effectively able to increase the number of attack steps needed by the adversary by at least 600% for the graphs that are representative of real-world network topologies. In addition, even when the graphs were highly connected, our engine was still able to increase the attacker's required amount of attack steps by at least 50%.



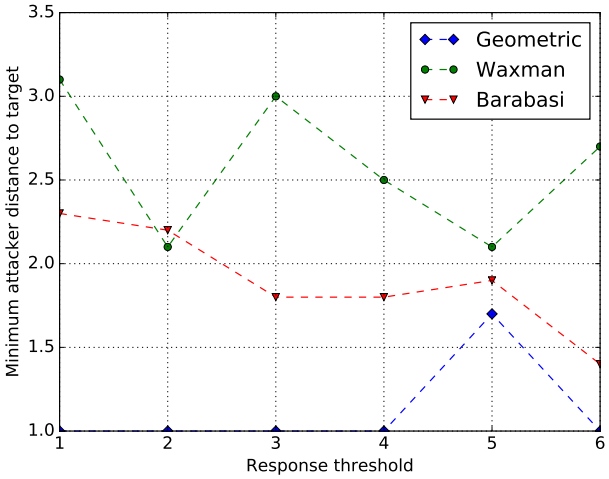
(a) Average % increase in attack steps



(b) Average defender payoff



(c) Average attacker distance from σ

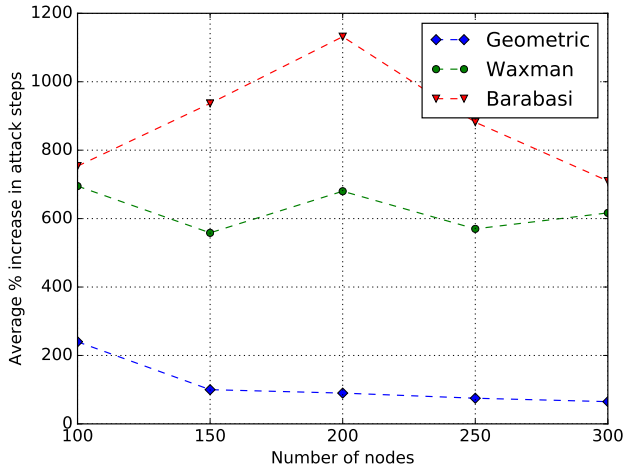


(d) Attacker's minimum distance from σ

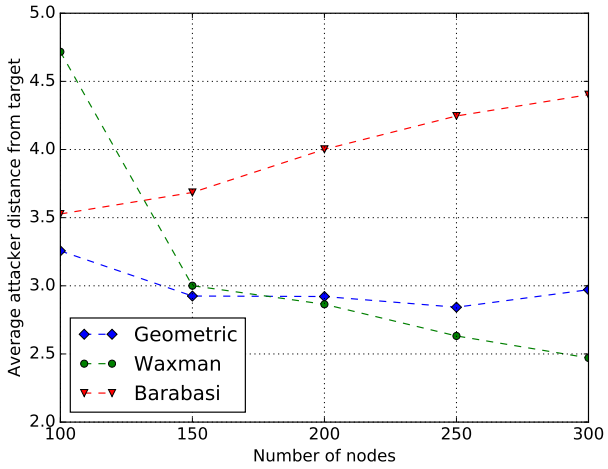
Figure 6.4: Performance evaluation of our response engine with varying threshold values. Figure 6.4a shows that our engine was able to increase the number of compromises needed by the attacker by at least 55%. Figure 6.4b illustrates that the zero-sum game's payoff for the defender decreases almost linearly as the `threshold` increases. Figure 6.4c shows that the average attacker's distance from σ is very close to the NSG's diameter, while Figure 6.4d shows that, with the exception of the geometric NSG, our engine was able to keep that attacker from reaching the target data server σ. It was able, however, in the geometric NSG case, to increase the number of compromises needed to reach σ by at least 55%.

6.5.2 Scalability

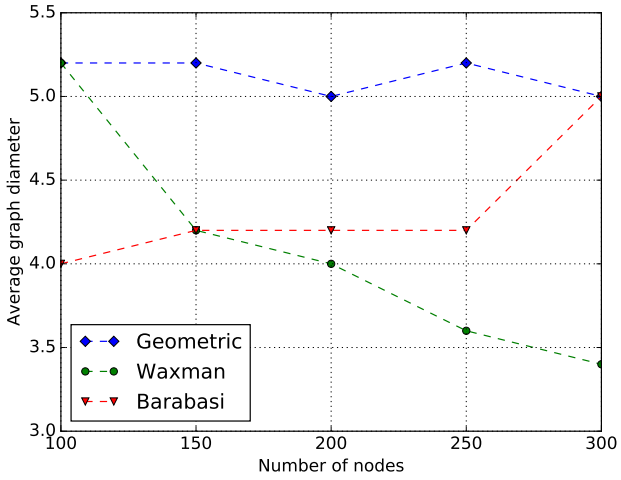
Next, we measured the scalability of our response engine as the network grew in size. We varied the number of nodes in the network from 100 to 300 in steps of 50 and measured the



(a) Average % increase in attack steps



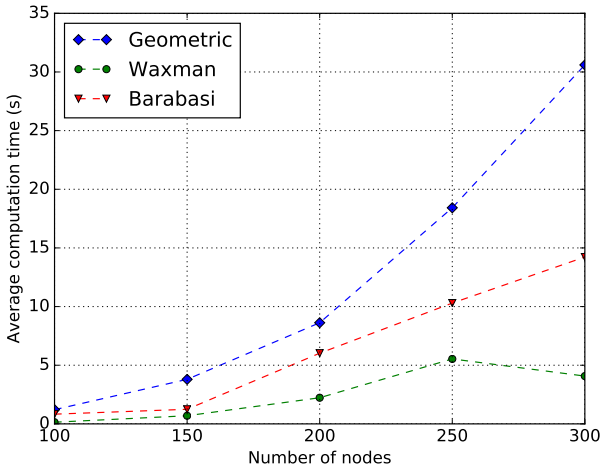
(b) Average attacker distance from σ



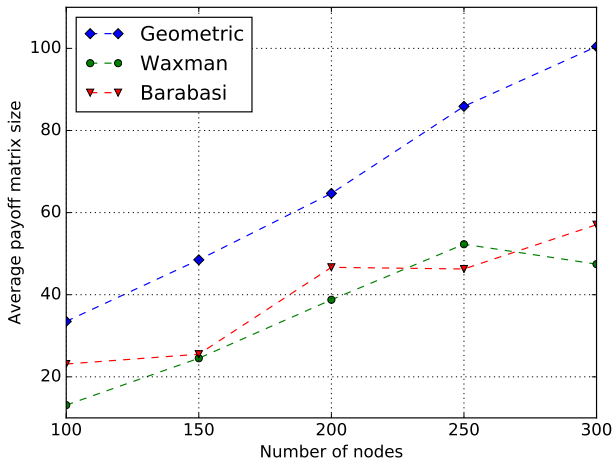
(c) Average graph diameter

Figure 6.5: Performance evaluation of our response engine with increasing number of nodes in the network. Figure 6.5a shows that our engine maintains high levels of performance even when the network grows larger. The engine is also capable of keeping the attacker at an average distance close to the graph's diameter in the cases of the Waxman and Barabási NSGs, as shown in Figures 6.5b and 6.5c.

average percentage increase in attack steps as well as the attacker's average distance from the target σ. Figure 6.5 shows our results for averages measured over five random NSGs generated by each of the NSG generation algorithms. We set the defender's `threshold` values to those that achieved a maximum average increase in attack steps as shown in Figure 6.4a, which are 5 for geometric NSGs, 2 for Barabási NSGs, and 3 for Waxman NSGs.



(a) Average time (s) to solve matrix game



(b) Average size of the matrix game

Figure 6.6: Computational performance evaluation of the engine for larger networks. Our response engine scales well with the increase in the size of the network.

As shown in Figure 6.5a, our response engine can scale well as the size of the network increases, providing average percentage increases in attack steps between 550% and 700% for Waxman NSGs, 750% and 1150% for Barabási NSGs, and 50% and 220% for geometric NSGs. These results show that as the number of nodes, and thus the number of connection edges, increases in the network, our engine is able to maintain high-performance levels and delay possible attackers, even when they have more room to evade the engine's responses and move laterally in the network. This is further confirmed by the results shown in Figures 6.5b and 6.5c. For the Waxman and Barabási NSGs, the response engine is always capable of keeping the attacker at an average distance from the target server equal to the diameter of the graph. For the geometric NSGs, the attacker is always capable of getting close to and reaching the target server, regardless of the diameter of the graph. Our engine, however, is always capable of increasing the number of attack steps required by at least 50%, even for larger networks.

6.5.3 Computational performance

Finally, we evaluated the computational performance of our game engine as the scale of the network increased from 100 to 300 nodes. We used the same values for `threshold` as in the previous subsection, and measured the average time to solve for the saddle-point strategies as well as the average size of the matrix game generated during the simulation. Since all of the payoff matrices we generated are square, we report on the number of rows

in the matrix games. The rows correspond to the number of available attack or response actions for the players (i.e., for a state s, we report on $|\mathcal{A}_s| = |\mathcal{D}_s|$). Our engine makes use of the `ExternalLogitSolver` solver from the Gambit software framework [200] to solve for the saddle-point strategies at each step of the simulation. In computing our metrics, we averaged the computation time and matrix size over 10 random graphs from each algorithm, and we limited the number of steps in the simulation (i.e., the number of game turns) to 10.

Figure 6.6b shows that for all NSG-generation algorithms, the size of the payoff matrices for the generated zero-sum game increases almost linearly with the increase in the size of the nodes in the network. In other words, the average number of available actions for each player increases linearly with the size of the network. Consequently, Figure 6.6a shows that the computational time needed to obtain the saddle-point strategies scales very efficiently with the increase in the size of the network; the engine was able to solve 50×50 matrix games in 15 seconds, on the average. The short time is a promising result compared to the time needed by an administrator to analyze the observed alerts and deploy strategic response actions.

In summary, our results clearly show the merits of our game engine in slowing down the advance of an attacker that is moving laterally within an enterprise network, and its ability to protect a sensitive database server effectively from compromise. For all of the NSG-generation algorithms, our engine was able to increase the number of attack steps needed by an attacker to reach the sensitive server by at least 50%, with the value increasing to 600% for the Waxman and Barabási NSG-generation algorithms. The results also show that our engine is able to maintain proper performance as networks grow in size. Further, the computational resources required for obtaining the saddle-point strategies increased linearly with the number of the nodes in the network.

6.6 RELATED WORK

Several researchers have tackled the problem of selecting cyber actions as a response to intrusions. The space can be divided into three parts; automated response through rule-based methods, cost-sensitive methods, and security games.

In rule-based intrusion response, each kind of intrusion alert is tagged with a suitable response. The static nature of rule-based intrusion response makes it predictable and limits its ability to adapt to different attacker strategies. Researchers have extended rule-based intrusion response systems to become cost-sensitive; cost models range from manual assessment of costs to use of dependency graphs on the system components to compute a response action's cost. In all of those cases, the process of selecting a response minimizes the cost of

response actions over a set of predefined actions that are considered suitable for tackling a perceived threat. Stakhanova surveyed this class of systems in [194]. While cost-sensitive intrusion response systems minimize the cost of responding, they are still predictable by attackers, and a large effort is required in order to construct the cost models.

Bloem *et. al.* [202] tackled the problem of intrusion response as a resource allocation problem. Their goal was to manage the administrator's time, a critical and limited resource, by alternating between the administrator and an imperfect automated intrusion response system. The problem is modeled as a nonzero-sum game between automated responses and administrator responses, in which an attacker gain (utility) function is required. Obtaining such functions, however, is hard in practice, as attacker incentives are not known. The problem of finding attacker-centric metrics was tackled by ADAPT [203]. The ADAPT developers attempted to find a taxonomy of attack metrics that require knowledge of the cost of an attack and the benefit from the attack. ADAPT has created a framework for computing the metrics needed to set up games; however, assigning values to the parameters is still more of an art than a science.

Use of security games improved the state of IRSs, as they enabled modeling of the interaction between the attacker and defender, are less predictable, and can learn from previous attacker behavior [21, 22]. In [204], the authors model the security game as a two-player game between an attacker and a defender; the attacker has two actions (to attack or not attack), and the defender has two actions (to monitor or not monitor). The authors consider the interaction as a repeated game and find an equilibrium strategy. Nguyen *et. al.* [205] used fictitious play to address the issue of hidden payoff matrices. While this game setup is important on a high level and can be useful as a design guideline for IDSs, it does not help in low-level online response selection during a cyber attack.

To address the issue of high level abstraction in network security games, Zonouz [206] designed the Response and Recovery Engine (RRE), an online response engine modeled as a Stackelberg game between an attacker and a defender. Similar to work by Zhu and Başar [207], the authors model the system with an attack response tree (ART); the tree is then used to construct a competitive Markov decision process to find an optimal response. The state of the decision process is a vector of the probabilities of compromise of all the components in the system. The authors compute the minimax equilibrium to find an optimal response. The strategy is evaluated for both finite and infinite horizons. Scalability issues are tackled using finite lookahead. The game, however, has several limitations: (1) the model is sensitive to the assigned costs; (2) the model required *a priori* information on attacks and monitoring (conditional probabilities) which is not available; and (3) the system uses a hard-to-design ART to construct the game.

6.7 DISCUSSION AND FUTURE WORK

The goals of our response engine are to provide networked systems with the ability to maintain acceptable levels of operation in the presence of potentially malicious actors in the network, and to give administrators enough time to analyze security alerts and neutralize any threats, if present. Our results show that the engine is able to delay, and often prevent, an attacker from reaching a sensitive database server in an enterprise network. However, the response actions that our engine deploys can have negative impacts on the system's provided services and overall performance. For example, disconnecting certain nodes as part of our engine's response to an attacker can compromise other nodes' ability to reach the database service. This can have severe impacts on the system's resiliency, especially if it is part of a service provider's infrastructure. In the future, we plan to augment our engine with response action cost metrics that reflect their impact on the network's performance and resiliency. We plan to add support for a resiliency budget that the engine should always meet when making response action decisions. In addition, we will investigate deployment challenges for the response actions. We envision that with the adoption of Software Defined Networks (SDNs), the deployment of such actions will become easier. Our engine can be implemented as part of the SDN controller and use the southbound API to deploy its response actions.

In the context of APTs, attackers are often well-skilled, stealthy, and highly adaptive actors that can adapt to the changes in the network, including the response actions deployed by our engine. We will investigate more sophisticated models of attackers, specifically ones that can compromise more than one node in each attack step, and can adapt in response to our engine's deployed actions. In addition, knowledge of the attacker's strategies and goals would provide our response engine with the ability to make more informed strategic decisions about which response actions to deploy. Therefore, we plan to investigate online learning techniques that our engine can employ in order to predict, with high accuracy, an attacker's strategies and goals. However, the main challenge that we face in our framework's design and implementation is the lack of publicly available datasets that contain traces of attackers' lateral movements in large-scale enterprise networks. In addition to simulations, we will investigate alternative methods with which we can evaluate our response engine and the learning techniques that we devise.

6.8 CONCLUSION

Detection of and timely response to network intrusions go hand-in-hand when secure and resilient systems are being built. Without timely response, IDSs are of little value in the

face of APTs; the time delay between the sounding of IDS alarms and the manual response by network administrators allows attackers to move freely in the network.We have presented an efficient and scalable game-theoretic response engine that responds to an attacker's lateral movement in an enterprise network, and effectively protects a sensitive network node from compromise. Our response engine observes the network state as a network services graph that captures the different services running between the nodes in the network, augmented with a labeling function that captures the IDS alerts concerning suspicious lateral movements. It then selects an appropriate response action by solving for the saddle-point strategies of a defense-based zero-sum game, in which payoffs correspond to the differences between the shortest path from the attacker to a sensitive target node, and an acceptable engine safety distance threshold. We have implemented our response engine in a custom simulator and evaluated it for three different network graph generation algorithms. The results have shown that our engine is able to effectively delay, and often stop, an attacker from reaching a sensitive node in the network. The engine scales well with the size of the network, maintaining proper operation and efficiently managing computational resources. Our results show that the response engine constitutes a significant first step towards building secure and resilient systems that can detect, respond to, and eventually recover from malicious actors.

CHAPTER 7: CONCLUSION

The pervasiveness of targeted and sophisticated cyber attacks has pushed system designers to face the unfortunate reality that attacks are now the norm. With the increased dependence of our critical infrastructure on networking platforms, failure to address such attacks can have devastating consequences. The perfectly hardened and secure system is now a myth; system design has moved from an era focused on robustness and hardening, to one focused on cyber resilience. Much like fault-tolerant design, cyber-resilient systems admit the presence of malicious attacks and attempt to provide continuous service, even if in a degraded state, throughout the lifetime of an attack. Such systems then move into a period of graceful restoration of full service after threats have been detected and neutralized. Such continuity and restoration of service are a necessary safeguard against the drastic economic and societal consequences of cyber attacks.

In this dissertation, we set out to show that the cyber resilience posture of our networked infrastructure can be enhanced through systematic integration of sound theoretical analysis and practically realizable design. We first presented the motivation for cyber resilience and its various definitions in Chapter 2. We also illustrated how, in foundational work on TCP congestion control, the integration of control theory and practical design led to the emergence of a reliable communication protocol that can handle changes in the state of the network, and that still occupies that largest portion of today's Internet traffic.

To validate our thesis, we focused on two aspects of resilience: (1) inter-networking resilience, and (2) intra-networking resilience. At the inter-networking level, we focused on DDoS attacks, which continue to plague the Internet to this day. We presented CPuzzle (Chapter 3) and Midgard (Chapter 4) to combat DDoS attacks at the transport and network layers, respectively. At the core of CPuzzle and Midgard's designs is the observation that achieving resilience to DDoS attacks must involve end users, since the cost of launching such attacks is vastly smaller than the cost of defending against them in the network. CPuzzle is an enhancement to TCP that provides resilience to state-exhaustion attacks by forcing users and attackers to solve computational puzzles. We set the difficulty of the puzzles by using a sound game-theoretic model that uses easy-to-obtain parameters. Our analysis shows that the model illustrates an important trade-off between a TCP server's provisioning and the difficulty of the challenges it can ask its users to solve. Our results from deploying CPuzzle on the DETER testbed show that it increases resilience to state-exhaustion attacks by increasing the cost of launching an attack and effectively rate-limiting attacks that attempt to overwhelm the TCP server's state.

We designed Midgard (Chapter 4) to extend CPuzzle's results into the network layer and provide resilience to volumetric DDoS attacks. Current approaches to DDoS protection rely heavily on the availability of vast bandwidth resources that can absorb incoming attacks and filter out malicious traffic. That has effectively transformed the DDoS battlefield into a continuous bandwidth war between attackers and cloud providers. Unfortunately, recent DDoS attacks have shown that attackers are gaining the ability to compromise larger and larger botnets, especially with the advent of IoT devices with poor security measures, and are able to launch unprecedented attacks. Midgard attempts to enhance our networks' resilience to volumetric attacks by combining the benefits of cloud protection services with the benefits of employing client puzzles. We designed Midgard to be resilient to puzzle reuse by leveraging the puzzles' cryptographic properties. We then designed a controller that estimates each user's computational prowess and allocates bandwidth resources in a fairness-preserving manner. We deployed Midgard on a sample topology on DETER. Our results show that Midgard can effectively hinder the effects of volumetric DDoS attacks by rate-limiting misbehaving users and allocating more bandwidth to benign clients.

At the intra-networking level, we introduced sShield (Chapter 6), a game-theoretic engine that actively manipulates network topologies to protect high-value assets (such as data stores) from potential attackers who are moving laterally in a network. We modeled the problem as a zero-sum game between an attacker who aims to compromise a high-value asset and a defender who wants to prevent said attacker from reaching the asset. Using the game-theoretic decisions, sShield can manipulate the network connectivity to build a protective entourage around the asset to prevent the attacker from reaching it.

sShield's practicality is enabled by the advent of SDN that decouple a network's control plane (i.e., the logic used to make traffic decisions) from the data plane (the forwarding pipelines at each individual switch). Unfortunately, programmability and flexibility in the data plane can lead to design-time bugs, which poses serious security and reliability challenges and thus hinders the network's resilience. Therefore, we designed and implemented BiFrost (Chapter 5), a tool for the static verification of programmable data-plane programs that uses sequential circuit analysis and verification techniques. Our design of BiFrost is based on the observation that data plane programming languages describe a limited hardware pipeline with no dynamic memory allocation. Therefore, instead of treating data plane programs as general-purpose programs, BiFrost translates them into equivalent sequential circuits. BiFrost then leverages a rich set of sequential analysis, abstraction, synthesis, and verification techniques to scalably verify that the data plane programs do not read or write invalid header fields and that they satisfy user-defined properties. We evaluated BiFrost on a set of real-world data plane programs. We found two header-validity bugs in publicly

available programs and showed that BiFrost can efficiently verify large and complex data plane programs.

7.1 FUTURE DIRECTIONS

Our societies are becoming more and more dependent on interconnected devices for providing essential and critical services. It is therefore of paramount importance to adopt a "resiliency mindset" to safeguard our infrastructure against failures and an avalanche of targeted and sophisticated attacks. To that end, we believe that our designs must be built on a strong theoretical foundation that can be manifested in practical implementations. We believe that the goals we set out to achieve in this thesis are aligned with the recent move in the security community, pioneered by several government agencies [20, 40, 208, 209, 210], to establish a "science of security" that guides the performance of security research and experimentation.

We believe that our work in CPuzzle and Midgard shows that end-user involvement in the process of protecting our networks against DDoS attacks can be beneficial. In fact, such approaches have already been implemented to protect login forms from brute-force attacks, for example, through the use of CAPTCHAs. Unfortunately, such approaches require direct involvement of the user and can often be cumbersome, especially if requested frequently [211, 212]. In CPuzzle and Midgard, the user's involvement in the protection service is seamless and does not require direct intervention. Rather, our theoretical modeling takes into consideration the users' preferences in terms of the amount of computational effort they are willing to contribute to tolerate an attack. Therefore, we envision that the integration of theory and practicality showcased in our DDoS resilience designs can be extended to other applications and protocols, which will be especially important now that attackers are capable of easily acquiring botnets consisting of hundreds of thousands to millions of unsuspecting machines.

Furthermore, our design of sShield provides a motivating example of how network programmability can be leveraged to deploy dynamic network responses that can adapt the state of the network in the face of failures and attacks. Specifically, the decoupling of the control plane from the data plane allows network controllers to access a vast array of computational resources. For example, controllers can run on commodity machines or powerful servers. Such computational resources can be used to perform decision algorithms (such as the game-theoretic approach we presented) that are based on sound theoretical models with real-time inputs from the data plane. For example, our game-theoretic model in sShield can be extended to perform deception actions. Such actions would lead an attacker who

is moving laterally in a network to an isolated subnetwork (i.e., an active honeypot) where administrators can safely analyze the threat and take response actions.

Finally, the deployment of programmable networks must be accompanied by extensive testing and verification of network programs, especially as we are moving into the era of full programmability of both the control-plane and the data-plane. In that context, our design of BiFrost illustrates how repurposing traditional verification techniques for data-plane programs can achieve high levels of scalability and detect design bugs. However, some faults can occur because to the interactions among multiple forwarding devices. Therefore, we plan to extend BiFrost to support the verification of network-wide properties across multiple programmable switches. In addition, as shown in [213], attackers can use data plane devices to exploit security vulnerabilities in the control plane and carry out unauthorized traffic decisions. Therefore, network verification would not be complete if it did not support verification of the interactions between the data-plane and the control-plane devices, ensuring that one cannot be used to inject faults and exploit vulnerabilities in the other. We believe that the problem resembles that of verifying the properties that account for the interaction between software APIs and hardware devices [214, 215], and we plan to extend BiFrost to provide support for such verification efforts.

REFERENCES

[1] A. Kavanaugh, "The impact of computer networking on community: A social network analysis approach," in *Telecommunications Policy Research Conference*, 1999, pp. 27–29.

[2] S. Kottler, "February 28th DDoS incident report," March 2018. [Online]. Available: https://github.blog/2018-03-01-ddos-incident-report/

[3] M. Locklear, "Mirai botnet creators plead guilty to charges over 2016 attack," December 2017. [Online]. Available: https://www.engadget.com/2017/12/13/mirai-botnet-creators-guilty-plea/

[4] T. S. Bernard, T. Hsu, N. Perlroth, and R. Lieber, "Equifax says cyberattack may have affected 143 million in the U.S." September 2017. [Online]. Available: https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html

[5] B. Krebs, "3 million customer credit, debit cards stolen in Michaels, Aaron Brothers breaches," April 2014. [Online]. Available: https://krebsonsecurity.com/2015/09/inside-target-corp-days-after-2013-breach/

[6] B. Krebs, "Inside Target Corp., days after 2013 breach," September 2015. [Online]. Available: https://krebsonsecurity.com/2014/04/3-million-customer-credit-debit-cards-stolen-in-michaels-aaron-brothers-breaches/

[7] J. Davis, "32M patient records breached in first half of 2019, 88% caused by hacking," August 2019. [Online]. Available: https://healthitsecurity.com/news/32m-patient-records-breached-in-first-half-of-2019-88-caused-by-hacking

[8] K. Granville, "Facebook and Cambridge Analytica: What you need to know as fallout widens," March 2018. [Online]. Available: https://www.nytimes.com/2018/03/19/technology/facebook-cambridge-analytica-explained.html

[9] J. Mayer, "How russia helped swing the election for Trump," September 2018. [Online]. Available: https://www.newyorker.com/magazine/2018/10/01/how-russia-helped-to-swing-the-election-for-trump

[10] J. Lewis, *Economic Impact of Cybercrime, No Slowing Down.* McAfee, 2018.

[11] Y. Sverdlik, "One minute of data center downtime costs US$7,900 on average," December 2013. [Online]. Available: https://www.datacenterdynamics.com/news/one-minute-of-data-center-downtime-costs-us7900-on-average/

[12] D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: from dependability to security," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 48–65, Jan 2004.

[13] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491199 pp. 55–60.

[14] "Critical infrastructure security and resilience," Presidential Policy Directive, PPD-21, February 2012. [Online]. Available: https://obamawhitehouse.archives.gov/the-press-office/2013/02/12/presidential-policy-directive-critical-infrastructure-security-and-resil

[15] Y. I. Khan, E. Al-shaer, and U. Rauf, "Cyber resilience-by-construction: Modeling, measuring & verifying," in *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, ser. SafeConfig '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2809826.2809836 pp. 9–14.

[16] W. Harrop and A. Matteson, "Cyber resilience: A review of critical national infrastructure and cyber-security protection measures applied in the uk and usa," in *Proceedings of Current and Emerging Trends in Cyber Operations: Policy, Strategy and Practice*, F. Lemieux, Ed. London: Palgrave Macmillan UK, 2015, pp. 149–166.

[17] B. Schneier, "The process of security," April 2000. [Online]. Available: https://www.schneier.com/essays/archives/2000/04/the_process_of_secur.html

[18] B. Schneier, "Security orchestration for an uncertain world," March 2017. [Online]. Available: https://securityintelligence.com/security-orchestration-for-an-uncertain-world/

[19] B. Obama, "Improving critical infrastructure cybersecurity," Executive Order, February 2013. [Online]. Available: https://obamawhitehouse.archives.gov/the-press-office/2013/02/12/executive-order-improving-critical-infrastructure-cybersecurity

[20] C. Herley and P. C. van Oorschot, "SoK: Science, security and the elusive goal of security as a scientific pursuit," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 99–120.

[21] M. H. Manshaei, Q. Zhu, T. Alpcan, T. Başar, and J. Hubaux, "Game theory meets network security and privacy," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 25:1–25:39, July 2013. [Online]. Available: http://doi.acm.org/10.1145/2480741.2480742

[22] T. Alpcan and T. Başar, *Network Security: A Decision and Game-Theoretic Approach*. New York, NY, USA: Cambridge University Press, 2010.

[23] K. C. Nguyen, T. Alpcan, and T. Başar, "Fictitious play with time-invariant frequency update for network security," in *Proceedings of the IEEE International Conference on Control Applications*, Sept. 2010, pp. 65–70.

[24] T. Alpcan and T. Başar, "A game theoretic approach to decision and analysis in network intrusion detection," in *Proceedings of the 42nd IEEE Conference on Decision and Control*, vol. 3, Dec 2003, pp. 2595–2600.

[25] S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley, "RRE: A game-theoretic intrusion response and recovery engine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 2, pp. 395–406, Feb. 2014.

[26] Q. Zhu and T. Başar, "Dynamic policy-based IDS configuration," in *Proceedings of the 48th IEEE Conference on Decision and Control*, Dec. 2009, pp. 8600–8605.

[27] E. LeMay, M. D. Ford, K. Keefe, W. H. Sanders, and C. Muehrcke, "Model-based security metrics using adversary view security evaluation (advise)," in *2011 Eighth International Conference on Quantitative Evaluation of SysTems*, Sep. 2011, pp. 191–200.

[28] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The mobius modeling tool," in *Proceedings 9th International Workshop on Petri Nets and Performance Models*, Sep. 2001, pp. 241–250.

[29] X. Yang, D. Wetherall, and T. Anderson, "TVA: A DoS-limiting network architecture," in *IEEE/ACM Trans. Netw.*, vol. 16, no. 6, Dec. 2008, pp. 1267–1280.

[30] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu, "Portcullis: Protecting connection setup from denial-of-capability attacks," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 289–300.

[31] X. Liu, X. Yang, and Y. Xia, "Netfence: Preventing internet denial of service from inside out," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 255–266.

[32] X. Liu, X. Yang, and Y. Lu, "To filter or to authorize: Network-layer DoS defense against multimillion-node botnets," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 195–206.

[33] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/fayaz pp. 817–832.

[34] D. Dean and A. Stubblefield, "Using client puzzles to protect TLS," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. USENIX Association, 2001.

[35] W. Feng, E. Kaiser, and A. Luu, "Design and implementation of network puzzles," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 4, March 2005, pp. 2372–2382.

[36] W. H. Sanders, "Quantitative security metrics: Unattainable holy grail or a vital breakthrough within our reach?" *IEEE Security Privacy*, vol. 12, no. 2, pp. 67–69, Mar 2014.

[37] M. M. Swanson, N. Bartol, J. Sabato, J. Hash, and L. Graffo, "Security metrics guide for information technology systems," National Institute of Standards and Technology (NIST), Tech. Rep., 2003.

[38] V. Verendel, "Quantified security is a weak hypothesis: A critical survey of results and assumptions," in *Proceedings of the 2009 Workshop on New Security Paradigms Workshop*, ser. NSPW '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1719030.1719036 pp. 37–50.

[39] S. Stolfo, S. M. Bellovin, and D. Evans, "Measuring security," *IEEE Security Privacy*, vol. 9, no. 3, pp. 60–65, May 2011.

[40] J. M. Spring, T. Moore, and D. Pym, "Practicing a science of security: A philosophy of science perspective," in *Proceedings of the 2017 New Security Paradigms Workshop*, ser. NSPW 2017. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3171533.3171540 pp. 1–18.

[41] M. A. Noureddine, A. M. Fawaz, A. Hsu, C. Guldner, S. Vijay, T. Başar, and W. H. Sanders, "Revisiting client puzzles for state exhaustion attacks resilience," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 617–629.

[42] M. A. Noureddine, A. Fawaz, W. H. Sanders, and T. Başar, "A game-theoretic approach to respond to attacker lateral movement," in *Proceedings of the 7th Conference on Decision and Game Theory for Security (GameSec)*, November 2-4 2016.

[43] M. A. Noureddine, A. Hsu, M. Caesar, F. A. Zaraket, and W. H. Sanders, "P4aig: Circuit-level verification of p4 programs," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, June 2019, pp. 21–22.

[44] M. Simaan and J. B. Cruz, "On the stackelberg strategy in nonzero-sum games," *Journal of Optimization Theory and Applications*, vol. 11, no. 5, pp. 533–555, May 1973. [Online]. Available: https://doi.org/10.1007/BF00935665

[45] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. T. Wroclawski, and S. Schwab, "The DETER project: Advancing the science of cyber security experimentation and test," in *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, Nov 2010, pp. 1–7.

[46] M. A. Noureddine and F. A. Zaraket, "Model checking software with first order logic specifications using AIG solvers," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 741–763, Aug 2016.

[47] D. Fisher, R. Linger, H. Lipson, T. Longstaff, N. Mead, and R. Ellison, "Survivable network systems: An emerging discipline," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-97-TR-013, 1997. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12905

[48] R. J. Ellison, R. C. Linger, T. Longstaff, and N. R. Mead, "Survivable network system analysis: a case study," *IEEE Software*, vol. 16, no. 4, pp. 70–77, 1999.

[49] "Compute Engine Service Level Agreement (SLA)," Available at https://cloud.google.com/compute/sla.

[50] "Amazon Compute Service Level Agreement," Available at https://aws.amazon.com/compute/sla/.

[51] "Microst Azure SLA for Cloud Services," Available at https://azure.microsoft.com/en-us/support/legal/sla/cloud-services/v1_5/.

[52] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: Association for Computing Machinery, 1988. [Online]. Available: https://doi.org/10.1145/52324.52356 p. 314–329.

[53] B. Krebs, "Antivirus is dead: Long live antivirus!" May 2014. [Online]. Available: https://krebsonsecurity.com/2014/05/antivirus-is-dead-long-live-antivirus/

[54] R. Weisman, "Is antivirus dead?" May 2020. [Online]. Available: https://blog.storagecraft.com/antivirus-dead/

[55] O. Sukwong, H. Kim, and J. Hoe, "Commercial antivirus software effectiveness: An empirical study," *Computer*, vol. 44, no. 3, pp. 63–70, 2011.

[56] D. Yadron, "Symantec develops new attack on cyberhacking," May 2014. [Online]. Available: https://www.wsj.com/articles/symantec-develops-new-attack-on-cyberhacking-1399249948?tesla=y

[57] R. M. Smullyan et al., *Theory of formal systems*. Princeton University Press, 1961.

[58] R. Alur and D. Dill, "The theory of timed automata," in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 45–73.

[59] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC 793, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[60] P. Alcoy, S. Bjarnson, P. Bowen, C. Chui, K. Kasavchenko, and G. Sockrider, "Insight Into The Global Threat Landsace: NetScout Arbor's 13th Annual Worldwide Infrastructure Security Report," 2017. [Online]. Available: https://pages.arbornetworks.com/rs/082-KNA-087/images/13th_Worldwide_Infrastructure_Security_Report.pdf

[61] S. Hilton, "Dyn analysis summary of Friday October 21 attack," October 2016, Oracle Dyn. [Online]. Available: http://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/

[62] "Global DDoS threat landscape: Q4 2017," Imperva Incapsula. [Online]. Available: https://www.incapsula.com/ddos-report/ddos-report-q4-2017.html

[63] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai botnet," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis pp. 1093–1110.

[64] B. Krebs, "Source Code for IoT Botnet 'Mirai' Released," 2016. [Online]. Available: https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/

[65] T. Bienkowski, "No sooner did the ink dry: 1.7 tbps DDoS attack makes history," March 2018. [Online]. Available: https://www.netscout.com/blog/security-17tbps-ddos-attack-makes-history

[66] T. Vissers, T. Van Goethem, W. Joosen, and N. Nikiforakis, "Maneuvering around clouds: Bypassing cloud-based security providers," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1530–1541.

[67] X. Yang, D. Wetherall, and T. Anderson, "A DoS-limiting network architecture," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05. New York, NY, USA: ACM, 2005, pp. 241–252.

[68] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, pp. 62–73, July 2002.

[69] K. Whalen, "The economics of DDoS attacks," 2017, Arbor Networks. [Online]. Available: https://www.arbornetworks.com/blog/insight/economics-ddos-attacks/

[70] W.-C. Feng, "The case for TCP/IP puzzles," in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, ser. FDNA '03. New York, NY, USA: ACM, 2003, pp. 322–327.

[71] A. Juels and J. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *Proceedings of the 1999 Networks and Distributed System Security symposium (NDSS)*, March 1999.

[72] X. Wang and M. K. Reiter, "Defending against denial-of-service attacks with puzzle auctions," in *Proceedings of the 2003 Symposium on Security and Privacy*, May 2003, pp. 78–92.

[73] E. Nygren, S. Erb, A. Biryukov, D. Khovratovich, and A. Juels, "TLS client puzzles extension," Working Draft, IETF Secretariat, Internet-Draft, December 2016.

[74] Y. Nir and V. Smyslov, "Protecting Internet Key Exchange Protocol Version 2 (IKEv2) Implementations from Distributed Denial-of-Service Attacks," RFC 8019, November 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc8019.txt

[75] X. Wang and M. K. Reiter, "Mitigating bandwidth-exhaustion attacks using congestion puzzles," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 257–267.

[76] T. Başar and R. Srikant, "Revenue-maximizing pricing and capacity expansion in a many-users regime," in *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, 2002, pp. 294–301.

[77] H. Shen and T. Başar, "Incentive-based pricing for network games with complete and incomplete information," in *Advances in Dynamic Game Theory: Numerical Methods, Algorithms, and Applications to Ecology and Economics*, S. Jørgensen, M. Quincampoix, and T. L. Vincent, Eds. Boston, MA: Birkhäuser Boston, 2007, pp. 431–458.

[78] H. Shen and T. Basar, "Optimal nonlinear pricing for a monopolistic network service provider with complete and incomplete information," in *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 6, August 2007, pp. 1216–1223.

[79] D. Bernstein, "SYN cookies," 1997. [Online]. Available: https://cr.yp.to/syncookies.html

[80] W. Eddy, "TCP SYN flooding attacks and common mitigations," Internet Requests for Comments, RFC 4987, August 2007. [Online]. Available: https://tools.ietf.org/pdf/rfc4987.pdf

[81] J. Lemon, "Resisting SYN flood DoS attacks with a SYN cache," in *Proceedings of the 2002 BSD Conference (BSDC'02)*. Berkeley, CA, USA: USENIX Association, 2002, pp. 10–10.

[82] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '92. London, UK, UK: Springer-Verlag, 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=646757.705669 pp. 139–147.

[83] M. K. Franklin and D. Malkhi, "Auditable metering with lightweight security," in *Proceedings of the International Conference on Financial Cryptography*. Springer, 1997, pp. 151–160.

[84] T. J. McNevin, J.-M. Park, and R. Marchany, "pTCP: A client puzzle protocol for defending against resource exhaustion denial of service attacks." Department of Electrical and Computer Engineering, Virginia Tech, Tech. Rep. TR-ECE-04-10, October 2004.

[85] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[86] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research perspectives and challenges for bitcoin and cryptocurrencies," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 104–121.

[87] B. Laurie and R. Clayton, "Proof-of-work proves not to work; version 0.2," in *WEIS*, 2004.

[88] L. Chen, P. Morrissey, N. P. Smart, and B. Warinschi, "Security notions and generic constructions for client puzzles," in *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 505–523.

[89] B. Groza and B. Warinschi, "Cryptographic puzzles and DoS resilience, revisited," *Des. Codes Cryptography*, vol. 73, no. 1, pp. 177–207, Oct. 2014.

[90] D. Stebila, L. Kuppusamy, J. Rangasamy, C. Boyd, and J. G. Nieto, "Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols," in *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011*, ser. CT-RSA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 284–301.

[91] C. Sheng, "A general utility function for decision-making," in *Mathematical Modelling*, vol. 5, no. 4, 1984, pp. 265 – 274.

[92] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message syntax and routing," Internet Requests for Comments, RFC 7230, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230

[93] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.

[94] "ab - Apache HTTP server benchmarking tool," Apache. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[95] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab, "Design, deployment, and use of the DETER testbed," in *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test (DETER 2007)*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–1.

[96] I. Arghire, "You can DDoS an organization for just $10 per hour: Cybercrime Report," https://www.securityweek.com/you-can-ddos-organization-just-10-hour-cybercrime-report, March 2018, accessed: 08-01-2018.

[97] Cloudflare, "Memcached DDoS Attack," https://www.cloudflare.com/learning/ddos/memcached-ddos-attack/, 2018, accessed: 08-08-2018.

[98] J. Arteaga, D. Lewis, W. Mejia, E. Shuster, D. McEwan, and A. Zeigenhirt, "State of the Internet (SOTI) / Security: Web Attack," Akamai, 2018.

[99] X. Liu, X. Yang, and Y. Xia, "Netfence: preventing internet denial of service from inside out," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. –, Aug. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=2043164.1851214

[100] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing internet denial-of-service with capabilities," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 39–44, Jan. 2004. [Online]. Available: http://doi.acm.org/10.1145/972374.972382

[101] M. S. Kang, S. B. Lee, and V. D. Gligor, "The Crossfire attack," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 127–141.

[102] A. Studer and A. Perrig, "The Coremelt attack," in *Computer Security – ESORICS 2009*, M. Backes and P. Ning, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 37–52.

[103] X. Wang and M. K. Reiter, "Mitigating bandwidth-exhaustion attacks using congestion puzzles," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030118 pp. 257–267.

[104] Z. Liu, H. Jin, Y.-C. Hu, and M. Bailey, "Middlepolice: Toward enforcing destination-defined policies in the middle of the internet," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978306 pp. 1268–1279.

[105] T. Miu, A. Hui, W. Lee, D. Luo, A. Chung, and J. Wong, "Universal ddos mitigation bypass," *Black Hat USA*, 2013.

[106] T. T. Miu, W. Lee, A. K. Chung, D. X. Luo, A. K. Hui, and J. W. Wong, "Kill'em all–ddos protection total annihilation!" *Defcon 21*, 2013.

[107] T. Vissers, T. Van Goethem, W. Joosen, and N. Nikiforakis, "Maneuvering around clouds: Bypassing cloud-based security providers," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813633 pp. 1530–1541.

[108] Amazon, "Amazon AWS Shield: Managed DDoS Protection," https://aws.amazon.com/shield/, accessed: 08-08-2018.

[109] P. Mittal, D. Kim, Y.-C. Hu, and M. Caesar, "Mirage: Towards deployable ddos defense for web applications," *arXiv preprint arXiv:1110.1060*, 2011.

[110] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of Explicit Congestion Notification (ECN) to IP," Internet Requests for Comments, Internet Engineering Task Force, RFC 3168, September 2001. [Online]. Available: https://tools.ietf.org/html/rfc3168

[111] D. Black, "Relaxing restrictions on Explicit Congestion Notification (ECN) experimentation," Internet Requests for Comments, Internet Engineering Task Force, RFC 8311, January 2018. [Online]. Available: https://tools.ietf.org/html/rfc8311

[112] A. Back et al., "Hashcash-a denial of service counter-measure," 2002.

[113] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988. [Online]. Available: http://doi.acm.org/10.1145/52324.52356 pp. 314–329.

[114] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," in *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*, ser. SIGCOMM '87. New York, NY, USA: ACM, 1988. [Online]. Available: http://doi.acm.org/10.1145/55482.55484 pp. 2–7.

[115] M. Szymaniak, D. Presotto, G. Pierre, and M. van Steen, "Practical large-scale latency estimation," *Computer Networks*, vol. 52, no. 7, pp. 1343 – 1364, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128608000212

[116] P. Sharma, Z. Xu, S. Banerjee, and S.-J. Lee, "Estimating network proximity and latency," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 3, pp. 39–50, July 2006. [Online]. Available: http://doi.acm.org/10.1145/1140086.1140092

[117] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443 pp. 54–66.

[118] D. Zhang, C. Zheng, H. Zhang, and H. Yu, "Identification and analysis of skype peer-to-peer traffic," in *2010 Fifth International Conference on Internet and Web Applications and Services*, 2010, pp. 200–206.

[119] N. Spring, L. Peterson, A. Bavier, and V. Pai, "Using planetlab for network research: Myths, realities, and best practices," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, p. 17–24, Jan. 2006. [Online]. Available: https://doi.org/10.1145/1113361.1113368

[120] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten, "New client puzzle outsourcing techniques for DoS resistance," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 246–256.

[121] i. T. ultimate speed test tool for TCP, UDP and SCTP. [Online]. Available: https://iperf.fr

[122] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, jul 2014.

[123] M. V. Dumitru, D. Dumitrescu, and C. Raiciu, "Can we exploit buggy p4 programs?" in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3373360.3380836 p. 62–68.

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

[125] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2491185.2491199 p. 55–60.

[126] L. MacVittie, "Amazon outage casts a shadow on sdn," July 2012. [Online]. Available: https://devcentral.f5.com/s/articles/amazon-outage-casts-a-shadow-on-sdn

[127] M. Neves, B. Huffaker, K. Levchenko, and M. Barcellos, "Dynamic property enforcement in programmable data planes," in *2019 IFIP Networking Conference (IFIP Networking)*, 2019, pp. 1–9.

[128] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 5:1–5:7.

[129] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 490–503.

[130] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with Vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 518–532.

[131] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in P4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 4:1–4:7.

[132] N. McKeown, D. Talayco, G. Varghese, N. Lopes, N. Bjørner, and A. Rybalchenko, "Automatically verifying reachability and well-formedness in p4 networks," Stanford University, Tech. Rep. MSR-TR-2016-65, September 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/

[133] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 151–165.

[134] C. A. J. van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 814–819, July 2000.

[135] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: Association for Computing Machinery, 2006. [Online]. Available: https://doi.org/10.1145/1146909.1147048 p. 532–535.

[136] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Scalable min-register retiming under timing and initializability constraints," in *2008 45th ACM/IEEE Design Automation Conference*, June 2008, pp. 534–539.

[137] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 104–117.

[138] G. Cabodi, S. Quer, and F. Somenzi, "Optimizing sequential verification by retiming transformations," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: Association for Computing Machinery, 2000. [Online]. Available: https://doi.org/10.1145/337292.337591 p. 601–606.

[139] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2008, pp. 234–241.

[140] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '05. USA: IEEE Computer Society, 2005, p. 1076–1082.

[141] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2774993.2774999

[142] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/jin pp. 35–49.

[143] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3132747.3132764 p. 121–136.

[144] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *IEEE NDSS*, 2020.

[145] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with sdn data plane," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[146] G. Grigoryan and Y. Liu, "Lamp: Prompt layer 7 attack mitigation with programmable data planes," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018, pp. 1–4.

[147] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 19–27.

[148] The P4 language consortium, "$P4_{16}$ language specification," https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, May 2017. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[149] "NPL – network programming language specification v1.3," https://nplang.org/npl/specifications/, June 2019. [Online]. Available: https://nplang.org/npl/specifications/

[150] M. Eichholz, E. Campbell, N. Foster, G. Salvaneschi, and M. Mezini, "How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10804 pp. 12:1–12:28.

[151] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz, "Directions in active networks," *IEEE Communications Magazine*, vol. 36, no. 10, pp. 72–78, 1998.

[152] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," in *Proceedings DARPA Active Networks Conference and Exposition*, 2002, pp. 2–15.

[153] K. Psounis, "Active networks: Applications, security, safety, and architectures," *IEEE Communications Surveys*, vol. 2, no. 1, pp. 2–16, 1999.

[154] "Barefoot: The world's fastest & most programmable networks." [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/

[155] Browne, Clarke, Dill, and Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Transactions on Computers*, vol. C-35, no. 12, pp. 1035–1044, Dec 1986.

[156] Bochmann, "Hardware specification with temporal logic: An example," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 223–231, March 1982.

[157] Y. Malachi and S. S. Owicki, "Temporal specifications of self-timed systems," in *VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981. [Online]. Available: \url{https://doi.org/10.1007/978-3-642-68402-9_23} pp. 203–212.

[158] Y. Abarbanel-Vinov, N. Aizenbud-Reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal, and T. Yatzkar-Haham, "On the effective deployment of functional formal verification," *Formal Methods in System Design*, vol. 19, no. 1, pp. 35–44, 2001. [Online]. Available: https://doi.org/10.1023/A:1011219209077

[159] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl, "Formal verification made easy," *IBM Journal of Research and Development*, vol. 41, no. 4.5, pp. 567–576, July 1997.

[160] J. Markoff, "Flaw undermines accuracy of Pentium chips," The New York Times, Nov 1994. [Online]. Available: https://www.nytimes.com/1994/11/24/business/company-news-flaw-undermines-accuracy-of-pentium-chips.html

[161] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*, 2nd ed. MIT press, 2018.

[162] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *Proceedings of the 9th International Conference on Computer Aided Verification*, ser. CAV '97. Berlin, Heidelberg: Springer-Verlag, 1997, p. 72–83.

[163] T. Kropf, *Introduction to Formal Hardware Verification*. Springer Science & Business Media, 2013.

[164] S. Demri, F. Laroussinie, and P. Schnoebelen, "A parametric analysis of the state explosion problem in model checking," in *STACS 2002*, H. Alt and A. Ferreira, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 620–631.

[165] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[166] M. Budiu and C. Dodd, "The p416 programming language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, p. 5–14, Sep 2017. [Online]. Available: https://doi.org/10.1145/3139645.3139648

[167] R. Braden, D. Borman, C. Partridge, and W. W. Plummer, "Computing the internet checksum," Internet Requests for Comments, RFC Editor, RFC 1071, September 1988, http://www.rfc-editor.org/rfc/rfc1071.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1071.txt

[168] J. J. Dongarra and A. R. Hinds, "Unrolling loops in fortran," *Software: Practice and Experience*, vol. 9, no. 3, pp. 219–226, 1979. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090307

[169] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969. [Online]. Available: https://doi.org/10.1145/363235.363259

[170] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2011, pp. 125–134.

[171] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing – SAT 2012*, A. Cimatti and R. Sebastiani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 157–171.

[172] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, p. 290–301, Aug 2011. [Online]. Available: https://doi.org/10.1145/2043164.2018470

[173] N. M. P. Kazemian, G. Varghese, "Header space analysis: static checking for networks," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: https://doi.org/10.1145/3373360.3380843

[174] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, ser. NetAI'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3341216.3342206 p. 1–7.

[175] P. T. Suriya Kodeswaran, Mina Tahmasbi Arashloo and J. Rexford, "Tracking p4 program execution in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20. New York, NY, USA: ACM, 2020. [Online]. Available: https://doi.org/10.1145/3373360.3380843 pp. 117–122.

[176] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, p. 453–457, Aug 1975. [Online]. Available: https://doi.org/10.1145/360933.360975

[177] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[178] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2934872.2934881 p. 314–327.

[179] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of P4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 73–85.

[180] F. G. Alessandro Cimatti, Edmund Clarke and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, March 2000.

[181] M. A. Noureddine and F. A. Zaraket, "Model checking software with first order logic specifications using AIG solvers," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 741–763, Aug 2016.

[182] R. Di Pietro and L. V. Mancini, *Intrusion Detection Systems*. Springer, 2008.

[183] "Lateral movement: How do threat actors move deeper into your network," Trend Micro, Tech. Rep., 2003.

[184] N. Stakhanova, S. Basu, and J. Wong, "A taxonomy of intrusion response systems," *International Journal of Information and Computer Security*, vol. 1, no. 1-2, pp. 169–184, 2007. [Online]. Available: http://www.inderscienceonline.com/doi/abs/10.1504/IJICS.2007.012248

[185] A. Shameli-Sendi, N. Ezzati-Jivan, M. Jabbarifar, and M. Dagenais, "Intrusion response systems: Survey and taxonomy," *Int. J. Comput. Sci. Netw. Secur*, vol. 12, no. 1, pp. 1–14, 2012.

[186] R. Brewer, "Advanced persistent threats: Minimizing the damage," *Network Security*, vol. 2014, no. 4, pp. 5–9, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1353485814700406

[187] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, p. 80, 2011.

[188] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, May 2011.

[189] C. Bronk and E. Tikk-Rangas, "Hack or attack? Shamoon and the evolution of cyber conflict," Feb. 2013, available at ssrn: http://ssrn.com/abstract=2270860. [Online]. Available: http://ssrn.com/abstract=2270860

[190] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the Ukrainian power grid," White Paper, sANS Industrial Control Systems. 2016. [Online]. Available: http://www.nerc.com/pa/CI/ESISAC/Documents/E-ISAC_SANS_Ukraine_DUC_18Mar2016.pdf

[191] M. Roesch, "Snort: Lightweight intrusion detection for networks." in *Proceedings of USENIX LISA*, vol. 99, no. 1, 1999, pp. 229–238.

[192] "The Bro network security monitor," https://www.bro.org/. 2014.

[193] Trend Micro, "Understanding targeted attacks: Six components of targeted attacks," November 2015, [Online, Accessed: 06-05-2016]. http://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/targeted-attacks-six-components. [Online]. Available: \url{http://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/targeted-attacks-six-components}

[194] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," in *Proceedings of the 2003 ACM Workshop on Rapid Malcode*. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/948187.948190 pp. 11–18.

[195] G. Csardi and T. Nepusz, "The iGraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[196] B. M. Waxman, "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, pp. 1617–1622, Dec. 1988.

[197] R. Albert and A. Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, pp. 47–97, Jan 2002. [Online]. Available: http://link.aps.org/doi/10.1103/RevModPhys.74.47

[198] M. Nekovee, "Worm epidemics in wireless ad hoc networks," *New Journal of Physics*, vol. 9, no. 6, p. 189, 2007. [Online]. Available: http://stacks.iop.org/1367-2630/9/i=6/a=189

[199] M. Penrose, *Random geometric graphs*. Oxford University Press Oxford, 2003, vol. 5.

[200] R. D. McKelvey, A. M. McLennan, and T. L. Turocy, "Gambit: Software tools for game theory," Version 15.1.0, Tech. Rep., 2016. [Online]. Available: http://econweb.tamu.edu/gambit/

[201] E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 2016-06-16]. http://www.scipy.org/. [Online]. Available: http://www.scipy.org/

[202] M. Bloem, T. Alpcan, and T. Başar, "Intrusion response as a resource allocation problem," in *Proceedings of the 45th IEEE Conference on Decision and Control*, Dec. 2006, pp. 6283–6288.

[203] C. B. Simmons, S. G. Shiva, H. S. Bedi, and V. Shandilya, "ADAPT: A game inspired attack-defense and performance metric taxonomy," in *Proceedings of the 28th IFIP TC 11 International Conference*, ser. SEC 2013. Springer Berlin Heidelberg, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39218-4_26 pp. 344–365.

[204] T. Alpcan and T. Başar, "A game theoretic approach to decision and analysis in network intrusion detection," in *Proceedings of the 42nd IEEE Conference on Decision and Control*, vol. 3, Dec 2003, pp. 2595–2600.

[205] K. C. Nguyen, T. Alpcan, and T. Başar, "Fictitious play with time-invariant frequency update for network security," in *Proceedings of the IEEE International Conference on Control Applications*, Sept. 2010, pp. 65–70.

[206] S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley, "RRE: A game-theoretic intrusion response and recovery engine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 2, pp. 395–406, Feb. 2014.

[207] Q. Zhu and T. Başar, "Dynamic policy-based IDS configuration," in *Proceedings of the 48th IEEE Conference on Decision and Control*, Dec. 2009, pp. 8600–8605.

[208] D. McMorrow, "Science of cyber-security," The MITRE Corporation, Tech. Rep. JSR-10-102, November 2010.

[209] M. Bishop, "What is computer security?" *IEEE Security Privacy*, vol. 1, no. 1, pp. 67–69, 2003.

[210] S. Peisert and M. Bishop, "How to design computer security experiments," in *Proceedings of the Fifth World Conference on Information Security Education*, L. Futcher and R. Dodge, Eds. New York, NY: Springer US, 2007, pp. 141–148.

[211] J. Yan and A. S. El Ahmad, "Usability of captchas or usability issues in captcha design," in *Proceedings of the 4th Symposium on Usable Privacy and Security*, ser. SOUPS '08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1408664.1408671 p. 44–52.

[212] K. A. Kluever and R. Zanibbi, "Balancing usability and security in a video captcha," in *Proceedings of the 5th Symposium on Usable Privacy and Security*, ser. SOUPS '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1572532.1572551

[213] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, "Cross-app poisoning in software-defined networking," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243759 p. 648–663.

[214] B. Bailey, "Hardware and software co-verification employing deferred synchronization," mar 2002, uS Patent 6,356,862.

[215] L. Semeria and A. Ghosh, "Methodology for hardware/software co-verification in c/c++," in *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, 2000, pp. 405–408.