Finding Security Vulnerabilities in Network Protocols Using Methods of Formal Verification

Adi Sosnovich

Finding Security Vulnerabilities in Network Protocols Using Methods of Formal Verification

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Adi Sosnovich

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tevet 5777 Haifa January 2017

This research was carried out under the supervision of Prof. Orna Grumberg, in the Faculty of Computer Science.

Acknowledgements

First, I would like to express my sincere gratitude my supervisor, Prof. Orna Grumberg. Thank you for years of dedicated guidance and support throughout the production of this research and thesis. I am grateful for the opportunity to work with you under your supervision. My sincere thanks also goes to Dr. Gabi Nakibly. Thank you for years of collaboration, inspiration, and help, throughout my research. I would also like to thank to Prof. Michael Schapira. Thank you for you help, with very useful ideas and comments.

Last but not the least, I would like to thank my family: my parents and to my husband for supporting me spiritually throughout writing this thesis and my life in general.

The generous financial help of the Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research and Teaching Program and of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract **1**

Abbreviations and Notations **3**

1 Introduction **5**

2 Finding Vulnerabilities in OSPF Using Model Checking **11**

2.1 Perliminaries 11
2.2 Related Work 13
2.3 The OSPF Protocol 13
2.3.1 OSPF Basics 13
2.3.2 Threat Model 14
2.4 Modeling OSPF 15
2.4.1 The Concrete Model 15
2.4.2 Formal Model for OSPF 16
2.4.3 Specification 18
2.4.4 Experimental Data 18
2.4.5 Example of Attacks on OSPF 19
2.5 An Abstract Network and Its Matching Concrete Networks 20
2.5.1 Abstract Topology 22
2.5.2 Matching Abstract and Concrete Topologies 23
2.5.3 Global Abstract States 26
2.5.4 Matching Abstract and Concrete states 26
2.5.5 Abstract Transitions and Their Matching Concrete Transitions . 27
2.5.6 Examples of OSPF Attacks in the Abstract Model 29
2.6 Correctness of the Abstract Model 32
2.6.1 Flooding and Fight Back Destinations 33
2.6.2 Correctness Proof 37
2.7 Extension of the Concrete Model 42
2.7.1 Extended OSPF Basics 42
2.7.2 Extension Goals 44

2.7.3 Extended OSPF Modeling 46
2.7.4 Results 50
2.8 Conclusion 51

3 Analyzing BGP Traffic Attraction Attacks Using Model Checking 55
3.1 Preliminaries 55
3.2 Related Work 57
3.3 BGP Background 58
3.4 BGP Modeling 61
3.4.1 Model simplifications 61
3.4.2 Threat Model 62
3.4.3 The BGP Model 62
3.4.4 Attack Definitions and Specifications 66
3.5 Attacker Model Simplifications 67
3.5.1 Abstraction of Paths Originated By the Attacker 67
3.5.2 Reducing the Number of Messages Originated By the Attacker . 72
3.6 Reductions and Abstractions 75
3.6.1 Self-contained Fragments 75
3.6.2 Definite Routing Choice 80
3.6.3 Routing-preserving Path 85
3.6.4 Example of a Network Reduction 87
3.7 The BGP-SA Method 89
3.7.1 Reducing the Network Topology 90
3.7.2 Simulating the Trivial Attack 90
3.7.3 Generating the C Model 90
3.7.4 Applying Model Checking to the Implemented Model Using ExpliSAT 92
3.8 Experimental Results 93
3.8.1 Results on Internet Fragments 93
3.8.2 Example Demonstrating Model Checking Advantages 95
3.9 Conclusion 96
3.9.1 Possible Directions for Extensions 96

4 Formal Analysis of a Black Box OSPF Implementation 99
4.1 Preliminaries 99
4.1.1 Background in Symbolic Execution 101
4.1.2 OSPF Background 101
4.2 Black Box Analysis Procedures 102
4.2.1 The Method Flow 102
4.2.2 The OSPF Symbolic Model 103
4.2.3 Black Box Testing of the Generated Tests 106

4.2.4 Extending the Method to Multiple LSAs . . . 107
4.3 Evaluation . . . 109
4.3.1 Testbed . . . 110
4.3.2 Results With a Single LSA . . . 111
4.3.3 Results With Multiple LSAs . . . 117
4.4 Advantages and Limitations of Our Method . . . 118
4.5 Related Work . . . 119
4.5.1 Formal Black Box Analysis . . . 119
4.5.2 Symbolic Execution . . . 120
4.5.3 OSPF Analysis . . . 120
4.6 Conclusions . . . 121

5 Conclusions **123**

Hebrew Abstract **i**

List of Figures

2.1 Example of a concrete topology 16
2.2 Example of an abstract topology 16
2.3 A sketch of the router r procedure. 17
2.4 The experimented OSPF topology 19
2.5 Procedure of a singleton router 29
2.6 Flooding procedure of a singleton router 30
2.7 Abstract topology of attack #2 31
2.8 Abstract topology of attack #3 31
2.9 The structure of an LSA header 43
2.10 The topology used with the extended model 50

3.1 BGP network example 61
3.2 Algorithm for Computing a Self-Contained Fragment S of N. 80
3.3 Fragment example 81
3.4 Valid paths trees example 82
3.5 Algorithm for computing the tree T_o of all valid paths in N from the originating node o 83
3.6 Algorithm for computing $PRO(n)$ for each node n on the tree T_o 83
3.7 Algorithm for computing $drc(n)$ for $n \in Nodes$ 84
3.8 Reductions example 88
3.9 The BGP-SA Method 89
3.10 Partition of Node Types in the Extracted Fragment 89

4.1 The flow of our method 103
4.2 Systematic extension algorithm 109
4.3 Topology 1 111
4.4 Topology 2 112

Abstract

The routers and networks of the Internet are clustered into connected sets. Each such set is called an autonomous system (AS). Routing of data packets on the Internet works in two levels. The Border Gateway Protocol (BGP) is the routing protocol that determines through which ASes the packets will traverse. The Open Shortest Path First (OSPF) protocol is a widely used routing protocol that determines the path taken by the packets within each AS. Finding security vulnerabilities and attack strategies in these routing protocols is an important task and is significant for the Internet security.

Formal verification methods were originally developed to prove correctness of systems based on formal specifications. A very common approach in formal verification is model checking. Model checking is an efficient algorithm, that given a system model and a required specification, determines whether the system satisfies the specification requirements or not. When the answer is no, there is also a counterexample in the form of an undesired behavior of the system. In this thesis, we develop several methods to apply a systematic and automatic security analysis of these Internet routing protocols. We use methods and tools from the field of formal verification.

We first develop a security analysis method for finding built-in vulnerabilities in the OSPF protocol using model checking. We model parts of the OSPF standard and use a model checker tool to automatically find vulnerabilities which are inherent to the design of the protocol on simple network topologies. We then extend our analysis to more general network topologies. We develop a novel technique for parameterized networks that allows finding general attacks which are applicable to families of networks.

Next, we develop a security analysis method for the BGP protocol. We focus on traffic attraction attacks, where an attacker sends false routing advertisements to gain attraction of extra traffic in order to increase its revenue from customers, drop, tamper, or snoop on the packets. We use model checking to perform exhaustive search for attraction attacks on BGP. To deal with scalability issues of the entire Internet topology, we propose static methods to identify and automatically reduce Internet fragments of interest. Using a model checking tool we identify attacks as well as show that certain attraction scenarios are impossible on the Internet under the modeled attacker capabilities.

Finally, we propose a formal black box method to reveal non-standard protocol deviations in closed-source network devices. The method relies only on the ability to

test the targeted protocol implementation and observe its output. We use a model-based testing approach, which relies on a formal model of the protocol. We cope with scalability issues using optimizations that are tailored to analysis of network protocols. They allow reducing the number of generated tests without loss of functionality cover of the model. We evaluate our method against the OSPF protocol. We search for deviations in the OSPF implementation of Cisco – the largest networking vendor in the world.

Abbreviations and Notations

LSA	:	Link State Advertisement
$OSPF$	:	Open Shortest Path First
BGP	:	Border Gateway Protocol
AS	:	Autonomous System
ASN	:	Autonomous System Number
$LSDB$	:	LSA database
R	:	a router R
RID	:	Router ID
$R.LSDB$	:	The LSA database of a router R
$R.Q$	:	The incoming message queue of a router R

Chapter 1

Introduction

In this thesis we develop and implement methods to automatically find vulnerabilities and attack strategies in common Internet routing protocols. We focus on two of the most widely used protocols for routing on the Internet, the OSPF and BGP protocols. We use methods and tools of formal verification to allow a systematic and automatic search for attacks.

The routers and networks of the Internet are clustered into connected sets. Each such set is called an autonomous system (AS). Routing of data packets on the Internet works in two levels:

1. Inter-domain routing that determines through which ASes the packets will traverse. This level of routing is handled by a single routing protocol called the Border Gateway Protocol [54] (BGP).

2. Intra-domain routing that determines the path taken by the packets within each AS. This is determined independently in each AS. The most common examples of such routing protocols are OSPF [50], RIP [46], or IS-IS [20].

Finding security vulnerabilities and attack strategies in routing protocols is a complex task and is important for the Internet security.

Open Shortest Path First (OSPF) is one of the most widely deployed interior gateway routing protocols on the Internet. The most common attack vector against OSPF is spoofing of routing advertisements on behalf of a remote router. OSPF employs a self-defense "fight-back" mechanism that quickly reverts the effects of such attacks. Nonetheless, some attacks that evade the fight-back mechanism have been discovered, making it possible to persistently falsify routing advertisements [37, 51]. This type of attacks are the most serious threat to a routing protocol since they allow an attacker to gain persistent control over how traffic is routed throughout the network. This shows that despite its maturity, the OSPF specification is not without security flaws and may have still-unknown vulnerabilities. Manually identifying vulnerabilities in a complex protocol such as OSPF is a hard task which requires deep understanding and close acquaintance with the protocol.

BGP is the sole protocol used for inter-domain routing. In essence, BGP is the glue that holds the Internet together and which allows to connect between different ASes. The primary function of BGP is to exchange network reachability information between different ASes. A BGP manipulator may exploit vulnerabilities in the protocol to attract traffic towards its own AS. Attracting extra traffic enables the AS to increase revenue from customers, drop, tamper, or snoop on the packets. In recent years, there have been frequent occurrences of traffic attraction attacks on the Internet [67, 65, 63, 64, 43, 44]. Some of those attacks allowed oppressive governments to block their citizens from accessing certain websites. In other attacks the perpetrators eavesdropped or altered the communications of others, while in different attacks spammers sent millions of emails from IP addresses they do not own.

Formal verification methods were originally developed to prove correctness of systems based on formal specifications. A very common approach in formal verification is model checking. *Model checking* [24] is an efficient algorithm, that given a system model and a required specification, determines whether the system satisfies the specification requirements or not. When the answer is no, there is also a counterexample in the form of an undesired behavior of the system.

In this work we develop and implement methods to automatically find vulnerabilities and attack strategies in common Internet routing protocols. We focus on the OSPF and BGP protocols. We use methods and tools of formal verification to allow a systematic and automatic search for attacks. A thorough analysis which is implemented by formal verification tools has a major advantage over ad-hoc approaches, manual analysis, or simple testing techniques that randomly search for attacks. In the formal verification approach we model certain parts of the protocol. Then a systematic search over the execution paths of the model is applied. A common problem of this approach is that large-sized models cause scalability issues. We develop unique abstraction techniques to allow using compact abstract models for complex routing protocols over large network topologies.

In chapters 2 and 3 we use the model checking apporach to find vulnerabilities and attack strategies in OSPF and BGP. A network topology consists of regular nodes and an attacker. Each regular node runs the modeled protocol and an attacker node has predefined capabilities. Based on its modeled capabilities the attacker can apply manipulation techniques. We define specifications to identify certain types of attacks. A model checker tool systematically scans the execution paths of the model. If it finds a run that violates the specification, it returns a counterexample that represents a successful attack.

In chapter 4 we use another approach to find vulnerabilities in certain OSPF implementations. We use a *model-based testing* approach [11, 66], which relies on a formal model of the protocol in question. The model can be an abstraction of the protocol and refer to specific parts of it. Test cases are generated from the model itself, and are then executed on the system under test (SUT). We use symbolic execution to

systematically generate tests from our model. *Symbolic execution* [18] allows analyzing the execution paths of a program and generating corresponding test cases. The input variables of the program are defined as symbolic variables. Then, the program is symbolically run, where symbolic expressions represent values of the program variables. On each execution path a path-constraint is obtained in the form of a first order formula. Its solutions form a set of concrete values from which a test case is derived.

In each of the chapters 2, 3, and 4, we define a different type of security analysis. We start with an analysis of a protocol standard to reveal built-in vulnerabilities that are inherent to the protocol itself. In order to resolve such vulnerabilities it may be required to change the standard of the protocol. Next, we propose an analysis that allows revealing non-trivial attack strategies for a certain known vulnerability. Such an analysis can provide a better understanding of the implications of a known vulnerability and can reveal unknown strategies that exploit it. Finally, we develop a formal black box analysis to find deviations of a closed-source implementation of a protocol from its standard. Such deviations may reveal security vulnerabilities of the implementation itself. In the following, we describe the different approaches for security analysis of routing protocols that we focus on in our research.

Finding Built-in Vulnerabilities in a Network Protocol Standard

A built-in security vulnerability is inherent to the protocol itself , and attacks taking advantage of such vulnerabilities rely on legitimate functionality of the protocol. In order to handle such security vulnerabilities and prevent attacks, a modification of the protocol itself is required. The process of identifying built-in security vulnerabilities in network protocols is done mostly manually, in an ad-hoc manner. In addition, it requires deep familiarity with the protocol, and is therefore usually done by experts.

In Chapter 2 we model the OSPF routing protocol based on its standard and use a model checker tool to automatically find attacks and reveal built-in vulnerabilities. We develop a novel technique for parameterized networks, representing a family of networks, which is suitable for finding a counterexample (in our case an attack) on each member of the family. We define an abstract network topology that represents a family of concrete networks with varied sizes and topologies. We prove that an abstract attack found on an abstract network topology has a corresponding concrete attack on each member of the family. It allows finding general attacks which are applicable to families of networks. This work was published in [61] and in [52].

Finding Non-trivial Attack Strategies for a Certain Vulnerability

The BGP protocol is known to be vulnerable to traffic attraction attacks [35, 15]. In such attacks malicious Autonomous Systems manipulate BGP routing advertisements in order to attract traffic to, or through, their AS networks. Attracting extra traffic enables the AS to increase revenue from customers, drop, tamper, or snoop on the

packets. Although the vulnerability is known, there exist a variety of attack strategies which a manipulator can use to gain traffic attraction from certain ASes. Such strategies may depend on the network topology, the business relationships between Autonomous systems, and the location of the manipulator within the network.

In Chapter 3 we focus on this known vulnerability of BGP, and we develop a method to automatically search for non-trivial attack strategies that exploit this vulnerability. Our goal is to provide insights to where and how BGP traffic attraction attacks are possible. We use model checking to systematically reveal BGP traffic attraction attacks on the Internet, or prove their absence under certain conditions. We develop powerful reductions and abstractions that allow model checking to explore relatively small fragments of the Internet, yet obtain relevant results. Reductions are essential as the Internet nowadays includes roughly $50,000$ ASes. Our analysis method allows identifying safe nodes that are not amenable to traffic attraction attacks and can be exploited to reduce vulnerability of other nodes in the Internet. This work was published in [62].

Finding Vulnerabilities of a Network Protocol Black-box Implementation

The Internet infrastructure relies almost entirely on network protocols that are based on open standards. However, the overwhelming majority of network devices on the Internet, e.g. routers and switches, are proprietary and closed source. Hence, there is no straightforward way to analyze them. Specifically, one cannot easily and systematically identify deviations of a protocol implementation of a network device from the protocol's standard. Such deviations (either deliberate or inadvertent) are particularly important to identify since they present a non-standard functionality which have not been openly and rigorously analyzed by the networking and security community. Therefore, these deviations may degrade the security or resiliency of the network.

In Chapter 4 we propose a formal black box method to unearth non-standard protocol deviations in closed-source network devices. The method relies only on the ability to test the targeted protocol implementation and observe its output. We use a model-based testing approach [11, 66], which relies on a formal model of the protocol in question. We use *concolic execution* [34, 57] to systematically generate tests from our model. Concolic testing is a dynamic symbolic execution technique to systematically generate tests along different execution paths of a program. It involves concrete runs of the program over concrete input values alongside symbolic execution. Each concrete execution is on a different path. The paths are explored systematically and automatically until full coverage is achieved.

In general, such an approach has significant scalability issues in terms of number of tests needed to cover a desired functionality. We cope with these issues using efficient and practical optimizations that are tailored to the analysis of network protocols. They allow to significantly reduce the number of generated tests without loss of functionality

cover of the model.

The method we propose allowed us to implement the first practical tool to identify deviations of black box implementations of one of the most complex multi-party protocols on the Internet – the OSPF routing protocol. We search for deviations in the OSPF implementation of Cisco – the largest networking vendor in the world. Our evaluation identified numerous significant deviations. Some of them can be abused to compromise the security of a network. The deviations were confirmed by Cisco.

Chapter 2

Finding Vulnerabilities in OSPF Using Model Checking

2.1 Perliminaries

In this chapter we present a novel approach to automatically finding security vulnerabilities in the routing protocol *Open Shortest Path First* (OSPF) [50]. OSPF is the most widely used protocol for Internet routing, thus finding vulnerabilities which are inherent to the design of the protocol is significant for Internet security. Manually identifying vulnerabilities in a complex protocol such as OSPF is a hard task which requires deep understanding and close acquaintance with the protocol.

We propose to find vulnerabilities *automatically* by using model checking techniques. In order to use model checking for our purpose we build a model for the protocol when running on a given network topology; we include in the model an attacker with predefined capabilities; and we specify the absence of a state in which an attack succeeds (to be defined later). If the model checker finds a state violating the specification, it returns a counterexample leading to that state. The counterexample being a run of the protocol is, in fact, an *attack* on the protocol.

A high level description of the OSPF protocol is given below. OSPF runs on each router in a network of routers. Its goal is to distribute the full network topology to all routers. The routers send each other messages describing their partial view of the network topology. When a router gets a message from its neighbor, it updates its database accordingly and *floods* the message on to all of its *other* neighbors. OSPF includes a mechanism for fighting against possible attacks. If a router gets a message in its own name that it did not originate, then the router initiates a *"fight back"* message in order to correct the topology view of all other routers.

We start by modeling (concrete) networks with a fixed number of routers in a specific topology, where each router runs the OSPF protocol. The *attacker* is one of the routers running the same protocol, except that it can also send *fake* messages in the name of other routers, and can ignore messages sent to it. A *state* of the model consists of the

databases and message queues of all routers in the network. We say that an *attack succeeds* in a state if (at least) one of the routers has a fake message in its database, and no router has a message waiting to be sent. This means that no fight back is going to change the fake topology view of this router. Thus, the attack is persistent.

We ran the model checking tool CBMC [25] on several topologies. We note that the OSPF protocol is quite elaborate. Further, the size of the database of each router is proportional to the size of the network. We therefore limited the topology sizes in order to fit in the model checker capacity. We have found several attacks on small network topologies. The vulnerabilities revealed by the attacks we found are known and accepted by OSPF experts.

The limitation of the approach described so far is clear. It can only check a specific and small network topology which may expose only a part of the protocol's functionality. In order to allow for a good coverage of the protocol's functionality many other specific topologies need to be checked, taking more time and computing resources.

We therefore develop an approach which can search for attacks in a *parameterized network*, consisting of a *family of networks* with varied sizes and topologies. We define an *abstract network*, that represents such a family. The abstract network $\mathcal{A}$ has the property that if there is an attack on $\mathcal{A}$ then there is a corresponding attack on *each* of the (concrete) networks represented by $\mathcal{A}$. An abstract network allows revealing security vulnerabilities in the OSPF protocol, which can harm routing in huge networks with complex topologies. Finding such attacks directly on the huge networks is practically impossible. Abstraction is therefore essential.

The abstraction is defined on all levels of the model: We define an abstract topology which represents a family of concrete topologies. An abstract state represents a set of concrete states. The correspondence between abstract transitions and their concrete counterpart is more subtle. Each abstract transition represents a set of finite concrete *runs*, one in each of the concrete topologies represented by A. As a result, our abstract model is unusual: It under-approximates each member in a family of concrete models. That is, every run of the abstract model has a corresponding run in each of the concrete models represented by it. This is an important characteristic of our abstraction as it allows us to find *general* attacks on an abstract network which are manifested in each of the concrete models it represents. Thus, these attacks are applicable in a large (even infinite) number of networks. This indicates that they exploit fundamental vulnerabilities, which are applicable to many configurations of the network. This is in contrast to finding a specific attack that is only applicable for a single perhaps marginal network configuration.

In this part, we have found attacks on abstract networks manually. However, our abstract model can be implemented for instance in C to be used with CBMC, similarly to our implementation of the concrete model.

It should be noted that in principle, more attacks could be found on a concrete system that belongs to a family. However, in this work we are interested in finding

general attacks, that are robust to changes in the topology. These are usually the first attacks a network operator would like to know with regard to its network.

We emphasize that the contributions of this work go beyond the security analysis of OSPF. The abstract concept and definition can be beneficial for finding security vulnerabilities in other protocols as well.

2.2 Related Work

There are a few works that present a security analysis of the OSPF protocol. Most such works (e.g., [69, 70, 37, 51]) focus on LSA falsification attacks. Only two past works ([37] and [51]) present OSPF attacks with a persistent effect while evading a fight-back. This low number of works stands in contrast to the centrality of OSPF to Internet routing. This can be partially explained by the difficulty to do a manual and thorough security analysis of complex distributed network protocols.

There are some works that propose a security analysis of the design of network protocols based on model checking (e.g., [48, 49, 41]) . All past works check a given network configuration with a predetermined set of participants. In particular, some works (e.g., [47, 30, 45]) analyzed the security of OSPF and other routing protocols, while considering only a given network model. As other distributed network protocols the functionality of a routing protocol is highly dependent of the number of participants in the protocol and the network topology. Hence, current works that employ model checking for distributed network protocols may not cover the entire protocol's functionality.

Reasoning about families of systems, also known as *parameterized systems*, is a known research area (e.g. [33, 40, 27, 56, 7]). Most works present an abstract model which *over-approximates* all members in the family and is used to verify that they all satisfy a given property. We, on the other hand, define an abstract model which *under-approximates* each member in the family. Our abstract model is therefore most suitable for finding attacks on all members. To the best of our knowledge, no similar reasoning has been applied before to parameterized systems.

2.3 The OSPF Protocol

2.3.1 OSPF Basics

The Internet is clustered into sets of connected networks and routers called Autonomous Systems (AS). Each AS is administered by a single authority, such as a large organization, or an Internet service provider. Within each AS a routing protocol is run. Its aim is to allow routers to construct their routing tables, while dynamically adapting to changes in the AS topology. Open Shortest Path First (OSPF) [50] is currently used within most ASes on the Internet. It was developed and standardized by the IETF organization.

Each OSPF router composes a list of all its links to neighboring routers and their

costs. This list is termed *Link State Advertisement* (LSA). Each LSA is flooded throughout the AS. Every router compiles a database of the LSAs from all routers in the AS, thus having a complete view of the AS topology. This allows a router to calculate the least cost paths between it and every other router in the AS. As a result, the router's routing table is formed.

A new instance of each LSA is advertised periodically every 30 minutes, by default. Every LSA has a sequence number which is incremented with every new advertised instance. A more recent LSA instance with a higher sequence number will always take precedence over an older instance with a lower sequence number. An LSA includes the following fields: a) *src* - the router which just sent the LSA; b) *dest* - the router to which the LSA is destined; c) *orig* - the router which first advertised the LSA; d) *seq* - sequence number.

Two routers in the AS may be connected over a *point-to-point* link. A subset of two or more routers may be connected over a *transit network*. One router in every transit network is selected to act as a *designated router*. During the flooding of an LSA each router sends the LSA to all its neighbors (except the neighbor from which the LSA was received). To alleviate flooding load this rule has an exception: a non-designated router may flood an LSA over a transit network only to the designated router of that network. The designated router will send it to all the other routers in that transit network. Note that a router will only receive an LSA from one of its neighbors. An LSA having a *src* that is not one of the router's neighbors will be dropped.

A common goal for an OSPF attacker is to advertise a fake LSA on behalf of some other router in the AS. Such an attack changes the view other routers have of the AS topology and consequently changes their routing tables. The primary measure by which OSPF defends against such attacks is the "*fight-back*" mechanism. Once a victim router receives an instance of its own LSA which is newer than the last instance it originated, it immediately advertises a newer instance of the LSA with a higher sequence number which cancels out the false one. This mechanism prevents most OSPF attacks from persistently falsifying an LSA of another router. Another defense measure is the authentication of LSAs using a secret key shared by the routers of the autonomous system. An outside router that does not know the shared secret can not send LSAs to routers inside the autonomous system.

2.3.2 Threat Model

We adopt the common threat model found in the literature ([69, 70, 37, 51]). This model assumes the attacker has the ability to send LSAs to routers in the AS, and the routers process them as valid LSAs. This assumption can be trivially achieved by an insider, namely an attacker who gained control over just a *single* router in the AS.

The attacker can gain control of a router, for example, by remotely exploiting an implementation vulnerability on the router. Several such vulnerabilities have been

published in the past (e.g., CVE-2010-0581, CVE-2010-0580, and CVE-2009-2865).

Note that since the attacker has control of a legitimate router, the attacker knows the secret key used to authenticate the LSA messages. An outside router that does not know the shared secret cannot send LSAs to routers inside the autonomous system.

2.4 Modeling OSPF

2.4.1 The Concrete Model

In the following we present the concrete model for OSPF we used to find attacks. We note that our model is a simplified version of the real OSPF.

Our model assumes as a starting point a stable routing state in the AS. Namely, all the routers advertised their LSAs and calculated their routing tables. In particular, no LSA flooding is in progress or about to start. The LSA databases of all routers are complete and identical. Without loss of generality we assume that the sequence numbers of all the LSAs that have been advertised are 0. In addition, designated routers for all transit networks have been selected. The model is composed of three entities: (AS) *topology* which models a concrete topology of the AS, *Router* which models a legitimate router inside the AS; *Attacker* which models a malicious router inside the AS.

Autonomous System Topology Model

We denote the concrete topology by $T_c = (R, S, E, DR_c)$, where R is the set of routers, $S \subseteq 2^R$ is the set of transit networks, which we refer to as *sub-network*, $E \subseteq R \times R$ is a set of undirected edges, each representing a point-to-point link between two routers, and $DR_c : S \to R$ maps sub-networks to their designated routers. For simplicity of presentation we assume that each router belongs to at least one sub-network. We emphasize that the routers forming a sub-network are directly connected to each other as if they were forming a clique. Nonetheless, those connections are not part of the set E which only includes point-to-point links. Figure 2.1 depicts an example of a topology. The dashed circles marked as s_i are sub-networks, the circles r_i are routers, and lines connecting routers are edges. Bold circles represent designated routers.

Router Model

The router model executes the standard functionality of the protocol. We model only part of the functionality defined by the OSPF standard since a large model might be infeasible for model checking. Nonetheless, our model captures the protocol's essential operations which any attack must exploit. For example, flooding by its very nature must be exploited by any attack that aims to advertise false LSAs. The functionality we modeled includes: (1) LSA message structure. (2) Flooding procedure. (3) Designated router logic. (4) Fight-back mechanism.

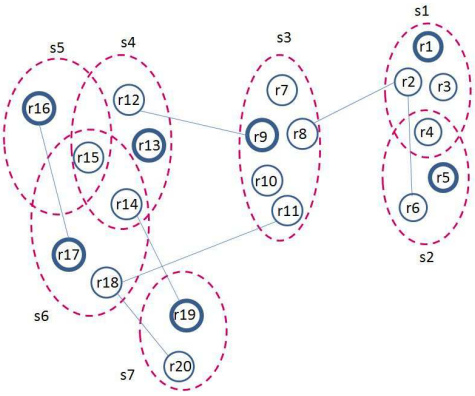

Figure 2.1: Example of a concrete topology

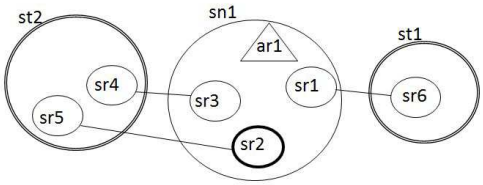

Figure 2.2: Example of an abstract topology

We do not model the actual contents of each LSA, i.e. the list of advertised links and their costs, because the LSA content has no material effect on the attack technique used to advertise a fake LSA. Figure 2.3 gives a high level overview of the router procedure. $r.Q$ denotes r's incoming message queue. A message $m = (src, dest, orig, seq)$. $r.DB$ denotes the set of LSA instances currently installed in r's database.

Attacker Model

In our work we assume that an attacker is one of the routers of the autonomous system. Other routers treat the attacker as a legitimate router. The attacker is free from the protocol's standard and is able to ignore incoming messages and to originate messages arbitrarily. In particular, an attacker may originate fake LSAs on behalf of other routers in the topology. The model indicates such LSAs by a special *isFake* flag, which is not part of the OSPF standard, and legitimate routers do not make use of it. This flag allows us to easily define the specifications for the model (see section 2.4.3). Note that since the attacker has control of a legitimate router, the attacker knows the secret key used to authenticate the LSA messages (see Section 2.3.2).

Another important capability of the attacker is sending an LSA to a non-neighbor destination through several links without being opened on the way. Thus, the intermediate routers will not process the message. We call this *unicast* sending. This is a trivial capability that is inherent to any IP network. Every router (malicious or benign) can send messages directly to remote routers. However, regular routers following the OSPF protocol do not use this capability when flooding LSA messages.

2.4.2 Formal Model for OSPF

The formal model we use for OSPF is a finite state machine with global states and transitions. In order to obtain a finite model suitable for model checking, we impose a predefined bound SB on the sequence number of messages, and a predefined bound K

```
if (r.Q not empty)
{
  m = pop-head(r.Q)
  if (m.dest ≠ r)
    send m according to r's routing table
  else //m.dest is r
  {
    if (m is newer than the copy in r.DB)
    {
      if (m.orig == r)
        fight-back
      else
        update r.DB and flood m
    }
    else
      ignore m
  }
}
```

Figure 2.3: A sketch of the router r procedure.

on the queue size of each router. It should be noted that in real OSPF such bounds exist as well. The queue of each router consists of up to K messages of the form $m = (src, dest, orig, seq, isFake)$, taken from the message domain $M = R \times R \times R \times \{0, ..., SB\} \times \{T, F\}$. The database of router r, $r.DB : R \to \{0, \ldots, SB\} \times \{T, F\}$, includes for each router r' the sequence number of the last message that was originated by r' and reached r , and the value of the flag *isFake* indicating whether this message was in fact originated by the attacker and not by r'.

Definition 2.1. A *global state* $\sigma = \{r.DB \,|r \in R\} \cup \{r.Q \,|r \in R\}$ consists of a database and a message queue for each of the routers in the topology, including the attacker.

Definition 2.2. An *r-transition* between two global states corresponds to an application of the router r procedure (which is either the procedure given in Figure 2.3 if r is a regular router, or the attacker's procedure if r is the attacker). Note that an r-transition may change, in addition to the queue and the database of r, the queues of some of its neighbors.

Definition 2.3. A *run* of the model consists of a sequence of global states $\sigma_1, \ldots \sigma_n$, such that for each i, a router r from R is chosen nondeterministically, and an r-transition is applied to σ_i, resulting in σ_{i+1}.

2.4.3 Specification

Our aim is to discover attacks on OSPF that allow an attacker to persistently falsify LSAs of legitimate routers. Our specification for the absence of a successful persistent attack requires that each state will satisfy at least one of the following two conditions:

1. No router has a fake LSA in its database.

2. At least one message resides in a router's queue.

The first condition verifies that the attacker has not fooled another router to install a fake LSA. The second condition relates to the attack's persistency. If not all the routers' queues are empty then the router whose LSA has been falsified might still fight back and revert the effect of the attack. Note that a state which violates the specification defines the outcome of a successful persistent attack regardless of a specific attack technique.

A model checker will search for a violation of the specification. When found, it will return a counterexample in the form of a run of the model which leads to a violating state. This run is actually an *attack* on OSPF.

2.4.4 Experimental Data

We have implemented in C our concrete model of OSPF, which is a simplified version of the protocol. The implementation is a rather small C program with a few hundreds of code lines. To find counterexamples, i.e. attacks, for which the above specification does not hold we use CBMC, a bounded model checker tool [25]. CBMC can check if a C program satisfies a specification along bounded runs. In our model, we bounded the number of cycles by 8, such that in each cycle any of the routers (including the attacker) can run their procedure once. In order to have a finite model which is rather small, we used a bound of $K = 4$ for the queue size, and a bound of $SB = 8$ for possible sequence numbers.

All our experiments were conducted on Intel Xeon X5650 with 32GB of memory. Table 2.1 details for several network topologies of different sizes, the number of variables and clauses in the CNF formula generated by CBMC, and the time it took to solve the formula using the solver MiniSAT2 [26] .

Table 2.1: For CNF formulas encoding topologies of different sizes, the number of variables and clauses in millions and the solving time in hours

#Routers	#Variables	#Clauses	Time
5	8M	21M	3.17h
6	17M	40M	7.07h
7	23M	55M	12.87h

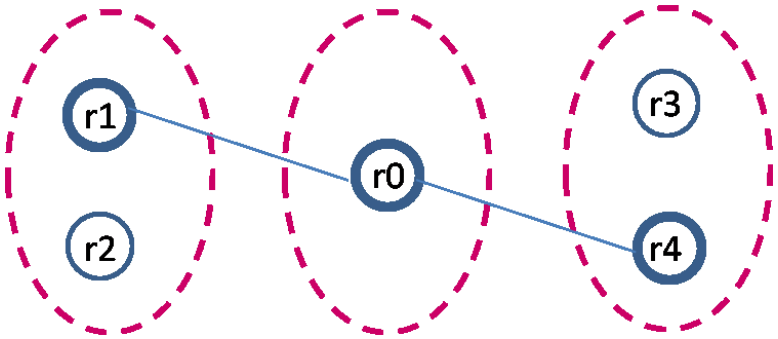

Figure 2.4: The experimented OSPF topology

2.4.5 Example of Attacks on OSPF

As mentioned before, when an attack is found the model checker CBMC outputs a path of global concrete states ending with a state that violates the specification. Figure 2.4 depicts an example of a topology with three sub-networks: $\{r1, r2\}$, $\{r3, r4\}$, and $\{r0\}$. $r1$ and $r4$ act as designated routers. The router $r0$ is attached to $r1$ and $r4$ using point-to-point links. In this topology $r3$ is the attacker. Note that although there are no edges between routers in the same sub-network, they are considered directly connected.

In the following we describe several attacks we found using the above concrete model having the topology depicted in Fig. 2.4. The state explosion problem of the model checking impedes finding more complex attacks which may only be exhibited on larger topologies.

Recall that our model is a simplified version of the real OSPF. As the OSPF standard is given in an English manuscript, we cannot formally prove that our model is an under-approximation of the real OSPF. However, an OSPF expert validated that attacks found in our model are also valid in the full OSPF protocol.

Attack #1

The attacker ($r3$) originates a fake LSA on behalf of $r4$ directly to $r2$ (using unicast sending), while falsifying the source to be $r1$. The fields of the fake LSA are: $src = r1$, $dest = r2$, $orig = r4$, $seq = 1$, and $isFake = true$. $r2$ receives this LSA while considering it to be a valid LSA sent by $r1$. Since the sequence number of the attacker's LSA is larger than that of the LSA instance installed in $r2$'s database, $r2$ installs the attacker's LSA in its database. Since $r2$ received the message from $r1$, it does not flood it back to it. Since $r2$ has no other links no further messages are sent in the topology. Hence, the specification of our model is violated.

Attack #2

The following attack relies on the fact that the routers' queues are bounded. Note that any real-life router must bound its queue size that is dependent on the size of memory space in the router. The attacker continuously sends the following message many times: ($src = r3$, $dest = r4$, $orig = r0$, $seq = 1$, $isFake = true$). The number of sent copies should be larger than the bound on the size of the routers' queues. The messages are received by $r4$ which floods the first message to $r0$. $r0$ then originates a fight-back message m' with $seq = 2$. Since the queue of $r4$ is full, m' will be discarded leaving $r4$ with the fake message installed in the database. All subsequent fake messages flooded to $r0$ will not trigger fight-back, since their sequence number (1) is smaller than that of the last message originated by $r0$ (m' with $seq = 2$). We note that the OSPF standard makes use of a reliable delivery of messages by leveraging acknowledgment messages. Hence a real router retransmits a message until it receives an acknowledgment. Our model does not include this functionality. Nonetheless, this attack would still be feasible in real life if the attacker continued sending messages to keep $r4$'s queue full.

Attack #3

The following attack was first described in [51]. The attacker sends the following two LSA messages: $m1 = (src = r3, dest = r4, orig = r1, seq = 1, isFake = true)$ and $m2 = (src = r3, dest = r4, orig = r1, seq = 2, isFake = true)$. First, $m1$ is received and installed by $r4$. Then, $r4$ floods it to $r0$. Afterwards, $m2$ is received by $r4$. Since it has a higher sequence number than $m1$, $m2$ supersedes it in $r4$'s database. $m2$ is also flooded to $r0$. $r0$ processes and sends both messages to $r1$, while $m2$ is the last to be installed in its database. Once $r1$ receives $m1$ it immediately originates a fight-back message $m3$ with $seq = 2$ and floods it to all its neighbors. $r1$ then receives $m2$. Since $m2$ and $m3$ have equal sequence number (2), $m2$ is not considered newer than $m3$, hence $r1$ does not send another fight-back message and ignores $m2$. Once $r0$ receives $m3$, it does not consider it newer than $m2$ which is currently installed in its database. Hence, it ignores $m3$. Since $r4$ installed the fake message $m2$ and no more messages are waiting to be sent the specification of our model is violated.

2.5 An Abstract Network and Its Matching Concrete Networks

In the previous section we showed how attacks can be found on concrete models. Due to the state explosion problem, the models that can be handled are very small and hence restricted in their topologies. We would like to extend our search for attacks to larger and more complex topologies. Further, we are interested in *general* attacks, which are insensitive to most of the details of the topology and therefore can be applied in a family of topologies.

In order to achieve that, we define an *abstract model* which can represent a family of concrete models. The models in the family are similar in some aspects of their topologies but may differ in many other aspects.

The abstract model consists of an abstract topology which includes abstract components representing many routers and sub-networks, and of an abstract protocol which is an adjustment of OSPF to the abstract components.

We define several levels of abstract components. The most abstract component is the *sub-topology*, which represents any number of concrete sub-networks. The edges between the sub-topology and the rest of the topology are not abstracted. As a result, routers within the sub-topology which are connected to these edges remain un-abstracted as well. These routers are called *singleton routers*. The concrete routers they represent are called *visible*. All other routers within the sub-topology and the edges among them are fully abstracted, and are referred to as *invisible*.

Another abstract component is the *abstract router* which represents a set of concrete routers, all contained within the same sub-network, and have no edges outside of the sub-network. An *abstract sub-network* consists of a set of abstract routers and a set of singleton routers. As with sub-topologies, the singleton routers in a sub-network are un-abstracted. They represent a single concrete router whose edges are un-abstracted too. We require that each singleton router belongs to either a sub-topology or a nonempty set of abstract sub-networks.

The intuition behind the definition of an abstract topology is as follows. The un-abstracted routers are those that may participate in an attack. The others are needed to form a topology that brings un-abstracted routers to manifest more of their OSPF functionality and thus to possibly expose more security vulnerabilities. Moreover, abstracted routers allow to show that a found attack is general and applicable to a family of topologies.

Clearly, the attacker is always an (un-abstracted) singleton router. Moreover, the messages sent by the attacker are un-abstracted as well. That is, their originator, source, and destination fields refer to singleton routers.

We impose some constraints on abstract sub-topologies, to guarantee that for every abstract transition and every concrete topology represented by the abstract topology, there can be found a corresponding finite concrete run.

For a sub-topology st, recall that each singleton router in st represents a single concrete visible router. We require that in the part of the concrete topology which is represented by st, each of its visible routers must belong to a different sub-network. Also, visible routers in st may not be directly connected to each other, but should be connected to at least one invisible router. Further, the invisible routers in st form a strongly connected component. These constraints guarantee that if a message is flooded to st by a singleton router r, then there is a concrete run along which the message is opened by all invisible routers prior to being opened by any other singleton router.

While these constraints seem quite restrictive, our abstract topologies still represent

a large variety of topologies of different sizes. As shown in Section 2.5.6, some nontrivial attacks were found on them. Many of these constraints can be removed for the price of much more complex definitions and correctness proof. We choose to present a simpler version here, and to demonstrate its usability.

2.5.1 Abstract Topology

Formally, an abstract topology is denoted by $T_A = (SR, ST, AR, SN, E_A, DR_A)$, where

- SR is a set of abstract singleton routers, each of which representing a single concrete router.
- $ST \subseteq 2^{SR}$ is a set of sub-topologies. A sub-topology contains a set of abstract singleton routers from which there are edges to other components in the abstract topology. Each sub-topology represents a set of concrete sub-networks which forms a strongly connected component in the concrete topology.
- AR is a set of abstract routers. Each abstract router represents a set of concrete routers within the same sub-network in the concrete topology.
- $SN \subseteq 2^{AR \cup SR}$ is a set of abstract sub-networks. Each abstract sub-network represents a single concrete sub-network.
- $E_A \subseteq SR \times SR$ is a set of undirected edges, each representing a point to point link between two abstract singleton routers.
- $DR_A : SN \to SR$ is a function that maps sub-networks to their designated router, which must be from SR.

We will use the following notations:

- sr denotes a singleton router in SR
- st denotes a sub-topology in ST
- ar denotes an abstract router in AR
- sn denotes a sub-network in SN
- r denotes a concrete router in R
- s denotes a concrete sub-network in S

Example 2.4. The following abstract topology is depicted in Figure 2.2:
$T_A = (SR, ST, AR, SN, E_A, DR_A)$, where:

- $SR = \{sr_1, sr_2, sr_3, sr_4, sr_5, sr_6\}$
- $ST = \{st_1, st_2\}$, where: $st_1 = \{sr_6\}$, $st_2 = \{sr_4, sr_5\}$

- $AR = \{ar_1\}$
- $SN = \{sn_1\}$, where: $sn_1 = \{ar_1, sr_1, sr_2, sr_3\}$
- $E_A = \{(sr_4, sr_3), (sr_2, sr_5), (sr_6, sr_1)\}$
- $DR_A = \{(sn_1, sr_2)\}$

Note that connections between routers within the same sub-network are not depicted in the figure, similarly to the concrete case.

Below we detail the constraints imposed on the abstract topology T_A:

1. A singleton router cannot be in both a sub-network and a sub-topology, but has to be in either a sub-topology or a sub-network:
   $\forall sr, sn : (sr \in sn \Rightarrow \neg\exists st : (sr \in st))$
   $\forall sr, st\,(sr \in st \Rightarrow \neg\exists sn : (sr \in sn))$
   $\forall sr\,((\exists st\,[sr \in st]) \vee (\exists s\,[sr \in s]))$
2. A singleton router cannot be in more than one sub-topology:
   $\forall st_1, st_2 : (st_1 \neq st_2 \Rightarrow st_1 \cap st_2 = \emptyset)$
3. E_A does not include edges between routers within the same sub-topology:
   $\forall sr_1, sr_2, \forall st\,(((sr_1 \in st) \wedge (sr_2 \in st) \wedge (sr_1 \neq sr_2)) \Rightarrow ((sr_1, sr_2) \notin E_A))$

2.5.2 Matching Abstract and Concrete Topologies

We define a matching relation between abstract and concrete topologies. The matching relation adhere to the intuitive explanation given above. Let $T_A = (SR, ST, AR, SN, E_A, DR_A)$ be an abstract topology and $T_C = (R, S, E, DR_C)$ be a concrete topology. A relation

$$H \subseteq (SR \times R) \cup \left(AR \times 2^R\right) \cup (SN \times S) \cup \left(ST \times 2^S\right) \cup (E_A \times E)$$

is a *matching relation* between T_A and T_C if it satisfies the set of constraints defined below.

Example 2.5. The relation H, given below, is a matching relation between T_A from Figure 2.2 and T_C from Figure 2.1.

- $H \cap (SR \times R) = \{(sr1, r8), (sr2, r9), (sr3, r11), (sr4, r18), (sr5, r12), (sr6, r2)\}$
- $H \cap \left(AR \times 2^R\right) = \{(ar1, \{r7, r10\})\}$
- $H \cap (SN \times S) = \{(sn1, s3)\}$
- $H \cap \left(ST \times 2^S\right) = \{(st1, \{s1, s2\}), (st2, \{s4, s5, s6, s7\})\}$

- $H \cap (E_A \times E) = \{((sr1, sr6), (r8, r2)), ((sr4, sr3), (r18, r11)),$
  $((sr2, sr5), (r9, r12))\}$

Below we detail the constraints imposed on the matching relation H:

- Constraints related to the matching between SR and R:

  1. $H \cap (SR \times R)$ is a 1-1 function. We will denote this part of H by abuse of notation: $H : SR \to R$. In fact, all other parts of H are also constrained to be a 1-1 function, and thus this notation will apply to all parts.
  2. An abstract sub-network sn contains a singleton router sr if and only if the matched concrete sub-network $H(sn)$ contains the matched router $H(sr)$:
     $\forall sr, sn\,[(sr \in sn \Leftrightarrow H(sr) \in H(sn))]$
  3. Given the sub-topology st and a singleton router $sr \in st$, all concrete sub-networks that contain $H(sr)$ must be matched to the sub-topology (because a sub-topology cannot represent a partial sub-network).
     $\forall sr, \forall st\,[sr \in st \Rightarrow (\forall s\,[H(sr) \in s \Rightarrow s \in H(st)])]$
  4. For each singleton router in a sub-topology, there must be a matching concrete router in the matched part of the concrete topology.
     $$\forall sr, \forall st \left(sr \in st \Rightarrow H(sr) \in \bigcup_{s \in H(st)} s \right)$$
  5. Singleton routers within the same sub-topology must be matched with concrete routers which are in different sub-networks on the concrete topology.

     $$\forall sr_1, sr_2, st\,([sr_1 \in st \wedge sr_2 \in st] \Rightarrow (\neg \exists s\,(H(sr_1) \in s \wedge H(sr_2) \in s)))$$

- Constraints related to the matching between AR and 2^R:

  1. $H \cap \left(AR \times 2^R\right)$is a 1-1 function.
  2. Any 2 sets of concrete routers that are matched with different abstract routers must be disjoint.
     $\forall ar_1 \neq ar_2\,[H(ar_1) \cap H(ar_2) = \emptyset]$
  3. The set of concrete routers that is matched with an abstract router must be in the concrete sub-network that is matched with the abstract sub-network of the abstract router.
     $\forall ar, \forall sn\,[ar \in sn \Rightarrow H(ar) \subseteq H(sn)]$
  4. $\forall ar\,[H(ar) \neq \emptyset]$

- Constraints related to the matching between SN and S:

  1. $H \cap (SN \times S)$ is a 1-1 function.

2. For each router r in a concrete sub-network which is matched to an abstract sub-network in the abstract topology, there should either exist a matching singleton router in the matching abstract-sub-network, or an abstract router in the abstract sub-network that is matched with a set of concrete routers in the concrete sub-network, which includes this concrete router r.

$$\forall sn, \forall r \left[r \in H\left(sn\right) \Rightarrow \left(\left(\exists sr : \left(r = H\left(sr\right) \right) \right) \vee \left(\exists ar : \left(r \in H\left(ar\right) \right) \right) \right) \right]$$

- Constraints related to the matching between ST and 2^S:

1. $H \cap \left(ST \times 2^S \right)$ is a 1-1 function.
2. Any two sets of concrete sub-networks which are matched with different sub-topologies in the abstract topology must be disjoint.
   $\forall st_1 \neq st_2 \left[\left(\bigcup_{s \in H(st_1)} s \right) \cap \left(\bigcup_{s \in H(st_2)} s \right) = \varnothing \right]$
3. For each $st \in ST$, the sub-graph formed by $\bigcup_{s \in H(st)} s$ in the concrete network is a strongly connected component, taking into account the implicit links within sub-networks, which are not part of E.
4. $\forall st \left[H\left(st\right) \neq \emptyset \right]$
5. For each $st \in ST$, the sub-graph in the concrete network formed by:

   $$\left(\bigcup_{s \in H(st)} s \right) \setminus \left(r \in R | \exists sr \left(H\left(sr\right) = r \right) \right)$$

   is a strongly connected component, taking into account the implicit links within sub-networks, which are not part of E.

- Constraints related to the matching between E_A and E:

1. $H \cap \left(E_A \times E \right)$ is a 1-1 function.
2. An edge between two singleton routers in the abstract topology must be matched with an edge between two concrete routers, which are matched with the singleton routers.
   $\forall \left(sr_1, sr_2 \right) \in E_A, \ \forall \left(r_1, r_2 \right) \in E :$
   $\left[\left(\left(sr_1, sr_2 \right), \left(r_1, r_2 \right) \right) \in H \Rightarrow H\left(sr_1\right) = r_1 \wedge H\left(sr_2\right) = r_2 \right]$
3. A concrete edge between routers that have matching abstract singleton routers, should be matched with the edge between the singleton routers.
   $\forall r_1, r_2 [\left(\left(r_1, r_2 \right) \in E \wedge \exists sr_1 \left(H\left(sr_1\right) = r_1 \right) \wedge \exists sr_2 \left(H\left(sr_2\right) = r_2 \right) \right) \Rightarrow$
   $\left(\left(sr_1, sr_2 \right) \in E_A \right) \wedge \left(\left(sr_1, sr_2 \right), \left(r_1, r_2 \right) \right) \in H]$
4. There is no concrete edge between concrete router which is represented by an abstract router in the abstract topology.
   $\left(r_1, r_2 \right) \in E \Rightarrow \left(\neg \exists ar : \left(r_1 \in H\left(ar\right) \vee r_2 \in H\left(ar\right) \right) \right)$

- More global constraints on the matching relation H

  1. Each singleton router that is a designated router in the abstract topology, should be matched with a concrete router which is also designated.
     $\forall sn, \forall sr \left[DR_A(sn) = sr \Rightarrow DR_C(H(sn)) = H(sr) \right]$
  2. For each concrete router which is designated, if it has a matched singleton router, then it also should be designated in the matching sub-network.
     $\forall s, \forall r, \forall sn \left[(DR_C(s) = r) \wedge (H(sn) = s) \Rightarrow H(DR_A(sn)) = r \right]$
  3. Any two different types of abstract components are matched with disjoint sets of concrete routers in the concrete topology.
     - $\forall st, \forall sr \left(\left(\bigcup_{s \in H(st)} s \right) \bigcap \{ H(sr) \,|sr \notin st \} = \varnothing \right)$
     - $\forall st, \forall ar \left(\left(\bigcup_{s \in H(st)} s \right) \bigcap H(ar) = \varnothing \right)$
     - $\forall sr, \forall ar \left(H(ar) \bigcap \{ H(sr) \} = \varnothing \right)$

2.5.3 Global Abstract States

Let T_A be an abstract topology and let $AC = ST \cup AR \cup SR$ be the set of components in the abstract topology. *Abstract messages* consist of the same fields as concrete messages. The message domain in the abstract model is $M = AC \times AC \times ORIGS \times \{0, ..., SB\} \times \{T, F\}$, where $ORIGS \subseteq SR$ is a predefined set of originators which can be used by the attacker in its messages.

An *abstract state* is defined by $\sigma_A = \{ac.DB \,|ac \in AC\} \cup \{ac.Q \,|ac \in AR \cup SR\}$, where for every component $ac \in AC$, the structure of its database is identical to that of a concrete component, $ac.DB : ORIGS \rightarrow \{0, ..., SB\} \times \{T, F\}$, except that here it is only defined for the subset $ORIGS \subseteq SR$. In addition, for every $ac \in AR \cup SR$, $ac.Q$ is a queue of up to K messages. The database is restricted to $ORIGS$ since in our setting (see section 2.4.1) only the attacker originates messages, and those messages have $orig \in ORIGS$. Thus, there is no need for $ac.DB$ to contain entries of other originators.

Note that, we do not define a queue for sub-topologies st, since flooding within st is always described as a single abstract transition. Each singleton router in st has a queue. Thus, a queue for st would have represented the queues of all invisible routers, matched to st. However, the queues of all invisible routers are empty whenever the abstract transition begins or ends. Thus, there is no need to represent their content.

2.5.4 Matching Abstract and Concrete states

Let T_A and T_C be an abstract and concrete topologies and let H be their matching relation. In order to define a matching between abstract and concrete states, we first define a matching between abstract and concrete databases and queues.

We use h to denote a function that matches abstract databases, messages, queues, and global states to sets of their concrete counterparts.

1. An abstract database DB_A matches a concrete database DB_C, denoted $DB_C \in h(DB_A)$, if for each $o \in ORIGS$, the entry for o in DB_A is identical to the entry of $H(o)$ in DB_C.

2. An abstract message m and a concrete message m' match, denoted $m' \in h(m)$, if $m'.src \in H(m.src)$, $m'.dest \in H(m.dest)$, $m'.orig = H(m.orig)$, $m'.seq = m.seq$, and $m'.isFake = m.isFake$.

   Since *orig* is a singleton router and since *seq* and $IsFake$ are un-abstracted, they have a single matching.

3. An abstract queue matches a concrete queue if

   (a) For a singleton router sr, each message m in its queue is matched with a sequence of (one or more) concrete messages in $h(m)$.

   The reason for matching more than one concrete message with m is that an abstract transition may add only one message to the queue. On the other hand, the concrete run that correspond to this transition consists of several concrete transitions, each of which may add a matching message to the queue. This is because, when sr is part of a sub-topology st, then the invisible routers represented by st may flood the message several times to sr via different paths in the sub-topology.

   (b) For an abstract router ar, its queue represents the queues of all concrete routers matched with ar. Here the sizes of the queues are identical since a message received by ar corresponds to single messages received by each r in $H(ar)$ from the designated router. No other messages are sent among routers in $H(ar)$.

We can now define matching of abstract and concrete states. Let σ_C be a concrete global state and σ_A be an abstract global state. $\sigma_C \in h(\sigma_A)$ if the following conditions hold:

1. $\forall ac \in AR \cup SR\,[\forall r \in H(ac)\,(r.Q \in h(ac.Q))]$. That is, queues of matching components must match.

2. $\forall ac \in SR \cup ST \cup AR\,[\forall r \in H(ac)\,(r.DB \in h(ac.DB))]$. That is, databases of matching components must match.

2.5.5 Abstract Transitions and Their Matching Concrete Transitions

An *abstract transition* between two global abstract states corresponds to an application of the procedure of one of the abstract components. The abstract model includes

procedures for a singleton router, an abstract router, and an attacker. Our model does not include a procedure for a sub-topology. Instead, its behavior is defined as part of the procedure of singleton routers included in it.

A high-level description of the procedure of a singleton router sr is given in Figure 2.5. It is similar to the procedure of a concrete router, except that it does not handle messages whose destination is not sr. This is because in the abstract model such messages are sent by unicast directly to their destination. The singleton router procedure can perform either flooding or fight back. Figure 2.6 describes the flooding procedure performed by a singleton router (as part of its procedure). $FD_A(sr, m.src)$ returns the flooding destinations, i.e. set of abstract components to which sr floods a message m obtained from component src. The fight back procedure is similar, except that FD_A is replaced by the fight back destinations, FBD_A. The statement $ac_1.Q' = ac_1.Q \bullet \{m_{sr \to ac_1}\}$ performs an update of ac_1's queue. The resulting queue, $ac_1.Q'$, is obtained by concatenating the old queue $ac_1.Q$ with a message which is identical to m, except that its src is sr and its destination is ac_1.

The procedure of an abstract router is simpler. It only installs a message from its queue in its database and does not perform flood or fight back. This is because it is part of a single abstract sub-network, and is not connected by any edges.

An ac-abstract transition corresponds to a single application of the procedure for abstract component ac. This transition may represent either a single concrete transition or a sequence of concrete transitions (i.e., a concrete run), depending on the type of ac and on the message content. Below we detail a few non-trivial cases where abstract transitions correspond to a concrete run. For every concrete topology T_C represented by an abstract topology T_A and for every abstract transition in T_A, a corresponding concrete run as detailed below can be found in T_C.

Case 1

Consider an abstract transition in which a singleton router sr floods a message m, where sr is within a sub-topology st, and st belongs to the flooding destinations of sr. In such a case, the concrete run represented by the abstract transition includes, in addition to the flooding done by sr, the flooding applied by the invisible routers in $H(st)$. By the end of this run, all invisible routers within st have already removed m from their queue, updated their databases (if their databases were less updated), and flooded m further to the rest of the visible routers in $H(st)$.

Case 2

Consider an abstract transition in which a singleton router sr in a sub-topology st floods a message m, where $m.src = st$. This abstract transition represents a concrete run in which $H(sr)$ floods m. In addition, invisible routers in $H(st)$, which are included in the flooding destinations of $H(sr)$, remove m from their queue and ignore it.

Case 3

Consider an abstract transition in which the attacker sends a message m by unicast to a destination which is not one of its neighbors. That is, the message m is added to the queue of its destination. This abstract transition represents a sequence of concrete transitions in which each router on the routing path which is not the destination, sends the message according to its routing table, without opening the message.

Case 4

Abstract transition taken by an abstract router ar represents a sequence of similar concrete transitions taken by each of the concrete routers represented by ar exactly once.

```
singleton router procedure(sr)
if (sr.Q not empty)
{
  m = pop-head(Q)
    if (m is newer than the copy in sr.DB)
    {
      if (m.orig == sr)
        fight back(sr, m)
      else
        update sr.DB and flood(sr, m)
    }
    else
      ignore m
  }
}
```

Figure 2.5: Procedure of a singleton router

2.5.6 Examples of OSPF Attacks in the Abstract Model

In this section we describe a few attacks, found on different abstract models which we picked manually.

Attack #1

This attack has been found on the abstract topology T_A, presented in Figure 2.2. The attacker is $sr2$. The set of predefined originators is $ORIGS = \{sr1\}$. The attacker originates a fake message on behalf of $sr1$: $m = (src = sr2, dest = sr5, orig = sr1, seq = 1, isFake = T)$. $sr5$ receives this message while considering it to be a valid message, sent by $sr2$. Since the sequence number of m is larger than that of the message instance installed in $sr5$'s database, $sr5$ installs m in its database, and

```
flood(sr, m)
For each
ac_1 ∈ FD_A(sr, m.src) ∩ (AR ∪ SR))
{
  ac_1.Q' = ac_1.Q . {m_{sr→ac_1}}
}
For each st ∈ FD_A(sr, m.src) ∩ ST
{
  if (st.DB[m.orig].seq < m.seq)
  {
    st.DB'[m.orig] = (m.seq, m.isFake)
    For each sr_1 ∈ FD_A(st, sr)
      sr_1.Q' = sr_1.Q . {m_{st→sr_1}}
  }
}
```

Figure 2.6: Flooding procedure of a singleton router

floods it. The fake message will be flooded and installed in the databases of $st2, sr4$, and $sr3$. When m is installed by sr3, it will be flooded to the attacker $sr2$, since $sr2$ is the designated router of the sub-network $sn1$. The attacker will choose to ignore m, thus preventing this message from being flooded to $sr1$, and avoiding fight back. Since no more messages are waiting to be sent, the specification is violated.

Formally, the abstract path that represents the attack can be described as:

$$\pi : \sigma_0 \xrightarrow{sr2} \sigma_1 \xrightarrow{sr5} \sigma_2 \xrightarrow{sr4} \sigma_3 \xrightarrow{sr3} \sigma_4 \xrightarrow{sr2} \sigma_5$$

We denote on each abstract transition, the abstract component in $SR \cup AR$ for which the procedure was taken. The contents of the abstract global states can be described as follows, where we will denote only the changes between two consequent states:

- $\sigma_0 = [<>]$
- $\sigma_1 = [sr5.Q =< m_{sr2 \to sr5} >]$
- $\sigma_2 = [sr5.DB[sr1] = (1, T)\,; st2.DB[sr1] = (1, T)\,; sr4.Q =< m_{st2 \to sr4} >; sr5.Q =<>]$
- $\sigma_3 = [sr4.DB[sr1] = (1, T)\,; sr3.Q =< m_{sr4 \to sr3} >; sr4.Q =<>]$
- $\sigma_4 = [sr3.DB[sr1] = (1, T)\,; sr2.Q =< m_{sr3 \to sr2} >; sr3.Q =<>]$
- $\sigma_5 = [sr2.Q =<>]$

A matching concrete path that represents this attack on the matching concrete topology from Figure 2.1 can be denoted as:

$\pi' : {\sigma_0}' \xrightarrow{r9} {\sigma_1}' \xrightarrow{r12} {\sigma_2}' \xrightarrow{r13} {\sigma_3}' \xrightarrow{r14} {\sigma_4}' \xrightarrow{r15} {\sigma_5}' \xrightarrow{r19} {\sigma_6}' \xrightarrow{r20} {\sigma_7}' \xrightarrow{r17} {\sigma_8}' \xrightarrow{r17} {\sigma_9}' \xrightarrow{r16} {\sigma_{10}}' \xrightarrow{r16} {\sigma_{11}}' \xrightarrow{r17} {\sigma_{12}}' \xrightarrow{r15} {\sigma_{13}}' \xrightarrow{r18} {\sigma_{14}}' \xrightarrow{r18} {\sigma_{15}}' \xrightarrow{r17} {\sigma_{16}}' \xrightarrow{r11} {\sigma_{17}}' \xrightarrow{r9} {\sigma_{18}}'$

Matching between abstract and concrete transitions:
The abstract transition $\sigma_0 \xrightarrow{sr2} \sigma_1$ matches the concrete transition:${\sigma_0}' \xrightarrow{r9} {\sigma_1}'$. The abstract transition $\sigma_1 \xrightarrow{sr5} \sigma_2$ matches the sequence of concrete transitions: ${\sigma_1}' \xrightarrow{r12} {\sigma_2}' \xrightarrow{r13} {\sigma_3}' \xrightarrow{r14} {\sigma_4}' \xrightarrow{r15} {\sigma_5}' \xrightarrow{r19} {\sigma_6}' \xrightarrow{r20} {\sigma_7}' \xrightarrow{r17} {\sigma_8}' \xrightarrow{r17} {\sigma_9}' \xrightarrow{r16} {\sigma_{10}}' \xrightarrow{r16} {\sigma_{11}}' \xrightarrow{r17} {\sigma_{12}}' \xrightarrow{r15} {\sigma_{13}}'$. The abstract transition $\sigma_2 \xrightarrow{sr4} \sigma_3$ matches the sequence of concrete transitions: ${\sigma_{13}}' \xrightarrow{r18} {\sigma_{14}}' \xrightarrow{r18} {\sigma_{15}}' \xrightarrow{r17} {\sigma_{16}}'$. The abstract transition $\sigma_3 \xrightarrow{sr3} \sigma_4$ matches the concrete transition ${\sigma_{16}}' \xrightarrow{r11} {\sigma_{17}}'$. The abstract transition $\sigma_4 \xrightarrow{sr2} \sigma_5$ matches the concrete transition ${\sigma_{17}}' \xrightarrow{r9} {\sigma_{18}}'$.

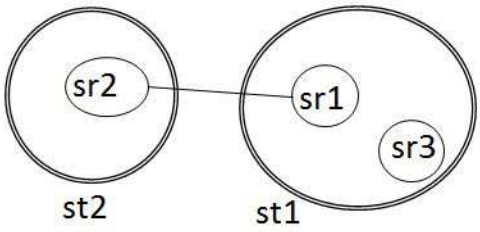

Figure 2.7: Abstract topology of attack #2

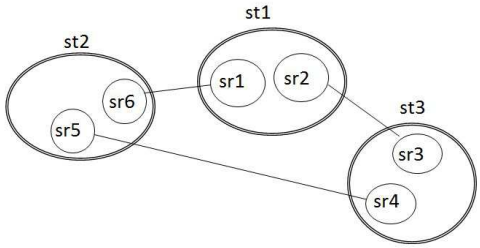

Figure 2.8: Abstract topology of attack #3

Attack #2

T_A is the abstract topology presented in Figure 2.7. The attacker is $sr3$. The set of predefined originators is $ORIGS = \{sr1\}$. The attacker originates a fake message on behalf of $sr1$: $m = (src = sr1, dest = sr2, orig = sr1, seq = 1, isFake = T)$, which is sent by unicast to $sr2$. $sr2$ installs the fake message in its database and floods it only to the sub-topology $st2$ due to the flooding rules of OSPF. Therefore, in the final state the queues of all abstract components are empty, and the databases of $sr2$ and $st2$ are installed with the fake message. Thus, the specification is violated.

Formally, the abstract path that represents the attack can be described as:
$\pi : s_0 \xrightarrow{sr1} s_3 \xrightarrow{sr2} s_2$

The contents of the abstract global states can be described as follows, where we will denote only the changes between two consequent states:

- $\sigma_0 = [<>]$
- $\sigma_1 = [sr2.Q =< m_{sr1 \to sr2} >]$
- $\sigma_2 = [sr2.DB[sr1] = (1, T)\,; st2.DB[sr1] = (1, T)\,; sr2.Q =<>]$

Attack #3

T_A is the abstract topology presented in Figure 2.8. The attacker is $sr3$. The set of predefined originators is $ORIGS = \{sr2\}$. The attacker sends the following two LSAs (using unicast sending): $m1 = (src = sr3, dest = sr2, orig = sr2, seq = 1, isFake = T)$ and $m2 = (src = sr4, dest = sr5, orig = sr2, seq = 2, isFake = T)$. As a result, $sr2$ sends a fight back message $m3$ with $orig = sr2, seq = 2, isFake = F$, but $sr5$ opens $m3$ after it has already installed $m2$ in its database, and will thus ignore the fight back message and will remain with the fake message.

Formally, the abstract path that represents the attack can be described as:
$\pi = \sigma_0 \xrightarrow{sr3} \sigma_1 \xrightarrow{sr3} \sigma_2 \xrightarrow{sr2} \sigma_3 \xrightarrow{sr5} \sigma_4 \xrightarrow{sr6} \sigma_5 \xrightarrow{sr1} \sigma_6 \xrightarrow{sr1} \sigma_7 \xrightarrow{sr3} \sigma_8 \xrightarrow{sr6} \sigma_9$

The contents of the abstract global states can be described as follows, where we will denote only the changes between two consequent states:

- $\sigma_0 = [<>]$
- $\sigma_1 = [sr2.Q =< m_{sr3 \to sr2} >]$
- $\sigma_2 = [sr5.Q =< m'_{sr4 \to sr5} >]$
- $\sigma_3 = [sr2.Q =<>; sr2.DB[sr2] = (2, F)\,; st1.DB[sr2] = (2, F)\,;$
  $sr1.Q =< m''_{st1 \to cr1} >; sr3.Q =< m''_{sr2 \to sr3} >]$
- $\sigma_4 = [sr5.DB[sr2] = (2, T)\,; st2.DB[sr2] = (2, T)\,; sr5.Q =<>; sr6.Q =< m'_{st2 \to sr6} >]$
- $\sigma_5 = [sr6.DB[sr2] = (2, T)\,; sr6.Q =<>; sr1.Q =< m''_{st1 \to sr1}, m'_{sr6 \to sr1} >]$
- $\sigma_6 = [sr1.DB[sr2] = (2, F)\,; sr1.Q =< m'_{sr6 \to sr1} >; sr6.Q =< m''_{sr1 \to sr6} >]$
- $\sigma_7 = [sr1.Q =<>]$
- $\sigma_8 = [sr3.Q =<>]$
- $\sigma_9 = [sr6.Q =<>]$

2.6 Correctness of the Abstract Model

Theorem 2.6. Let T_A and T_C be an abstract and concrete topologies and let H be their matching relation. Then, for each finite abstract run $\sigma_1, \ldots \sigma_n$, there exists a corresponding finite concrete run $\sigma'_1, \ldots \sigma'_k$, such that $\sigma'_1 \in h(\sigma_1)$ and $\sigma'_k \in h(\sigma_n)$.

Corollary 2.7. An abstract attack found on an abstract topology T_A, has a corresponding attack on each matching topology T_C .

Proof Sketch

- We show that for each abstract transition, there is a concrete finite run, such that the initial and final states of the transition and of the run are matching.
- An abstract attack is an abstract run for which the final state violates our specification. A concrete state matching an abstract state which violates the specification, also violates the specification. Thus, the corresponding paths are concrete attacks.
- The proof is based on the matching relation H and on the function h, defined in section 2.5.

In Section 2.6.1 we define the concept of flooding and fight back destinations in the concrete and abstract models. Then, in Section 2.6.2 we give the full proof of Theorem 2.6.

2.6.1 Flooding and Fight Back Destinations

Flooding Destinations in the Concrete Model

We define a flooding destinations function $FD_C : R \times R \to 2^R$ such that given a destination router *dest* and a source router *src*, $FD_C(dest, src)$ is the set of routers to which *dest* will flood an LSA message received from *src*.

$FD_C(dest, src)$ is the minimal set that contains the following routers:

1. For each $(dest, r') \in E$ if $src \neq r'$ then $r' \in FD_C(dest, src)$.
2. For each $s \in S$ such that $dest \in s$, if $DR(s) = dest$ then:

$$\left\{ r' \in s \,\middle|\, r' \neq dest \wedge src \neq r' \right\} \subseteq FD_C(dest, src)$$

   If $src \neq DR(s)$ then: $DR(s) \in FD_C(dest, src)$.

Fight Back Destinations in the Concrete Model

The fight back destinations function $FBD_C : R \to 2^R$ defines, for each router r, the set of routers to which r will flood an LSA fight back message originated by it. $FBD_C(r)$ is the minimal set that contains the following routers:

1. For each $s \in S$ such that $r \in s$, if $DR(s) = r$ then $\{r' \in s \,|\, r' \neq r\} \subseteq FBD_C(r)$ else $DR(s) \in FBD_C(r)$.
2. For each $(r, r') \in E$: $r' \in FBD_C(r)$.

Flooding Destinations in the Abstract Model

Let $AC = SR \cup AR \cup ST$. We define a function of flooding destinations: $FD_A : (SR \cup ST) \times (SR \cup ST) \rightarrow 2^{AC}$, such that given a destination component $dest \in SR \cup ST$ and a source component $src \in SR \cup ST$, $FD_A(dest, src)$ is the set of abstract components in AC to which $dest$ will flood an LSA message received from src.

1. $FD_A(dest, src)$ for $dest \in SR$ is the minimal set that contains the following routers:

   (a) For each $(dest, ac') \in E_A$, if $src \neq ac'$ then $ac' \in FD_A(dest, src)$.

   (b) For each $sn \in SN$ such that $dest \in sn$, if $DR(sn) = dest$ then $\{ac' \in sn \,|ac' \neq dest \wedge src \neq ac'\} \subseteq FD_A(dest, src)$. Else, if $src \neq DR(sn)$, then $DR(sn) \in FD_A(dest, src)$

   (c) For each $st \in ST$ such that $dest \in st$, If $src \neq st$ then $st \in FD_A(dest, src)$

   Note: in the abstract model, if src is a sub-topology, there might be flooding back to the sub-topology in the concrete model. However, since we assume messages originated from the attacker do not have an abstract src field, we can be sure that the sub-topology will ignore the flooded message (since the message must have arrived by regular flooding, thus it has installed the flooded LSA in its database). Thus, we do not include it in the flooding destination. (It actually defines another kind of a macro step. The hidden flooding will be taken into account in the translation of the abstract path into a concrete one).

2. $FD_A(dest, src)$ for $dest \in ST$ is the minimal set that contains the following routers: $\{ac' \in st | ac' \neq src\} \subseteq FD_A(dest, src)$

Fight Back Destinations in the Abstract Model

We define $FBD_A : SR \rightarrow 2^{AC}$ be a fight back destinations function, such that given an abstract singleton router sr,$FBD_A(sr)$ is the set of abstract components to which sr will flood an LSA fight back message originated by it. $FBD_A(sr)$ is the minimal set that contains the following abstract components in AC:

1. For each $(sr, ac') \in E_A$: $ac' \in FBD_A(sr)$.

2. For each $sn \in SN$ such that $sr \in sn$, if $DR(sn) = sr$ then $\{ac' \in sn \,|ac' \neq sr\} \subseteq FBD_A(sr)$, else $DR(sn) \in FBD_A(sr)$

3. For each $st \in ST$ such that $sr \in st$: $st \in FBD_A(sr)$

Matching Flooding Destinations

Lemma 2.8. Let T_A be an abstract topology and T_C a matching concrete topology, such that H is their matching relation. The following matching between flooding destinations exists:

1. $\forall r \in SR, \forall src \in SR \cup ST : \;\; c \in FD_A(r, src) \wedge c \notin ST \Rightarrow$
   $[\forall r' \in H(c) : (r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))]$

2. $\forall r \in R, \forall src \in R : \;\; \forall r' \in FD_C(r, src) :$
   $[(\exists c \notin ST : (r' \in H(c))) \Rightarrow ((r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q')))]$

We refer to matching floodings, i.e, an abstract message is flooded by r, and a matching concrete message is flooded by $H(r)$. Note that Q represents a component's queue before the flooding , and Q' represents the queue after the flooding. When denoting $r' \in H(c)$ for some $c \in AR \cup SR$, we actually refer to $r' = H(c)$ in the case where $c \in SR$.

Proof.

1. Let $r \in SR$, $src \in SR \cup ST$, such that r receives an LSA from src and floods it to $FD_A(r, src)$. We will prove the claim for any possible flooding destinations, according to the definition of the function FD_A:

   (a) For each $(r, c) \in E_A$ such that $src \neq c$, we know that $c \in FD_A(r, src)$ (and $c \in SR$ due to the definition of E_A). According to the matching defined between abstract and concrete topologies, we know that for each $(r, c) \in E_A$ there exists a matching edge of the form: $(H(r), H(c)) \in E$.

   If $src \in SR$, then it can be immediately inferred that $src \neq c \Rightarrow H(src) \neq H(c)$. Therefore, according to the definition of FD_C, we can then infer that $H(c) \in FD_C(H(r), H(src))$. This implies that $H(c).Q \in h(c.Q) \Rightarrow H(c).Q' \in h(c.Q')$. If the message is flooded to c in the abstract model, then a matching message is flooded to $H(c)$ in the concrete model. Thus, the matching of their queues is preserved after the flooding.

   If $src \in ST$, then we know that in the concrete model, the concrete source, denoted as src_C, that flooded the message to $H(r)$ had to be an invisible router (because there is no direct link between any two visible routers in a sub-topology). This implies that $H(c) \neq src_C$. Therefore, according to the definition of the FD_C function: $H(c) \in FD_C(H(r), src_C)$. Thus, the abstract flooding to c matches the concrete flooding to $H(c)$, and we can infer that $H(c).Q \in h(c.Q) \Rightarrow H(c).Q' \in h(c.Q')$.

   (b) If $DR(sn) = r$ then $C_{sn} = \{c \in sn \,|\, c \neq r \wedge src \neq c\} \subseteq FD_A(r, src)$. According to the matching with the concrete topology, $DR(H(sn)) = H(r)$.

For each $c \in C_{sn}$ such that $c \in SR$, we know that $H(c) \in H(sn)$. Also, we know hat $c \neq r \Rightarrow H(c) \neq H(r)$. In addition, since $c \neq src$, it implies that the concrete source denoted as src_C is necessarily not $H(c)$ as explained in the previous case (a). Therefore, $H(c) \in FD_C(H(r), src_c)$. If $src \in SR$ then $src_c = H(src)$ and if $src \in ST$ then $src_c \in H(src)$ such that src_c is a neighbor of $H(r)$. This implies that for each $c \in C_{sn}$ such that $c \in SR$, $H(c).Q \in h(c.Q) \Rightarrow H(c).Q' \in h(c.Q')$.

For each $c \in C_{sn}$ such that $c \in AR$, we know that $H(c) \subseteq H(sn)$. Also, we know that $c \neq r \Rightarrow \forall r' \in H(c) : [r' \neq H(r)]$. In addition, since $c \neq src$, it implies that $\forall r' \in H(c) : [r' \neq src_C]$. Therefore, $\forall r' \in H(c) : [r' \in FD_C(H(r), src_c)]$. This implies that for each $c \in C_{sn}$ such that $c \in AR$, $\forall r' \in H(c) : (r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))$. If $DR(sn) \neq r$ and $src \neq DR(sn)$, then $DR(sn) \in FD_A(r, src)$. According to the matching with the concrete topology, we can infer that $DR(H(sn)) \neq H(r)$ and that $DR(H(sn)) \neq src_C$. Therefore, $DR(H(sn)) \in FD_C(H(r), src_c)$. This implies that for $c = DR(sn)$, $H(c).Q \in h(c.Q) \Rightarrow H(c).Q' \in h(c.Q')$.

(c) If $\exists st \in ST : (r \in st)$ and $src \neq st$, then $st \in FD_A(r, src)$. This case is irrelevant to the lemma, because it only refers to a flooding destination in ST. However, the lemma does not refer to such flooding destinations.

2. Let $r \in R, src \in R$ such that r receives an LSA from src and floods it to $FD_C(r, src)$. We will prove the claim for any possible flooding destinations, according to the definition of FD_C:

(a) For each $(r, r') \in E$, if $src \neq r'$ then $r' \in FD_C(r, src)$. If $\exists c \notin ST : (r' \in H(c))$, then either $c \in SR$ or $c \in AR$. If $c \in AR$, then it would contradict the fact that there is no matching edge in the abstract topology. This implies that $c \in SR$. Therefore, there is a matching edge $(c, c') \in E_A$ such that $r' = H(c), r = H(c')$. Since $src \neq r'$, we can conclude that : $src_A \neq c$ (the abstract source is denoted as src_A), and therefore $c \in FD_A(c', src_A)$. This implies that $(r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))$.

(b) For each $s \in S$ such that $r \in s$:
If $DR(s) = r$, then $C_s = \{r' \in s \,|\, r' \neq r \wedge src \neq r'\} \subseteq FD_C(r, src)$. If $\exists c \notin ST : (r' \in H(c))$, then there exists $sn \in SN$ such that $H(sn) = s$. For each $r' \in C_s$ there exists $c \in SR \cup AR$ such that $r' \in H(c)$. Also, there exists $c' \in SR$ such that $r = H(c')$ and $DR(sn) = c'$.

If $c \in SR$, then $r' \neq r$ implies that $c \neq c'$. In addition, $src \neq r'$ implies that $src_A \neq c$. This is because the flooded messages are matching, and thus their sources are matching. Therefore, $c \in FD_A(c', src_A)$. This implies that $(r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))$.

If $c \in AR$ then $r' \neq r$ implies that $c \neq c'$ (because r is necessarily mapped by a concrete router which is a designated router). Also, $src_A \neq c'$ because abstract router cannot be a source in the abstract model. Therefore, $c \in FD_A(c', src_A)$, which implies $(r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))$.
If $DR(s) \neq r$ and $src \neq DR(s)$ then $DR(s) \in FD_C(r, src)$. If $\exists c \notin ST : (DR(s) \in H(c))$, then there exists $sn \in SN$ such that $H(sn) = s$. Therefore, there exists $c' \in SR$ such that $H(c') = r$, and thus $DR(s) \neq r \Rightarrow DR(sn) \neq c'$. Also, $src \neq DR(s)$ implies that $DR(sn) \neq src_A$. Therefore, $DR(sn) \in FD_A(c', src_A)$, which implies that for $r' = DR(s)$ and $c = DR(sn)$:$(r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))$.

□

Matching Fight Back Destinations

Lemma 2.9. Let T_A be an abstract topology, and T_C a matching concrete topology, such that H is their matching relation. The following matching between flooding destinations exists:

1. $\forall r \in SR: \ c \in FBD_A(r) \wedge c \notin ST \Rightarrow$
   $[\forall r' \in H(c) : (r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q'))]$

2. $\forall r \in R: \ \forall r' \in FBD_C(r) :$
   $[(\exists c \notin ST : (r' \in H(c))) \Rightarrow ((r'.Q \in h(c.Q) \Rightarrow r'.Q' \in h(c.Q')))]$

The proof of this lemma is very similar to the proof of Lemma 2.8.

2.6.2 Correctness Proof

Below we prove Theorem 2.6.

Proof. Let T_A be an abstract topology, T_C a matching concrete topology, and H their matching relation. We will prove for each possible abstract transition the following:

1. There exists a translation to a sequence of concrete transitions, starting from a concrete state that matches the initial state of the abstract transition.

2. The final concrete state of the translation matches the final abstract state of the original abstract transition.

3. At the final concrete state of the translation, all queues of concrete routers which are represented by a sub-topology in the abstract topology are empty, given that they were empty at the first state of the transition sequence.

Abstract Transitions of a Singleton Router

First, we will refer to the possible abstract transitions that can be taken by a singleton router. Let $\sigma_1 \xrightarrow{sr} \sigma_2$ be an abstract transition taken by $sr \in SR$. Let ${\sigma_1}'$ be a concrete state that satisfies: ${\sigma_1}' \in h(\sigma_1)$. Let $sr.Q_1 =< m_1, m_2, ..., m_n >$ be the queue of sr in state σ_1 for some $n \geq 1$. Let $H(sr).Q_1 =< {m_1}', m_2', ..., {m_p}' >$ be the queue of $H(sr)$ in state ${\sigma_1}'$ for some $p \geq 1$. We will refer to the following cases, when each requires a different translation to the corresponding concrete run:

- If $\neg \exists st \in ST : (sr \in st)$, then the abstract transition represents a single concrete transition, because a sub-topology is not involved in the potential flooding. In addition, the abstract message must be matched with a single concrete message, since the source of the message cannot be a sub-topology (since sr is not in a sub-topology). Therefore, the translation to the concrete transition is straight-forward (whether the abstract transition was ignoring, flooding or sending a fight back). The matching proof is also simple in this case, and is relying on the lemmas from Section 2.6.1. In addition, for any sub-topology, the queues of the hidden routers remain empty after such a transition, because this transition does not involve any sub-topology.

- If $\exists st \in ST : (sr \in st)$, then the corresponding concrete run may involve several concrete transitions. Let ${m_1}', ..., {m_t}'$ be the matched sequence of messages for some $1 \leq t \leq p$ (if $m_1.src \notin ST$ then $t = 1$). We will refer to the possible actions performed in the abstract transition, and will show their corresponding translations.

  1. If $m_1.seq \leq sr.DB[m_1.orig].seq$, then sr removes m_1 from its queue and ignores it. The translation to the sequence of concrete transitions in that case is: ${\sigma_1}' \xrightarrow{H(sr)} {\sigma_2}' \xrightarrow{H(sr)} \ldots \xrightarrow{H(sr)} {\sigma_{t+1}}'$. Due to the matching of the initial states (${\sigma_1}' \in h(\sigma_1)$), the concrete router $H(sr)$ will ignore all these messages that are matched with the abstract message m_1. The matching proof in this case is also trivial (removing matching parts from matching abstract and concrete queues results in matching queues at the final states). In addition, since this abstract transition does not involve any flooding to a sub-topology, it is also implied that for any sub-topology the queues of the hidden routers remain empty after such a transition.
  2. If $m_1.seq > sr.DB[m_1.orig].seq \wedge m_1.orig \neq sr$, then sr removes m_1 from its queue, installs it in its database, and floods it. However, we need to refer to two possibilities in this case.
     (a) If $m_1.src \in ST$,then the translation will include concrete transitions taken by $H(sr)$, such that at the first one it will flood the message, and there might be more transitions of ignoring, if the abstract message is

matched with a sequence of concrete messages. However, since $H(sr)$ may flood the message also to concrete routers that are invisible in the abstract topology and represented by the sub-topology, the translation will also include concrete transitions taken by these routers. It is guaranteed that these routers will ignore this message flooded by $H(sr)$, since $m_1.src \in ST$ and due to the restriction in our abstract model that src in a message sent by the attacker is a singleton router. As a result, we can infer that the sub-topology has already installed m_1 in its database before flooding it to sr. Thus, it is assured that if $H(sr)$ floods this message back to any of the invisible routers represented by the sub-topology, they will ignore it. Thus, at the end of the translation it is assured that the queues of concrete routers represented by the sub-topology are empty. The concrete translation will be composed of three parts at most:

i. The first transition is taken by $H(sr)$, in which it updates its database and floods $m_1{}'$.

ii. If there are more "copies" of this message in its queue (of $H(sr)$), i.e.: $t > 1$, then the next t transitions will be taken by $H(sr)$, in which it will ignore the messages $m_2{}', ..., m_t{}'$.

iii. Let $\{r_1{}', ..., r_i{}'\} \subseteq \bigcup_{s \in H(m_1.src)} s$ be the set of invisible concrete routers to which $H(sr)$ floods $m_1{}'$, for some $i \geq 0$. If this set is not empty ($i > 0$), then the next i transitions will be taken by each of the routers in this set, and in each transition the router $r_j{}'$ will ignore the message flooded by $H(sr)$.

To prove the matching of the final abstract and concrete state, we shall first present these final states. In the abstract transition, we can infer that the final abstract state is σ_2 such that:

i. $sr.Q_2 =< m_2, ...m_n >$

ii. $sr.DB_2[m_1.orig] = (m_1.seq, m_1.isFake)$

iii. $\forall c \in FD_A(sr, m_1.src) : \left(c.Q_2 = c.Q_1 \centerdot \left\{m_{1[sr \to c]}\right\}\right)$

In the concrete model, due to the sequence of transitions we have shown, we can infer (assuming that the index of the final concrete state is k) that the final concrete state is $\sigma_k{}'$ such that:

i. $H(sr).Q_k =< m_{t+1}{}', ...m_p{}' >$

ii. $H(sr).DB_k[m_1{}'.orig] = (m_1{}'.seq, m_1{}'.isFake)$

iii. $\forall c \in FD_C(H(sr), m_1{}'.src) : \left(c.Q_k = c.Q_1 \centerdot \left\{m_1{}'_{[H(sr) \to c]}\right\}\right)$

Therefore, the matching between the final states is similar to the previous cases: the databases of matching routers were updated by matching messages, the queues remain matching due to the removal of matching parts, and the queues of neighbors are matching bases on Lemma 2.9.

Note that the state of invisible routers was not changed at the end of these transitions.

(b) If $m_1.src \notin ST$, then the translation should include the concrete run of flooding the message within the sub-topology. (since $FD_A(sr, m_1.src) \cap ST \neq \emptyset$). Let $st \in ST$ be the sub-topology for which $r \in st$. i.e., $FD_A(sr, m_1.src) \cap ST = \{st\}$. The corresponding concrete run will be composed of three parts at most:

i. The first transition is taken by $H(sr)$, in which it updates its database and floods $m_1{}'$.

ii. If there are more copies of this message in its queue (of $H(sr)$), i.e.: $t > 1$, then the next t transitions will be taken by $H(sr)$, in which it will ignore the messages $m_2{}', ..., m_t{}'$.

iii. The last part will be composed of transitions taken by invisible routers in $\bigcup_{s \in H(st)} s$, until all their queues become empty. At the end of this sequence, it is assured that the message will be flooded to the queues of the remaining visible routers in the sub-topology (not including sr itself), due to the constraints we imposed on a sub-topology structure.

It is clear how to generate the concrete transitions described at the first two parts. For the third part, it is possible to choose any interleaving for which all invisible routers apply their concrete procedures until all their queues are emptied. It is easy to prove that there exists such an interleaving, based on the constraints mentioned in the sub-topology definition, and in particular - based on the fact that all invisible routers form a strongly connected component in the concrete topology. After all queues of invisible routers are emptied, their databases are installed with the flooded LSA, and the flooded LSA has been inserted to the queues of other visible routers. This can be proved if we assume by negation that there exists an invisible router for which its database was not installed with the flooded LSA, but it has a neighbor that was installed with that LSA (we can assume that there is necessarily such a neighbor, because at the beginning the message was inserted to at lease one of the invisible routers' queue which is an immediate neighbor of sr, the router which flooded the message, and due to the connectivity of the set of invisible routers).

Based on that, we can prove the matching of the final states. We denote by R_S The set of invisible routers within the sub-topology in the concrete topology. Let $\sigma_k{}'$ be the final concrete state after the sequence of transitions described above are taken, starting from $\sigma_1{}'$.

– If $st.DB_1[m_1.orig].seq < m_1.seq$: In that case, the final abstract

state is:
$\sigma_2 = [sr.Q_2 =< m_2, ...m_n >;$
$sr.DB_2[m_1.orig] = (m_1.seq, m_1.isFake)\,;$
$\forall c \in FD_A\,(sr, m_1.src) : \left(c.Q_2 = c.Q_1 \centerdot \left\{m_{1_{[sr \to c]}}\right\}\right)\,;$
$st.DB_2[m_1.orig] = (m_1.seq, m_1.isFake)\,;$
$\forall sr' \neq sr \in st : \left(sr'.Q_2 = sr'.Q_1 \centerdot \left\{m_{1_{[st \to sr']}}\right\}\right)]$
The final concrete state is, based on what we proved before:
$\sigma_k{}' = [H\,(sr)\,.Q_k =< m_{t+1}{}', ...m_p{}' >;$
$H\,(sr)\,.DB_k[m_1{}'.orig] = (m_1{}'.seq, m_1{}'.isFake)\,;$
$\forall c \in FD_C\,(H\,(sr)\,, m_1{}'.src) : \left(c.Q_k = c.Q_1 \centerdot \left\{m_1{}'_{[H(sr) \to c]}\right\}\right)\,;$
$\forall r \in R_S : (r.DB_k\,[m_1{}'.orig] = (m_1{}'.seq, m_1{}'.isFake))\,;$
$\forall sr' \neq sr \in st(\exists t \geq 1[(H(sr').Q_k = H(sr').Q_i \centerdot \{m_1{}'', m_2{}'', .., m_t{}''\}) \wedge$
$\forall i(m_i{}'' \in h(m_{1_{st \to H(sr')}}))]]$
It can be easily proved that these states are matching.

- If $st.LSA\_DB[m_1.orig].seq \geq m_1.seq$, then the matching proof is very similar, and the only difference is that the databases contents of the sub-topology was not changed, and that the message was not flooded to other visible routers in the sub-topology.

3. If $m_1.seq > sr.DB\,[m_1.orig]\,.seq \ \wedge \ m_1.orig = r$, then there is a fight back message which is flooded within a sub-topology. The concrete translation is similar to the regular flooding case, except that in this case the flooded message is a fight back message. Thus, it is flooded to neighbors according to the fight back destinations function instead of the flooding destinations function.

Abstract Transitions of an Abstract Router

We will now refer to the possible abstract transitions that can be taken by an abstract router. Assuming that $r \in AR$ and $\sigma_1{}' \in h\,(\sigma_1)$, we will show the concrete translation and will prove that $\sigma_k{}' \in h\,(\sigma_2)$. Let $r.Q_1 =< m_1, m_2, ..., m_n >$ be the queue of r in state σ_1 for some $n \geq 1$. Let $r'.Q_1 =< m_1{}', m_2{}', ..., m_n{}' >$ be the queue of each $r' \in H\,(r)$ in state $\sigma_1{}'$. Let $H\,(r) = \{r_1{}', ..., r_{k-1}{}'\}$ for some $k \geq 2$. Concrete translation: $\sigma_1{}' \xrightarrow{r_1{}'} \sigma_2{}' \xrightarrow{r_2{}'} ... \xrightarrow{r_{k-1}{}'} \sigma_k{}'$. The concrete translation is a sequence of concrete transitions, such that each transition is taken by a different router represented by the abstract router in the abstract topology (such that all router represented by the abstract router have taken exactly one transition), in some order.
The matching proof of the final states is straight forward. In the final states the queues are matching, since in the abstract transition the first abstract message was removed from the queue, and in the concrete run the matching message was removed from the queues of all routers matched with the abstract router. If the abstract router installed the LSA in its databases, then all concrete routers in the matched concrete run has

installed the LSA, thus the databases are also matching at the end of the transitions. In addition, the queues of invisible routers remain empty after this transition because no sub-topology is involved in this macro-step.

Abstract Transitions of an Attacker

Finally, we will refer to possible abstract transitions taken by an attacker. We refer to translation of an abstract transition in which a message is sent from the attacker to its destination (the message arrives to the destination within one abstract step). We define this macro-step as enabled when the queues of routers within the routing path are empty. Therefore, it can be directly translated into concrete transition taken by concrete routers in the concrete routing path. Other possible actions of the attacker can be translated similarly to the translation shown for a concrete router. Let $\sigma_1 \xrightarrow{r} \sigma_2$ be the abstract transition, and let ${\sigma_1}' \xrightarrow{r_1} {\sigma_2}' \xrightarrow{r_2} ... \xrightarrow{r_{k-1}} {\sigma_k}'$ be the corresponding translation to concrete transitions. Let m be the message sent in the abstract transition. Therefore, the change between σ_1 and σ_2 is: $(m.dest).Q_2 = (m.dest).Q_1 \bullet \{m\}$. In the concrete topology, let $r_i = nextHop(r_{i-1}, m.dest)$ for each $i > 1$, and let r_1 be a neighbor of the attacker r. Therefore, in each concrete transition, ${\sigma_i}' \xrightarrow{r_i} {\sigma_{i+1}}'$, the message $m' \in h(m)$ is removed from the queue of r_{i-1} and added to the queue of r_i. In the final state ${\sigma_k}'$, it is added to the queue of $m'.dest$: $(m'.dest).Q_k = (m'.dest).Q_1 \bullet \{m'\}$. There are no other changes in ${\sigma_k}'$ relative to ${\sigma_1}'$, and therefore ${\sigma_k}' \in h(\sigma_2)$. (Since $m' \in h(m)$, it implies that the queues of $m.dest$ and $m'.dest$ are matching). Thus, the final states are matching. The queues of invisible routers remain empty, since the destination is necessarily a singleton router in the abstract topology.

□

2.7 Extension of the Concrete Model

2.7.1 Extended OSPF Basics

Below we give an extended overview of the OSPF protocol based on [50]. OSPF is a link state routing protocol: each router advertises an LSA containing the links to neighboring networks and routers and their associated costs. Each LSA is flooded throughout the Autonomous System (AS). Routers construct a complete view of the AS topology by compiling all the LSAs they receive into a single database. From this global view routers compute their routing tables. Each router is identified by a parameter called router ID.

A local network having two or more routers directly attached to it is called a transit network. A router connected to a transit network advertises a link to the network rather than to the neighboring routers. In addition, one of the neighboring routers is chosen to act as a designated router. This router advertises an LSA on behalf of the local network, in addition to its own LSA, advertising links back from the network to all the routers attached to the network.

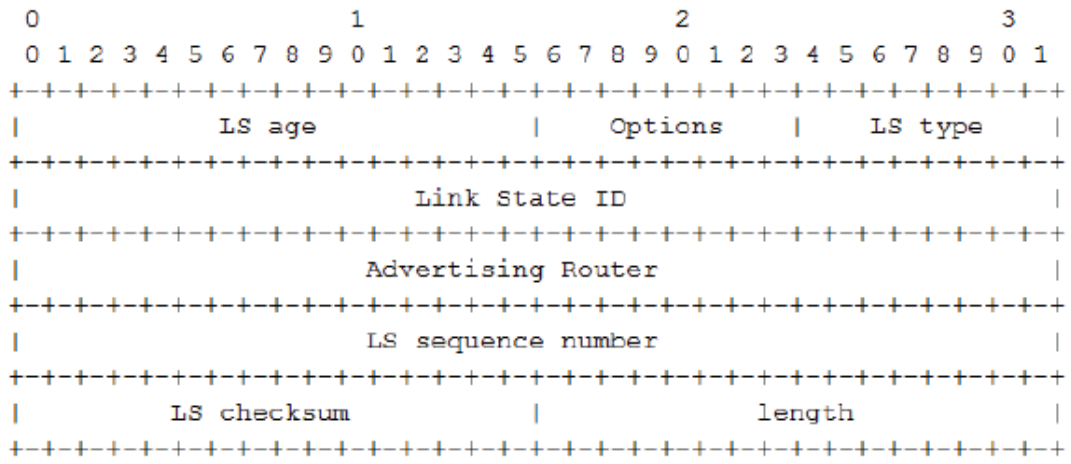

Figure 2.9: The structure of an LSA header

Each LSA is advertised periodically every 30 minutes, by default. An LSA includes a Sequence Number field, which is incremented for every new instance. A fresh LSA instance with a higher sequence number will always take precedence over an older instance with a lower sequence number. In addition, an LSA includes an Age field indicating the elapsed time since the LSA's origination. When it reaches 1 hour, the LSA instance is removed from the LSA database.

The sequence number field is 32 bit long. Once the sequence number reaches its maximum value it needs to wrap to zero in the next LSA instance. To do that the LSA instance with the maximum sequence number (MaxSeqNum) is first flushed by advertising another instance having the maximum sequence number and an age field of 1 hour. This instance replaces the current LSA instance but then immediately removed from the LSDB due to its age. Therefore, no instance of that LSA is kept in the LSDB. At this stage a fresh instance of the LSA with a sequence number of zero will be advertised.

For scalability reasons, OSPF allows an AS to be partitioned into areas. Flooding is confined to a single area while routing information is disseminated between different areas through a special backbone area. For simplicity of the presentation we shall consider only the case where an AS is composed of a single area.

The OSPF defines four basic types of LSA: Router-LSA, Network-LSA, Summary-LSA and AS-external-LSA. The most common type is Router-LSA, which is used to advertise the links of a given router. For brevity, throughout this section we refer to a Router-LSA as simply LSA.

All LSAs begin with a common 20 byte header. Figure 2.9 depicts the LSA header. A description of the LSA header fields follows.

- LS age – The time in seconds since the LSA was originated.

- Options – Optional capabilities supported by the described portion of the routing domain.
- LS type – The type of LSA.
- Link State ID – Identifies the part of the AS that is being described by the LSA.
- Advertising Router – Identifies the router that originated the LSA.
- LS sequence number – The sequence number of the LSA.
- LS checksum – The checksum of the complete contents of the LSA.
- Length – The length in bytes of the LSA.

The standard specifies (Sec. 12.1 of [50]) that an LSA is identified by the three fields: LS type, Link State ID, and Advertising Router.

2.7.2 Extension Goals

The goal of our extension is to extend our systematic analysis by model checking of the OSPF protocol. It is motivated by a new OSPF vulnerability that was found during a manual analysis of the protocol ([52]). We will refer to it as the *new vulnerability*. We extend the modeled functionalities of the fight-back mechanism and the routing table calculation.

Below we describe the additional vulnerabilities that should be captured by our extended modeling of the fight-back mechanism and were not captured in our original model from Section 2.4.

- **The periodic injection attack** : Jones et al. [37] introduced the first known attack that evades the fight-back mechanism – called 'periodic injection'. The attack exploits a vulnerability in which the fight-back mechanism is triggered only *after* the router has flooded the false LSA (which was advertised on its behalf) to its neighbors. Since the OSPF specification prevents a router from originating two instances of its own LSA within less than 5 seconds, the victim router must wait for that time period before it is able to send the fight-back LSA. Hence, if the attacker periodically advertises the false LSA at least once every 5 seconds, it will ultimately prevent the victim from issuing any fight-back LSAs.
- **The new vulnerability:** The new vulnerability [52] was found during a manual analysis of the OSPF standard. The Link State ID field of a Router-LSA equals the ID of the router whose links are described by the LSA. Moreover, since a Router-LSA is originated by the router described by that LSA, the Advertising Router field of the LSA will also equal the router's ID. Therefore, we note that in every Router-LSA, the above two fields – Link State ID and Advertising Router

– are expected to have the same exact value. The first part of the vulnerability arises because the standard neglects to specify a sanity check to verify this equality on LSA reception. Thus, a Router-LSA is considered valid even if it has different values in these two fields. In the following we explain how an attacker can benefit from this.

According to Section 13.4 of [50], a router is triggered to send a fight-back LSA only if it receives a false LSA in which

> "the Advertising Router is equal to the router's own Router ID."

This means that no fight-back shall be triggered by the victim router as long as the Advertising Router field of a received LSA is *not* equal to the victim router's ID. This is true even if the Link State ID field of that LSA equals the victim router's ID. Namely, no fight-back is triggered even if the false LSA claims to describe the links of the victim router itself.

An attacker can exploit the above vulnerability as follows. Let us assume the attacker wishes to advertise a false LSA on behalf of some victim router, R_v. The attacker will advertise on its local links the false LSA while having a header in which:

- Link State ID = ID of R_v,
- Advertising Router $\neq$ ID of R_v.

The false LSA shall be flooded throughout the AS as usual and will eventually be received by all routers, including the victim router R_v. The OSPF standard guarantees that such false LSA will not trigger fight-back by R_v. Consequently, all routers within the AS – including R_v – install the false LSA in their LSA databases. Thus, the attacker achieves complete poisoning of the LSA databases of all routers within the AS.

However, does this LSA database poisoning indeed poison the routing tables as well? Recall that an LSA is identified by the combination of the following three fields:

1. LS type,
2. Advertising Router,
3. and Link State ID.

This means that the false LSA advertised by the attacker has a different identifier than that of the valid LSA of the victim router. They differ in the Advertising Router field's value. The value of a valid LSA equals that of the victim router's ID while the value of the false LSA does not. Therefore, the false LSA should *not* replace the valid LSA in the LSA databases. Consequently, following the attack, both LSAs are expected to reside in the LSA databases of all routers.

Now let us turn to the second part of the vulnerability. The OSPF standard (Section 16.1) specifies that during the routing table calculation phase LSAs are looked up in the LSA database

> "based on the Vertex ID".

The vertex ID refers in the standard to the LSA's Link State ID field. This means that while a router calculates its routing table, it identifies LSAs on the basis of their Link State ID field only and not on the basis of the full LSA identifier, which also includes the Advertising Router and LS type fields. The fact that the lookup is based on the partial identifier is explicitly reiterated in footnote 14 of the standard, which also explains the motivation behind this:

> "[14] There is one instance where a lookup must be done based on partial information. This is during the routing table calculation, when a network-LSA must be found based solely on its Link State ID. The lookup in this case is still well defined, since no two network-LSAs can have the same Link State ID."

Namely, the standard motivates this lookup by saying that during the routing table calculation the full identifier of Network-LSAs is not known. However, it seems that the standard unduly generalized this partial information lookup to Router-LSAs as well, despite the full identifier of a Router-LSA being known during the routing calculation phase[1].

The above gives rise to the following question: which LSA will be fetched from the LSA database during the routing table calculation: the valid LSA of the victim router or the false LSA advertised by the attacker? Remember, both LSAs reside side by side in the LSA database of each router in the AS. Both LSAs have the exact same value in their Link State ID field – the ID of the victim router. Unfortunately, the standard fails to answer this question; it does not consider the case where two different LSAs have the same Link State ID field. Because this question is not addressed in the standard, the answer is thus dependent on the implementation. An OSPF implementation that fetches the valid LSA during the routing table calculation is oblivious to the attack. However, an OSPF implementation that fetches the false LSA is completely vulnerable to the attack.

2.7.3 Extended OSPF Modeling

We extend the formal model of OSPF described in Section 2.4.1. In this extension, the fight-back mechanism and routing table calculation are modeled in more detail. Such

[1]Note that for Router-LSAs, the Advertising Router field must be equal to the Link State ID field and hence must also be known.

Listing 2.1: The model's main loop

```
main()
{
  loop {
    for every router R
    {
      Router(R); //execute R's procedure

      if (R.flooding_timer > 0)
         R.flooding_timer--;

      for every LSA in R.LSA_DB
      {
        LSA.counter++;

        //violation of this assertion
        //indicates a successful attack
        assert(
          LSA.mark == false ||
          LSA.originated_by_attacker == false ||
          LSA.counter < MIN_COUNTER
        );
      }
    }

    Attacker(); //execute attacker's procedure

    num_cycles++;
  }
}
```

extended modeling allows us to better search for attacks that may exploit intricate details of the fight-back mechanism.

The model is composed of a fixed network topology that contains legitimate OSPF routers and a single malicious router. The modeled functionality includes the LSA message structure, the LSA flooding procedure, the fight-back mechanism, and the routing table calculation. In the following we include a pseudo-code that gives a general overview of the model we used. The next sections explain the pseudo-code.

Listing 2.2: The attacker's procedure

```
Attacker()
{
  LSA = generate_arbitrary_LSA();
  flood(LSA);
}
```

The Main Function

The main function of the model is a loop, where in each loop iteration any of the routers (including the attacker) can run their procedure once. Each loop iteration is considered a cycle. The flooding timers of each router are decremented in every cycle (An explanation of the flooding timers is given below as part of the router model description).

We *mark* LSAs which are deemed to affect the routing table. We count the number of cycles each LSA resides in an LSA database. The last portion of the main loop is the assertion of the model's specification. A detailed explanation of the assertion is given below as part of the specification description.

The Attacker Model

The attacker generates an LSA with arbitrary content. This models an attacker with complete control over the false LSAs it sends out. Note that the identity of the victim router on whose behalf the false LSA is sent is not predefined. It can be any one of the legitimate routers. The model checker will cover every possible false LSA during its search for an attack. Note that the attack may be composed of a sequence of false LSAs sent by the attacker.

The Router Model

In the initial state, each router R has in its LSA database LSAs originated by R, and for each link described in these LSAs, there is also an LSA in the database with a link back to R. Namely, there is a link from one of R's neighbors back to R. We need these links in our model because, as per the OSPF standard, a link can be considered in the routing table calculation only if there are links in the reverse direction.

Each router has a flooding timer that determines how many cycles it should wait before originating an LSA. When the timer is set to 0 the router can originate its own LSA without any delay. After the router floods its own LSA, its timer is set to MinLSInterval, a predefined constant that determines by how many cycles to delay the next LSA origination. The timer is decreased by 1 in each cycle for which its value is greater than 0.

When the router R receives an LSA, it first checks that it is valid (i.e., properly formatted and received from a valid neighbor). If so, it checks whether such an LSA exists in its LSA database. If it exists and is considered newer than the database copy or if it does not exist in the database, then R floods this LSA. Afterward it checks whether the LSA is self-originated. A self-originated LSA is a one whose Advertising Router field equals R's ID. If this conditions holds, a fight-back is triggered, in which case the LSA will not be installed in the database. The fight-back itself is delayed due to the flooding of the false LSA prior to discovering that it was self-originated.

Otherwise, R adds this LSA to its LSA database and triggers the routing table calculation. The calculation logic must be modeled since not every LSA installed in the

LSA database of a router eventually affects that router's routing table.

The Routing Table Calculation

To keep the model compact, we do not model the entire routing table calculation logic. We are not interested in modeling the calculated routing table, but only the functionality that determines which LSAs will ultimately affect it. We start by considering R as reachable (it is the root of its shortest path's tree). Then, a lookup for R's LSA in its database is performed using the Link State ID field. Note that the fetched LSA may be either a valid LSA of R or a malicious LSA crafted by the attacker in a previous stage. For each advertised link in the fetched LSA, let W denote the neighbor on the other side of that link. If an LSA for W is found and contains a link back to R, then R's LSA is marked and W is considered reachable in the shortest path tree. This loop is executed for every reachable router.

According to the OSPF standard, a Summary-LSA installed in the database affects the routing table if and only if its Advertising Router field contains an ID of a reachable router. Therefore, our model marks such LSAs only if this condition holds.

Specification

We say that an attack is successful if there is an LSA installed in a router for which all the following three conditions hold:

1. The LSA was originated by the attacker.

2. The LSA is marked, i.e., it affects the routing table calculation.

3. The LSA's counter exceeds MIN_COUNTER (where MIN_COUNTER is some large value).

The last condition ensures that the attack is persistent, i.e., it is not reverted by the fight-back mechanism. The MIN_COUNTER is a predefined constant that determines the persistency of the attack. In our model, when an LSA that was originated by the attacker replaces an older instance of that LSA which was also originated by the attacker, then the counter value of the older LSA is copied to that of the newer one. This allows our model to find attacks in which the false LSA's instances are constantly changed by the attacker. These are also considered persistent attacks since the attacker has an uninterrupted effect on the routing table.

This specification is coded as an assertion in the model's main loop. If the model checker finds a state of the model which violates this assertion, then it must be the case that the attacker managed to persistently affect the routing table calculation while evading the fight-back mechanism.

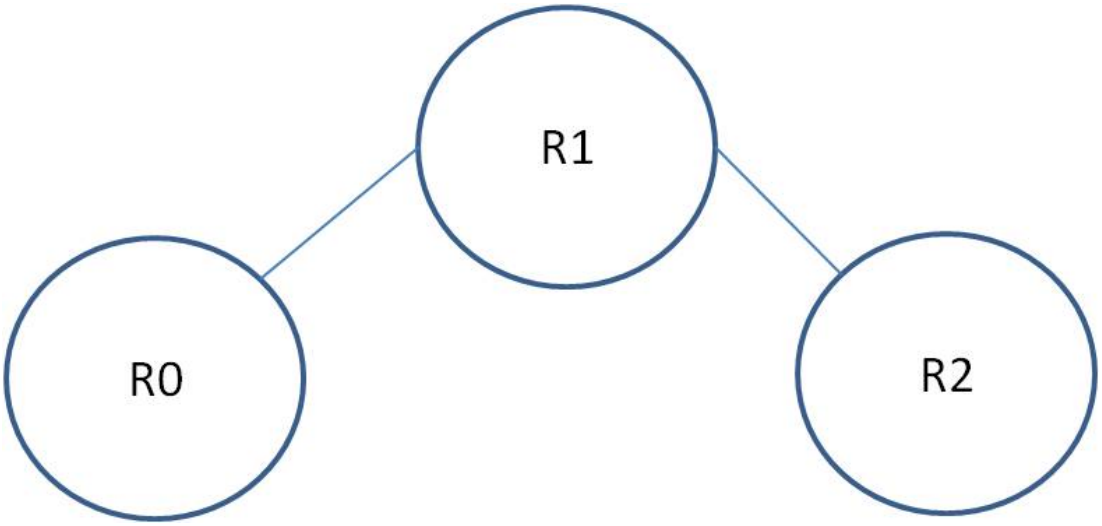

Figure 2.10: The topology used with the extended model

2.7.4 Results

We implemented the above model in C and used the CBMC tool [23] to perform the model checking. We used a simple fixed network topology with three routers as depicted in Figure 2.10. The routers $R1, R2$ are the legitimate routers and $R0$ is the attacker. The links between the routers are of type point-to-point.

Attacks Found in Our Model

By running the model checker with our OSPF model implementation, we recreated the following attacks:

1. The disguised LSA attack was found, where the attacker sent two LSAs – a trigger LSA and a disguised LSA. This attack was also found on the original model as described in the third attack of Section 2.4.5.

2. The periodic injection attack – where the attacker periodically sent LSAs on behalf of some router at a higher rate than MinLSInterval. Each LSA sent by the attacker had a sequence number greater than the previous one. Therefore, the delayed fight-back was also replaced and delayed for each new instance. Consequently, a sequence of malicious LSAs were present in the victim's LSA database more than MIN_COUNTER cycles, and the fight-back was never sent due to reset of the flooding timer with every received false LSA.

3. An attack that exploits the new vulnerability, originally found in a manual analysis as described in [52]. The output contained an execution path that described the above attack, where the LSA sent by the attacker was a Router-LSA having a Link

State ID equal to the victim's ID but having a different value for the Advertising Router.

Correcting the Vulnerabilities in the Model

The CBMC model checker halts after it finds the first counterexample for the assertion in our model. Therefore, in order to find new vulnerabilities, we had to correct the ones found above.

1. In order to avoid receiving the disguised LSA attack, we changed the condition that triggers the fight-back mechanism, such that not only newer self-originated LSA instances will trigger it, but also instances that are considered identical to the LSA in the database, and were actually originated by the attacker. In that case the disguised LSA will also trigger a fight-back.

2. In order to avoid receiving the periodic injection attack, we set the MinLSInterval to 0. In that case the fight-back is not delayed thus the attack cannot be persistent.

3. In order to avoid receiving the new attack, we added an assumption that during LSA generation by the attacker, the Link State ID field must equal the Advertising Router field. This condition was actually used in the patches released by most vendors who were found vulnerable to the attack.

We reran CBMC on the corrected model and no new attacks were found. This indicates that, once the above conditions are enforced, an attacker will not be able to craft a malicious LSA such that it will not trigger the fight-back mechanism while still affecting the routing table calculation.

Nonetheless, we note that model checking is only as good as the model itself. The above result cannot be considered "proof" that no further weaknesses exist in OSPF in general since we omitted details of OSPF operation that are not relevant to the fight-back mechanism. Moreover, since we used a bounded model checker, namely one that does not explore all the states of the model, the above result cannot be considered "proof" that no other weaknesses in the fight-back mechanism exist. Nonetheless, we believe this result provides a strong indication that there are no additional weaknesses in this mechanism.

2.8 Conclusion

In this chapter we developed and implemented a formal analysis of the OSPF protocol using model checking to allow finding built-in vulnerabilities in the standard of the protocol. We focused on persistent poisoning attacks and defined a corresponding specification. We found *general* attacks which are applicable to families of networks and demonstrated *security vulnerabilities* in the OSPF protocol. We developed a novel

technique for *parameterized networks* which is suitable for finding a counterexample (in our case an attack) on each member of the family.

In the last part we extended our modeling of the fight-back mechanism. Using model checking, we sought other persistent attacks that aim to evade the fight-back mechanism. The results give an indication that once the known weaknesses are mitigated no other weakness in the fight-back mechanism exists.

Listing 2.3: A valid router procedure

```
Router(R)
{
  //fetch the LSA at the top of the
  //router's incoming queue
  LSA = R.in_queue.dequeue()

  if (!R.valid(LSA))
    continue;

  if(LSA is newer than one in R.LSA_DB)
  {
    R.flood(LSA);

    if (R.is_self_originated(LSA))
    {
      R.flooding_timer = minLSInterval;
      FB_LSA = R.generate_fight_back();

      //replace existing queued fight-back
      //with the new one.
      if (R.flooding_queue.size() > 0)
        R.flooding_queue.dequeue()
      R.flooding_queue.enqueue(FB_LSA);
    }

    else
    {
      R.LSA_DB.update(LSA) ;
      R.calcRT_flag = True ;
    }
  }

  if(R.flooding_timer == 0)
  {
    if (R.flooding_queue.size() > 0)
    {
      LSA = R.flooding_queue.dequeue()
      R.flood(LSA);
    }
  }

  R.RoutingTableCalculation();
}
```

Listing 2.4: The routing table calculation procedure

```
RoutingTableCalculation()
{
  if (!calcRT_flag)
    continue;

  calcRT_flag = False;
  //R is the current router
  R.reachable = True;

  for every V where V.reachable == True
  {
    //lookup in an arbitrary order within LSA database for V's LSA
    V_LSA = DB_lookup(V) ;

    for every link L in V_LSA
    {
      W = L.neighbor;
      W_LSA = DB_lookup(W);
      if (W_LSA has link back to V)
      {
        //V_LSA affects the routing table
        V_LSA.mark = True;
        W.reachable = True;
      }
    }
  }

  for (every Summary-LSA in LSA_DB)
  {
    if (LSA.AdvertisingRouter.reachable == True)
    //LSA affects the routing table
    LSA.mark = True;
  }
}
```

Chapter 3

Analyzing BGP Traffic Attraction Attacks Using Model Checking

3.1 Preliminaries

In this chapter we combine static examination and model checking to examine fragments of the Internet and either identify possible attacks on their routing protocol or prove that specific attacks are not possible.

The Internet is composed of Autonomous Systems (ASes). Each AS is administered by a single entity (such as an Internet service provider, or an enterprise) and it may include dozens to many thousands of networks and routers. Inter-domain routing determines through which ASes packets will traverse. Routing on this level is handled throughout the Internet by a single routing protocol called the *Border Gateway Protocol* [54] (BGP).

It is well known that the Internet is vulnerable to traffic attacks [35, 15]. In such attacks malicious Autonomous Systems manipulate BGP routing advertisements in order to attract traffic to, or through, their AS networks. Attracting extra traffic enables the AS to increase revenue from customers, drop, tamper, or snoop on the packets. In the recent past, there have been frequent occurrences of traffic attraction attacks on the Internet [67, 65, 63, 64, 43, 44]. Some of those attacks allowed oppressive governments to block their citizens from accessing certain websites. In other attacks the perpetrators eavesdropped or altered the communications of others, while in different attacks spammers sent millions of emails from IP addresses they do not own. In one type of attack scenario the traffic is diverted through the attacker's AS network and then forwarded to its real destination, which allows the attacker to become a "man-in-the-middle" between the source of the traffic and its final destination. Such attacks are called *interception attacks*. In another type of attack scenario, the traffic is not

forwarded to its real destination, which allows the attacker to impersonate the real destination or simply block access to it. Such attacks are called *attraction attacks*. In the sequel, when we refer to any attack of these types we call it a *traffic attack*.

Our goal is to provide insights to where and how BGP traffic attacks are possible. Note that BGP is the sole protocol used throughout the Internet for inter-domain routing, hence its importance. We develop a method that exploits model checking to systematically reveal BGP traffic attacks on the Internet, or prove their absence under certain conditions. Our method is based on powerful reductions and abstractions that allow model checking to explore relatively small fragments of the Internet, yet obtain relevant results. Reductions are essential as the Internet nowadays includes roughly $50,000$ ASes.

In a normal mode of the BGP operation, when no attacker is present, an AS node receives from some of its neighbors their choice of routing path to the destination. When AS A announces a routing update to its neighbor AS B consisting of a target node n and a path π, it means that A announces to B that it is willing to carry packets destined to n from B, and that packets will traverse over the path π. From the announced routing paths, the node chooses its most preferred route (according to business relationship between the entities that administer the ASes, length of path, etc.) and sends it further to some of its neighbors. Its announced path may, in turn, influence the choice of preferred paths of its neighbors. In contrast, an attacker may send its neighbors faulty routing paths whose goal is to convince them and other AS nodes in the Internet to route through the attacker on their way to the destination.

Our static examination investigates the announcements flowing throughout the Internet. The basic idea is that if announcements cannot flow from one part of the Internet to another then nodes in the first part cannot influence the routing decisions of nodes in the second part. Our first reduction is thus based on BGP policies that determine the flow of announcements in the Internet. Given a destination and an attacker, we *statically* identify on the full Internet topology a *self-contained fragment* S that consists of a set of nodes, including the destination and attacker. S is defined so that nodes in S may send announcements inside and outside of S, but nodes outside of S never send announcements to nodes in S. Thus, the routing choices of nodes in S are not influenced by routing choices of the rest of the Internet.

We can now isolate S from the rest of the Internet and apply model checking only to it in order to search for an attack strategy that attracts traffic to the attacker. Since routing decisions in S are made autonomically, an attack strategy found on S will attract the same nodes from S when the full Internet is considered. This result allows to significantly reduce the processing burden on model checking while searching for attacks on the Internet. Similarly, if we show that no attack strategy manages to attract traffic from certain victims in S then the attacker will not manage to attract traffic from those victims in the full Internet as well. Thus, by searching a small fragment we find attacks on the full Internet or show their absence.

The second reduction we suggest is applied within a self-contained fragment S to further reduce it. We *statically* identify nodes in S that for all BGP runs choose the same route to the destination (that does not pass through the attacker), regardless of the attacker's behavior. Such nodes are considered *safe* with respect to the destination and the attacker of S.

The advantage of this reduction is twofold. First, safe nodes can be safely removed from the model, thus easing the burden on model checking. Second, nodes that wish to improve their routing security may decide to route through safe nodes, thus avoiding traffic attacks from this specific attacker. We further elaborate on the latter in Section 3.9.

Our third reduction is based on an abstraction. We can *statically* identify a *routing-preserving* set of nodes that all make the same routing choices. Such a set can be replaced by a single node with similar behavior without changing routing decisions of other nodes in the network.

Note that all three reductions are computed statically by investigating the Internet topology and are therefore easy to compute.

We implemented our method, called BGP-SA, for BGP Security Analysis. We first extracted from the Internet self-contained fragments, which are defined by a destination and an attacker nodes, and applied reductions to them. We chose the attacker and the destination nodes either arbitrarily or in order to reconstruct known recent attacks. In order to apply model checking, we modeled the BGP protocol for each AS node. We also modeled an attacker with predefined capabilities. The BGP model is written in C. We considered several specifications which allow to reveal different types of attacks. We ran IBM's model checking tool ExpliSAT [22] on self-contained, reduced fragments.

We found interception attacks. One of those attacks reconstructs a recent known attack where Syria attracted traffic destined to YouTube [63]. In other cases we showed that some attraction scenarios are impossible under the modeled attacker capabilities. In the latter case, model checking could also reveal additional *safe* nodes.

3.2 Related Work

Ref. [35] discusses the security of BGP and its vulnerability to different attacks. It shows that an attacker may employ non-trivial and non-intuitive attack strategies in order to maximize its gain. This was shown by giving anecdotal evidence (obtained manually) for each attack strategy in specific parts of the Internet. In our work we develop reductions and use model checking to systematically and automatically search for BGP traffic attacks on the Internet.

There are some past works that use formal methods to analyze convergence properties of BGP. [14] uses a static model of BGP path selection and analyzes configurations of BGP policy. [13] uses static and dynamic models to reason about BGP convergence. [55] analyzes convergence of routing policies with an SMT solver. We use a different

modeling to reason about traffic attraction scenarios on the Internet. Our modeling implements runs of the protocol until stabilization, includes an attacker, and is based on the routing policy used by most ASes on the Internet. Our model includes parts of BGP that are most relevant to the analysis of traffic attraction, and is based on the model presented in [35].

A formal analysis dealing with the security of secured variants of BGP was presented in [17] . They suggested a formal security model for path-vector routing protocols, such as BGP. Using their security model they have proven manually by reduction that S-BGP (a more secured variant of BGP) satisfied part of the security requirements of their model, and showed how the protocol could be modified to meet all requirements.

To the best of our knowledge, there are no previous works that employed model checking tools to find attacks and reveal security vulnerabilities automatically on BGP.

3.3 BGP Background

The routers and networks of the Internet are clustered into connected sets. Each such set is called an autonomous system (AS). As of the end of 2014, there are roughly 50,000 autonomous systems on the Internet. An AS is usually administered by a single network operator, such as an ISP (Internet service provider), an enterprise, a university, etc. Each AS has a predefined routing policy determined by the network operator. An autonomous system is assigned a globally unique number, sometimes called an Autonomous System Number (ASN).

Routing of data packets on the Internet works in two levels:

1. Inter-domain routing that determines through which ASes the packets will traverse. This level of routing is handled by a single routing protocol called the Border Gateway Protocol [54] (BGP).

2. Intra-domain routing that determines the path taken by the packets within each AS. This is determined independently in each AS. Each network operator is free to choose any routing protocol to employ within its AS. The most common examples of such routing protocols are OSPF [50], RIP [46], or IS-IS [20].

Note that BGP is the sole protocol used for inter-domain routing. In essence, BGP is the glue that holds the Internet together and which allows to connect between different ASes. The currently used version of BGP is number 4. The protocol's standard is specified by the IETF (Internet Engineering Task Force) standardization body in [54]. The primary function of BGP is to exchange network reachability information between different ASes.

Each AS periodically announces to all its neighboring ASes (i.e., the ASes to which it is directly connected) routing updates. A routing update consists of the identity of a target network and a path that consists of a sequence of ASes that starts from the

advertising AS and leads to the AS in which the target network resides. Note that BGP advertises routing updates pertaining to networks residing within ASes (not to ASes themselves), while the routing path is at the AS level. When AS A advertises a routing update to its neighbor AS B consisting of a target network n and a path π, it means that A announces to B that it is willing to carry packets destined to n from B, and that packets will traverse over the path π. This routing information will then be propagated by AS B to its neighbors, after prepending itself to π. The propagation of routing information by one AS to all its neighbors is a matter of a policy determined by that AS. We shall elaborate on this in the following.

An AS can also send route withdrawals. A withdrawal retracts an earlier routing update. When an AS A receives a withdrawal message from its neighbor B, A removes the route that was previously advertised by B from its routing table and propagates the withdrawal to its neighbors.

Every AS stores the routing updates learned from its neighboring ASes in a data structure called Adj-RIBs-In. If several routes were advertised for the same target network by different neighboring ASes, then the AS must choose its most preferable one. Once a route is chosen all packets destined to that target network will be routed via the neighboring AS that announced the chosen route. The chosen routes for all target networks on the Internet are stored in a data structure called Loc-RIB. Choosing the most preferable route is a matter of policy specific to each AS. We will refer to it as the *preference policy*.

As noted above, each AS propagates to its neighbors the routing updates it receives. Only routes within the Loc-RIB may be propagated. Namely, an AS can only propagate a route it has chosen as its most preferable one. Before propagating a route the AS must prepend itself to that route. An AS may choose a subset of its neighbors to which a route is propagated. This is a matter of policy specific to each AS. We call it an *export policy*.

Preference and Export Policies

As noted above, the preference and export policies are a local matter for each AS determined by the network operator. These policies usually abide by business relationships and commercial agreements between the different network operators. While in reality there are many types of business relationships and agreements, the following two relationships are widely believed to capture the majority of the economic relationships [32].

- Customer-provider – in such a relationship the customer pays the provider for connectivity. Usually, the provider AS is larger and better connected than the customer AS. For example, the AS administered by Sprint is a provider of the AS of Xerox corporation. Xerox pays money to Sprint for connecting Xerox to

the rest of the Internet through Sprint. In this chapter we denote this kind of relationship with arrow from customer to provider.

- Peer-peer – in such a relationship the two peer ASes agree to transit each other's traffic at no cost. Usually, the two ASes are of comparable size and connectivity. For example, the ASes administered by Sprint and NTT are peers. Each provides the other connectivity to parts of the Internet it may not have access to. In this chapter we denote this kind of relationship with an undirected line between the two ASes.

Based on the above business relationships the following is a well-accepted model for the preference and export policies [32].

Preference Policy.. This policy is based on the following simple rationale. An AS has an economic incentive to prefer forwarding traffic via customer (that pays him) over a peer (where no money is exchanged) over a provider (that he must pay). Combined with the fact that routing must be loop free and preferably on short routes the following policy is defined:

1. Reject a routing update that contains a route if the AS itself already appears on the announced route.
2. Prefer routes that were announced by a customer over routes announced by a peer over routes announced by a provider.
3. Among the most preferable routes choose the shortest ones, i.e., the ones which traverse the fewest ASes.
4. If there are multiple such paths, choose the one that was announced by the AS with the lowest ASN.

Export Policy.. This policy is based on the following simple rationale. An AS is willing to carry traffic to or from other ASes only if it gets paid to do so. Based on this rationale the following policy is

- AS B will announce to AS A a route via AS C if and only if at least one of A and C are customers of B.

To illustrate the above policies consider the topology depicted in Figure 3.1. Let us consider the routing of AS 9 to AS 0. There are three possible paths: (9,3,2,1,0), (9,4,5,0), and (9,7,1,0). Due to the above preference policy 9 will favor the first route over the second route which is favored over the third route. This is because the first route is announced by a customer AS (i.e., 3), while the second and third routes are announced by a peer (4) and provider (7) ASes, respectively. Note that the chosen route (9,3,2,1,0) will be propagated to 7 and 4, according to the above export policy.

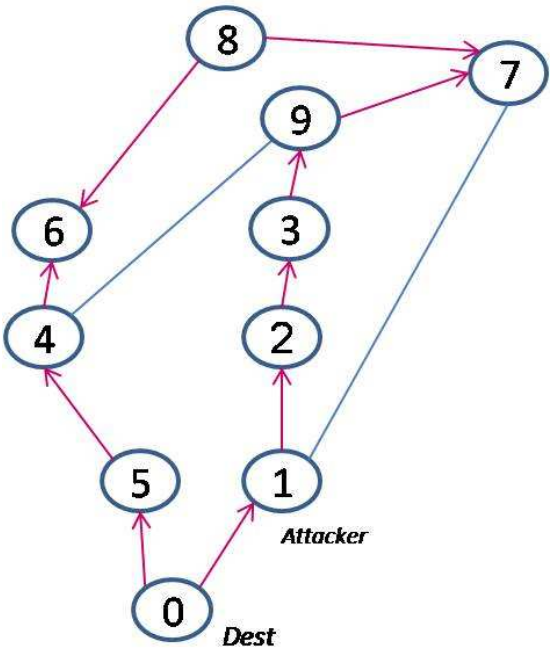

Figure 3.1: BGP network example

3.4 BGP Modeling

We use a BGP standard model acceptable in the literature (e.g. [17, 38, 35]) to facilitate the analysis of traffic attacks using false route advertisements. The model includes all the relevant parts of the protocol that deal with the dissemination and processing of route advertisements and withdrawals. In particular, the mechanisms of route distribution and route preference are modeled, including malicious routes originated by an attacker.

3.4.1 Model simplifications

Motivation

To simplify the model implementation we apply several abstractions and simplifications as detailed below. The goal is to generate a simple and compact model on which a model checker tool can run.

Single Destination Simplification

We assume a single destination, called *Dest*, such that the other ASes want to send traffic to a target network within *Dest*. We can focus on a single destination because routing announcements referring to different destinations flow independently of each other. Namely, the routing to one destination does not influence the routing to another destination. As a result, in our model a routing update does not include the identity of

the target network.

Simplifications and Abstractions of the Attacker

The simplifications related to the modeled attacker are described in Section 3.5.

3.4.2 Threat Model

We use a similar thread model to the one used in [35]. The network contains a single attacker AS node and the other nodes are regular nodes. The attacker can originate arbitrary path announcements and can use arbitrary export policy. The regular nodes make use of all route announcements as legitimate. Thus, when a regular node receives a false route announcement originated by the attacker, it may use the false path in its routing table.

Note that we model a BGP version that does not include mechanisms to validate the routing announcements. There are more secure BGP variants. In Section 3.9.1 we discuss possible extensions of the model that include these more secure variants of BGP. A more secure variant includes some validation mechanisms of BGP announcements, such that the attacker cannot announce any path he wants.

3.4.3 The BGP Model

Modeling the BGP Network

A BGP network N is a tuple $N = (Nodes, Links, Dest, Attacker)$ where $Nodes$ is a set of Autonomous System (AS) nodes in the network graph. $Links$ is a set of node pairs with one of the following types: customer-provider or peer-to-peer, representing the business relationships between ASes in the network. $Dest$ is an AS from $Nodes$ representing a single destination node that contains the target network to which all other nodes build routing paths. $Attacker$ is a node from $Nodes$ representing an AS that can send false routing advertisements to achieve traffic attraction or interception.

$Dest$ and the $Attacker$ are called the *originators* of N. All other nodes are called *regular* nodes. The originators are considered concrete. Concrete nodes are not eliminated or abstracted by our reductions.

For a node n, we define $neighbors(n)$ to be the set of nodes linked to it by any type of link. $customer\text{-}neighbors(n)$ is the set of neighbors that are customers of n.

Example 3.1. Consider the BGP network presented in Figure 3.1. $Nodes = \{0, 1, \dots 9\}$, $Links$ consists of customer-provider links such as $(1 \rightarrow 2)$ and $(9 \rightarrow 7)$, and also peer-to-peer links such as $(4 - 9)$ and $(1 - 7)$. For node 9, $neighbors(9) = \{3, 4, 7\}$ while $customer\text{-}neighbors(9) = \{3\}$.

A *path* in N is a sequence $\pi = (n_1, \dots, n_k)$ of nodes in $Nodes$, such that for every $1 \leq i < k$, n_i and n_{i+1} are connected by an edge (of any kind) from $Links$.

Modeling BGP Messages

A BGP message can be one of the following message types:

- **Routing Update:** A Routing update consists of a path with a sequence of ASes. Formally, a routing update is of the form $\langle n_1, n_2, ..., n_k \rangle$ where $n_i \in Nodes$ for all $1 \leq i \leq k$. When an AS n receives a routing update $\langle n_1, n_2, ..., n_k \rangle$ from its neighbor n_k, it means that it can use the routing path via $n_k \rightarrow ... \rightarrow n_1$ towards the destination.

  For example, consider the BGP network presented in Figure 3.1. When $AS2$ receives a routing update of the form $\langle 0, 1 \rangle$ from its neighbor $AS1$, it means that $AS2$ can use the routing path via $1 \rightarrow 0$ towards the destination. When $AS3$ receives a routing update of the form $\langle 0, 1, 2 \rangle$ from its neighbor $AS2$, it means that $AS3$ can use the routing path via $2 \rightarrow 1 \rightarrow 0$ towards the destination.

- **Route Withdrawal:**

  A route withdrawal is a message of the form $\langle WITHDRAW, n \rangle$, where $n \in Nodes$. When an AS n' receives a route withdrawal $\langle WITHDRAW, n \rangle$ from its neighbor n, it means that n retracts its previously announced routing update that was sent to n' and n' can no longer use it to route towards the destination.

  For example, consider the BGP network presented in Figure 3.1. When $AS2$ receives a route withdrawal of the form $\langle WITHDRAW, 1 \rangle$ from its neighbor $AS1$, it means that $AS1$ retracts its previously announced routing update and that $AS2$ cannot use it as a routing path. Thus, if $AS2$ previously received a routing update of the form $\langle 0, 1 \rangle$ from its neighbor $AS1$ and then received the route withdrawal $\langle WITHDRAW, 1 \rangle$ from $AS1$, it means that $AS2$ can no longer use the routing path via $1 \rightarrow 0$ towards the destination.

Local Configurations

The *local configuration* of a regular AS n consists of:

- A message queue $Q(n)$ containing incoming BGP messages. A BGP message can be a routing update or a route withdrawal as described previously.

- A Routing Information Base $RIB(n)$ containing a set of possible routes to $Dest$ via $neighbors(n)$. Formally, $RIB(n)$ is a mapping of nodes from $neighbors(n)$ to routing paths. Let ϵ represent an invalid routing path. For each $n' \in neighbors(n)$ there is a single element (n', π) in $RIB(n)$, where π is a routing update or ϵ. We have that $(n_k, \langle n_1, ..., n_k \rangle) \in RIB(n)$ if and only if n received from its neighbor n_k the routing update $\langle n_1, ..., n_k \rangle$. If $(n_k, \epsilon) \in RIB(n)$, it means that n has no available routing path via its neighbor n_k. Note that this means that either n

has not received any routing update from its neighbor n_k or that the most recent message it received from n_k was a route withdrawal $\langle WITHDRAW, n_k \rangle$.

- The chosen routing path which is the most preferred is denoted $chosen(n)$.

  For example, if $RIB(3) = \{(9, \langle 0, 4, 5, 9 \rangle), (2, \langle 0, 1, 2 \rangle)\}$ and if the most preferred routing path of $AS3$ is via $AS2$, then $chosen(3) = \langle 0, 1, 2 \rangle$. Note that $chosen(n)$ is determined based on the content of $RIB(n)$ and on the preference policy as described in Section 3.3.

Global Configuration

A (global) *configuration* of N consists of the local configurations of all nodes.

A BGP Run

A run of the BGP protocol on network N starts from an *initial* configuration in which all queues are empty and all RIBs contain invalid routing paths. Initially *Dest* sends announcements to all its neighbors. The run terminates after all nodes in N terminate their run and their queues are empty. In particular, the originators have already sent out all their announcements. The final configuration of a run is called *stable*.

The set of actions performed by the various AS nodes in the network is denoted by *Acts*. An action performed by a node n is *enabled* in a configuration C if n is a regular node and $Q(n)$ is not empty in C or if n is an originator which is not in its terminating configuration. Note that, in a stable configuration no action is enabled.

A *run* in N is a sequence $r = C_1, \alpha_1, C_2, \ldots, \alpha_{k-1}, C_k$ where C_1 is the initial configuration, C_k is a stable configuration, and for every $1 \leq i < k$, C_{i+1} is obtained from C_i by applying the action α_i, which is enabled at C_i. We will often be interested in referring to export actions along a run. We denote by $export(n, n')$ the action of node n exporting a BGP message to its neighbor n'.

The set of all runs of N is denoted by R.

AS protocols

Below we give a high level description of the protocols in our model that run by the different type of AS nodes in the BGP network.

Dest Protocol:

The destination AS *Dest* sends a single routing update of the form $\langle Dest \rangle$ to each of its neighbors and then terminates. *Dest* ignores messages sent by any other ASes in the network. This is because it does not try to build routing paths to the target network which is located within its own AS. Therefore, its queue is always empty.

Attacker Protocol:

An attacker can originate a predefined bounded number *Bound* of arbitrary path announcements to any of its neighbors. Its path announcements are of the form $\langle n_1, .., n_k, attacker \rangle$, where $0 \leq k$ and $n_i \in Nodes$ for all $0 \leq i \leq k$. Any path announcement it originates can be sent to any subset of its neighbors.

Note that a routing path announced by the attacker does not necessarily represent a real routing path in the network graph. It may contain repeating occurrences of AS nodes and does not necessarily lead to Dest.

The regular AS nodes use any routing update as a legitimate routing path leading to an AS in which the target network resides, regardless of whether it was originated by the attacker or by Dest (see Section 3.4.2).

Regular AS Protocol of Node n:

A regular AS node uses export and preference policies as detailed in Section 3.3. AS n is *enabled* when $Q(n) \neq \emptyset$. Let M be a given message on $Q(n)$, and let n' be the neighbor from which M was sent to n.

- If M is a routing update and n is on the announced routing path of M, reject M.

  The rejection process includes setting the path via n' on $RIB(n)$ to invalid. If $chosen(n)$ was via n' of the form $\langle n_1, ..., n_k, n' \rangle$ before the rejection, n also sends a route withdrawal message of the form $\langle WITHDRAW, n \rangle$ based on its export policy. Finally, $chosen(n)$ is updated based on the prefernce policy and the current state of $RIB(n)$ (after invalidating the path via n' on $RIB(n)$).

- Otherwise, if the announcement M is a new route $\langle n_1, ..., n_k, n' \rangle$, update $RIB(n)$ with $(n', \langle n_1, ..., n_k, n' \rangle)$ and then update $chosen(n)$ based on the updated $RIB(n)$.

- If M is a withdrawal $\langle WITHDRAW, n' \rangle$, invalidate the routing path via n' in $RIB(n)$. If the chosen routing path was invalidated, a withdrawal is sent based on the export policy. Additionally, $chosen(n)$ is updated based on the prefernce policy and the current state of $RIB(n)$ (after invalidating the routing path via n' on $RIB(n)$).

- If the update modifies $chosen(n)$ in one of the previous cases, export the new chosen routing path (if exists), based on the export policy. If $chosen(n) = \langle n_1, ..., n_k \rangle$, then the exported routing path is of the form : $\langle n_1, ..., n_k, n \rangle$.

Convergence

It was shown that for a BGP network with normal ASes using the modeled preference and export policies, convergence is guaranteed [32]. It was also shown that in the presence of an attacker with our modeled capabilities, convergence is guaranteed as well

[42]. Moreover, there is a single stable configuration for which the network converges, regardless of the interleaving. For a BGP run r we denote by $stable(r)$ the stable configuration of the BGP network obtained at the end of the run r.

3.4.4 Attack Definitions and Specifications

Attacker Goal

The goal of the attacker in our model is to achieve traffic attraction or interception. We say that a node n is *attracted* by the attacker if in the stable configuration, $chosen(n)$ is a routing path on which the attacker appears. A node n is *intercepted* by the attacker if it is attracted, and in addition the attacker has an available routing path to the destination.

Successful Attack

A successful attack is a BGP run such that its final stable configuration satisfies the attacker goal. The attack strategy can be represented by the sequence of actions preformed by the attacker during the attack, where each of its actions contains the sent announcement and a set of neighbors to which it was sent.

Normal Outcome

The normal outcome is the final routing choices of all ASes in N when the attacker acts like a regular AS. Formally, it is defined by $chosen(n)$ for every regular node $n \in Nodes$ on the final configuration $stable(r)$, where r is a BGP run on which the attacker acts like a regular AS.

Trivial Attack Strategy

In the trivial strategy the attacker sends a false advertisement to all its neighbors on which it announces that the target network is located within its own AS. The announcement sent by the attacker in the trivial attack is of the form: $\langle attacker \rangle$.

Specifications

To measure how successful a traffic attraction or interception attack is, we suggest several types of specifications. Except for the last one, they all compare the result of the attack to the normal outcome of the protocol run and to the result of the trivial attack, when applicable. Below are examples of possible specifications:

1. **Non-trivial traffic attraction** from certain victims: The attacker can successfully attract traffic from some victims, while it fails to do so in the normal run and the trivial attack.

2. **Non-trivial traffic interception** from certain victims: The attacker can successfully apply traffic interception on some victims, while it fails to do so in the normal run and the trivial attack.

3. **Shorten-path** : The attacker manages to attract traffic from certain victims and shorten their routing paths with respect to the trivial attack. Shorter routing paths may be considered more attractive and have a potential to attract more traffic from the rest of the non-modeled ASes on the Internet.

4. **Quantified** traffic attraction or interception: The attacker can attract traffic from more than n nodes, where n is a predefined constant. No specific victims are specified.

We choose to use specifications (1) and (2) in our method and experiments.

3.5 Attacker Model Simplifications

In this section we detail additional abstractions and simplifications of the attacker model that we used.

3.5.1 Abstraction of Paths Originated By the Attacker

Motivation

We noticed that in the attacker model as described in Section 3.4.3 many paths originated by the attacker may have a similar effect on the final configuration of the BGP run. The order of the nodes and their number of repetitions within the announced routing path have no effect on the routing choices of the regular ASes. We are only interested in the information of which AS nodes appear on the originated routing path at least once. If a node n appears on a path and the path is propagated to n, n will reject the path once it receives it.

The Abstraction

Let $LoopNodes$ be a set of nodes from $Nodes \setminus \{attacker\}$ that are predefined as nodes for which an attacker can make use of, in order to apply some of its manipulation strategies. Such nodes cannot be abstracted away from the network, and are thus predefined.

An abstract routing announcement that the attacker can originate is of the form:

$$\langle [*, attacker], length, LN \rangle$$

where:

- $length \geq 1$ is the length of the path.

- $LN \subseteq LoopNodes$ is the set of AS nodes which appear on the path at least once.

- The component $[*, attacker]$ denotes a sequence of AS nodes on the announced path, where $*$ denotes an abstraction for any sequence of $length - 1$ AS nodes from $LN \cup \{attacker\}$.

- Any such path originated by the attacker should satisfy the condition: $|LN| \leq length - 1$. This is because the number of AS nodes that appear on the path cannot exceed the length of the path.

The model is adjusted to the above abstraction:

- Since an abstract routing announcement originated by the attacker is propagated and exported by regular nodes, the general form of an abstract routing announcement in the model is: $\langle [*, attacker, n_1, ..., n_k], length, LN \rangle$.

- When an AS n receives an abstract routing update, it is of the general form: $\langle [*, attacker, n_1, ..., n_k], length, LN \rangle$. A node n appears on the abstract routing path if and only if $n \in LN$ or $n \in \{n_1, ..., n_k\}$. If n chooses to route via an abstract announcement $\langle [*, attacker, n_1, ..., n_k], length, LN \rangle$, it will export an updated abstract announcement $\langle [*, attacker, n_1, ..., n_k, n], length{+}1, LN \rangle$ to some of its neighbors based on the export policy. Additionally, the received abstract announcement is stored in $n.RIB$ for the corresponding neighbor n_k.

For an abstract routing announcement π we denote by $concrete(\pi)$ the set of concrete routing announcements that it represents.

Definition 3.2. If $\pi = \langle [*, attacker, n_1, ..., n_k], length, LN \rangle$, then $concrete(\pi)$ is the set of all concrete routing announcements of the form $\langle p_1, ..., p_i, attacker, n_1, ..., n_k \rangle$, such that the following conditions hold:

- $\{p_1, ..., p_i\} = LN$ or $\{p_1, ..., p_i\} = LN \cup \{attacker\}$

- $i + k + 1 = length$

Example 3.3. If the attacker is $AS1$ and it originates the following abstract announcement: $\langle [*, 1], 3, \{2\} \rangle$, it represents the following set of concrete routing updates: $\langle 2, 1, 1 \rangle$, $\langle 1, 2, 1 \rangle$, $\langle 2, 2, 1 \rangle$.

Definition 3.4. A concrete announcement π and an abstract announcement π' are considered *matching* if and only if $\pi \in concrete(\pi')$.

AS Abstract Configuration

A *concrete configuration* of an AS n is composed of $n.Q$ and $n.RIB$ (note that $n.RIB$ uniquely defines $chosen(n)$ based on the modeled preference policy). It contains only concrete routing paths as modeled in Section 3.4.3.

An *abstract configuration* of an AS n is composed of $n.Q'$ and $n.RIB'$ which may contain abstract routing announcements.

An abstract configuration S' represents a set of concrete configurations $concrete(S')$. For a concrete configuration S of a node n we say that $S \in concrete(S')$ if and only if for every abstract routing path π' in $n.Q'$ or in $n.RIB'$ within the abstract configuration S', a matching concrete routing path π (such that $\pi \in concrete(\pi')$) is in the corresponding content of $n.Q$ or $n.RIB$ within the configuration S. Additionally, for every concrete routing path π in $n.Q$ or in $n.RIB$ within the concrete configuration S a matching abstract routing path π' is in the corresponding content of $n.Q'$ or $n.RIB'$ within the abstract configuration S'. That is, the abstract content of $n.Q'$ and $n.RIB'$ represents the concrete content of $n.Q$ and $n.RIB$.

Definition 3.5. A concrete configuration S and an abstract configuration S' are considered *matching* if and only if $S \in concrete(S')$.

Lemma 3.6. Let S, S' be concrete and abstract configurations of a node n such that $S \in concrete(S')$. Let $\langle n_1, \pi_1 \rangle$ be within the concrete RIB and let $\langle n_1, \pi_1' \rangle$ be a matching routing path within the abstract RIB'. In the concrete model, n prefers the routing path π_1 via n_1 if and only if in the abstract model, n prefers the routing path π_1' via n_1. Additionally, n exports its chosen routing path to the same set of neighbors in both the concrete and abstract models.

Proof. The preference policy of n can be affected by the link type from which a route announcement is received, by the length of a route announcement, and by the ASN of the neighbor from which the announcement is received. The information of the route length and of the neighbor's ASN which exports the announcement is kept in the abstraction. The link types are identical in both models (the abstraction has no effect on the network topology). Thus, in both of the abstract and concrete models the above values are equal for matching route announcements. Therefore, the results of a node's preference policy on matching concrete and abstract configurations are the same.

The export policy depends on the link types between n and its neighbors. Since the link types are identical in both models (the network topology is unchanged), the set of neighbors to which n exports its chosen routing path is the same in both models.

□

Global Abstract Configuration

A global abstract configuration contains the set of all abstract configurations of ASes within the network N.

A global abstract configuration S' and a global concrete configuration S are *matching* if and only if for every node n the abstract configuration of n matches the concrete configuration of n. (That is, the concrete configuration is represented by the abstract configuration). A global abstract configuration S' represents a set of global concrete configurations $concrete(S')$ such that every concrete configuration $S \in concrete(S')$ matches the global abstract configuration.

Equivalence of the Abstract Attacker and the Concrete Attacker

Let M be the BGP model as described in Section 3.4.3, and let M' be the BGP model with the abstraction of routing announcements that the attacker originates. Let N be a BGP network. Let R be the set of all possible BGP runs on N as modeled by M, and let R' be the set of all possible BGP runs on N as modeled by M'. We denote by $stable(R)$ the set of all $stable(r)$ for every run r in R.

Lemma 3.7. Let π' be an abstract routing announcement and let π be a matching concrete routing announcement. Let S' be a global abstract configuration and let S be a matching global concrete configuration. After an AS n receives the above routing announcement and applies its procedure in the abstract and concrete model, the obtained global configurations are matching.

Proof. The AS n appears on the routing announcement π if and only if it appears on π', based on the definition of matching abstract and concrete routing announcements. Thus, n rejects the announcement in the abstract model if and only if it rejects it in the concrete model. Let p be the node that exported the announcement to n. If n rejects the announcement and sends a withdrawal, it sends the withdrawal to the same set of neighbors on the abstract and concrete model based on its export policy. If n does not reject the path, it is added to $n.RIB$ as $\langle p, \pi \rangle$ and to $n.RIB'$ as $\langle p, \pi' \rangle$. Thus, the content of $n.RIB$ and of $n.RIB'$ remains matching after the update. As a result, based on Lemma 3.6, $chosen(n)$ is determined similarly on both the abstract and concrete model. If the new routing announcement is chosen and exported, it is added to the queues of the same neighbors on both the abstract and concrete models. Thus, the queues contents of the abstract and concrete configurations remain matching after the update. Finally, we can conclude that the concrete and abstract global configurations S, S' remain matching after n terminates its procedure.

□

Lemma 3.8. If $LoopNodes = Nodes \setminus \{attacker\}$ in M', then $stable(R)$ and $stable(R')$ are matching. That is, for every final configuration S in $stable(R)$ there exists a corre-

sponding final configuration S' in $stable(R')$ such that $S \in concrete(S')$. Additionally, for every final configuration S' in $stable(R')$ there exists at least one final configuration S in $stable(R)$ such that $S \in concrete(S')$.

Proof. Note that if $LoopNodes \neq Nodes \setminus \{attacker\}$ then the set of announcements that the attacker can originate in the abstract model cannot represent all concrete announcements that it can originate in the concrete model. For example, if n is a regular node such that $n \notin LoopNodes$, then a concrete announcement on which n appears cannot be represented in the abstract model. Thus, there exists a run in the concrete model on which n receives an announcement originated by the attacker and rejects it to prevent a routing loop, whereas there is no corresponding run in the abstract model due to the limitation of $LoopNodes$.

If $LoopNodes = Nodes \setminus \{attacker\}$ then every concrete run has a representing run in the abstract model and vice versa. Let r be a concrete run in the model M. Let K be the set of announcements that the attacker originates during that run. Let K' be a matching set of abstract announcements. Let r' be an abstract run obtained from r by replacing the concrete announcements from K with abstract announcements from K' and by adjusting the RIB and queues contents. Note that both runs r, r' begin from matching global configurations. The initial configuration in both models M, M' is defined as a global configuration for which every node's queue is empty and every node's RIB contains invalid routing paths. Thus, by induction on the length of the run r and based on Lemma 3.7 it is possible to obtain such a matching run r', since both runs begin from matching global configurations. Therefore, $stable(r)$ and $stable(r')$ are matching, and thus r' is a matching run that represents r in the abstract model. In the second direction, let r' be an abstract run from the model M'. Let K' be the set of abstract messages sent by the attacker on this run. Thus, we can obtain a corresponding set K of matching concrete announcements. Based on the abstraction definition every abstract announcement represents at least one concrete announcement. Thus, a concrete run r obtained from r by replacing the abstract announcements from K' with concrete announcements from K is a matching run on the concrete model. Therefore, $stable(r)$ and $stable(r')$ are matching, and thus r is a matching run that is represented by r' in the abstract model.

□

Corollary 3.9. If $LoopNodes = Nodes \setminus \{attacker\}$, for every traffic attraction attack in M there is a corresponding attack in M' and vice versa.

The attack specifications we use are based on the final configurations of the runs in our model. We showed that every abstract final configuration has a corresponding concrete final configuration and vice versa, if $LoopNodes = Nodes \setminus \{attacker\}$. Thus, it follows that for every traffic attraction attack in M there is a corresponding attack in M' and vice versa.

From the above it follows that the modeled attacker in the concrete and abstract versions are equivalent when $LoopNodes = Nodes \setminus \{attacker\}$.

3.5.2 Reducing the Number of Messages Originated By the Attacker

In Section 3.4.3 we described the attacker capabilities and mentioned that it could originate a predefined bounded number $Bound$ of arbitrary path announcements to any of its neighbors.

In this simplification we use a $Bound = 1$ and prove that an attacker with this bound is equivalent to an attacker with any larger finite bound.

Lemma 3.10. If $chosen(n)$ for some node n is defined, then the routing path π it represents can be of one of the following forms:

- $\pi = (Dest, n_1, n_2, .., n_k)$ such that $k \geq 0$ and $n_i \notin \{Dest, Attacker\}$ for all $1 \leq i \leq k$.
- $\pi = (*, Attacker, n_1, n_2, .., n_k)$ such that $k \geq 0$ and $n_i \notin \{Dest, Attacker\}$ for all $1 \leq i \leq k$. (The * denotes an abstract sequence of nodes from LoopNodes as described in Section 3.5.1).

The correctness of the lemma above follows from the definition of the protocol. Routing announcements are propagated by export actions. On each export action of a path π by a regular node n, the node appends itself to π. Announcements origination is only executed by originating nodes, and therefore an announced routing path can only be of the form $(Dest)$ or $(*, Attacker)$. The originating nodes do not export any announcements that they receive from their neighbors. Therefore, a regular node n can have in its RIB only routing paths of the form $(Dest, n_1, n_2, .., n_k)$ or $(*, Attacker, n_1, n_2, .., n_k)$. The sequence $n_1, n_2, .., n_k$ represents a real path on the network topology. Through this path the message was exported until n received it from n_k.

Definition 3.11. Let n be a regular node such that $chosen(n) = \pi$. From the previous lemma, $\pi = (Dest, n_1, n_2, .., n_k)$ or $\pi = (*, Attacker, n_1, n_2, .., n_k)$, for some $k \geq 0$, and such that $n_i \notin \{Dest, Attacker\}$ for all $1 \leq i \leq k$.

- We define the *regular suffix* of π, denoted $RegSuf(\pi)$, as:
  $RegSuf(\pi) = (n_1, n_2, .., n_k)$.
- We define the originator of π, denoted $orig(\pi)$ as:
  $orig(\pi) = O$, such that $O = Dest$ if $\pi = (Dest, n_1, n_2, .., n_k)$ and $O = Attacker$ if $\pi = (*, Attacker, n_1, n_2, .., n_k)$.

Lemma 3.12. Let r be a BGP run and let x be a regular node in the network topology. If in the final configuration $stable(r)$ the node x has a chosen routing path such that $chosen(x) = \pi$, and such that:

- $RegSuf(\pi) = (n_1, n_2, .., n_k)$
- $orig(\pi) = O$,

then the following holds in the configuration $stable(r)$:

- $orig(chosen(n_i)) = O$ for all $1 \leq i \leq k$.
- $RegSuf(chosen(n_1)) = ()$
- $RegSuf(chosen(n_i)) = (n_1, n_2, .., n_{i-1})$ for all $1 < i \leq k$

Example 3.13. Consider the topology from Figure 3.1. If in the final configuration $AS3$ routes via the path $3 \to 2 \to 1 \to 0$, then $AS2$ in the final configuration routes via the path $2 \to 1 \to 0$, and $AS1$ in the final configuration routes via the path $1 \to 0$.

Proof. If in the final configuration of r the node x has a routing path π to the destination such that $RegSuf(\pi) = (n_1, n_2, .., n_k)$ and $orig(\pi) = O$, it means that this routing path was originated by O and was exported by nodes along the path $(n_1, n_2, .., n_k)$ on the network topology. This follows from the definition of the protocol. n_k was the node that exported the routing path π to x.

Therefore, during the run r, node n_1 exported the routing path originated from O to n_2. When a node exports a routing path, its exported path is determined by *chosen*. Thus, while exporting, it satisfied the conditions: $RegSuf(chosen(n_1)) = ()$ and $orig(chosen(n_1)) = O$. Similarly, n_2 exported the routing path to n_3. While exporting, it satisfied the conditions: $RegSuf(chosen(n_2)) = (n_1)$ and $orig(chosen(n_2)) = O$. We can generalize this to each node n_i on the path $(n_1, n_2, .., n_k)$. Therefore, each such node, while exporting the relevant routing path originated by O, satisfied the conditions from the lemma.

Assume by negation that in the final configuration $stable(r)$ at least one of the nodes violates the conditions of the lemma. Let n_i be the node with the minimal index (that is, closest to O on the path $(O, n_1, n_2, .., n_k)$) that violates the conditions on the final configuration.

Therefore, the node n_i had changed its routing choice later on the run r (after exporting to n_{i+1} the routing path originated from O that was eventually exported to x).

Let M' be the announcement that changed the routing choice of n_i afterwards. Let M be the original announcement the n_i received from n_{i-1} before its export to n_{i+1}. We will refer to all possible cases:

- M' was received by n_i from the same neighbor n_{i-1} that sent the original message M. This means that n_{i-1} has also changed its routing choice, since it exported a different routing path afterwards. This contradicts the fact that n_i is the first node on the path $(n_1, n_2, .., n_k)$ that changed its routing choice.

- M' was received from a different neighbor y. This means that n_i prefers the route via y over the route via n_{i-1}. Let $Exp(n_i, y)$ be the set of nodes to which n_i exports a route via y, and let $Exp(n_i, n_{i-1})$ be the set of nodes to which n_i exports a route via n_{i-1}. We know that $n_{i+1} \in Exp(n_i, n_{i-1})$. Since n_i prefers the route via its neighbor y, it follows that the link type between n_i and y is either the same or a more preferred link type than the link type between n_i and n_{i-1}. Based on the export policy it follows that $Exp(n_i, n_{i-1}) \subseteq Exp(n_i, y)$. This means that $n_{i+1} \in Exp(n_i, y)$. Thus, the new preferred routing path is necessarily propagated via a path from n_i to x. This contradicts the fact that x routes via the original routing path $x \to n_k ... \to n_1 \to O$ in the final configuration.

- If M' was a route withdrawal, the only case where it could change the routing choice of n_i would be if it was received from n_{i-1}. However, this means that n_{i-1} has also changed its routing choice. Thus, we get a contradiction similarly to the first case.

□

Correctness of the Simplification

Theorem 3.14. Let M_1, M_2 be two different route announcements originated by the attacker. Let r be a BGP run in which the attacker sends to its neighbor A the route announcement $M1$ and later the route announcement $M2$. No other route announcements are sent from the attacker to A. In the final configuration of r, $M1$ has no effect on any of the routing choices for any node in the network. Thus, r is equivalent to a similar run in which the attacker only sends $M2$ to its neighbor A.

Proof. Let $M_1 = \langle [*, attacker], length_1, LN_1 \rangle$ and $M_2 = \langle [*, attacker], length_2, LN_2 \rangle$. Assume by negation that there exists a node x that its final configuration of the run r is affected by $M1$. We will refer to all possible cases for the effect of $M1$ on the final configuration of x:

- x is routing via M_1 in its final configuration. Based on Lemma 3.12 it follows that every node on a path from the attacker to x is routing via M_1 in the final configuration, including A which is the neighbor of the attacker. However, since A received from the attacker the message M_2 after M_1, it could not have kept its routing choice via M_1. The update of $M2$ have replaced $M1$ in its RIB, and in particular its chosen routing path could not have remained $M1$ once it was no

longer in its RIB. Thus, in the final configuration the routing choice of A is not via $M1$, which contradicts the assumption that x is routing via $M1$ in the final configuration.

- LN_1 affects the routing choice of x. This means that x has received M_1, rejected this routing path, and following the rejection it has chosen to route via a different neighbor. Let y be the neighbor that exported $M1$ to x. At the time of the export y has chosen to route via $M1$. However, in the final configuration y could not have kept this routing choice (as proved in the previous case above). Thus, y has changed its routing choice after the export. One of the two options could be:

  1. $M2$ was propagated and exported to y. In that case y would export $M2$ to x (since it also exported $M1$ to x). In that case only $LN2$ could have an effect on x and the effect of $LN1$ would be irrelevant.
  2. $M2$ was not propagated to y due to an earlier loop. Let z be the node that rejected $M2$ to prevent a routing loop. As a result, z had to invalidate the route via the neighbor from which $M2$ was exported and to send a route withdrawal to retract its previous routing choice via $M1$. In that case the withdrawal would be propagated to y (to retract $M1$), and then y would export the withdrawal to x. This also means that the previous effect of $LN1$ is overridden by the new withdrawal of y.

  This means that y had either sent a withdrawal to x or had sent a new route announcement to x. Thus, LN_1 could not have had an effect on x afterwards. Once it received the new message from y, only the new message could have an effect on the routing choice of x.

□

The above proof can be generalized to any finite number of messages sent by the attacker in a similar way. Thus, if during a BGP run the attacker sends a sequence of messages $M_1, ..., M_k$ to the same neighbor A, only the last message M_k can have an effect on the routing choices of the AS nodes. Thus, an attacker that can send up to one message per each neighbor is equivalent to an attacker that can send any finite number of messages per each neighbor.

3.6 Reductions and Abstractions

The goal of our reductions is to obtain a manageable sized fragment of the large network which is suitable for identifying BGP traffic attacks or show their absence.

3.6.1 Self-contained Fragments

The extraction of a self-contained fragment is our main reduction that significantly reduces the initial network, such as the full Internet topology. The reduction is based on

preserving the flow of announcements in the network during a BGP run. The following is a central notion in our analysis of the flow. It directly follows from the export policy (see Section 3.3).

Definition 3.15. A path $\pi = (n_1, \ldots, n_k)$ in N is ***valid*** if n_1 is an originating node, no node is repeated on π, and for every $1 < i < k$, at least one of n_{i-1} and n_{i+1} is a customer of n_i. Further, no n_i is an originating node except n_1 and possibly n_k.

Examples of valid paths in the network N of Figure 3.1 are $(0, 5, 4, 6, 8)$ and $(0, 5, 4, 9, 3, 2, 1)$. Note that $(0, 5, 4, 6, 8, 7)$ is not a valid path, since both 6 and 7 are not customers of 8. The following is a key observation about valid paths.

Lemma 3.16. Export actions can only be performed along valid paths for every BGP run in a network N. Thus, for a regular node n and an action $export(n, n')$, the nodes n, n' are neighbors along a valid path in N.

Proof. In the first part we prove the lemma for export actions of route announcements. Every export of a route announcement is a result of an original route announcement originated by an originating node O. Thus, when a regular node k receives a route announcement from its neighbor l, there exists an export path from O to l through which the announcement was propagated.

Assume that an action $export(n, n')$ is performed in a BGP run with a route announcement that was originated by O. We prove that n, n' are neighbors along a valid path which starts from O by induction on the distance of n from O along the export path of the message that O originated.

Base: In the base case the distance is 1, which means that n is a neighbor of O. According to the export policy of a regular node n, if it receives a route announcement from O and it chooses to use it, it will export the route to its neighbor n' if and only if at least one of the nodes O or n' are customers of n. This means that the path O, n, n' is a valid path by definition.

Induction hypothesis: Assume the correctness for distance k.

Induction step: Let n be a node for which its distance on the export path from O is $k+1$. Let n'' be the neighbor of n' on the export path, so that n'' exported the message originated by O to n. Based on the induction hypothesis, the export path from O to n'' is a valid path. This is due to the fact that the distance of n'' from O on the export path is k. Since n is a regular node and since it exports the message originated by O to its neighbor n', it means that based on its export policy at least n'' or n' is a customer of n. Thus, the suffix of n, n' of the export path from O follows the condition of valid paths. Therefore, the export action is indeed performed along a valid path, which is the export path from O to n'.

Now we prove the lemma for export actions of route withdrawals. Note that based on the model, none of the originating nodes originates a route withdrawal. Thus, when

a node receives a route withdrawal, there exists a withdrawal path along which the withdrawal is propagated. Such a withdrawal path begins with a node that originates a withdrawal invalidating its routing choice to prevent a routing loop. Thus, the first node that originates a withdrawal along a withdrawal path is a node that received a route announcement which contained itself on the announced path. Let p be a node that receives a route announcements that it rejects due to a routing loop. Thus, based on the proof above, there exists a valid path π from an originator O to p through which the announcement was propagated to p. As a result, p originates a withdrawal. Let n be a node that received a withdrawal that was originally created by p. We prove that a withdrawal export from n to n' is along a valid path from O via p to n'. As in the previous proof, we prove by induction on the distance of n from p along the export path of the withdrawal that p originated.

Base: In the base case the distance is 1, which means that n is a neighbor of p. We know that there is a valid path from O to p through which a route announcement was propagated to p. Then, p rejected the path from its neighbor p' to prevent a routing loop, invalidated its routing choice via p' and sent a withdrawal based on its export policy. n received the withdrawal from p, which means that either p' or n is $p's$ customer. Then, n invalidated its routing choice via p and exported a withdrawal to its neighbor n'. This means that either p or n' is a customer of n. Thus, after appending the nodes n, n' to the valid path from O to p, we get a valid path from O to n'. This means that the export action of the withdrawal from n to n' was indeed performed along a valid path.

Induction hypothesis: Assume the correctness for distance k.

Induction step: Let n be a node for which its distance on the export path from p is $k+1$. Let n'' be the neighbor of n on the export path, so that n'' exported the withdrawal originated by p to n. Based on the induction hypothesis, the export path from p to n'' is along a valid path. This is due to the fact that the distance of n'' from p on the export path is k. Since n is a regular node and since it exports the withdrawal originated by p to its neighbor n', it means that based on its export policy at least n'' or n' is a customer of n. Thus, the suffix of n, n' of the export path from p follows the condition of valid paths. Therefore, the export action is indeed performed along a valid path.

□

Corollary 3.17. Let n be a regular node. If there is no valid path in N with edge from node n to node n' then there is no run in N along which $export(n, n')$ is performed.

Note, however, that the contrary is not true. There might be an edge (n, n') on a valid path but still no $export(n, n')$ is performed. This is due to the preference policy of nodes.

We say that n *cannot export* to n' if there is no run in which the action $export(n, n')$ is performed.

Definition 3.18. Let N be a network and let $S \subseteq Nodes$ be a subset of its nodes that includes all originators of N. S is a ***self-contained fragment*** of N if for every $n \in (Nodes \setminus S)$, n cannot export to any $n' \in S$. This means that nodes outside of S cannot change routing decisions of nodes in S.

The following lemma describes the significance of self-contained fragments.

Lemma 3.19. Let N be a network and let S be a self-contained fragment of N. Then, any traffic attack found on S can occur on N as well. Moreover, if we prove that a traffic attack is not possible in S then the corresponding attack is not possible in N as well.

Proof. Let N be a network and let S be a self-contained fragment of N. A traffic attack is determined by the set of routing announcements that the attacker originates and by the set of its neighbors for which the announcements are sent. Let K be a set of pairs (π, E) where π is a routing announcement originated by the attacker and E is a subset of the attacker's neighbors to which π is sent. Thus, such a set K defines a traffic attack on S. Let $r_K(S)$ be a run in S obtained by the attacker's actions in K. Let $r_K(N)$ be a corresponding run in N obtained by the same attacker's actions in K. We show that for each $n \in S$, $chosen(n)$ in $stable(r_K(S))$ is identical to its counterpart in $stable(r_K(N))$.

Note that since the attacker and the destination nodes are in S, we can represent the run $r_K(N)$ as a concatenation of runs: r_1, r_2. In the first part r_1 only the procedures of nodes within S are applied until stabilization within S (note that nodes in S can export to nodes outside of S but such export actions have no effect on nodes in S since the procedures of nodes in $N \setminus S$ are not applied in this part of the run). In the second part $r2$ the procedures of nodes within N are applied until stabilization within N. Clearly, $r1$ is identical to $r_K(S)$, except for the export actions to nodes outside of S within $r1$. At the end of $r1$ only the queues of nodes within $N \setminus S$ may contain messages. We can choose this specific interleaving for the run $r_K(N)$ since it has no effect on the final configuration $stable(r_K(N))$ (see Section 3.4.3). Therefore, it follows that $stable(r2) = stable(r_K(N))$.

Let n be some node in S, and let π be $chosen(n)$ in $stable(r_K(S))$ and π' be $chosen(n)$ in $stable(r_K(N))$. Assume by negation that $\pi \neq \pi'$. From the above it follows that $chosen(n)$ in $stable(r1)$ is equal to π. Thus, during $r2$ there was an export to n from one of is neighbors n', such that following the $export(n', n)$, n had changed its routing choice from π to some other route. Recall that in the initial configuration of $r2$ only nodes within $N \setminus S$ have messages in their queues. The queues of nodes within S are empty and the originating nodes are in their terminating state (all the originating nodes are within S). Since n is within S, it means that $export(n', n)$ was along an export path which started from some node $n'' \in N \setminus S$ during $r2$. Based on Lemma 3.16, export actions can only be performed along valid paths. Thus, it follows that there is a valid path from some originating node O of the form: $(O, ..., n'', ..., n', n)$.

The export actions on the path from O to n'' were during the run $r1$ ($n'' \in N \setminus S$ had a message in its queue at $stable(r1)$). The export actions on the path from n'' to n via n' were during the run $r2$. Note that $n \in S$ and $n'' \in N \setminus S$. This means that there was an export from some node in $N \setminus S$ to some node in S. However, this contradicts the fact that S is a self-contained fragment of N. Thus, it follows that the assumption was wrong and that in $stable(r2) : chosen(n) = \pi$. There could not have been an export action from a node outside of S to a node in S, and therefore in $r2$ only nodes in $N \setminus S$ are activated. Therefore, any traffic attack found on S can occur on N as well.

□

The lemma implies that instead of searching a huge network N (such as the Internet) we can identify a (relatively small) self-contained fragment, isolate it from the rest of the network, and search for possible attacks on it. Assume an attacker (in S) can attract traffic from a node n' in S. Then since nodes outside of S do not send n' alternative routing options, they cannot "convince" n' to change its routing choice and avoid the route through the attacker. Thus, a traffic attack which is successful in S is also successful in N. Similarly, if a certain node is definitely *not* routing through the attacker in S then the same holds in N as well.

The only attack that might be more successful on N than on S is the quantified attack (see Section 3.4.4, specification 4) that requires a minimum number of attracted nodes. This is because nodes outside of S may be attracted by an attacker in S, thus increase the number of attracted nodes.

Fragment Importance

Following Lemma 3.19 , it should be noted that the fragment concept is of great importance for applying significant reductions on BGP networks. The set of announcements that a node within the fragment can receive during any BGP run with an attacker within the fragment on the whole Internet is equal to its counterpart on a similar run that is applied to the fragment only. Therefore, the set of chosen routing paths within the fragment is equal as well, due to the deterministic preference policy of each node. Thus, the task of applying model checking on the whole Internet is reduced to applying it on a self-contained fragment when searching for BGP traffic attacks with our suggested specifications. Additionally, the fragment concept may be useful for other BGP-based formal analyses that require substantial reductions on large networks.

Computing Self-contained Fragments

The algorithm for computing a self-contained fragment for given destination and attacker nodes is given in Figure 3.2. $\Pi_{n,c,O}$ denotes the set of all nodes on valid paths from any $o \in O$ to n via c.

Given a network $N = (Nodes, Links, Dest, Attacker)$, we describe the computation of a set of nodes which forms a self-contained fragment. The resulting S includes $Dest$

```
1: function FRAGMENT(N)
2:     S = O ∪ neighbors(O)
3:     while continue == True do
4:         S_add = ∅
5:         for ∀n ∈ S and ∀c ∈ neighbors(n) : c ∉ S ∪ S_add do
6:             S_add = S_add ∪ Π_{n,c,O}
7:         end for
8:         if S == S ∪ S_add then
9:             continue = False
10:        else
11:            S = S ∪ S_add
12:        end if
13:    end while
14:    Return (S)
15: end function
```

Figure 3.2: Algorithm for Computing a Self-Contained Fragment S of N.

and *Attacker* and excludes some of N's nodes that cannot export any announcement to S.

Initially, only the set of originators $O = \{Dest, Attacker\}$ and their neighbors are in S (line 2). A node c outside of S is inserted to S if c is a neighbor of some $n \in S$, and c is on a valid path from some originator in O to n (lines 5-6). The algorithm terminates when for every $c \notin S$ which is a neighbor of some $n \in S$, c is not on a valid path from an originator to n and therefore (by Corollary 3.17) c cannot export to n.

Example for a Self-contained Fragment Extraction

Consider the 10-nodes-sized network, presented in part A of Figure 3.3. The grey node 48685 is the attacker. The yellow node 22561 is the destination. The thick lines in part A represent the arrow direction of the customer-provider links. In practice the initial network can be much larger. Applying the fragment extraction algorithm results in:

1. Initialization: Insert O and their neighbors. $S = \{22561, 48685, 209, 25934, 6677\}$
2. Add $c = 3257$, due to valid path : $(o = 22561, 209, 3257, n = 6677)$
3. Add $c = 5580$, due to valid path : $(o = 22561, 209, 5580, n = 25934)$

The remaining nodes are not added. For example, 3303 does not appear on any valid path in the original network, and is therefore dropped during the construction of a self-contained fragment. After applying this phase we remain with 7 nodes as presented in part B of Figure 3.3.

3.6.2 Definite Routing Choice

In this reduction we identify nodes that never route via the attacker. If for all runs of BGP on a network N, a node n chooses to route through a specific path π originated by *Dest* that does not pass through the attacker, then π is the *definite routing choice*

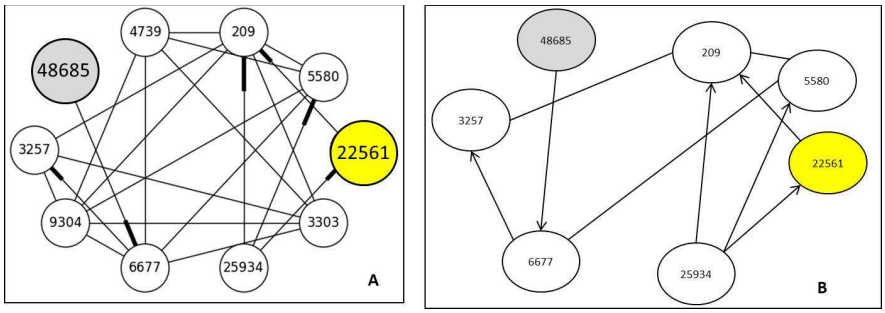

Figure 3.3: Fragment example

of n, denoted $drc(n)$. We consider such nodes as *safe*, since they cannot be attracted by the attacker.

For example, in Figure 3.1, $drc(5) = [0]$ and $drc(4) = [5, 0]$. Node 5 is a neighbor of *Dest* and its link to *Dest* is more preferred than its other link. Therefore, since the announcement from *Dest* is guaranteed to be sent to 5, it will always prefer this path regardless of other paths it might get from 4. For a similar reason, and since 5 is guaranteed to export its path to 4, node 4 will always prefer the route via 5. On the other hand, $drc(9)$ is undefined since on different runs its choice of routing may change as a result of the announcements sent by the attacker (which may change from run to run).

$drc(n)$, when defined, is $chosen(n)$ in every run, regardless of the attacker's actions. Consequently, the export actions of n are also determined. We can therefore eliminate such n from our network and initiate a BGP run from a configuration in which the results of its export is already in the queues of the appropriate neighbors. This may significantly reduce the network size to which model checking is applied.

Computing Trees of Valid Paths

For a given originating node n, the tree of all valid paths starting at n, denoted $Valid(n)$, can easily be constructed. Figure 3.4 presents the trees $Valid(0)$ and $Valid(1)$ of the two originating nodes of the network of Figure 3.1.

The following procedure computes a tree of all valid paths in the network For each originating node.

Algorithm ValidTree(N, o) (Figure 3.5) returns the tree T_o of all valid paths in N, starting from the originating node o. The algorithm works iteratively, adding a successor n' to a node $n \notin O$ on the tree if at least one of $father(n)$ and n' is a customer of n. It further checks that n' does not appear on $path(o, n)$, which is the path in the tree from the root o to n. Note that $father(n)$ and $path(o, n)$ are defined over T_o while the notions of neighbor and customer are taken from N.

Figure 3.4 depicts the trees of valid paths originated from the destination node and from the attacker in the BGP network of figure 3.1.

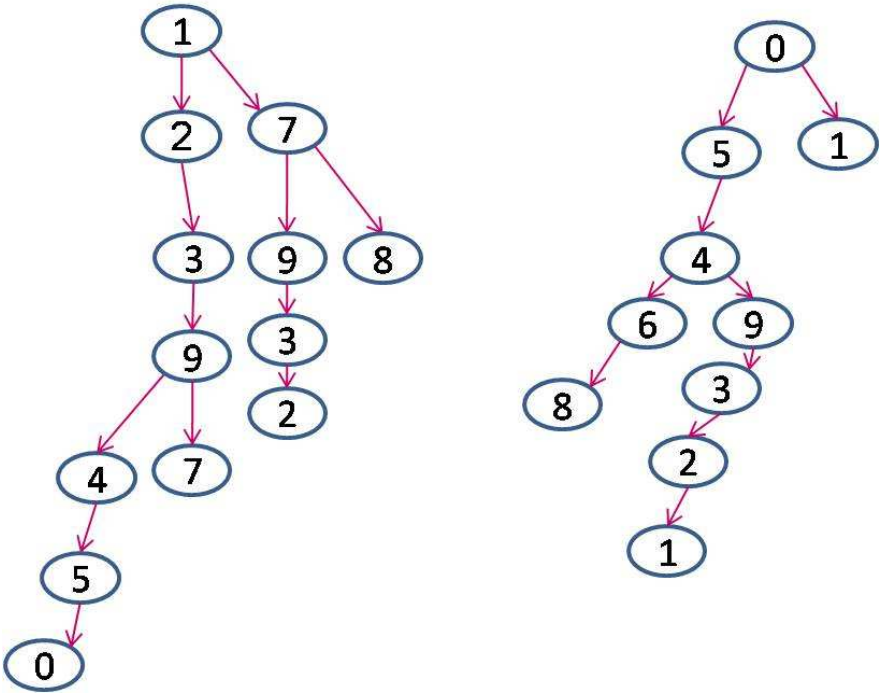

Figure 3.4: Valid paths trees example

Computing Potential Routing Options (PRO) for Nodes in N

The set of Potential Routing Options of a node n, denoted $PRO(n)$, is the set of pairs $(n', length)$ such that there is a valid path $\pi = [n_1, ..., n_k]$ where n' is a neighbor of n on π and the path from n' to the originator n_1 is of length $length$. That is, for some i, $n' = n_{i-1}, n = n_i$ and $length = i - 1$.

PRO(n) includes an over-approximation of the announcements originated by Dest, that n may obtain from its neighbors on any BGP run. It also includes announcements that are produced as a result of an attacker's announcement, but those appear with the shortest length only (see example below). For example, in the BGP network of Figure 3.1, $PRO(9) = (4,3),(3,3),(7,2)$. The pair $(4,3)$ represents the following valid path from Dest: $(0,5,4)$. Since it is a valid path, a BGP run can potentially contain export actions along the valid path $(0,5,4,9)$ (starting from the announcement originated by Dest). In such a case, $AS9$ receives this route announcement $(0,5,4)$ from its neighbor $AS4$. Thus, the pair $(4,3)$ in $PRO(9)$ represents the fact that $AS9$ can potentially receive a route announcement with $length = 3$ from its neighbor $AS4$. The pair $(3,3)$ represents the following valid path from Attacker: $(1,2,3)$. Since it is a valid path, a BGP run can potentially contain export actions along the valid path $(1,2,3,9)$ (starting from the announcement originated by Attacker). In such a case, $AS9$ receives a route announcement $(*,1,2,3)$ from its neighbor $AS3$. Thus, the pair $(3,3)$ in $PRO(9)$ represents the fact that $AS9$ can potentially receive a route announcement with minimal length: $length = 3$ from its neighbor $AS3$. Similarly, the pair $(7,2)$ in

```
1: function ValidTree(N, o)
2:     Set o to be the root of T_o
3:     Add neighbors(o) to be direct sons of o
4:     for every unprocessed leaf n ∉ O in the tree do
5:         for every n' ∈ neighbors(n), n' not on path(o, n) do
6:             if father(n) or n' is a customer of n then
7:                 add n' to be a direct son of n
8:             end if
9:         end for
10:    end for
11:    Return (T_o)
12: end function
```

Figure 3.5: Algorithm for computing the tree T_o of all valid paths in N from the originating node o

```
1: function Pro(N)
2:     for each originator node o do
3:         T_o = ValidTree(N, o)
4:         for every node n and its successor n' on T_o do
5:             add the pair ⟨n, |path(o, n)|⟩ to PRO(n')
6:         end for
7:     end for
8: end function
```

Figure 3.6: Algorithm for computing $PRO(n)$ for each node n on the tree T_o

$PRO(9)$ represents the fact that $AS9$ can potentially receive a route announcement with minimal length: $length = 2$ from its neighbor $AS7$ (the export actions are along the valid path $(1, 7, 9)$). Note that,$(3, 3)$ and $(7, 2)$ represent paths from the attacker. The attacker may also originate announcements with longer length. However, we do not keep them in PRO since they are less attractive than those included in PRO. If for all runs of N, n chooses to route though a specific $(n', length) \in PRO(n)$ which is originated by Dest, then $(n', length)$ is the definite routing choice of n, denoted $drc(n)$. For example, in Figure 3.1, $drc(5) = (0, 1)$ and $drc(4) = (5, 2)$. On the other hand, drc(9) is undefined since on different runs its choice of routing may change as a result of the announcements sent by the attacker (which may change from run to run).

Once we have a set of trees for each originator, computing PRO is done by traversing each of the trees from root to leaves and collect for each node the announcement that can potentially be sent to it by its father on the trees. The Algorithm is presented in Figure 3.6.

Computing Definite Routing Choices

For a node n, a routing path to $Dest$ is definite if it is obtained from a neighbor that definitely routes to $Dest$ through the announced path, regardless of the attacker's behavior. Further, the announced path is preferred by n to all other potentially routing options in its $PRO(n)$. Note that, due to the non-deterministic behavior of the attacker,

```
1: function DRC(N)
2:     for each node n in N do
3:         Compute PRO(n); DRO(n) = ∅; drc(n) =⊥
4:     end for
5:     C = (neighbors(Dest) \ O)
6:     for each n ∈ C, add ⟨Dest, 1⟩ to DRO(n)
7:     while C ≠ ∅ do
8:         Choose n from C
9:         if ∃o ∈ DRO(n)(∀p ∈ PRO(n)[preferred(o, p)]) then
10:            Let o be the most preferable
11:            drc(n) = o
12:            Add to C all nodes n′ ∉ O s.t. n exports Update(o, n) to n′ and drc(n′) =⊥
13:        end if
14:        Remove n from C
15:    end while
16: end function
```

Figure 3.7: Algorithm for computing $drc(n)$ for $n \in Nodes$

some nodes may not choose the same path in all possible runs. In that case, their drc remains undefined.

Algorithm DRC, depicted in Figure 3.7, accumulates in $DRO(n)$ those $drc(n_1)$ obtained from neighbors n_1 of n. It then compares them to all routing option in $PRO(n)$. If a certain route o is preferable to all other ($preferred(o, p)$ for all routes p) then o becomes $drc(n)$. The announcement $o = \langle n_1, length \rangle$ is updated by n to be $Update(o, n) = \langle n, length + 1 \rangle$. $Update(o, n)$ is inserted to $DRO(n_2)$ of all neighbors n_2 to which n exports $drc(n)$, according to the BGP export policy.

Lemma 3.20. Let N be a network and S a self-contained fragment of N. Then for every node $n \in S$, $PRO(n)$ obtained by PRO(S) is equal to $PRO(n)$ obtained by PRO(N).

Proof. The computation of PRO is based on the valid path trees. Let n be a node in S. If the pair $\langle n', length \rangle$ is added to $PRO(n)$ during the computation on S, it means that there is a valid path from some originator o in S to n via the neighbor n'. Since S is contained within N, that valid path also exists in N. Therefore, the pair $\langle n', length \rangle$ is also added to $PRO(n)$ during the computation on N. In the other direction, let $\langle n', length \rangle$ be a pair added to $PRO(n)$ during the computation on N. Thus, there is a valid path π from some originator o to n via the neighbor n'. Since $n \in S$ and based on the definition of a self contained fragment, all nodes on the valid path π are in S. This is because there is no valid path from an originator to a node within S via some node outside of S. Thus, the pair $\langle n', length \rangle$ is also added to $PRO(n)$ during the computation on S (since the relevant valid path is within S). □

Lemma 3.21. Let N be a network and S a self-contained fragment of N. Then for every node $n \in S$, $drc(n)$ obtained by DRC(S) is equal to $drc(n)$ obtained by DRC(N).

The lemma states that definite routing choices of $n \in S$ cannot be changed by $n' \notin N \setminus S$.

Proof. Let n be a node in N. Let $A = drc(n)$ obtained by DRC(S), and let $B = drc(n)$ obtained by DRC(N). The computation of drc for every node n depends on the computation of PRO. Since $PRO(n)$ computed on N is equivalent to $PRO(n)$ computed on S, it follows that the computation of $drc(n)$ is also equivalent for the network N and its fragment S. □

Removal of Nodes With Definite Routing Choices

After the computation of definite routing choices, the network size can be reduced. We remove the nodes with definite routing choices from the network, and add their definite announcements to the queues of their neighbors based on their export policies. Thus, the initial configuration of the network after the reduction is equivalent to the configuration obtained after applying the procedures of Dest and the nodes with definite routing choices from the initial configuration. Let r be a run that starts from the regular initial configuration on a network without the reduction of definite routing choices. Let r' be the corresponding run (with similar attacker's actions) on a reduced network that starts from the corresponding initial configuration. Then $stable(r)$ and $stable(r')$ are equivalent.

3.6.3 Routing-preserving Path

Another source of reduction is the abstraction of routing-preserving paths.

A path $\pi = (n_1, \ldots, n_k)$ is *routing-preserving* if for every run r of N, in the final (stable) configuration of r one of the two cases holds: either for all $1 < i \leq k$, n_i chooses to route through n_{i-1}, or for all $1 \leq i < k$, n_i chooses to route through n_{i+1}.

Intuitively, for every run of the protocol, the nodes on a routing-preserving path all agree on the same route to the destination. As a result, we can replace such a path with a single node (an *abstraction* of the path) without changing the routing of other nodes in the network. The protocol of an abstract node is adjusted such that it exports announcements with lengths that match the number of nodes in the path it represents. An example of a routing-preserving path in Figure 3.1 is $(2, 3, 9)$.

To find routing-preserving sequences in the network, the following steps are applied iteratively until no more abstractions can be found.

- For each regular node n_1 with exactly two neighbors n_2, n_3 such that at least $n2$ is regular (not an originator), check if all conditions below hold:
  1. n_1 is a provider of n_3
  2. n_1 is a customer of n_2
  3. For each potential routing next-hop n' of n_2 such that $n' \neq n_1$, n_2 is peer or customer of n'.
  4. $n_1, n_2 \notin LoopNodes$

5. There is no link between n_2 and n_3.

- If the previous step satisfied all conditions, n_1 is eliminated and can be represented by n_2. n_2 is connected to n_3 by the same link type. (That is, n_2 is a provider of n_3).

In a single step of the abstraction, $n2$ becomes an *abstract node* that represents the sequence of nodes n_2, n_1.

The abstraction is applicable only for nodes that are not in $LoopNodes$. Any node in $LoopNodes$ should remain concrete in the network topology and cannot be abstracted. Thus, it is guaranteed that the abstracted nodes never reject route announcements originated by the attacker. Note that if n_1 is in $LoopNodes$, then there may be a case where n_1 does not agree with n_2 on the chosen routing path to the destination. If n_2 chooses to route via its neighbor n' over the routing path $\langle \pi, length, LN \rangle$ such that $n_1 \in LN$, then n_1 will reject this path and its routing choice will not be similar to the choice of n_2. Similarly, if n_2 is in $LoopNodes$, then there may be a case where n_2 does not agree with n_1 on the chosen routing path to the destination. If n_1 chooses to route via its neighbor n_3 over the routing path $\langle \pi, length, LN \rangle$ such that $n_2 \in LN$, then n_2 will reject this path and its routing choice will not be similar to the choice of n_1.

In the following we formally define the abstraction for a single step. It can be generalized to a sequence of steps (where the abstract node can represent any finite number of concrete nodes).

Definition 3.22. Let (n_2, n_1, n_3) be a sequence of nodes in the network graph that satisfies the above conditions. Then, (n_2, n_1) is a *routing-preserving path*. After the abstraction step is applied, the node n_2 is an *abstract node* that represents the concrete sequence of nodes (n_2, n_1).

The protocol of the abstract node n_2 is adjusted as follows:

- If n_2 receives a routing update from n_3 of the form $\langle \pi, length, LN \rangle$, then it stores in its RIB the adjusted path $\langle \pi, length + 1, LN \rangle$. That is, the length is increased by 1 because in the concrete model the routing update is actually exported to n_1 before it is exported from n_1 to n_2. If n_2 chooses to use this routing update, it exports the adjusted routing update with length: $length + 2$.

- If n_2 receives a routing update from another neighbor n' that is not n_3 of the form $\langle \pi, length, LN \rangle$, then it stores in its RIB the path $\langle \pi, length, LN \rangle$. If n_2 chooses to use this routing update, it exports to n_3 the adjusted routing update with length: $length + 2$. That is, the length is increased by 1 because in the concrete model the routing update is actually exported to n_1 before it is exported from n_1 to n_3.

Correctness: Since n_1 has two neighbors n_2, n_3, it may only route via n_2 or n_3. Additionally, based on the export policy and the link types, it is guaranteed that n_1

exports its routing choice to its other neighbor. That is, if n_1 chooses a route via n_2 it exports it to n_3. Additionally, if n_1 chooses a route via n_3 it exports it to n_2. We will show that in any case, the nodes of the abstracted sequence n_2, n_1 agree on the same routing path to the destination:

- If the abstract node n_2 routes via n_3, it means that in the concrete network the node n_3 exports its routing path to n_1. Since n_1 is the provider of n_3, it is guaranteed that n_3 chooses to route via this path and exports is to n_2 (based on the normal preference and export policies). Since n_2 is the provider of n_1 (and is only peer or customer of its other neighbors), it is guaranteed that n_2 chooses to route via this path. Thus, in this case both n_2, n_1 agree on the same path to the destination (which is via n_3).

- If the abstract node n_2 routes via n' such that $n' \neq n_3$, it means that n_2 exports this path to n_1 (since n_2 is the provider of n_1). It is guaranteed that n_1 chooses to route via n_2. If n_1 had another routing path it could choose, it would have been via its other neighbor n_3. However, if n_1 could route via n_3, then both n_2 and n_1 would choose this route (as shown in the previous case). Since in this case n_2 routes via another neighbor n', it follows that n_3 does not export any path to n_1. Thus, in this case both n_2, n_1 agree on the same path to the destination (which is via n').

- If the abstract node n_2 has no routing path to the destination, it means that in the concrete network n_3 does not export a routing path to n_1 and every neighbor n' of n_2 does not export a routing path to n_2. Thus, in this case both nodes n_2, n_1 do not have a routing path to the destination.

Therefore, in any case, the nodes of the abstracted sequence all agree on the same routing path to the destination. In the final configuration, the nodes n_2, n_1 are attracted by the attacker if and only if the abstract node n_2 is attracted by the attacker. The abstraction thus allows reducing the number of nodes in the network by adjusting the protocol of the relevant nodes as described above.

3.6.4 Example of a Network Reduction

In this example we revisit the example from Section 3.6.1 where the fragment extraction was demonstrated, and complete it with the additional reduction of definite routing choices and the abstraction of a routing-preserving path.

To demonstrate the complete reduction process consider the 10-nodes-sized randomly chosen sub-network from the Internet, presented in part A of figure 3.8. The grey node 48685 is the attacker. The yellow node 22561 is the destination. In practice the initial network can be much larger.

- Applying fragments computation results in:

1. Initialization: Insert O and their neighbors.
   $S = \{22561, 48685\} \cup \{209, 25934, 6677\}$
2. Add $c = 3257$, due to valid path from $o = 22561$ to $n = 6677$:
   $(22561, 209, 3257, 6677)$.
3. Add $c = 5580$, due to valid path from $o = 22561$ to $n = 25934$:
   $(22561, 209, 5580, 25934)$.

The remaining nodes are not added. For example, 3303 does not appear on any valid path in the original network, and is therefore dropped during the construction of a self-contained fragment. After applying this phase we remain with 7 nodes as presented in part B of figure 3.8.

- The reduced network contains ASes with definite routing choices: 209 , 25934, and 5580. These nodes cannot be attracted by the attacker in any case. Part C of figure 3.8 shows the resulting network. The arrow represents the definite export sent by 209 to 3257 on a peer link. The other removed nodes do not export their definite routing choices to any remaining node.

- The nodes $3257, 6677$ are a routing-preserving sequence and thus can be represented by a single abstract node, as presented in the resulting network D of Figure 3.8.

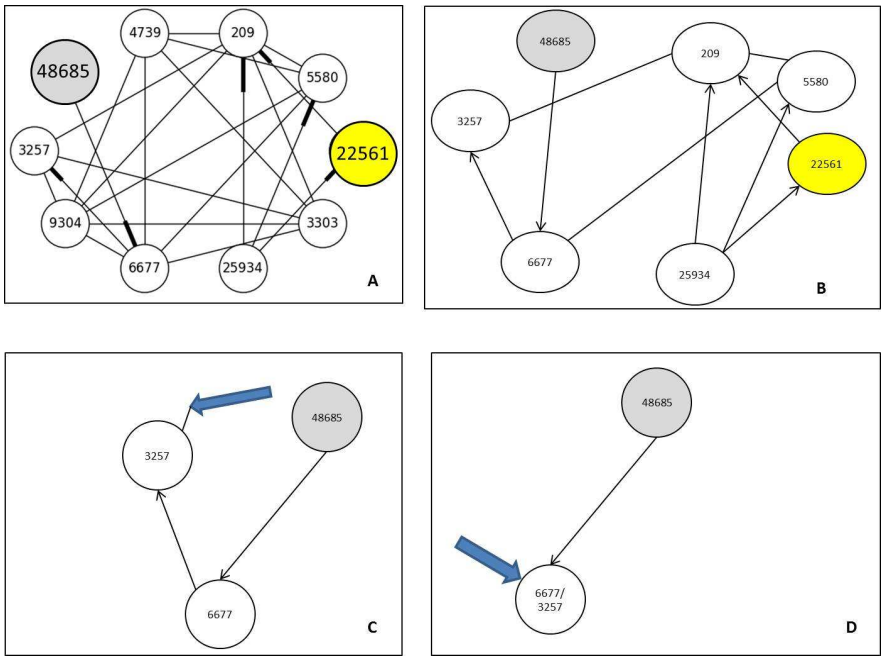

Figure 3.8: Reductions example

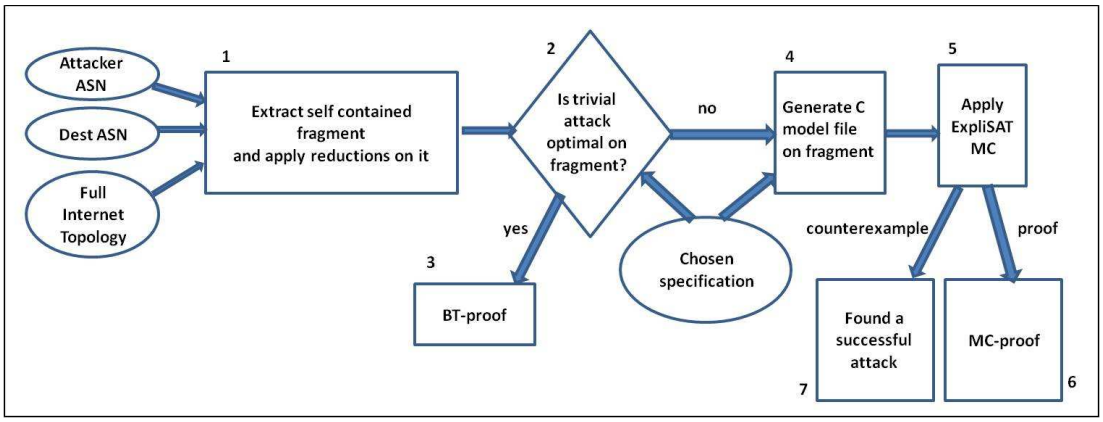

Figure 3.9: The BGP-SA Method

3.7 The BGP-SA Method

Our suggested method, called BGP-SA, for *BGP Security Analysis*, uses reductions and model checking to apply a formal analysis of BGP attraction attacks on a large network topology. We use model checking to perform a systematic search for traffic attacks. A systematic search is essential in order to reveal non-trivial attraction strategies on topologies from the Internet. It has a major advantage over simple testing techniques that randomly search for attacks. The model checker we use can perform full verification, thus it can also prove that no traffic attack is possible under certain conditions.

The BGP-SA method is composed of several stages, as depicted in Figure 3.9. Below we describe them in details.

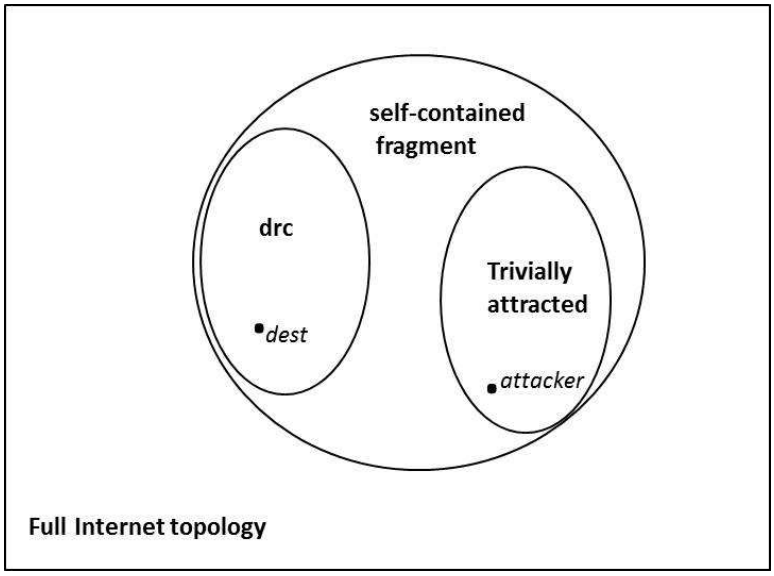

Figure 3.10: Partition of Node Types in the Extracted Fragment

3.7.1 Reducing the Network Topology

The input to the BGP-SA method consists of the full network topology, the chosen attacker and destination ASN, and the chosen specification. Given this input, we first extract a self-contained fragment and apply additional reductions and abstractions. (see square 1 of Figure 3.9). The extraction and reduction algorithms are explained in Section 3.6. The output is a reduced fragment that contains the nodes within the extracted fragment S, without those for which *drc* is defined. (See Figure 3.10).

3.7.2 Simulating the Trivial Attack

Here we explain items 2-3 of Figure 3.9. Given a reduced fragment, we run a simulation of the trivial attack on it. If the chosen specification is traffic attraction and if all the nodes in the reduced fragment are trivially attracted, then the attacker cannot improve its attraction results. If the chosen specification is traffic interception and if the trivial attack satisfies the interception condition additionally to attracting all nodes in the reduced fragment, then again the attacker cannot improve its attraction results.

In both cases it is considered a proof (denoted *BT-proof* for Best Trivial attraction proof) that within the fragment the attacker does not have a strategy which is better than the trivial one. When BT-proof is obtained, the analysis is terminated and model checking is not needed. Otherwise, the nodes of interest for searching attraction scenarios are the remaining nodes that are neither trivially attracted nor have a defined *drc*, as presented in Figure 3.10.

3.7.3 Generating the C Model

Given the reduced fragment and the chosen specification, we generate a model written in C on which the analysis is applied (see square 4 of Figure 3.9). Code 3.1- 3.3 depicts a pseudo-code of the generated code in high level, and below we give more details of it.

- Code 3.1 describes the procedures that implement nodes in our model. AS_Proc is the procedure of a regular AS. Its path preference and export policy are as explained in Section 3.3. The attacker has two procedures: $Arbitrary\_Attacker\_Proc$ is the procedure of an attacker that originates arbitrary path announcements and sends them to arbitrary neighbors. $Trivial\_Attacker\_Proc$ is the procedure of an attacker that applies the trivial attack and announces itself as the destination to all its neighbors. $Dest\_Proc$ is the procedure of $Dest$, in which it announces itself as the destination to all its neighbors.

- Code 3.2 describes the function implementing a BGP run in our model. The input parameter of this function is the type of run: normal – where the attacker acts as a regular AS, trivial – where the attacker applies the trivial attack, or arbitrary - where the attacker acts arbitrarily. The function is composed of a loop, where

at each loop iteration each one of the AS procedures is activated once. A stable configuration is achieved when no message is sent by any AS and all the queues are empty. The function returns the routing results at the stable configuration which include $chosen(n)$ for each node n in the network, where $chosen(n)$ is the preferred route of n.

- Code 3.3 describes the main function in the model and the assertion statement that implements the specification. The main function is composed of three calls to the function BGP_run, with the three types of run: normal, trivial, and arbitrary. The routing results of the three runs are saved. Then, to implement the attraction specification, a Boolean flag is set true if there exists some victim that is attracted by the attacker only in the arbitrary run, and not in the normal and trivial runs. The assertion requires that this Boolean flag is false. Therefore, if the assertion is violated, the violating run represents a successful attraction attack. To implement the interception specification, a constraint that the attacker has a routing path to the real destination should be added.

Code 3.1: Node Procedures

```
AS_Proc(){
        check incoming announcement and set chosen path;
        if(chosen path was changed)
                export new chosen path;
}
Arbitrary_Attacker_Proc(){
        Path p = nondeterministic_path();
        Neighbors G = nondeterministic_neighbors();
        foreach(n in G)
                send p to n;
}
Trivial_Attacker_Proc(){
        //attacker pretends to be dest
        Path p = <attacker>;
        send p to all neighbors;
}
Dest_Proc(){
        Path p = <dest>;
        send p to all neighbors ;
}
```

Code 3.2: BGP Run

```
enum RunType {normal, trivial, arbitrary};
typedef  Map<Node,Path>
Routing_Results ;
Topology fragment;

Routing_Results BGP_run(RunType type){
        clear AS states;
        Dest_Proc();
        while(!stable_state())  {
                for(AS in fragment){
                        if (AS is attacker and type == trivial)
                                Trivial_Attacker_Proc();
                        else if(AS is attacker and type == arbitrary)
                                Arbitrary_Attacker_Proc();
                        else
                                AS_Proc();
                }
        }
        return routing results; //chosen paths of all nodes
}
```

Code 3.3: Main Function with Attraction Specification

```
Routing_Results results[3];
int main(){
        results[normal] = BGP_run(normal);
        results[trivial] = BGP_run(trivial);
        results[arbitrary] = BGP_run(arbitrary);
        bool isSomeVictimAttracted = false;
        for(AS in fragment){
                if(AS routes via attacker in arbitrary run and not
                in normal and trivial runs)
                        isSomeVictimAttracted = true;
        }
        assert(!isSomeVictimAttracted);
}
```

3.7.4 Applying Model Checking to the Implemented Model Using ExpliSAT

Here we explain squares 5-7 of Figure 3.9. After the C code of the model is generated on the fragment, we apply model checking using IBM's model checking tool ExpliSAT [22].

The model checker systematically scans all possible execution paths of the C program. If it finds a run that violates the assertion, it returns a counterexample that represents a successful attack. If the model checker terminates without any counterexample, it is considered a proof that our attacker cannot perform the specified attack on the fragment. This is denoted as *MC-proof*.

3.8 Experimental Results

We applied our BGP-SA method on Internet fragments and used IBM's model checking tool ExpliSAT [22] to search for traffic attacks. The model checker can run on multiple cores. The experiments were performed on a 64-cores machine with AMD Opteron(tm) Processor 6376, 125GB RAM, and 64-bit Linux.

ExpliSAT Model Checker

ExpliSAT [22] verifies C programs containing assumptions and assertions. To use ExpliSAT we implement our model in C. Our specifications are negated and added as assertions on stable states. The model checker returns a counterexample if there is a violating run, and it can also perform *full verification* and automatically prove that no violating run is possible.

ExpliSAT combines explicit state model checking and SAT-based symbolic model checking. It traverses every feasible execution path of the program, and uses a SAT solver to verify assertions. It performs as many loop iterations as needed, and therefore full verification is possible and no loop bounds are required.

	Fragment size (#nodes)	**Reduced size (#nodes)**	**Trivial attraction (#nodes)**	**Specification**	**Result**	**Time (min)**	**Dest ASN**	**Attacker ASN**
1	16	11	9	attraction	BT proof	-	31132	16987
2	17	6	4	attraction	BT proof	-	9314	7772
3	22	10	8	attraction	BT proof	-	11669	36291
4	29	9	5	attraction	MC proof	1.5	29117	15137
5	15	13	10	attraction	MC proof	1	12431	18491
6	36	18	7	attraction	MC proof	17	19969	13537
7	69	27	17	attraction	MC proof	340	8296	20091
8	15	13	invalid	interception	counterexample	0.1	12431	18491
9	28	10	invalid	interception	counterexample	0.5	19361	32977
10	80	48	invalid	interception	counterexample	13	9218	43571
11	81	31	invalid	interception	counterexample	9	37177	40473
12	114	30	invalid	interception	counterexample	18	36040	29386
13	71	68	65	interception	N/A	>12h	30894	1290
14	10	-	4	interception	counterexample	0.1	-	-

Table 3.1: Results of BGP-SA Application on Fragments Extracted from the Full Internet Topology

3.8.1 Results on Internet Fragments

We performed experiments on self-contained fragments extracted from the full Internet topology. The ASes links from the Internet are from [19] and are relevant to October 2014.

Table 3.1 presents the results of applying our method. The fragments in lines 1-13 are based on randomly chosen destination and attacker from the Internet, with the exception of line 12 which is obtained by choosing the attacker and destination according to a recent attack where Syria attracted traffic destined to Youtube [63]. Line 14 is explained in Section 3.8.2. The first two columns specify the number of nodes in the extracted self-contained fragment and in the reduced fragment. The third column specifies the number of nodes attracted by the attacker on the trivial attack. The value is invalid if the specification is interception and the trivial attack does not satisfy the interception condition, by which the attacker should have an available routing path to the destination. The specification we used for each instance appears on the fourth column, and is either attraction or interception, which correspond to the specifications defined in Section 3.4.4. Note that in the interception specification, if the trivial attack fails to satisfy the interception condition, we only compare the attraction to the normal outcome. The result column specifies any of the possible results that are described in Section 3.7. The N/A result describes ExpiSAT runs that did not terminate. The last two columns specify the chosen ASN from the Internet of the destination and attacker nodes, from which the fragment was extracted.

We analyzed the normal outcome of BGP on each of the fragments and found that normally the attacker could not achieve any attraction at all, or only attraction from one or two nodes that were also trivially attracted. Therefore, we do not specify the attraction of normal outcome in the table.

The experiments show that the reductions we apply are significant. The simple BGP simulations of the trivial attack allow us to avoid applying model checking on fragments in which the attacker manages to achieve optimal attraction results by the trivial attack.

When we used ExpliSAT with the attraction specification, we got proofs that no better attack strategy exists. It can be explained by the fact that the trivial attack strategy can be considered most efficient in many cases. Consider for instance line 4 on which we got a proof by ExpliSAT. It should be noted that 2 nodes in the fragment are not trivially attracted and do not have definite routing choices, but still there is no attack strategy capable of attracting traffic from them. Thus, these two nodes are also considered *safe*, in addition to the nodes with definite routing choices.

For the interception instances in lines 8-12 the trivial attack failed to achieve the interception goal and ExpliSAT found simple interception attacks. In line 10 for instance, the attacker exported announcements to all its neighbors, such that one of its neighbors appeared on the announced path and thus was rejected by it (as explained about AS policy in Section 3.3). That neighbor allowed the attacker to have an available path to the destination. The attacker managed to attract traffic from 3 nodes, where in the trivial attack it managed to attract 4 nodes, but had no route to the destination. In lines 8 and 9, the attacker simply did not export announcement to one of its 3 neighbors. It achieved attraction of 1 nodes and 7 nodes, respectively. In the trivial attack it

managed to attract 16 nodes and 10 nodes, respectively. Line 12 was performed on a fragment from a recent attack [63]. The fragment reduction was significant in this case. We found that the trivial attack attracted 12 nodes but did not satisfy the interception condition. The model checker found an attack strategy that achieved interception and attracted 11 nodes. The attacker sent false announcements to 3 of its 4 neighbors in the found interception attack. In line 13 the trivial attack achieved the interception goal. ExpliSAT searched for a better interception strategy, but its run did not terminate.

3.8.2 Example Demonstrating Model Checking Advantages

Here we explain line 14 in the table. The network is taken from Figure 3.1. The network is a variation of the one presented in [35], where the goal was to show a non-trivial interception attack. We did not apply our reductions on this network topology.

In the normal outcome and trivial attack, the attacker fails to attract traffic from $AS8$. In the attack strategy suggested in [35] the attacker avoids exporting its path to $AS2$, and only exports it to $AS7$. The result is that $AS7$ chooses a shorter path directly via the attacker, and as a result $AS8$ prefers this shorter path. Thus, the attacker manages to apply traffic interception on $AS8$.

Line 14 of Table 3.1 specifies the experiment we performed on this topology with our BGP model. ExpliSAT automatically found a counterexample with greater attraction. It returned a counterexample in which the attacker exported announcements both to $AS7$ and to $AS2$. The announcement exported to $AS2$ contained $AS9$ on the sent path. Therefore, $AS9$ ignored that announcement, and did not export it to $AS7$. Thus, $AS7$ chose the shorter path via the attacker. Eventually, the attacker managed to achieve attraction from $AS8$, $AS2$, and $AS3$. Note that with the strategy suggested by [35] only $AS8$ is attracted. An alternative attack that could attract even more nodes to the attacker is to export to $AS2$ an announcement that contains $AS7$ instead of $AS9$ on the sent path. That way it can achieve attraction from $AS9$ as well.

From the above analysis we may conclude that by sending an announcement that creates a loop an attacker can better control on where the propagation of some path should be blocked in order to achieve better attraction results.

It should be noted that some versions of BGP are more secure [39] and may prevent the attacker from sending paths that do not exist in the network. On such versions the attacker cannot apply the loop strategy. Therefore, the loop strategy may have an advantage over the no-export strategy only in the absence of certain BGP security mechanisms.

Note that applying the fragment extraction and reductions would prevent from getting the counterexample. However, by extending the specification and defining that a scenario in which some node is routing via the attacker through a shorter path is also considered a successful attack, we were able to find that counterexample on the reduced topology as well. That shorter routing path can potentially attract more nodes

from outside the fragment. Given the counterexample, a simulation can be applied on a larger topology. In our case, the counterexample reveals that the routing path of $AS7$ via the attacker can be shortened with respect to its length in the trivial attack, and a simulation of the found attack on the larger topology reveals that $AS8$ is a new attracted node as a result.

3.9 Conclusion

In this work we propose a method to reveal possible attacks on Internet routing or prove that certain attacks are not possible. We develop substantial reduction techniques that enable to apply model checking in order to formally analyze BGP traffic attacks on the Internet. The use of model checking has a major advantage due to the systematic search, by which it can reveal unexpected or more sophisticated attacks. This is demonstrated in Section 3.8.2, where during an experiment that was done to reconstruct a known attack, the model checker automatically found a different attack strategy that achieved better attraction results than expected.

One obvious implication of our work is a better understanding of the vulnerability of the Internet to traffic attacks. Nonetheless, our suggested method can also be practical and useful for a network operator to increase its resilience to such attacks. In some cases a network operator may fear a traffic attack from potential attacking ASes. For example, telecommunication companies may fear their traffic be attracted by ASes that belong to adversary governments. Such governments can exploit these attacks in order to eavesdrop on traffic of consumers of those telecommunication companies. In such cases, the network operator can use our method in order to discover the identity of the ASes which the attacking AS can not attract traffic from. Once these *safe* ASes are known the network operator may form links to these ASes and prefer routes announced by those ASes, thereby eliminating the chances to be attracted by the attacker.

In scenarios where the network operator cannot link or prefer routes announced by safe ASes the network operator can use our method to choose to link to specific ASes that will force the attacking AS to attract traffic from many ASes in order to attract traffic from the network operator's own AS. In such a case, the footprint of the attack would be substantially increased and therefore the chances of attack being noticed and stopped also increases. In such a way the network operator will be able to increase its robustness to attacks.

3.9.1 Possible Directions for Extensions

Below we refer to several possible extensions of our method and modeling of BGP that may allow capturing additional aspects in the security analysis:

- Modeling more secure BGP variants: We used a BGP variant without any validation mechanisms of routing announcements. There exist more secure BGP

variants [35]. For example, with origin authentication it is verified that the last AS node on an announced path indeed contains the target network using a trusted database. Thus, if the attacker originates a false path where the last AS is not Dest, the regular nodes will ignore this path. Another example for a more secure variant is the Secure Origin BGP. In this version an announced path is validated to physically exist in the AS level topology of the network, using a trusted database. A more secure version is the Secure BGP variant, where path verification is used. It guarantees that an AS A can announce a path to its neighbors only if A received this path from one of its neighbors.

The model can be adjusted to reflect such secure variants by limiting the attacker capabilities. Given a validation mechanism, we can limit the attacker to originate routing announcements that are not ignored by regular nodes as a result of the validation mechanism.

- Using additional specifications: In Section 3.4.4 we suggested several types of specifications. It is possible to extend the analysis with some specifications that we did not use and implement in our method. For example, the shorten-path specification allows finding nodes within a fragment which can potentially attract more traffic from the non-modeled nodes outside the fragment. The quantitative specification allows finding scenarios where an attacker can achieve attraction which is greater than some given bound.

- Modeling more than one attacker: We used a single attacker node in our analysis. It can be extended to include multiple attacker nodes that can cooperate. The analysis can be used to find whether a group of attackers may have more traffic attraction capabilities with respect to the capabilities of each attacker separately.

Chapter 4

Formal Analysis of a Black Box OSPF Implementation

4.1 Preliminaries

The Internet owes much of its success to open standards. These standards are being developed in an iterative, rigorous and open process. They are a fruit of extensive deliberations, trial implementations and testings. Furthermore, open standards are thoroughly documented and freely available, so they can be readily scrutinize at any time even after their creation. It is generally believed that open standards led to a more robust and secure Internet.

In stark contrast to the open nature of Internet standards, the Internet infrastructure predominantly relies on proprietary and closed-source devices, such as routers and switches, made by large vendors like Cisco, Juniper and Huawei. A device's vendor can add, remove or alter the standardized functionality of a protocol as it sees fit, as long as interoperability to other vendors' implementations is preserved. In some cases, even this interoperability is not entirely kept. It is not uncommon to have two network devices of different vendors that cannot co-operate seamlessly (e.g., [31]). Possible motivations of the vendors' deviations from the standardized functionality vary ([8]): development cost reduction, optimization of the protocol functions, or supplemental of features that add value to the vendor's customers. Additionally, inadvertent deviations may rise due to misunderstanding of the standard or negligence to completely implement the standard.

Due to the closed-source nature of network devices the networking and security community do not have clear visibility to these functional deviations. Identifying them is crucial in order to assess their full impact on a network's resiliency, efficiency and security. To address this problem we leverage formal analysis methods that assist in identifying deviations of a network protocol implementation from its standard. Our analysis is black box, that is, can be conducted without requiring access to the implementation's source code or binary code. We only assume the ability to send packets to the device and observe its external behavior. This behavior includes the packets sent by the device

and information that is explicitly available through its user interface. Due to the black box analysis nature of our method, it can be readily applied to any network device with minimal changes.

We propose to use a model-based testing approach [11, 66]. Generally speaking, in this approach a reference model of a system under test (SUT) is formulated. The model embodies the desired functionality for that system. Based on the model tests are generated, usually automatically. Each test has a desired outcome as determined by the model. The tests are then executed against the SUT and the resulting outcome is compared to the desired result. In our case the model is formally defined according to the standard of the protocol. The SUT is the network device's implementation of the protocol. A failed test indicates a deviation of the implementation from the standard. We use concolic execution [34, 57] to systematically generate tests from our model. Symbolic execution allows analyzing the execution paths of a program and generating corresponding test cases. Concolic testing is a dynamic symbolic execution technique to systematically generate tests along different execution paths of a program. It involves concrete runs of the program over concrete input values alongside symbolic execution. Each concrete execution is on a different path. The paths are explored systematically and automatically until full coverage is achieved.

The model-based testing approach has been successfully employed to find bugs in open source software. Moreover, there have been positive attempts to employ the approach for finding bugs in open-source implementations of one-to-one network protocols such as TCP and UDP [16]. However, many network protocols may involve multiple participants. Examples of such protocols are routing protocols [50, 46], spanning tree protocols [6], and VoIP protocols [28, 5]. Due to scalability issues the use of model-based testing has been inhibited in the realm of these complex, multi-party protocols. The functionality of such protocols depend on the dynamics between the participants, their relative locations in the network, and the role each participant takes as part of the protocol. Such protocols may expose parts of their functionality only in specific complex interactions between the protocol's participants. Therefore, the number of tests that verify the protocol functionality may be prohibitively high.

To cope with the scalability issues of model-based testing of complex multi-party protocols we propose practical optimizations that significantly reduce the number of generated tests without loss of functionality cover of the model. Our main optimization merges different tests that pass through a joint intermediate state. Namely, we merge two long test scenarios that reach the same intermediate state into a single shorter test scenario that starts from the intermediate joint state. This optimization is especially useful for test scenarios in which multiple packets are sent. For example, consider two non-identical sequences of packets, $P1$ and $P2$, that are sent during two test scenarios, $t1$ and $t2$, respectively. Assume that the model ends up in the same final state following each of the two tests. Therefore, a test having a sequence of packets of the form $P1||P$ (the sequence of packets in $P1$ followed by a sequence of packets in P) can be merged

with a test that has the sequence $P2||P$. The merged test shall have a sequence of packets P and it should be executed from initial state that is identical to the intermediate joint state of the original two tests.

The method we propose allowed us to implement the first practical tool to identify deviations of black box implementations of one of the most complex multi-party protocols on the Internet – the OSPF routing protocol [50]. The OSPF protocol is a widely used intra-domain routing protocol which is deployed in many enterprise and ISP networks. We searched for deviations in three versions of Cisco's implementation of OSPF in IOS[1]. The later version was released in 2016. In total, we found 7 significant deviations. Most of them compromise the security and resiliency of the network. The deviations were acknowledged by Cisco to exist in the versions we tested.

4.1.1 Background in Symbolic Execution

Symbolic execution [18] allows analyzing the execution paths of a program and generating corresponding test cases. The input variables of the program are defined as symbolic variables. Then, the program is symbolically run, where symbolic expressions represent values of the program variables. On each execution path a path-constraint is obtained by collecting all the symbolic expressions that correspond to conditional branches on that path. The path-constraint is a quantifier-free first-order formula over the symbolic variables. Its solutions form a set of concrete values of the input variables for which the program runs via the same execution path. A test that covers this path is then derived from this solution, containing concrete values of the input variables.

Concolic testing ([34, 57]) is a dynamic symbolic execution technique to systematically generate tests along different execution paths of a program. It involves concrete runs of the program over concrete input values alongside symbolic execution. Initially, some random concrete input values are chosen. During a run of the program with this input, symbolic constraints are gathered over the conditional branches of the current execution. Thus, at the end of the run the symbolic path constraint is obtained. A constraint solver is then used to construct the next concrete execution on a different path. This can be achieved, for instance, by negating the last conjunct on the path-constraint not already negated. A new solution for the variant of the path constraint with negations should necessarily steer a new concrete execution over a different path. This process is repeated systematically and automatically. Finally, the process terminates based on some time limit, coverage criteria, or when full coverage is achieved.

4.1.2 OSPF Background

OSPF (Open Shortest Path First) is a widely used intra-domain routing protocol, namely used within collections of networks, each of which is called an autonomous

[1]Cisco's IOS is a software family that implements all networking and operating system functionality in many of Cisco's routers and switches.

system. We focus on the most popular version of the protocol – version 2 [50]. It is used in many autonomous systems (AS) on the Internet.

A detailed background of the OSPF protocol is given in Section 2.7.1 of Chapter 2.

4.2 Black Box Analysis Procedures

Our goal is to find deviations of an OSPF implementation from the the protocol's standard functionality as defined in [50].

We use a model-based testing approach by modeling central parts of the OSPF protocol based on its standard. We use a concolic testing tool, named mini-mc [1], to generate test cases that cover our model's execution paths. The constraint solver it uses is z3 [2].

The OSPF model needs not be fully detailed. It may model parts of the protocol that are relevant to the current analysis and may abstract away details that are irrelevant. Thus, we may split the analysis according to several goals, each concentrating on a specific part of the protocol.

4.2.1 The Method Flow

The method flow is depicted in Figure 4.1. Below we explain each numbered box on that figure:

1. Given a fixed network topology the OSPF model produces a run of the protocol. Each execution path of the model is represented by a concrete run that starts from the *standard initial state* on the chosen topology. In the standard initial state the LSDBs of all routers are complete and consistent, containing all the LSAs originated by the routers in the network topology. The LSDBs correctly reflect the network topology view when they are consistent. During the run concrete LSAs are sent to the routers, and the run terminates in a stable state after no LSAs are sent between any routers on the network.

   The OSPF model is implemented in Python. The symbolic variables are determined in advance. A description of the model and the symbolic variables is given in Section 4.2.2.

2. Applying the concolic execution tool on our OSPF symbolic model generates a test file for each execution path of the model. Each test file contains the sent LSAs and the initial and final LSDBs content for all routers in the network. There is a finite number of execution paths due to the finite domain of the symbolic variables (explained in Section 4.2.2). Each execution path is finite and reaches a stable state in which no LSAs are sent.

3. Each generated test file is executed on the SUT. The script activates the routers, initializes them according to the model initial state, sends the LSAs from the

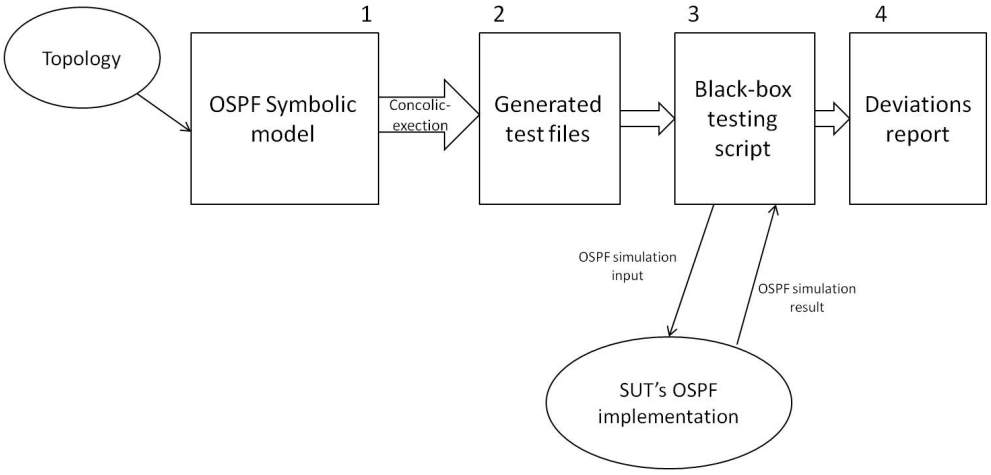

Figure 4.1: The flow of our method

input test file using Scapy [4], reads the LSDBs on the stable state and compares them with the expected LSDBs from the test file (previously obtained from the model). If any deviation is detected, it generates a report about the failed test. A detailed description of the black box testing script is given in Section 4.2.3.

4. A failed test represents a deviation between the SUT's OSPF implementation and the OSPF standard. Note that a found deviation may pose a security vulnerability in the SUT, but this may not necessarily be the case. It requires further analysis to infer what kind of effect the deviation has on security or correctness of the implementation. The analysis requires comparing the traces of all messages exchanged between the routers during the run of the test, both on the model and on the black box implementation. The model trace represents the expected behavior based on the standard. Thus, comparing the model trace with the implementation trace allows tracking and analyzing the different routers' behavior that causes the different final state.

   In order to avoid repeating deviation reports that follow from the same deviation and to find new deviations, we update the model, when possible, to reflect the found deviation. Thus, the final states of the tests generated from the updated model are reflecting the deviations that were already found on the tested implementation.

4.2.2 The OSPF Symbolic Model

We modeled the core parts of the OSPF protocol based on its standard and applied it on a fixed network topology. The model includes the following core functions: the LSA structure, the flooding procedure, fight-back mechanism, and the LSA purge procedure.

We leverage an OSPF model that we proposed in Chapter 2. The model was previously used in the context of model checking to find vulnerabilities. In this chapter we use it to generate tests by choosing symbolic variables and applying concolic execution methods. We then apply the generated tests on black box OSPF implementations. We extend the modeled functionalities of our OSPF model from Chapter 2 by adding the modeling of LSA purge procedure when the sequence number reaches to MaxSequenceNum. In the following, we will refer to our model as the OSPF *reference model*.

A run of the model simulates the protocol behavior, and returns the resulting LSDBs of the routers given the input LSAs to be sent from an initial state.

Below we detail the main aspects of the OSPF reference model that we implemented and used for the black box analysis.

The Modeled LSA Structure

An LSA in our model contains the following fields: LS type, Link State ID (LSID), Advertising Router (AR), Sequence Number (SeqNum), LS age, and Links List. The LS age is abstracted into a Boolean flag indicating whether it is MaxAge or not.

An LSA message contains the LSA itself, and in addition, the source and destination fields.

State

A state in our model is the set of LSDBs of all routers and the state of the routers' incoming queues. A state is considered *stable* when all routes' queues are empty.

The Symbolic Variables

We used symbolic variables within the LSAs that are sent, and within the LSDBs of the routers' initial states. Below we detail the symbolic variables and their domains.

- Each sent LSA message has the following symbolic variables:
  - Sequence number – the seqNum field is a number in the range $[0, maxSeq]$, where maxSeq is a predefined constant in the model.
  - Destination – the destination field is one of the routers within the chosen topology.
  - Advertising Router – the AR field is the router ID of any router within the network topology.
  - LSID – the LSID field is the router ID of any router within the network topology.

The symbolic LSA message is of the form:

$$\langle type, src, dest, LSID, AR, seq, links, age \rangle$$

The remaining fields that are not symbolic are concretely assigned as follows:

- $type = routerLSA$
- $links = []$, i.e, an empty list of links
- src corresponds to $dest$: The src field is determined deterministically based on the $dest$ field. We know in advance based on the given topology through which route the LSA arrives to its $dest$, and we have to set the src value to be equal the router ID that is the neighbor of $dest$ which forwards the LSA to it. Otherwise, the $dest$ router ignores that LSA.
- $age = 0$ (assigned as not MaxAge).

- The LSDBs initialization is based on the standard initial state. The sequence numbers of the LSAs are symbolically initialized with additional symbolic variables. Their range is $[0, k]$, where k a predefined constant smaller than MaxSeqNum.

The Main Function

The main function of the model has input symbolic arguments as described above. The run of the model starts with initialization of the LSDBs according to the standard initial state. Afterwards, for each symbolic LSA, the LSA is sent to its destination and then a loop is applied. On each loop iteration every router runs its procedure once.

The Router Procedure

When a router R receives an LSA, it checks if the LSA exists in its LSDB. If it does not exist or is considered newer than the existing instance, the router floods the LSA and updates its LSDB accordingly. If the LSA is self-originated, a fight-back is triggered. If the sequenceNumber of the fight-back LSA reaches the MaxSeqNum, the router originates an LSA with MaxSeqNum and MaxAge, and then a new LSA with InitialSeqNum. The MaxAge LSA triggers the other routers to purge R's LSA from their LSDBs.

Interleaving

The model simulates the routers run sequentially, in a round-robin scheduling.

When a single LSA is sent, the interleaving does not affect the final state. As long as only one 'external' LSA is being sent among the routers, all interleaving of subsequent LSAs result in the same final state. For multiple LSAs in the model we consider specific interleaving on which every LSA is sent separately in specific order. After each LSA is sent, the routers are activated until stabilization is achieved. Only then the next LSA is sent, and so on. Thus, for every number of sent LSAs, we can expect that an actual run on the SUT would terminate in a similar final state as in our model.

4.2.3 Black Box Testing of the Generated Tests

The black box testing script interacts with the SUT's OSPF implementation. The script's input is the set of test files to run that was generated in the previous stage. For each test file the script applies a corresponding test run on the SUT. The results of the run are compared with the expected results that are specified in the test file. After completing a run of a test file on the SUT and before applying a new run of another test file, the routers are restarted to avoid side effects from previous runs on the following runs. The output of the script is the list of failed tests for which the results of the SUT's run did not match the expected results obtained from the model.

Let $(mInitialState, mLSAs, mFinalState)$ be the specified initial state, sent LSAs, and final state within a test file obtained from the model. We detail below the significant procedures applied in this stage by the black box testing script to allow the comparison between the runs of the OSPF model and the SUT:

Initialization of the Routers

The initial state of the routers on the SUT when they are initially activated or after being restarted is the *standard initial state*, on which the LSDBs of all routers are complete and consistent, containing all of the LSAs originated by the routers in the network topology. The LSDBs correctly reflect the network topology view when they are consistent, and the routing tables of the routers are computed based on the LSDBs content. The sequence numbers of the LSAs are arbitrary in the standard initial state of the routers within the SUT.

Since $mInitialState$ is the standard initial state, the contents of the LSAs and routing tables of the routers within the SUT after restart should match the model initial state as specified in the test file. However, the sequence numbers of LSAs within $mInitialState$ have concrete values in terms of the model domains, whereas the LSAs of the routers on the SUT have arbitrary sequence numbers. We have to make sure that the sequence numbers of the routers' LSAs are consistent with those specified in $mInitialState$. This is because the initial sequence numbers of the LSAs may have an effect on the final state. Let $mSeq_A, mSeq_B$ be the initial sequence numbers of two routers A, B based on a test generated from the model, and let $seqA, seqB$ be the initial sequence numbers of the two routers on the SUT. It is expected that on the SUT the initial state would match as follows: $seqA - seqB = mSeq_A - mSeq_B$.

The OSPF implementation does not allow to manually set the sequence numbers of the routers' LSAs. Therefore, we artificially apply such initialization by sending each router a self-originated LSA of its own. For instance, if routers A and B have the following initial sequence numbers in $mInitialState$: $mSeq_A = 2, mSeq_B = 0$, and the arbitrary initial state of the routers' LSAs in the SUT contains the following sequence numbers: $Seq_A = 0x80000005, Seq_B = 0x8000000F$, then we send to A an LSA with $seq = 0x80000012$, and to B an LSA with $seq = 0x80000010$. Thus, finally after the

fight back, we have a state with $Seq_A = 0x80000013, Seq_B = 0x80000011$ that matches the initial state of the model.

Sending the Symbolic LSA

To send the LSA from the test file, we take into account the concrete values of the symbolic variables within the generated tests. We use Scapy to generate a corresponding LSA packet to be sent on the SUT. The sequence number of the LSA to be sent is based on the initial sequence number of the initial mathcing LSA on the SUT and of the sequence number of the sent LSA as specified in the test file. For example, consider a case where $mLSAs$ contains a sent LSA with the fields:$AR = 1, LSID = 1, seqNum = 3$. This means that on the test from the model, an LSA was sent on behalf of R_1 with $seqNum = 3$. In order to translate it to an LSA to be sent on the SUT, we need to consider the following values: let $mSeq_1 = 1$ be the initial sequence number of $R1$'s LSA on $mInitialState$, and let $Seq_1 = 0x80000005$ be the initial sequence number of $R1$'s LSA on the SUT after applying the initialization process in the previous stage. Then, the sequence number of the LSA to be sent on behalf of $R1$ on the SUT would be $Seq_1 + (seqNum - mSeq_1)$, or in that specific case: $0x80000005 + (3 - 1) = 0x80000007$.

Comparison of Matching States

To compare the final LSDBs from the model and from the test run on the SUT, we check that for each LSA in the model LSDB there is a matching LSA in the SUT's LSDB and vice versa. LSAs are considered matching if the following fields are matching: LS type, LSID, AR, Links. The links list is considered matching if for every link in the SUT there is a matching link in the model and vice versa. The sequence numbers specified in the final state of the generated test file are given as symbolic expressions in terms of the symbolic input variables. Thus, if the test file states that the final expected sequence of the LSA of R_1 is $symbR1Initial + 1$, then we check that the matching LSA from the SUT's concrete run has a sequence number that is larger by 1 from the initial sequence number of $R1$'s LSA at the initialization process. Note that a symbolic expression of a final sequence number in the model test file can be a function of symbolic variables that refer to sequence numbers within the sent LSAs, or of symbolic variables that refer to the sequence numbers within the initial LSAs of the routers.

4.2.4 Extending the Method to Multiple LSAs

In this section we describe our extension to the method above with multiple LSAs.

The naive approach to generate tests with more than one LSA would be to use several instances of symbolic LSAs in the model. However, the disadvantage is that the number of symbolic variables is multiplied in the number of symbolic LSAs. This may result in the path-explosion problem due to the exponential growth in the number

of program paths to execute. We observed this problem when we initially tried this approach on our experiments (see Section 4.3.3).

We noticed that the above approach may result in generation of many redundant similar tests along different execution paths. Let P_1 be the set of all model paths with one LSA, starting from the standard initial state as described in the previous section. Let P_2 be the set of all model paths with two LSAs, starting from the same standard initial state. Let $P_1(S)$ be the set of all model paths within P_1 that are terminating in the model state S. We note that all symbolic executions of the model that start from the same model state are equivalent, regardless of the prefix path leading to that state. Thus, instead of exploring similar execution paths from each final state of each path in $P_1(S)$, we can apply it only once by using the model state S as the initial state, and by using only one symbolic LSA from that state.

We suggest the following approaches to apply our method on multiple LSAs more efficiently:

- **Systematically Increasing Depth of Explored Paths:** a pseudo code describing the systematic extension is given in Figure 4.2. RIS is the set of reachable states from which exploration via concolic execution has not been applied yet. ERS is the set of explored reachable states from which the exploration of concolic execution has already been applied. For each reachable state we also keep a sequence of corresponding LSAs from which the reachable state is obtained. For example, consider the pair $(S, (LSA_1, LSA_2))$, where S is a reachable state and (LSA_1, LSA_2) are the corresponding LSAs. The notation means that if a run of the model starts from the standard initial state and these LSAs are sent one by one, eventually the final state observed on the model is S.

  In line 1 of Figure 4.2 we initialize the set of reachable initial states with the standard initial state. In line 2 we initialize the set of explored reachable states with the empty set. K represents the current depth of the generated tests. It is initialized to 1, since on the first iteration we generate tests with a single LSA sent from the standard initial state. In each loop iteration for a new depth, lines 6-10 are applied. In line 6 we apply our method from the previous section on every state in RIS with a single symbolic LSA. The generated tests are kept and sent to the black box testing script. In order to initialize the SUT based on the new initial state, we adjust the initialization process in the black box testing script. The adjustment requires sending the corresponding sequence of LSAs that are kept with the reachable state, as part of the initialization. Afterwards, in line 7, we analyze the generated tests from the previous stage. For each generated tests with K LSAs we detect its final state. The set of all reachable states (and their corresponding LSA sequences) from the generated tests are kept in RS. In line 8, we add to the set of explored reachable states the set of the reachable states from which the method was applied on the current iteration. Then, in line 9, the

```
1.   RIS = {standard}
2.   ERS = φ
3.   K = 1
4.   while (K < bound )
5.   {
6.      generated-tests = applyMethod(RIS)
7.      RS = extract-reachable-states(generated-tests)
8.      ERS = ERS ∪ RIS
9.      RIS = RS \ ERS
10.     K = K + 1
11.  }
```

Figure 4.2: Systematic extension algorithm

set of reachable states from which exploration of depth $K+1$ should be applied, is updated, by removing the set of all explored reachable states from the set of the detected reachable states of the last iteration. Finally, K is increased by 1 towards the next iteration.

We should note that each reachable state may actually have multiple sequences of LSAs leading to it in our model. We choose to use one representative of LSA sequence out of all possible sequences.

- **Arbitrary Number of LSAs:** In this approach we use a reachable initial state that may require an arbitrary number of LSAs, and then apply concolic execution with a single symbolic LSA. The arbitrary reachable initial state may be achieved by random simulation of the protocol with arbirary number of LSAs, starting from the standard initial state. This process requires adjustment of the initialization process in the black box script as well. The sequence of LSAs leading to the new reachable initial state should be provided. Let $(S, (LSA_1, LSA_2, ..., LSA_k))$ be the reachable state and its corresponding LSA sequence. Let $F_1, F_2, ..., F_n$ be the set of test files generated from concolic execution with S assigned as the model initial state and with a single symbolic LSA. When running the generated tests on the SUT, we should adjust the initialization by sending the sequence of initializing LSAs $(LSA_1, LSA_2, ..., LSA_k)$. Then, we should compare the state of the routers on the SUT with the given model state S and make sure that they are matching. Only then we can send the LSA from the generated test file F_i, and finally compare the final states of the SUT and the test file.

4.3 Evaluation

In this section we describe an evaluation of our method against Cisco's IOS implementation of the OSPF protocol. IOS is a family of software used on most Cisco's routers and

switches. It includes operating system functions as well as various networking functions including routing.

Our focus in this section is on Cisco's implementation, nonetheless our method can be similarly and quite easily applied to other OSPF implementations as well. To adapt our tool to a different implementation one would need only to change the commands issued to fetch the LSDB and routing table and parse their output as needed.

Although the deviations we found in the Cisco implementation is of interest by themselves, our main aim in this section is to verify that indeed the method we propose is efficient and practical for finding protocol deviations even in complex network standards.

4.3.1 Testbed

To test Cisco's OSPF implementation we used alternately two network simulation software: GNS3 [3] and VIRL [53]. Both software suites allow to simulate a network of multiple routers, each running an emulation of an actual IOS image (identical to the images used in real Cisco routers).

Throughout the evaluation we simulated the topologies depicted in Figure 4.3 (Topology 1) and in Figure 4.4 (Topology 2). Both topologies consist of the two link types we model: point-to-point and transit networks. Topology 2 is based on Topology 1 while adding extra links to introduce cycles in the topology graph. Topology cycles allow the same LSA to be received via more than one link, thereby exposing more functionality details in the LSA flooding procedure. However, note that the two chosen topologies are simple. Each contains only 5 routers and 5 or 7 links. This is with the explicit intention of showing the power of our method with regard to functionality coverage. The extensive coverage allows to unearth protocol deviations even in simple topologies that may seemingly do not expose the full complexity of the protocol.

We emulated the routers in those topologies using images of three stable IOS versions as detailed in Table 4.1. These versions were evaluated due to the large time gap between their release dates – 5 years in total. This time gap leads us to assume that the there are non-negligible changes in the code base between the three versions, even though the core functionality of the OSPF standard remained the same during this time period. The changes may be due to new proprietary features, optimizations of protocol functions or bug fixes. These changes in the code base allowed us to verify that our method indeed is capable of identifying different deviations in the different versions of the same vendor's implementation.

As noted above, during each test we crafted and sent packets using the Scapy software. The packets are received by the emulated network through the network interface of the entity called 'cloud 1' (see Figures 4.3 and 4.4).

The outcome of each test is the contents of the LSDB and the routing table of every router in the network. We extract this information by connecting to each router using a Telnet session and issuing the relevant CLI commands. In Cisco these commands

IOS Version	Release date
15.1(4)M, release software (fc1)	Mar. 2011
15.2(4)S7, release software (fc4)	Apr. 2015
15.6(2)T, release software (fc4)	Mar. 2016

Table 4.1: Cisco's IOS versions tested for deviations

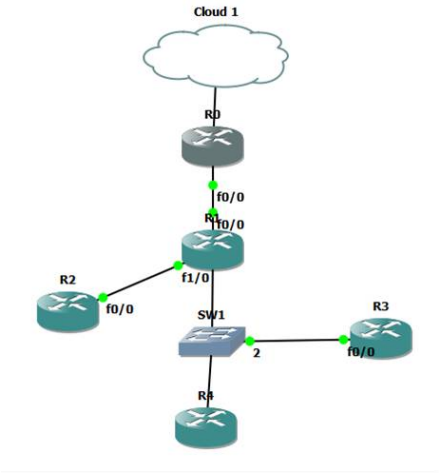

Figure 4.3: Topology 1

are `'show ip route'` to fetch a router's routing table, and `'show ip ospf database [router|network]'` to fetch the Router-LSAs or Network-LSAs, respectively, of a router's LSDB. We then parse the output of such commands and compare it to the expected results. Every test is preceded by a reset of all routers using the `'reload'` command.

4.3.2 Results With a Single LSA

Initially we applied our method with a single symbolic LSA. Table 4.2 summarizes the number of generated tests per each topology and model version. Note that the number of generated tests is reduced in versions 15.2, 15.6 compared to version 15.1. This is due to a fix applied in the newer versions, for which the model was updated. Thus, the

Topology	IOS veresion	# Generated tests	# Found deviations
Topology 1	15.1	395	7
Topology 1	15.2, 15.6	94	3
Topology 2	15.2, 15.6	104	2

Table 4.2: A summary of the number of generated tests and found deviations per each topology and IOS version with a single symbolic LSA

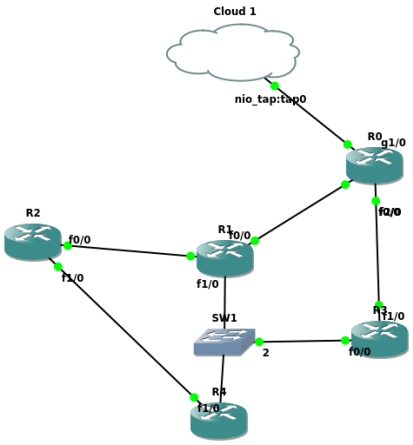

Figure 4.4: Topology 2

Table 4.3: Summary of found deviations on the three IOS versions, categorized by types

Deviation category	**15.1**	**15.2, 15.6**
Harmed Routing	{1,2,3,4,5,6}	{1,4}
Affected Stability	{4,5,6}	{4}
Non-vulnerability	{7}	{7}

number of model paths was reduced on the updated model that reflected the applied fix.

We found 7 deviations in version 15.1 with Topology 1. We consider 6 of them as security vulnerabilities. 3 of them were also reproduced in versions 15.2 and 15.6 with Topology 1, and the others were fixed in that version. In Topology 2 we tested the implementation of versions 15.2 and 15.6, and no new deviations were found. Two of the found deviations (1,4) were reproduced on this topology.

Table 4.3 categorizes the found deviations into several types according to the versions on which they were observed. We should note that the deviations in the category that affected stability included routers' behaviors on which two routers repeatedly sent similar LSAs to each other for many iterations.

Below is the detailed list of the found deviations and their description.

1. **Rogue LSA With Maximum Sequence Number:**

   - **Description**: A false LSA having the maximum sequence number was sent by unicast to a router R on behalf of R itself. The router originated an LSA with MaxSeqNum and MaxAge. However, it unexpectedly did not originate its LSA with InitialSeqNum. The routing tables of the other routers were affected due to that missing LSA of R.
   - **Comments:** This deviation is topology-dependent. The same behavior was

observed on the corresponding tests of $R0$, $R1$, and $R2$ (with $dest = LSID = AR = R0$ or $R2$), but on routers $R3$ and $R4$ the expected behavior was observed on Topology 1. However, on Topology 2, the same behavior was observed on all routers.

- **Impact:** This gap allows an attacker to send a spoofed LSA that persistently harms the routing in the network.
- **Status:** This deviation was acknowledged by Cisco. We found it in all IOS versions we tested.

2. **Inconsistent LSA With $LSID \neq AR$ Poisons LSDBs and Routing Tables:**

   - **Description**: A false inconsistent LSA with $LSID \neq AR$ was sent to a router R, where the LSID was equal to R's ID. The false LSA unexpectedly replaced the correct LSA of R on its own LSDB and on other routers' LSDBs as well. The routing tables of these routers were re-calculated based on that false LSA, and consequently no OSPF-derived route existed in their routing tables
   - **Sent LSA:**
     $\langle src = R0, dest = R1, LSID = R1, AR = R4, seq = n, links = [] \rangle$
   - **Detailed Description**: The false LSA sent to $R1$ had an LSID with $R1$'s ID and Advertising Router with $R4$'s ID. In the initial state the sequence number of $R4$'s LSA is less than n. In the final state the LSA originated by $R1$ is unexpectedly replaced with the sent LSA at the LSDBs of $R1, R2, R3$. Thus, these LSDBs remain poisoned at the end of this test. This is not in accordance with the OSPF specification which says that an LSA is identified by the fields $LSID, AR$, and type. The sent LSA should not have replaced the LSA of R1 since their AR values are different. $R4$ responds with a fight-back since the sent LSA contains $AR = R4$. Thus, its LSA's sequence number is increased and its value is $n + 1$ on the final state for all routers' LSDBs.
   - **Impact**: This deviation allows an attacker to send a spoofed LSA that persistently harms the routing in the network.
   - **Status:** This deviation was already known and was published in [52]. We found it in version 15.1, and it was fixed in versions 15.2, 15.6.

3. **Inconsistent LSA With Lower Sequence Number Causes a 'Fight-Back':**

   - **Description:** A false inconsistent LSA with $LSID \neq AR$ was sent to a router R, with $LSID = R$ and $AR = R'$. An unexpected fight-back LSA was originated by R', even though the sent LSA had lower sequence number than its own LSA. This is not in accordance with the OSPF specification

which says that a new LSA (a fight-back) should be sent in response to a self-originating false LSA only if that false LSA is newer than the current LSA (see Sec. 13.1 in the RFC).

- **Sent LSA:**
  $\langle src = R0, dest = R1, LSID = R1, AR = R2, seq = n, links = [] \rangle$
- **Detailed Description:** A false LSA having $LSID = R1$ and $AR = R2$ was sent to R1 from Cloud 1. This LSA has a sequence number n that was larger than the seq of $R1$s' initial LSA, but smaller than the seq of $R2$'s initial LSA. R1 floods the false LSA to all its neighbors including R2. R2 initially sends a fight-back LSA with a seq smaller than the seq of its own LSA currently installed in its DB (it simply sent an LSA that has a seq increased by 1 compared to the seq of the false LSA). This is not in accordance with the OSPF specification which says that a new LSA (a fight-back) should be sent in response to a self-originating false LSA only if that false LSA is newer than the current LSA (see Sec. 13.1 in the RFC). This is not the case in our test. As noted, the false LSA had a sequence number that is smaller than that of the current LSA. Eventually, R2 originates a new fight-back LSA with a seq that is increased by 1 compared to the LSA installed in its DB only after R1 sends to R2 the LSA with the updated seq.
- **Impact:** The impact is similar to the one described in the previous deviation, since the final state on both scenarios is similar. This scenario describes an additional deviation with respect to the previous one, but it has no additional effect on the calculated routing tables.
- **Status:** We are not aware of any report of this deviation in version 15.1. Due to the fix mentioned in the previous deviation, we did not reproduce this deviation on the later version, as expected.

4. **Incorrect MaxAge LSA origination during fight-back:**

   - **Description:**[2] A false LSA having the maximum sequence number was sent on behalf of some router R to another router R'. The origination of the MaxAge LSA by R' deviates from the specification of the protocol. In some cases this can result in non-stability of the routers, where R keeps sending the false LSA and R' keeps sending the MaxAge LSA.
   - **Impact:** This deviation allows an attacker to send a spoofed LSA that disrupts the routing in the network.
   - **Status:** This deviation was acknowledged by Cisco. We found it in all IOS versions we tested.

[2] Further details cannot be given here since at the time of the thesis publication Cisco has not issued a patch yet.

5. **Inconsistent Fight Back Response:**

   - **Description:** A false LSA with $LSID \neq AR$ was sent to a router R. One of the routers originated a fight-back response. Then, unexpectedly, its neighbor kept re-sending to that router the original false LSA, and the other router kept sending a new fight-back response with incremented sequence number. This behavior was repeated for many such iterations until stabilization. In addition, the following message showed up in the console: "Detected router with duplicate router ID " (with RID of the router that originated the fight-back).
   - **Sent LSA:**
     $\langle src = R1, dest = R3, LSID = R0, AR = R4, seq = 0x7, links = [] \rangle$
   - **Detailed Description:** A false LSA having LSID=$R0$ and Advertising Router=$R4$ was sent to R3 from Cloud 1. It is sent with seq = 0x7. The false LSA is then flooded from $R3$ to $R4$. $R4$ replies with a fight-back having seq=0x8. In response, R3 sends to R4 the LSA with the updated seq=0x9, which is larger than the seq of the fightback (it was the initial sequence num of $R4$). Then, R4 sends an updated LSA with seq=0xA. Until this point the behavior is as previously described on gap 3 (R4 should not have sent a fight-back since the LSA sequence number was less than its own LSA). Then, $R3$ unexpectedly re-sends to R4 the original (false) LSA with seq=0x7. Eventually, R4 sends an updated LSA with seq=0xB. This behavior goes on repeatedly, and for each such iteration R4 eventually sends an LSA with seq increased by 1. The last packet sent by R4 has a seq=0x16, and then there is a stable state. Additionally, the following message shows up: "Detected router with duplicate router ID" with the ID of $R4$.
   - **Comments:** The gap was only observed on several specific tests from all tests for which the failure was related to deviation #3.
   - **Impact:** The observed behavior includes the behavior described on deviations #2 and #3, but also affects the stability of the routers. It was observed on specific routers within the topology on several specific combinations of values for the LSA fields.
   - **Status:** Since deviation #3 was fixed after version 15.1, this deviation was not observed on the later version.

6. **Inconsistent Fight Back Response for MaxSeq-1**

   - **Description:** A false LSA with $LSID \neq AR$ and $seq = MaxSeq - 1$ was sent to a router R. One of the routers originated fight-back with $seq = MaxSeq$ and $age = MaxAge$. Consequently, its neighbor kept sending the original false LSA over and over again, and it took many such iterations

until stabilization. Additionally, the following message showed up in the console: "Detected router with duplicate router ID" (with RID of the router that originated the fight-back).

- **Sent LSA:**
  $\langle src = R1, dest = R4, LSID = R0, AR = R4, seq = MaxSeq - 1, links = [] \rangle$
- **Detailed Description:** We send a false LSA having LSID=R0 and Advertising Router=R4 to R4. The LSA had a seq=MaxSeq-1. R4 responds with a fight-back having seq=MaxSeq and age=MaxAge. However, R3 keeps sending the false LSA having seq=MaxSeq-1 over and over again, and it takes many iterations till stable state is achieved. Additionally, the following message shows up: "Detected router with duplicate router ID" with ID of $R4$.
- **Comments:** The gap was only observed on several specific tests from all tests that match to deviation #2, with seqNum = maxSeq-1.
- **Impact:** The observed behavior includes the behavior observed on deviation 2, but also affects the stability of the routers.
- **Status:** Since deviation #2 was fixed on later versions, this deviation was not observed on later versions as well.

7. **Re-flooding of LSA Arriving from DR by Unicast:**

- **Description:** A fake LSA on behalf of $R1$ was sent to $R3$ by unicast on Topology 1. The LSA is (unexpetedly) flooded by $R3$ to $R4$ and then to $R1$, resulting in a fight-back originated by $R1$. On the modeled behavior $R3$ does not re-flood the sent LSA, since it is received from the designated router $R1$.
- **Sent LSA:**
  $\langle src = R1, dest = R3, LSID = R1, AR = R1, seq = n, links = [] \rangle$
- **Detailed Description:** The above LSA is sent to $R3$. $R3$ receives it by unicast from $R1$. On the observed behavior, $R3$ floods the LSA by unicast to $R4$, and then $R4$ floods the LSA by unicast to $R1$. This results in a fight-back LSA that $R1$ originates. On the modeled behavior, $R3$ does not flood the sent LSA since it was sent from $R1$ which is the Designated Router. The RFC mentions the following: "If the new LSA was received on this interface, and it was received from either the Designated Router or the Backup Designated Router, chances are that all the neighbors have received the LSA already. Therefore, examine the next interface." Thus, our model follows the RFC instructions. However, based on the RFC, the flooding from the DR within a broadcast network is always expected to be by broadcast and not unicast: "The only packets not sent as unicasts are on broadcast networks; on these networks Hello packets are sent to the multicast

Topology	IOS veresion	# Unique reachable states (depth 1)
Topology 1	15.1	107
Topology 1	15.2, 15.6	9

Table 4.4: A summary of the analysis for generating tests of depth 2

destination AllSPFRouters, the Designated Router and its Backup send both Link State Update Packets and Link State Acknowledgment Packets to the multicast address AllSPFRouters, while all other routers send both their Link State Update and Link State Acknowledgment Packets to the multicast address AllDRouters.". The RFC assumes this is always true and does not contain any instruction to verify that. Thus, we infer that Cisco's implementation verifies whether the LSA packet sent from the DR was indeed flooded by multicast and not unicast as expected. If it is sent by unicast, they add the additional re-flooding as described on the observed behavior, to make sure that neighbors receive the LSA.

- **Impact:** This deviation demonstrates improved security of the implementation with respect to our model which is based on the standard. The final state on our model results in a poisoned LSDB of $R3$ with a fake LSA by $R1$, whereas on Cisco's implementation it is prevented by a fightback due to the re-flooding of the unicast LSA. The standard does not refer to the case on which a router receives an LSA from the designated router by unicast (as in our test), and assumes it is always sent by mulitcast from a designated router.
- **Status:** We found this deviation in all IOS versions we tested for Topology 1. Since it is not a security vulnerability we have not brought it to Cisco's attention so we have no confirmation from them on this deviation.

4.3.3 Results With Multiple LSAs

In the second stage we extended our analysis for multiple LSAs in Topology 1. We initially tried to generate tests by directly using two symbolic LSAs. However, this resulted in the generation of thousands of test files, where the concolic execution process did not terminate its run. This can be referred as the path-explosion problem.

We therefore applied our extended method for dealing with multiple LSAs, as detailed in Section 4.2.4. We analyzed the reachable states of the generated tests from the previous stage (with a single LSA). As specified in Table 4.4, for version 15.1 we identified 107 new reachable states out of 395 generated tests. For versions 15.2 and 15.6, we identified only 9 new reachable states out of 95 generated tests.

We partially applied the analysis of depth 2, for two new reachable states, on all IOS versions.

The new reachable states that we chose are:

1. The LSDBs of all routers contain a spoofed LSA of R0 with an empty list of links.
2. The LSDB of R1 contains its own LSA with $MaxSeqNum$, and the other routers' LSDBs are missing the LSA of R1.

The generated tests contained two sent LSAs, where the first LSA is leading to one of the above chosen reachable states. We did not find any new deviations on this analysis. However, we did observe test failures that were related to the previously found deviation #1 specified in the previous section. For instance, the following scenario was observed:

- Sent LSA #1: an LSA with MaxSeqNum was sent on behalf of R to R.
- Sent LSA #2: an LSA with MaxSeqNum was sent on behalf of another router R' to R.
- Final state: On Cisco's implementation, in the final state both LSAs of R and R' were initialized with InitialSeqNum on all routers' LSDBs.

The expected final state from the model was that only the LSA of R' would be initialized, and the LSA of R would remain unchanged (i.e., with MaxSeqNum on R's LSDB and missing from the other routers' LSDBs, as described on deviation#1). This scenario demonstrates that the second LSA unexpectedly affected the state of the routers w.r.t. a different LSA of another router (R) as well. The result was that R completed its expected procedure only after the second LSA was sent. This new observed behavior can be described as a more complete view of deviation#1 that was initially found in the 1-depth analysis. Increasing the depth of the analysis has the potential to reveal additional consequences of previously found deviation, as demonstrated in this case.

4.4 Advantages and Limitations of Our Method

The method we use allows focusing on specific protocol functionality on which an exhaustive testing can be applied for any implementation of the protocol, and black box in particular. It can be used to test various implementations of different versions and vendors, using the same model and the same set of tests, per each chosen network topology. Since the test generation is systematic and exhaustive with high coverage, it is very effective in finding deviations of a protocol from its standard. The effectiveness is demonstrated by the large number of deviations that we found on an OSPF implementation with a single symbolic LSA and a standard initial state.

Further, our tool can be easily adapted to search for deviation in a different vendor's implementation. One should only adapt the commands required to fetch LSDB and routing table from the routers and parse them as needed. Adapting our method to a

different network protocol is also straightforward, however it requires a new reference model and adjustments of the black box testing script, including the comparison method.

In Section 4.5 we present some previous work related to black box analysis. Many past works used the approach of automatic model inference. In our use case of comparing black box implementations with the standard of the protocol this approach would require to apply such inference for every new version that needs to be tested. Then, specific predefined properties would have to be tested on the inferred models. In our appraoch a reference model has to be implemented once, and its generated tests can be directly applied on every new implementation version, without inference process. It should also be noted that even though our method requires knowledge of the protocol and manual abstractions, it may also be required for the inference method. The inference process may require non-trivial abstractions to enable inferring a certain part of the protocol in the form of a regular automaton with some abstracted alphabet. This abstraction process is non-trivial and requires knowledge of the protocol.

The limitations of our approach include the manual implementation of a reference model, the manual decision of symbolic variables and implementation of a protocol-specific black box testing script.

Furthermore, as for all black box analysis methods our method can not guarantee full coverage of the implementation code. The generated tests only provide high coverage of the model. Therefore, our method is not geared towards finding software vulnerabilities in corner cases that stem from buffer overflow, race conditions and the like.

Additionally, despite our optimizations as the number of sent messages is increased or the topology size is larger, the number of symbolic variables and their domains may grow and lead to the path explosion problem. In this work, we do not explore the topology size limit of our method, nonetheless we have shown in previous section that large topologies are not necessarily needed to expose non-trivial deviations.

4.5 Related Work

4.5.1 Formal Black Box Analysis

There are some past works that used model based approaches for black box analysis. For example, in [71], a black box analysis was applied on networked applications for fault detection by analyzing traces of system calls. A network model component was used to derive a global ordering of the system calls by simulating syscalls. The analysis was then used to find deviations from expected network semantics on certain points in the ordered execution. In [16] a technique for rigorous protocol specification is developed and applied on the TCP and UDP protocols. The specification is written as operational semantics definition in higher order logic. A specification-based testing approach is used to test some implementations. It is based on capturing SUT traces and using a checker, written above HOL, that performs symbolic evaluation of the rigorous specification

along the captured traces. In [12] a black box approach was used by modeling real-time embedded systems environment in UML. They proposed some heuristics for test case generation by investigating selection mechanisms of test cases. They focused on random testing, adaptive random testing and search-based testing.

Apart from the model based approach, some other past works ([29, 36, 21]) approached the black box analysis task using active learning algorithms ([10, 58]) to automatically infer a model of the black box system in the form of an automaton. In [29] the inference algorithm was used to learn an automaton model for a fragment of the TCP protocol on two different implementations. The inferred models were then compared to obtain fingerprinting of these implementations. It should be noted that an abstraction of the TCP packets was used in the learning process. Thus, applying the method to a black box implementation still requires some knowledge of the protocol itself. [21] used similar inference methods to learn models of botnet Command and Control protocols. To analyze the inferred model, they defined certain properties to check on the inferred state-machine. In [36] a black box implementation of the MSN Instant Messaging Protocol is automatically inferred. A fuzz testing technique is used to analyze the inferred model and to search for inputs that can crash the implementation.

4.5.2 Symbolic Execution

Symbolic execution is a very effective and common technique for test generation. It is used to analyze protocols (e.g. [60],[59],[9]) mostly when the implementation is white box with available source code. For example, in [59], an analysis that uses symbolic execution is applied directly on the source code of network protocol implementations, such as DHCP and Zeroconf. It is used to test the protocol implementation against its specification. The specification from an RFC document is translated into a specification in a rule-based language. The input packets generated from the tests are then used to detect violations of the specified rules.

4.5.3 OSPF Analysis

There have been several works that analyzed the OSPF standard itself for security vulnerabilities [52, 61, 37]. A few of these works have used a formal model of the OSPF standard as in this work. However, these works have used the model in a formal analysis process to identify security issues in the model, namely they identify states in which the model has an undesirable property from a security point of view. In contrast, we use the formal model as a benchmark to test implementations. Thereby allowing us to find security issues in specific implementations of the standard rather than in the standard itself.

Few earlier works have also addressed deviations in OSPF implementations. In Ref. [69, 70, 68] the authors identify some deviations in OSPF implementations. All of the identified deviations relate to incorrectly wrapping of the LS sequence number field,

thereby potentially causing false LSAs to remain in the LSADB. The deviations found in these works are a result of an ad-hoc manual analysis.

4.6 Conclusions

In this chapter we developed and implemented a black box method to find deviations of a closed-source protocol implementation from its standard. We used a model based approach on which we modeled core parts of the protocol's standard and used it as a reference model. We have shown that the method is efficient and practical for finding deviations for complex multi-party protocols. We have done so by applying the method to the OSPF protocol – a complex and widely used routing protocol. We tested three versions of Cisco's implementation and found different deviations in each. The method uses concolic execution to generate tests with high coverage, and thus it allowed us to find 7 significant deviations of the tested implementations even in relatively simple topology. Most of the deviations we found also pose security vulnerabilities.

Chapter 5

Conclusions

In this work we developed several methods of a systematic and automatic search for attacks within common Internet routing protocols. We used methods and tools of formal verification, such as model checking and model based testing. We built models for the BGP and OSPF protocols, and developed unique abstraction techniques to cope with scalability issues.

Initially, we focused on a search for built-in vulnerabilities in the OSPF protocol. We started by modeling OSPF on (concrete) networks with a fixed number of routers in a specific topology. By using the model checking tool CBMC, we found several simple attacks on OSPF. In order to search for attacks in a family of networks with varied sizes and topologies, we defined the concept of an abstract network which represents such a family. The attacks we have found on abstract networks revealed security vulnerabilities in the OSPF protocol, which can harm routing in huge networks with complex topologies. Finding such attacks directly on the huge networks is practically impossible.

Next, we searched for non-trivial traffic attraction scenarios in the BGP protocol over the entire Internet topology. We used model checking to perform exhaustive search for attraction attacks on BGP. We developed a method which extracts and reduces fragments from the Internet. In order to apply model checking, we modeled the BGP protocol and also modeled an attacker with predefined capabilities. Our specifications allowed revealing different types of attraction attacks. Using a model checking tool we identified attacks as well as showed that certain attraction scenarios were impossible on the Internet under the modeled attacker capabilities

Finally, we proposed a formal black box method to unearth non-standard protocol deviations in closed-source network devices. The method relies only on the ability to test the targeted protocol implementation and observe its output. We used a model-based testing approach, which relies on a formal model of the protocol. We developed piratical optimizations to allow reducing the number of generated tests without loss of functionality cover of the model. We evaluated our method against the OSPF protocol and searched for deviations in the OSPF implementation of Cisco. Our evaluation identified numerous significant deviations. Some of them can be abused to compromise

the security of a network. The deviations were confirmed by Cisco.

Beyond the above results, our models and methods can be extended to further search for vulnerabilities in the OSPF and BGP protocols. Such extensions may include additional functionality modeling, other attack types and specifications, and more security mechanisms that may limit the attacker capabilities. The abstraction techniques and methodologies can be useful for finding security vulnerabilities in other protocols as well. We believe that our work is an important step in developing methodologies that take advantage of formal verification methods for finding security vulnerabilities in network protocols.

Bibliography

[1] `https://github.com/xiw/mini-mc`.

[2] `https://github.com/Z3Prover/z3`.

[3] Graphical network emulator. http://www.gns3.net.

[4] Scapy. `http://www.secdev.org/projects/scapy/`.

[5] H.323 : Packet-based multimedia communications systems. *Recommendation H.323 (12/09)*, 2009.

[6] IEEE standard for local and metropolitan area networks–media access control (MAC) bridges and virtual bridged local area networks. *IEEE Std 802.1Q-2016*, 2016.

[7] P. Abdulla. Regular model checking. *STTT*, 14(2), 2012.

[8] Paul Adamczyk, Munawar Hafiz, and Ralph E Johnson. Non-compliant and proud: A case study of http compliance. 2008.

[9] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[10] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[11] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[12] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *IFIP International Conference on Testing Software and Systems*, pages 95–110. Springer, 2010.

[13] Matvey Arye, Rob Harrison, and Richard Wang. The next 10,000 bgp gadgets.

[14] Matvey Arye, Rob Harrison, Richard Wang, Pamela Zave, and Jennifer Rexford. Toward a lightweight model of bgp safety. *Proc. of WRiPE*, 2011.

[15] Hitesh Ballani, Paul Francis, and Xinyang Zhang. A study of prefix hijacking and interception in the internet. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 265–276. ACM, 2007.

[16] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 265–276. ACM, 2005.

[17] Alexandra Boldyreva and Robert Lychev. Provable security of s-bgp and other path vector protocols: model, analysis and extensions. In *ACM Conference on Computer and Communications Security*, pages 541–552, 2012.

[18] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[19] CAIDA. Inferred AS Relationships Dataset. `http://data.caida.org/datasets/as-relationships/serial-1/20141001.as-rel.txt.bz2`, October 2014.

[20] R. Callon. Use of OSI IS-IS for routing in TCP/IP and dual environments. IETF RFC 1195, December 1990.

[21] Chia Yuan Cho, Eui Chul Richard Shin, Dawn Song, et al. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439. ACM, 2010.

[22] Hana Chockler, Dmitry Pidan, and Sitvanit Ruah. Improving representative computation in ExpliSAT. In *Haifa Verification Conference (HVC)*, LNCS 8244, Haifa, Israel, 2013.

[23] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 168–176. 2004.

[24] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[25] E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. TACAS'04.

[26] N.S. Niklas Een. Minsat 2.0 - http://minisat.se/minisat.html 2008.

[27] E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME*, 2003.

[28] J. Rosenberg et. al. Sip: Session initiation protocol. IETF RFC 3261, June 2002.

[29] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning fragments of the tcp network protocol. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 78–93. Springer, 2014.

[30] B. Fortz. On the evaluation of the reliability of OSPF routing in IP networks. Technical report, Institut dadministration et de gestion, 2001.

[31] Graham Francis. The sip survey 2015. `https://www.thesipschool.com/files/TheSIPSurvey2015.pdf`, November 2015.

[32] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking (TON)*, 9(6):681–692, 2001.

[33] S. German and P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3), 1992.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[35] Sharon Goldberg, Michael Schapira, Peter Hummon, and Jennifer Rexford. How secure are secure interdomain routing protocols? *Computer Networks*, 70:260–287, 2014.

[36] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pages 114–123. IEEE, 2008.

[37] E. Jones and O. Le Moigne. OSPF security vulnerabilities analysis. Internet-Draft draft-ietf-rpsec-ospf-vuln-02, IETF, June 2006.

[38] Josh Karlin, Stephanie Forrest, and Jennifer Rexford. Pretty good bgp: Improving bgp by cautiously adopting routes. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 290–299. IEEE, 2006.

[39] Stephen Kent, Charles Lynn, Joanne Mikkelson, and Karen Seo. Secure border gateway protocol (s-bgp. *IEEE Journal on Selected Areas in Communications*, 18:103–116, 2000.

[40] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich ssertional languages. In *CAV*, volume 1254 of *LNCS*, Haifa, Israel, 1997.

[41] Jing Liu, Xinming Ye, Jun Zhang, and Jun Li. Security verification of 802.11i 4-way handshake protocol. In *Communications, 2008.*

[42] Robert Lychev, Sharon Goldberg, and Michael Schapira. Network-destabilizing attacks. *arXiv preprint arXiv:1203.1681*, 2012.

[43] Doug Madory. Sprint, Windstream: Latest ISPs to hijack foreign networks. `http://research.dyn.com/2014/09/latest-isps-to-hijack/`, September 2014.

[44] Doug Madory. The Vast World of Fraudulent Routing. `http://research.dyn.com/2015/01/vast-world-of-fraudulent-routing/`, January 2015.

[45] S. U. R. Malik, S. K. Srinivasan, S. U. Khan, and L. Wang. A methodology for OSPF routing protocol verification. In *12th International Conference on Scalable Computing and Communications (ScalCom)*, 2012.

[46] G. Malkin. RIP version 2. IETF RFC 2453, November 1998.

[47] P. Matousek, J. Ráb, O. Rysavy, and M. Svéda. A formal model for network-wide security analysis. In *Engineering of Computer Based Systems, 2008.*

[48] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murphi. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[49] John C. Mitchell, Arnab Roy, Paul Rowe, and Andre Scedrov. Analysis of EAP-GPSK authentication protocol. In *ACNS*, 2008.

[50] J. Moy. OSPF version 2. IETF RFC 2328, April 1998.

[51] G. Nakibly, D. Gonikman, A. Kirshon, and D. Boneh. Persistent OSPF attacks. In *NDSS*, 2012.

[52] Gabi Nakibly, Adi Sosnovich, Eitan Menahem, Ariel Waizel, and Yuval Elovici. OSPF vulnerability to persistent poisoning attacks: A systematic analysis. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 336–345. ACM, 2014.

[53] Joel Obstfeld, Simon Knight, Ed Kern, Qiang Sheng Wang, Tom Bryan, and Dan Bourque. Virl: the virtual internet routing lab. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 577–578, 2014.

[54] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). IETF RFC 4271, January 2006.

[55] Yiqing Ren, Wenchao Zhou, Anduo Wang, Limin Jia, Alexander JT Gurney, Boon Thau Loo, and Jennifer Rexford. Fsr: formal analysis and implementation toolkit for safe inter-domain routing. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 440–441. ACM, 2011.

[56] M. Saksena, O. Wibling, and B. Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In *TACAS*, 2008.

[57] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[58] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *International Symposium on Formal Methods*, pages 207–222. Springer, 2009.

[59] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. Symbexnet: testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, 2014.

[60] JaeSeung Song, Tiejun Ma, Cristian Cadar, and Peter Pietzuch. Rule-based verification of network protocol implementations using symbolic execution. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–8. IEEE, 2011.

[61] Adi Sosnovich, Orna Grumberg, and Gabi Nakibly. Finding security vulnerabilities in a network protocol using parameterized systems. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 724–739, 2013.

[62] Adi Sosnovich, Orna Grumberg, and Gabi Nakibly. Analyzing internet routing security using model checking. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 112–129, 2015.

[63] Andree Toonk. BGP hijack incident by Syrian Telecommunications Establishment. `http://www.bgpmon.net/bgp-hijack-incident-by-syrian-telecommunications-establishment/`, December 2014.

[64] Andree Toonk. Hijack event today by Indosat. `http://www.bgpmon.net/hijack-event-today-by-indosat/`, April 2014.

[65] Andree Toonk. The Canadian Bitcoin Hijack. `http://www.bgpmon.net/the-canadian-bitcoin-hijack/`, August 2014.

[66] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[67] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. Mind your blocks: On the stealthiness of malicious BGP hijacks. 2015.

[68] Brain Vetter, Feiyi Wang, and Shyhtsun Felix Wu. An experimental study of insider attacks for ospf routing protocol. In *Network Protocols, 1997. Proceedings., 1997 International Conference on*, pages 293–300. IEEE, 1997.

[69] Feiyi Wang, Brian Vetter, and Shyhtsun Felix Wu. Secure routing protocols: Theory and practice. Technical report, North Carolina State University, May 1997.

[70] S. F. Wu and et. al. JiNao: Design and implementation of a scalable intrusion detection system for the OSPF routing protocol. *ACM Transactions on Computer Systems*, 2, 1999.

[71] Yanyan Zhuang, Eleni Gessiou, Steven Portzer, Fraida Fund, Monzur Muhammad, Ivan Beschastnikh, and Justin Cappos. Netcheck: Network diagnoses from blackbox traces. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 115–128, 2014.

יכולות התוקף הממודלות.

לבסוף, אנו מציעים שיטה פורמלית לאנליזה של מימוש פרוטוקול שהוא קופסה שחורה לצורך מציאת חריגות בין המימוש לבין התקן של הפרוטוקול. אנו מסתמכים רק על האפשרות לבחון את התנהגות המימוש של הפרוטוקול ואת הפלט שלו. אנו משתמשים בגישת ביצוע טסטים מבוססי מודל. שיטה זו מבוססת על מודל פורמלי של הפרוטוקול הנדון. כמו כן אנו מפתחים אופטימיזציות פרקטיות לצורך הקטנת מספר הטסטים הנוצרים מתוך המודל ללא איבוד כיסוי פונקציונליות, המותאמות לפרוטוקולי רשת. אנו מיישמים את השיטה על פרוטוקול OSPF ובוחנים את המימוש של Cisco, שהוא הנפוץ ביותר בשימוש ברשת האינטרנט. השיטה מאפשרת למצוא הבדלים משמעותיים בין מימוש הפרוטקול לבין התקן שלו אשר מהווים חולשות אבטחה במימוש ויכולים לאפשר לתוקף להשפיע על הניתוב ברשת.

תקציר

הנתבים והרשתות באינטרנט מחולקים לקבוצות קשירות. כל קבוצה כזו מכונה רשת אוטונומית. ניתוב של חבילות מידע באינטרנט פועל בשתי רמות. פרוטוקול Border Gateway Protocol, המכונה BGP, הוא פרוטוקול ניתוב הקובע דרך אילו רשתות אוטונומיות חבילות מידע יועברו ברשת האינטרנט. פרוטוקול Open Shortest Path First, המכונה OSPF, הוא פרוטוקול נפוץ מאוד הקובע את מסלולי הניתוב בתוך רשת אוטונומית. מציאת חולשות אבטחה ואסטרטגיות התקפה בפרוטוקולי ניתוב אלו היא משימה מורכבת וחשובה באבטחת האינטרנט. התקפות על פרוטוקולי ניתוב יכולות להפריע לניתוב ברשת ולמנוע מחבילות מידע מלהגיע ליעדן.

שיטות אימות פורמלי פותחו במקור כדי להוכיח נכונות של מערכות ביחס למפרטים פורמליים. בדיקת מודל היא שיטה מאוד נפוצה של אימות פורמלי. בדיקת מודל מורכבת מאלגוריתם יעיל אשר בהינתן מודל של מערכת ומפרט פורמלי, קובע האם המערכת מספקת את המפרט או לא. במידה ולא, האלגוריתם מחזיר דוגמא נגדית בצורת התנהגות לא רצויה של המערכת. בתזה זו, אנו מפתחים מספר שיטות לביצוע אנליזה למציאת חולשות אבטחה באופן שיטתי ואוטומטי של פרוטוקולי ניתוב ברשת האינטרנט. לצורך כך אנו משתמשים בשיטות וכלים של אימות פורמלי.

תחילה, אנו מפתחים אנליזה למציאת חולשות מובנות בתקן של פרוטוקול OSPF בעזרת בדיקת מודל. אנו ממדלים חלקים מהפרוטוקול על פי התקן שלו ומשתמשים בכלי לבדיקת מודל כדי למצוא באופן אוטומטי חולשות מובנות בפרוטוקול על טופולוגיות רשת פשוטות. באמצעות הגדרת מפרט לזיהוי התקפות שיכולות להשפיע באופן ממושך על טבלאות הניתוב של נתבים, הכלי לבדיקת מודל מחפש אחר התנהגויות כאלו במודל. לאחר מכן אנו מרחיבים את האנליזה לטופולוגיות רשת כלליות יותר. אנו מפתחים שיטה חדשה לרשתות פרמטריות אשר מתאימה למציאת דוגמה נגדית (התקפה במקרה שלנו) על כל רשת במשפחה של רשתות. השיטה מאפשרת למצוא התקפות כלליות שניתנות ליישום על משפחות של רשתות.

בהמשך, אנו מפתחים אנליזה למציאת התקפות על BGP. אנו מתמקדים בהתקפות משיכת תעבורה, אשר בהן תוקף יכול לשלוח פרסומי נתיבים מזויפים לצורך השגת תעבורה עודפת דרכו למטרות כגון הגדלת הרווח מלקוחות, או להשגת המידע המועבר בחבילות דרכו. אנו משתמשים בבדיקת מודל לביצוע חיפוש שיטתי אחר אסטרטגיות התקפה למשיכת תעבורה. הדבר דורש שיטות לצורך התמודדות עם רשת גדולה כמו רשת האינטרנט. אנו מציעים שיטות סטטיות לזיהוי חלקים מרשת האינטרנט שבהם ניתן להתמקד, לפני ביצוע בדיקת המודל. לשם ביצוע בדיקת מודל, אנו ממדלים חלקים מ BGP בנוכחות תוקף עם יכולות מוגדרות מראש. המפרטים שאנו מציעים מאפשרים לחשוף סוגים שונים של התקפות משיכת תעבורה. באמצעות שימוש בכלי לבדיקת מודל אנו מזהים התקפות אפשריות על רשת האינטרנט וכן מראים שחלק מהתרחישים למשיכת תעבורה אינם אפשריים תחת

המחקר בוצע בהנחייתה של פרופסור ארנה גרימברג, בפקולטה למדעי המחשב.

תודות

ראשית, ברצוני להביע את תודתי הכנה למנחה שלי, פרופ' ארנה גרימברג. תודה על שנים של הנחיה ותמיכה במהלך ביצוע המחקר וכתיבת התזה. אני אסירת תודה על ההזדמנות לעבוד איתך תחת ההנחיה שלך. כמו כן, ברצוני להביע גם תודתי הכנה לדר' גבי נקיבלי. תודה על שנים של שיתוף פעולה, ועל ההשראה והעזרה במהלך המחקר. בנוסף ארצה להודות לפרופ' מיכאל שפירא. תודה על ההשראה, העזרה, הרעיונות וההערות.

לבסוף, ארצה להודות למשפחתי: להוריי ולבעלי, על התמיכה והעידוד במהלך שנות המחקר ובחיי באופן כללי.

אני מודה לטכניון ולמלגת רנדי ל. ומלווין ר. ברלין בתכנית למחקר ולימוד אבטחת סייבר על התמיכה הכספית הנדיבה בהשתלמותי.

מציאת חולשות אבטחה בפרוטוקולי רשת תוך שימוש בשיטות לאימות פורמלי

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

עדי סוסנוביץ

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
טבת התשע"ז חיפה ינואר 2017

מציאת חולשות אבטחה בפרוטוקולי רשת תוך שימוש בשיטות לאימות פורמלי

עדי סוסנוביץ