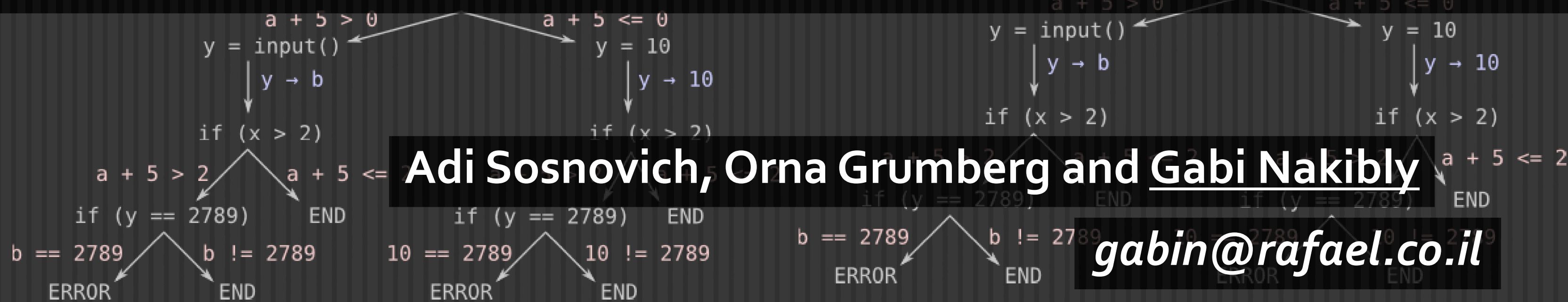
AUTOMATED DETECTION OF VULNERABILITIES IN BLACK-BOX ROUTERS

How to Find Vulnerabilities in Dozens of Router Models Without Using IDA



Adi Sosnovich, Orna Grumberg and Gabi Nakibly

gabin@rafael.co.il

INTRODUCTION – GABI NAKIBLY

- Chief research scientist at the National Cyber and Electronics Research Center
  - Operated by Rafael – Advanced Defense Systems ltd.
- A former Visiting Scholar at Stanford University
- Senior adjunct lecturer and research associate at the Technion – Israel institute of Technology
- I mostly spend my days doing network security research.

OUTLINE

- Motivation
- The method we propose
- What is symbolic execution
- Unique optimizations that make our method scalable
- Application of the method to Cisco’s OSPF implementation
- The vulnerabilities we have found

MOTIVATION

- Network protocols are based on open standards
- However, the Internet runs mostly on proprietary and closed-source routers
- A hidden deviation of a device's implementation from a protocol standard may create a logical vulnerability
- However, finding deviations in closed-source routers demands great efforts.

RESEARCH GOAL

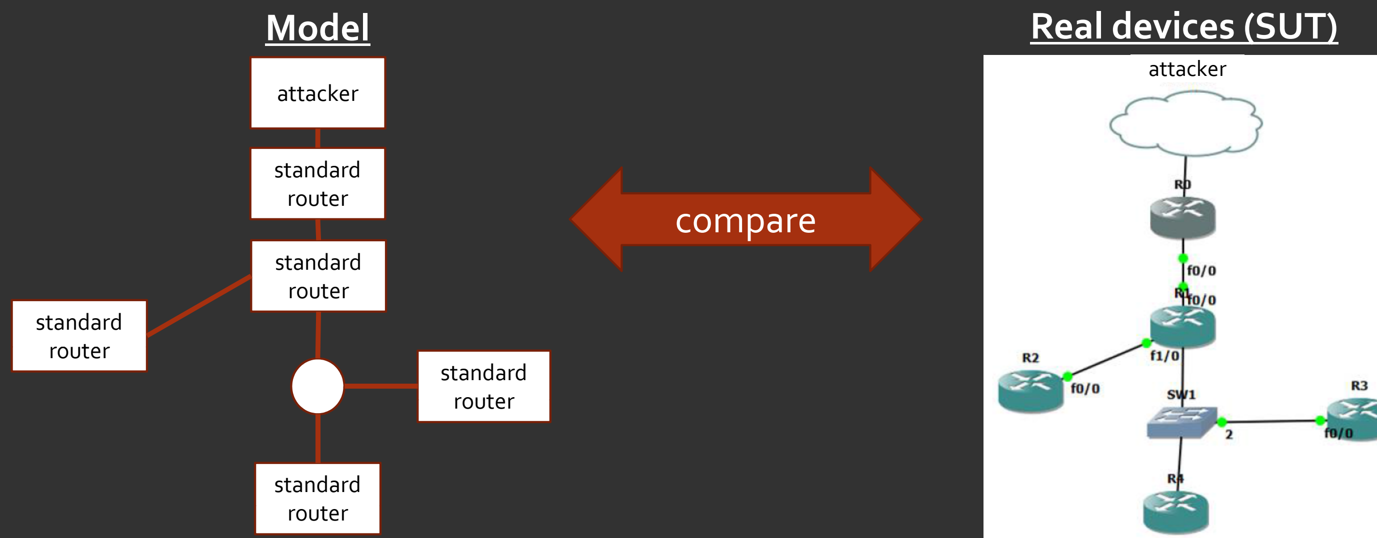
- An **automated** **formal** **black-box** method that unearths implementation deviations in closed-source network devices
  - Black-box: no need to access the binary or source code of the device!
  - Formal: a systematic method based on rigorous foundations
  - Automated: once a manual setup phase is compete no manual assistance is needed.

THE BASIC IDEA IN A NUTSHELL

- Compare the behavior of a router (called SUT) to that of a model that captures the standard functionality of the network protocol (or part of it).
  - SUT = System Under Test

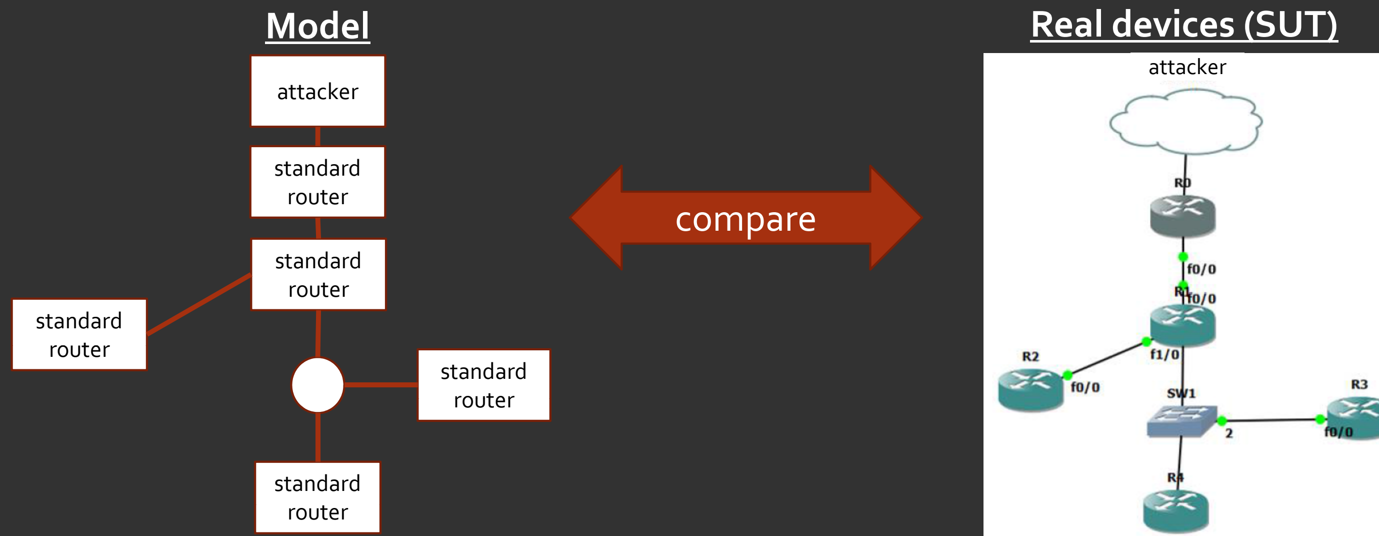
THE BASIC IDEA IN A NUTSHELL

- Compare the behavior of a router (called SUT) to that of a model that captures the standard functionality of the network protocol (or part of it).
  - SUT = System Under Test

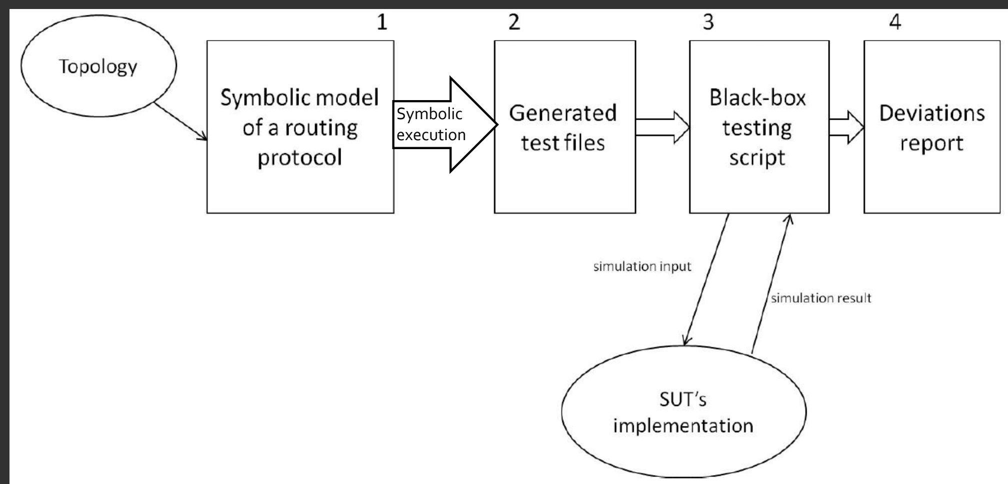


THE BASIC IDEA IN A NUTSHELL (CONT.)

- A naïve approach: manually select tests based on predefined heuristics
- Our approach: automatically generate tests based on symbolic execution that covers the entire model functionality.

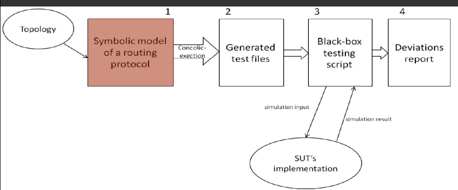


OUR METHOD IN A GLANCE



1. CREATE A MODEL OF A PROTOCOL

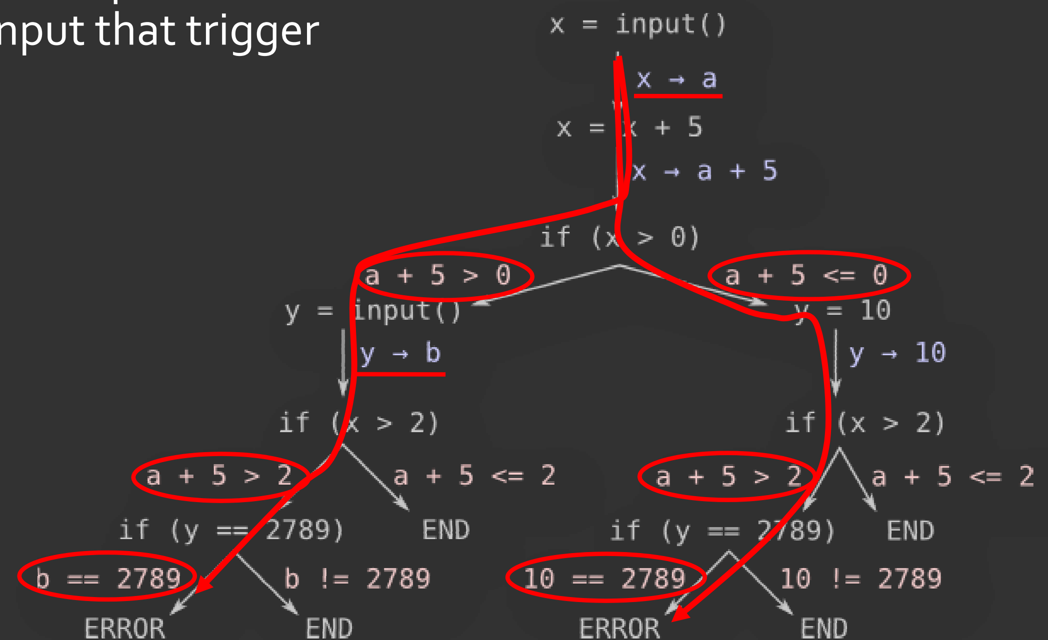
- The model is simply a program that captures the functionality of the standard (or part of it).
- The model simulates the execution of the protocol among the standard routers in a given topology.
- The model receives as an input protocol messages sent by an attacker.
  - These messages are the symbolic variables.




```
class Router:
    def __init__(self, ID ):
        self.queue = []
        self.DB = []
        self.originalDB = []
        self.ID = ID
        self.timer = 0
        self.delayedFB = []
        self.lookup_policy = 0
        self.routingTable = RoutingTable()
    def updateTimer(self):
        if self.timer>0:
            self.timer-=1
        return
    def addLSA(self,lsa):
        self.DB.append(lsa)
        return
    def flood(self,m,topology):
        debug.Print("flood function ---self.ID=  " + str(self.ID) + " src = " + str(m.src))
        src = m.src
        m.src = self.ID
        rlinks = topology.links[self.ID]
        for l in rlinks:
            if (src == l.linkID and  l.linkType!='transit') or l.linkType=='stub':
                debug.Print('continue')
                continue
            if (self.ID==3 and src==1):
                continue
            if (self.ID==4 and src==1):
                continue
            if l.linkID==self.ID and l.linkType=='transit':
                debug.Print('transit link')
                for n in topology.transitNetworks:
                    if n[1]==self.ID:
                        index=0
                        for rID in n[0]:
                            if rID == self.ID :
                                index+=1
                                continue
                            m2=m.duplicate()
                            m2.dest = rID
                            m2.interface = n[4][index]
                            topology.routers[m2.dest].queue.append(m2)
                            debug.Print("flood to " + str(m2.dest) + "  at interface " + str(m2.interface) + ", seq
                            index+=1
                    else :
                        m2 = m.duplicate()
                        m2.dest = n[1]
                        m2.interface = n[2]
                        topology.routers[m2.dest].queue.append(m2)
                        debug.Print("flood to " + str(m2.dest) + "  at interface " + str(m2.interface) + ", seq = "
            else:
                m2=m.duplicate()
                m2.dest = l.linkID
                m2.interface = topology.getLinkInterface(m2.dest,self.ID)
                topology.routers[m2.dest].queue.append(m2)
```

SYMBOLIC EXECUTION 101

- Symbolic execution allows to trace all execution paths of a program and generate the corresponding input that trigger each path.



2. GENERATE TEST CASES

- Run symbolic execution on the model of the protocol to cover **all** execution paths of the protocol's model.
- Each execution path is driven by a specific sequence of rogue protocol messages sent by the attacker.
  - Each such execution path represents a test case with an expected outcome.

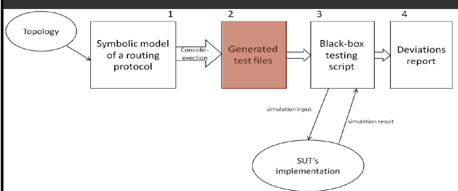
Test case example

Send routing MSG1 to R1

Send routing MSG2 to R2

Send routing MSG3 to R1

Expected outcome: routing states of all routers



3. EXECUTE TESTS

- Each generated test file is executed on the SUT.
- During the test execution the sequence of rogue protocol messages are sent to the network devices.
- At the end of the test the state of the devices are extracted.

<u>Execute test on SUT</u>

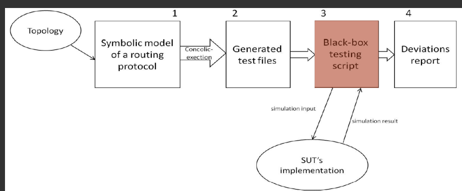
Send routing MSG1 to R1

Send routing MSG2 to R2

Send routing MSG3 to R1

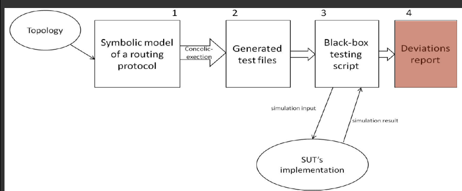
Actual outcome: routing states of all routers in SUT

Expected outcome: routing states of all rouetrs in model



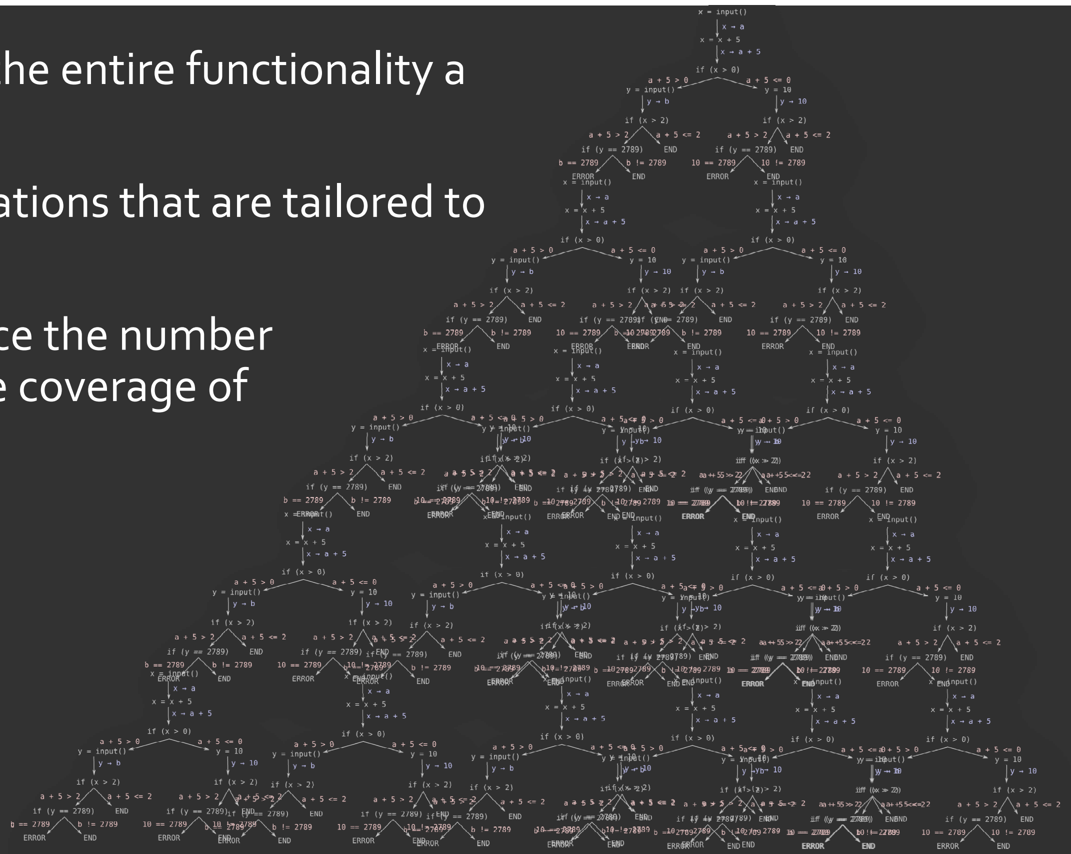
4. FIND DEVIATIONS

- A failed test represents a deviation of the protocol’s implementation from the protocol standard.
- The failed test is accompanied with traces of all messages exchanged between the devices during the run of the test, both on the model and on the SUT.
- Comparing these traces facilitates the analysis of the vulnerability.



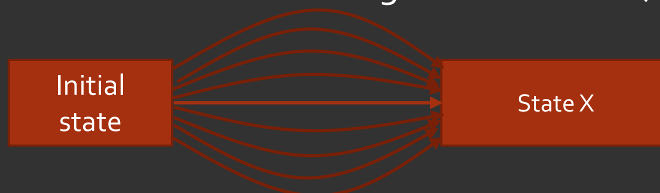
PATH EXPLOSION PROBLEM

- Many tests may be needed to cover the entire functionality a complex real-world protocol.
- We deal with it using unique optimizations that are tailored to testing network protocols.
- Our optimizations dramatically reduce the number generated tests without reducing the coverage of the protocol's model.

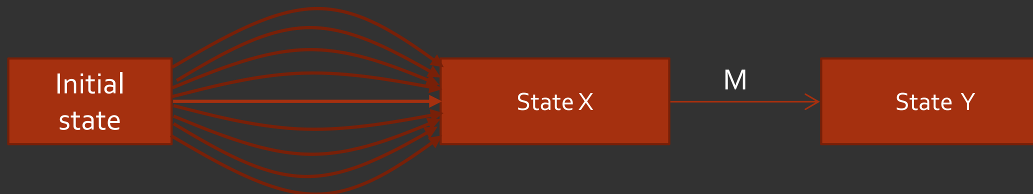


OUR MAIN OPTIMIZATION

- The optimization is due to following insight:
  - Let's assume we have the following 100 test cases (each is 1 message long):



- Then we shall have additional 100 test cases (each is 2 messages long):



- We can replace the above 100 (2-message) tests with the following single test:

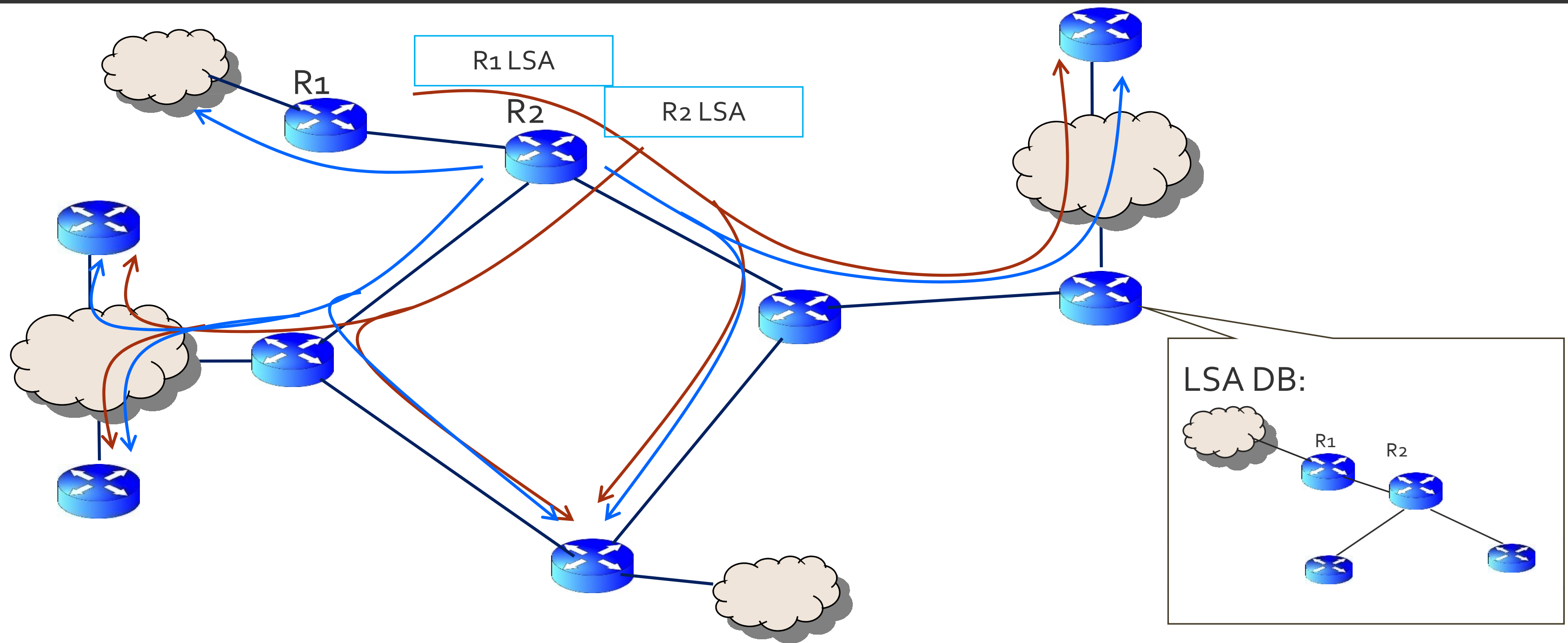
OSPF ANALYSIS

- We applied our method to find vulnerabilities in the OSPF implementation of routers.
- OSPF is one of the most widely used and most complex routing protocols on the Internet.

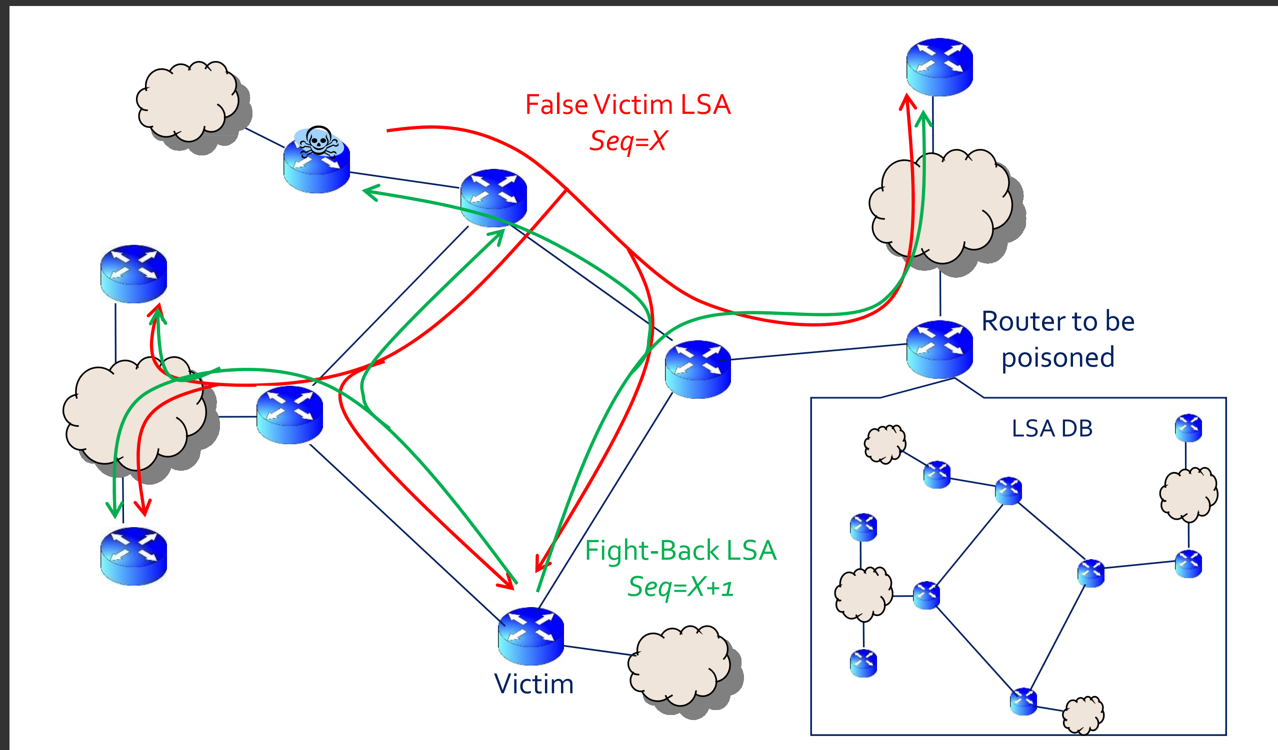
OSPF 101

- Every router advertises it’s links’ state (i.e. “who are my neighbors?”).
  - This is called Link State Advertisement (LSA).
- The LSAs are flooded throughout the network hop-by-hop.
- Every router receives the LSAs of all other routers.
  - This allows to build the topology map of the entire network.

OSPF 101 (CONT.)



THE FIGHT-BACK MECHANISM



THE ATTACKER

- Location: inside the network
  - Controls a legitimate router in an arbitrary location
  - This means it can flood LSAs to its neighbors
- Goal: persistent poisoning of routers' routing tables

OSPF MODEL

- We modeled the OSPF using a Python code having roughly 1000 LoC.
- The router's procedure implements the core functionality of OSPF.
  - relevant to the security against the above type of attack
- The model is at
  - https://github.com/gnakibli/ospf-model-test

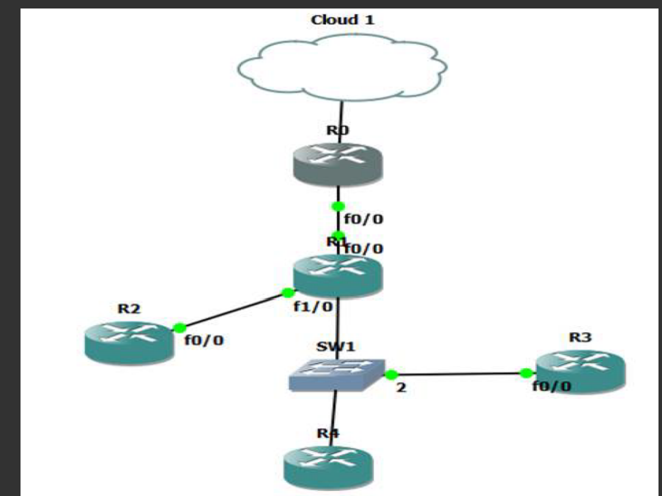
```python
class Router:
    def __init__(self, ID ):
        self.queue = []
        self.DB = []
        self.originalDB = []
        self.ID = ID
        self.timer = 0
        self.delayedFB = []
        self.lookup_policy = 0
        self.routingTable = RoutingTable()
    def updateTimer(self):
        if self.timer>0:
            self.timer-=1
        return
    def addLSA(self,lsa):
        self.DB.append(lsa)
        return
    def flood(self,m,topology):
        debug.Print("flood function ---self.ID=  " + str(self.ID) + " src = " + str(m.src))
        src = m.src
        m.src = self.ID
        rlinks = topology.links[self.ID]
        for l in rlinks:
            if (src == l.linkID and  l.linkType!='transit') or l.linkType=='stub':
                debug.Print('continue')
                continue
            if (self.ID==3 and src==1):
                continue
            if (self.ID==4 and src==1):
                continue
            if l.linkID==self.ID and l.linkType=='transit':
                debug.Print('transit link')
                for n in topology.transitNetworks:
                    if n[1]==self.ID:
                        index=0
                        for rID in n[0]:
                            if rID == self.ID :
                                index+=1
                                continue
                            m2=m.duplicate()
                            m2.dest = rID
                            m2.interface = n[4][index]
                            topology.routers[m2.dest].queue.append(m2)
                            debug.Print("flood to " + str(m2.dest) + "  at interface " + str(m2.interfa
                            index+=1
                    else :
                        m2 = m.duplicate()
                        m2.dest = n[1]
                        m2.interface = n[2]
                        topology.routers[m2.dest].queue.append(m2)
                        debug.Print("flood to " + str(m2.dest) + "  at interface " + str(m2.interface)
            else:
                m2=m.duplicate()
                m2.dest = l.linkID
                m2.interface = topology.getLinkInterface(m2.dest,self.ID)
                topology.routers[m2.dest].queue.append(m2)
```

CISCO TESTBED

- To test Cisco's OSPF implementation we used alternately two network emulation software: GNS3 and VIRL.
  - Both software suites allow to emulate a network of multiple routers, each running an actual IOS image (identical to the images used in real Cisco routers).
- We used the following IOS versions:

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

most recent



CISCO RESULTS

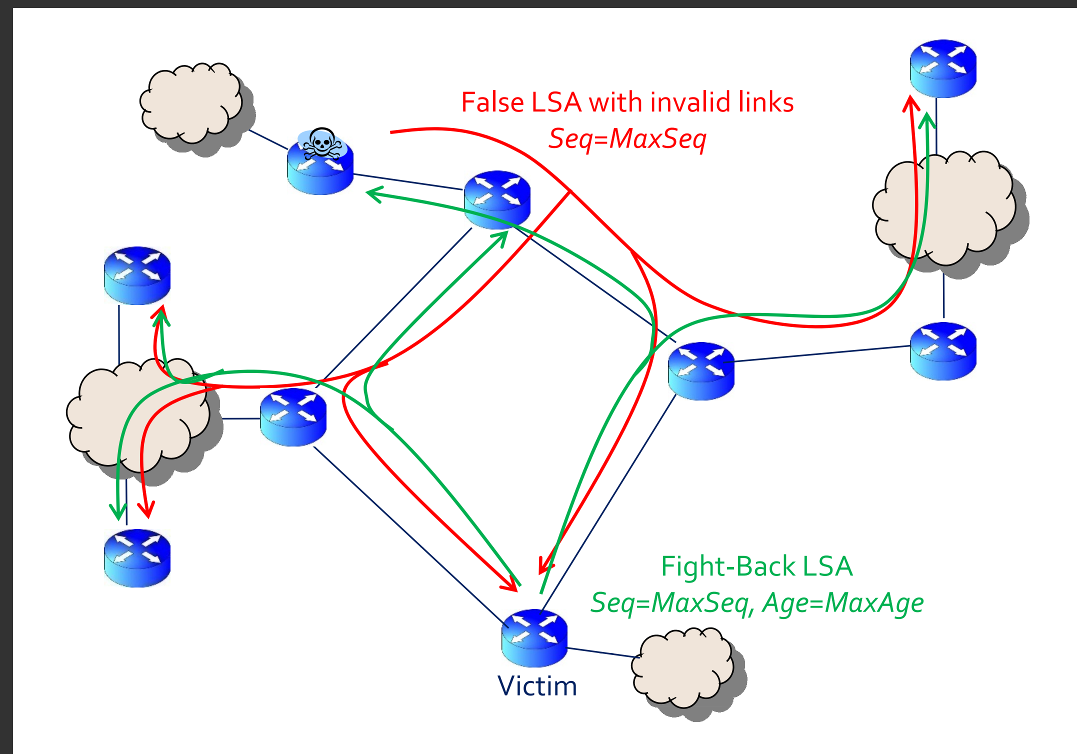
- For Cisco we discovered 7 deviations in all three IOS versions.
- 6 of those deviations presented logical vulnerabilities.
- 2 of them are vulnerabilities that exist in the most up-to-date IOS version.
  - CVE-2017-6770
- The new vulnerabilities allow an attacker to evade the fight-back mechanism and gain persistent control over the routing state of the network.
- The vulnerabilities affect <u>all</u> IOS, XE and ASA products as some NXOS products.

CISCO VULNERABILITIES

- OSPF determines which LSA is newer by examining the following values in order:
  - Sequence Number
  - Checksum

CISCO VULNERABILITIES (CONT.)

- An attacker sends a false LSA with
  - *Seq=MaxSeq*
- According to the standard the victim must flush the false LSA by sending the FB with:
  - *Seq=MaxSeq, Age=MaxAge*
  - The FB must include the **invalid** links.
    - This is to ensure that checksum values are identical.
    - However, Cisco sends the FB with the **valid** links!
    - Thus, potentially having a smaller checksum value as compared to that of the false LSA.
- Hence, the FB may be considered older than the false LSA and thus ignored!



QUAGGA

- Quagga – the most popular open-source routing suite on the Internet
- Similar vulnerability have been discovered in Quagga.
- CVE-2017-3224

IN SUMMARY

- We present an method that finds deviations of network protocol implementations.
- All our method need is a model of the protocol and you are good to go!
- Once you have a model you can test any implementation of that protocol fully automatically
  - allowing you to discover many vulnerabilities is a short time.
- Symbolic execution is an important tool that the security community should use more often.
- Code: https://github.com/gnakibli/ospf-model-test
- Shoot me an email if you have questions or comments: gabin@rafael.co.il