Automated Analysis and Debugging of Network Connectivity Policies

Karthick Jayaraman
Microsoft Azure
karjay@microsoft.com

Nikolaj Bjørner
Microsoft Research
nbjorner@microsoft.com

Geoff Outhred
Microsoft Azure
geoffo@microsoft.com

Charlie Kaufman
charliekaufman@outlook.com

ABSTRACT

Network connectivity policies are crucial for assuring the security and availability of large-scale datacenter. Managing these policies is fraught with complexity and operator errors. The difficulties are exacerbated when deploying large scale offerings of public cloud services where multiple tenants are hosted within customized isolation boundaries. In these large-scale settings it is impractical to depend on human effort or trial and error to maintain the correctness and consistency of policies.

We describe an approach for automatically validating network connectivity policies and its implementation in a tool called SecGuru. SecGuru can check selected properties of policies, e.g., is some traffic permitted or denied, and it can compare two policies yielding a *semantic diff* to summarize drifts. We use bit-vector logic to encode policies and semantic diffs; and the theorem prover Z3 as the underlying solver. A key contribution is a new algorithm for compactly enumerating symbolic diffs. We finally describe the experience of using SecGuru in Azure, a public cloud provider. Azure uses SecGuru for continuously monitoring policy configurations and alerting on errors, and also as a regression test suite to check policies before deployment. As a result of using SecGuru, today Azure proactively detects and avoids policy misconfigurations that lead to security and availability issues.

1. INTRODUCTION

Managing network connectivity restrictions in large-scale datacenters is a challenge that cannot rely on human inspection or trial and error. The large scale public cloud provider Azure is a case in point. It provides on-demand computing, storage, and networking resources to mutually distrusting customers. The infrastructure services and its customer services are hosted in custom isolation boundaries using network connectivity restrictions. For example, Azure management service interfaces are walled off from the Internet and arbitrary customer access. In addition, customer services are also isolated from one another. These restrictions are enforced in network devices such as routers and top-of-rack switches, hypervisor packet filters, and firewalls. Managing these restrictions is fraught with complexity.

An Example

We illustrate the challenges using an example. Datacenters and enterprises use a set of routers to connect their networks to the Internet backbone. These routers are called the Edge routers, and they enforce an access-control list (ACL) to enforce restrictions on traffic coming from the Internet. We will refer to it as the Edge ACL in this paper. Figure 1 provides a canonical example of an Edge ACL and typical maintenance operations done on it. The ACL in this example is authored in the Cisco IOS language. It is basically a set of rules that filter IP packets. They inspect header information of the packets and the rules determine whether the packets may pass through the device.

Each rule of a policy contains a packet filter, and typically comprises two portions, namely a traffic expression and an action. The traffic expression specifies a range of source and destination IP addresses, ports, and a protocol specifier. The expression 10.0.0.0/8 specifies an address range 10.0.0.0 to 10.255.255.255. That is, the first 8 bits are fixed and the remaining 24 (= 32-8) are varying. A wild card is indicated by *Any*. For ports, *Any* encodes the range from 0 to $2^{16}-1$. The action is either *Permit* or *Deny*. They indicate whether packets matching the range should be allowed through the firewall. We say each rule represents a single *cube*. This language has the first-applicable rule semantics, where the device processes an incoming packet per the first rule that matches its description. If no rules match, then the incoming packet is denied by default. Therefore, the order in which the rules appear is important. In certain policy languages, rules can be further compressed by using *multiple* ranges for the IP addresses and ports. For example, a filter on an IP address may contain two ranges 10.20.0.0/19; 10.40.0.0/19 where a semi-colon separates the two ranges. We call such rules as *multi-cubes*.

The left-hand side of Figure 1 shows an ACL instance prior to an update, and has four sections. The first section (lines 2-6) filters Internet traffic that targets private datacenter IP addresses. For example, line 3 in the ACL denies traffic targeting IP addresses in 10.0.0.0/8, which is a private address range per RFC1918 and should not be reachable from the Internet. The second section (lines 8-10) is for the anti spoofing ACL that filters Internet traffic that claims to come from within the datacenter network. The third section (lines 13-14) of the ACL permits traffic targeting datacenter IP addresses that should not have any port blocks. The fourth section (lines 17-24) of the ACL blocks a standard set of ports and protocols on all Internet traffic targeting any destination inside the datacenter network. Finally, the fifth section (lines 21-26) of the ACL permits traffic targeting datacenter IP addresses that will be subject to port and protocol restrictions of section four. Any

```
1   remark Isolating private addresses
2   deny ip 0.0.0.0/32 any
3   deny ip 10.0.0.0/8 any
4   deny ip 172.16.0.0/12 any
5   deny ip 192.0.2.0/24 any
6   ...
7   remark Anti spoofing ACLs
8   deny ip 128.30.0.0/15 any
9   deny ip 171.64.0.0/15 any
10  ...
11  remark permits for IPs without
12          port and protocol blocks
13  permit ip any 171.64.64.0/20
14  ....
15  remark standard port and protocol
16          blocks
17  deny   tcp any any eq 445
18  deny   udp any any eq 445
19  deny   tcp any any eq 593
20  deny   udp any any eq 593
21  ...
22  deny   53 any any
23  deny   55 any any
24  ...
25  remark permits for IPs with
26          port and protocol blocks
27  permit ip any 128.30.0.0/15
28  permit ip any 171.64.0.0/15
29  ...
```

```
1   remark Isolating private addresses
2   deny ip 0.0.0.0/32 any
3   deny ip 10.0.0.0/8 any
4   deny ip 172.16.0.0/12 any
5   deny ip 192.0.2.0/24 any
6   ...
7   remark Anti spoofing ACLs
8   deny ip 128.30.0.0/15 any
9   deny ip [171.64.0.0/18] any
10  ...
11  remark permits for IPs without
12          port and protocol blocks
13  permit ip any [171.64.64.0/18]
14  ....
15  remark standard port and protocol
16          blocks
17  deny   tcp any any eq 445
18  deny   udp any any eq 445
19  deny   tcp any any eq 593
20  deny   udp any any eq 593
21  ...
22  deny   53 any any
23  deny   55 any any
24  ...
25  remark permits for IPs with
26          port and protocol blocks
27  permit ip any 128.30.0.0/15
28  permit ip any 171.64.64.0/15
29  [permit ip any 128.230.0.0/16]
30  ...
```

Figure 1: Edge Network ACL : The left-side contains an example instance of the Edge ACL configuration. The right side contains the same instance with updates that are highlighted.

traffic targeting the IP addresses in section three will not be subject to port restrictions. However, if Internet traffic targets blocked ports and protocols on IP addresses in section five, then those packets will match the description in section four and will be blocked.

Ensuring the correctness of these policies is critical for both availability and security. For example, an incorrect deny rule in the Edge ACL can cause a connectivity outage to several services. Similarly, an incorrect allow rule may expose a protected management service to zero-day exploits or DDoS attacks.

The right-hand side instance highlights examples of typical maintenance updates done to the Edge ACL. In line 9, the update changes the address range in the anti-spoofing ACL from 171.64.0.0/15 to 171.64.0.0/18. This is because address ranges 171.64.64.0/18 and 171.64.128.0/17 are assigned to networks that interconnect with the Edge over the Internet. Therefore, the anti-spoofing ACL had to be revised to exclude those addresses. In line 12, the update changes the IP range without port and protocol blocks from 171.64.64.0/20 to 171.64.64.0/18. Finally, in line 26, the update adds permits for new block of IP addresses. These blocks do not have corresponding anti-spoofing ACLs because they are assigned to networks that connect to the Edge over the Internet. The rapid growth in both capacity and new services introduces a corresponding churn in IP addresses and updates to these policies.

Preserving the correctness of policies requires precisely understanding the impact of changes, and making sure we are always preserving the essential properties. For example, in Figure 1, the difference between the two policies because of the update in line 9 is that the revised policy allows the multicube described by $<$ $171.64.64.0/18; 171.64.128.0/17, *, *, *, ip >$, but the original policy does not. We need to make sure that the change assures this. Additionally, we need to make sure there is no regression on other properties such as isolating private addresses.

Manual reviews is infeasible at scale: Azure has several thousand network devices, hypervisor packet filters, and firewalls, and each of them enforce a policy and are subject to updates of the nature we described above. Some policies such as the Edge ACL have a few thousand rules. Moreover, the semantics of the rules vary depending on the type of the device. For example, the order of rules is relevant in network devices, but it is not for the hypervisor packet filter.

Our Approach

Our approach to checking policies is implemented in a tool called SecGuru. It automatically validates network connectivity policies at scale using a modern Satisfiability Modulo Theories solver (theorem prover) Z3 [8]. We first show how network connectivity policies are encoded into bit-vector logic and then show how to extract descriptive answers from Z3. In Azure, we deal with several types of network policy semantics. Bit-vector logic allows encoding in a straightforward way first-applicable rule semantics use by network devices and the default-deny semantics used by firewalls. In addition, there are stateful ACLs and stateless implementation of stateful ACLs. For example, network devices may filter traffic based on TCP SYN flags. The theory of bit-vectors allows us to model this and other Boolean combinations of flags accurately.

We also report on extensive experience in Azure; and we develop a set of benchmarks for evaluating our approach well beyond the current scale. SecGuru can be used to validate the correctness of a policy with respect to set of contracts, and also for assessing the impact of changes made to a firewall. A distinguished feature of SecGuru is the ability to enumerate symbolic differences between different versions of firewall configurations compactly. This allows operators to identify missing or superfluous rules directly by inspecting the output of SecGuru. In addition, the output is amenable to automatic rectification of the errors.

SecGuru now runs continuously in Azure, checking the

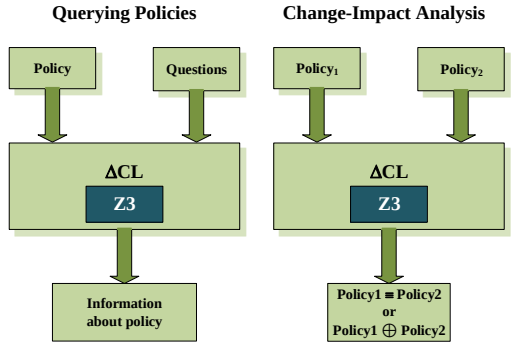

Figure 2: SecGuru

integrity of thousands of routers and firewalls servicing millions of machines. It is also used by operators to check for regressions when policies change as new services are put online or new requirements are imposed on the network. SecGuru scales very well and is efficient (spends typically a fraction of a second) in analyzing our production policies. Moreover, we also performed an evaluation of SecGuru using a set of synthetic benchmarks created based on characteristics of real policies. We designed these benchmarks to exercise SecGuru in worst-case scenarios of much larger-scale compared to our production policies. SecGuru has acceptable performance in all cases.

A general takeaway is that modern automated theorem proving technologies are suitable for encoding semantic properties of network policies and they can be used automatically and at scale in production environments.

Configuring network connectivity restrictions in data centers is well recognized as an important challenge [3, 31]. Section 7 summarizes numerous previous tools offering solutions to different aspects of network configuration. The related work has proposed a number of approaches for validating connectivity restrictions, using either custom data-structures and algorithms, and encoding into a specification language or directly into propositional satisfiability (SAT). None of the related work address compactly and comprehensively enumerate the differences between two policies. Such reports are essentially to reduce human effort to diagnose the problem, and are also amenable to automatic rectification.

Outline

The rest of the paper is organized as follows. Section 2 describes the architecture of SecGuru. Section 3 describes how policies are properties are encoded into bit-vector logic and the symbolic difference solver is given in Section 4. SecGuru is evaluated in Sections 5 and 6. Related work is reviewed in Section 7. Section 8 summarizes the results.

2. SECGURU

The core of SecGuru analysis engine is based on Z3, a Satisfiability Modulo Theories (SMT) solver, and uses the bit-vector logic support by Z3. The intuition behind this design is that firewall and router policies are essentially a set of constraints over IP addresses, ports, and protocol, each of which are bit-vectors of varying sizes. Therefore, analysis questions on these policies can be expressed as bit-vector

Status	Source Address	Src Port	Destination Address	Dst Port	Protocol
Permit	10.20.0.0/19	*Any*	157.55.252.0/30	*Any*	6
Deny	*Any*	*Any*	65.52.244.0/27	*Any*	4

Figure 3: Examples of contracts

logic formulas. As a consequence we represent policies and queries as logical formulas, and use satisfiability checking to extract answers. SecGuru relies on (compact) enumeration of satisfying assignments to provide detailed feedback. Modeling policy analysis questions as logical formulas allows the analysis to be semantic and agnostic of the low-level device syntax for access control.

SecGuru features two modes: contract validation and change-impact analysis (Figure 2).

Contract Validation

A contract is a property that should be preserved by a policy. It basically describes a set of traffic patterns that should be allowed or denied by the policy. For example, the Figure 3 below describes two contracts. The first contract describes a traffic pattern that should be accepted by the policy, and the second contract describes a traffic pattern that should be denied by the policy. Note that the contracts are agnostic of the low-level device syntax.

In the contract validation mode, SecGuru accepts a policy, (P), and a contract, (C), as input and provides one of the following results as output:

1. $C \rightarrow P$: The contract is preserved by the policy, i.e., the set of all traffic patterns described by C is a subset of the set of all traffic patterns accepted by the policy.
2. $C \rightarrow \neg P$: The contract is not preserved by the policy, i.e., traffic patterns accepted by C are denied by P.
3. $C \wedge P$: A proper subset of traffic patterns described in C is contained in the policy.

The contract validation mode can also be used as a mechanism to query the policy for information. When the response is (1) or (2), SecGuru may be additionally instructed to provide a listing of the specific rules that contributed to the decision. This may be useful for debugging problems with the policy. When the response is (3), SecGuru provides a compressed representation of this set.

The contract validation mode is particularly useful for maintaining complex policies. As each contract defines a property that should be preserved by the policy and is disconnected from the actual implementation, i.e., the low-level device syntax, it can be used as a regression test suite. New contracts can be added with the evolution of policies to improve the coverage of the policies. Contract validation can be performed prior to each update to make sure that updates preserve the essential properties.

Change-Impact Analysis

A common policy analysis scenario in Azure is ascertaining the impact of changes to a policy. Given a policy P_1 and a policy P_2, what is the impact of changing from policy P_1 to

policy P_2?. In the change-impact analysis mode, SecGuru accepts a policy P_1 and a policy P_2 as input and provides one of the following results as output:

1. $P_1 \equiv P_2$: Both P_1 and P_2 accept the same set of traffic patterns and reject the same set of traffic patterns.

2. $P_1 \not\equiv P_2$: P_1 and P_2 differ in the set of traffic patterns that they accept and reject. In addition, SecGuru also provides the following summary:

   - $P_1 \wedge \neg P_2$: Set of traffic patterns accepted by P_1, but not P_2.
   - $\neg P_1 \wedge P_2$: Set of traffic patterns rejected by P_1, but accepted by P_2.

In effect, this mode provides a semantic difference between the two policies, and can be used for evaluating changes to the policy and also for ascertaining how a policy has drifted away from the actual.

SecGuru *as a Production Monitoring Service*

Given the impact on both security and availability, Azure requires a pro-active and real-time method for detecting and fixing errors in the network connectivity policies. Thus, we have developed a monitoring infrastructure leveraging SecGuru for continuously validating network policies. The infrastructure is referred to as Azure network monitor (WaNetMon). Figure 4 contains a high-level architectural overview.

WaNetMon is designed as a real-time event-stream processing application. for this purpose. The monitoring servers that are part of WaNetMon poll network devices at regular intervals, collect configurations, and push them to an event stream called the configuration stream. Changes to the configurations create an update event in the stream. Similarly, policy contracts (device agnostic) for various devices types are stored in a database. The update events from the configuration stream triggers SecGuru to validate the updated configurations against their respective contracts, and pushes the results of validation into the device validation stream. WaNetMon features both reports and alerts based on the validation stream. Alerts raised due to validation failures are queued up for repair. The alert contains the detailed semantic difference that can be used to automatically deduce the changes needed to correct the policy configuration.

The next section describes in more detail how policies and queries are encoded into bit-vector logic and how semantic differences are enumerated succinctly.

3. FROM POLICIES TO BIT-VECTORS

We show how policies are directly encoded as predicates expressed in bit-vector logic.

3.1 Policies as Predicates

Bit-vectors are convenient for encoding IP headers. An IPv4 address is a 32 bit number and ports are 16 bit numbers. Protocols are also numerals using 16 bits. We can therefore write down each filter as a predicate with parameters that range over bit-vectors (32-bit, or 16-bit numerals). For example, r_1 and r_5 from Figure 1 have associated predicates:

$$r_1: \quad \begin{array}{l}(10.20.0.0 \le \mathit{srcIp} \le 10.20.31.255) \wedge \\ (157.55.252.0 \le \mathit{dstIp} \le 157.55.252.255) \wedge \\ \mathit{protocol} = 6\end{array}$$

$$r_5: \quad \begin{array}{l}(65.52.244.0 \le \mathit{dstIp} \le 65.52.247.255) \wedge \\ \mathit{protocol} = 4\end{array}$$

We use $r_i(\vec{x})$ to refer to the predicate associated with the i'th rule in a policy. The tuple $\vec{x}$ abbreviates $\langle \mathit{srcIp}, \mathit{srcPort}, \mathit{dstIp}, \mathit{dstPort}, \mathit{protocol}\rangle$. We use $r.\mathit{status}$ to access the status field of a rule. It is either *Allow* or *Deny*.

The meaning of a policy P is defined as a predicate $P(\vec{x})$ that evaluates to *true* when a packet with header $\vec{x}$ is allowed to pass through. Policies are given different semantics depending on where they are used. Hypervisor packet filters use a *Deny Overrides* convention. Router firewalls use a *First Applicable* convention. We summarize the semantics of policies according to these two conventions.

Definition 1 (Deny Overrides Policies). *Let* $\mathit{Allow} = \{r \in P \mid r.\mathit{status} = \mathit{Allow}\}$ *and likewise* $\mathit{Deny} = \{r \in P \mid r.\mathit{status} = \mathit{Deny}\}$. *The meaning of* P *with the Deny Overrides convention is the formula (linear in the size of the policy):*

$$P(\vec{x}) \quad = \quad (\bigvee_{r \in \mathit{Allow}} r(\vec{x})) \wedge (\bigwedge_{r \in \mathit{Deny}} \neg r(\vec{x}))$$

Thus, a packet is admitted if some Allow rule applies and none of the Deny rules apply.

Router firewall policies use the first applicable rule. Suppose a firewall has rules $r_1, \ldots, r_n$ that are either Allow or Deny rules, then the meaning is defined (linear in the size of the policy) by induction on n:

Definition 2 (First Applicable Policies). *Define* P, P_i *(for* $0 \le i < n$*) and* P_n *as:*

$$\begin{array}{lcll} P(\vec{x}) & = & P_1(\vec{x}) & \\ P_i(\vec{x}) & = & r_i(\vec{x}) \ \vee \ P_{i+1}(\vec{x}) & \textbf{if } r_i.\mathit{status} = \mathit{Allow} \\ P_i(\vec{x}) & = & \neg r_i(\vec{x}) \ \wedge \ P_{i+1}(\vec{x}) & \textbf{if } r_i.\mathit{status} = \mathit{Deny} \\ P_n(\vec{x}) & = & \mathit{false} & \end{array}$$

3.2 Solving Bit-vector Logic formulas

We showed how policies correspond to predicates over bit-vectors. Both policies using the Deny Overrides and the First Applicable semantics correspond to logical formulas. The predicates treat the parameters as bit-vectors and use comparison (less than, greater-than, equals) operations on the bit-vectors as unsigned numbers. Modern SMT solvers contain efficient decision procedures for bit-vector logic. Bit-vector logic expressive: it captures the operations that are common on machine represented fixed-precision integers, such as modular addition, subtraction, multiplication, bit-wise logical operations, and comparisons. The solvers leverage pre-processing simplifications at the level of bit-vectors and most solvers reduce formulas to propositional satisfiability where state-of-the-art SAT solving engines are used. We illustrated a direct encoding into bit-vector logic that is loss-less. The algorithms for solving bit-vector formulas is opaque. In the worst case the underlying SMT solver could use an algorithm that is asymptotically much worse than

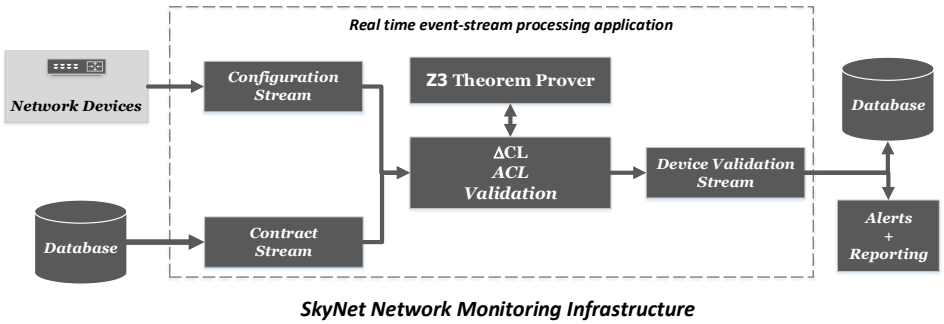

Figure 4: Continuous validation of network connectivity policies in production using SecGuru.

algorithms that have been specifically tuned to policy analysis (as for instance developed in [1, 4]), but as our evaluation shows, our approach easily scales an order of magnitude beyond what is required for modern data centers.

3.3 Leveraging SMT solver features

Encoding into bit-vector logic is flexible and high-level, but modern SMT solvers also provide features that are instrumental in solving problems efficiently. To give an example, one problem addressed in related work is to determine whether policies contain redundant rules [33]. A direct solution using our approach is to translate the original policy to a formula P and for each rule translate a policy without that rule into a formula P' and check for equivalence. The number of independent equivalence checks is linear in the number of rules. A more refined approach leverages incrementality supported by Z3 works by translating P into a formula P' where each *Allow* rule $r_i(\vec{x})$ formula is replaced by the strengthened formula $p_i \wedge r_i(\vec{x})$, and each *Deny* rule r_j is weakened to $p_j \vee r_j(\vec{x})$, where p_i, p_j are fresh predicates. We then assert the formula $P \not\equiv P'$. Suppose we want to check if the rule *Allow* rule r_k is redundant, then we check if the resulting state is satisfiable under the assumptions $\neg p_k \wedge (\bigwedge_{p_i \in Allow \setminus \{p_k\}} p_i) \wedge (\bigwedge_{p_j \in Deny} \neg p_j)$. The formula $P \not\equiv P'$ is asserted only once; and the underlying engine ensures that only state that depends on the changed assumptions has to be updated between rounds. We observed the incremental version to be more than twenty times faster for policies with a few hundred rules.

3.4 Complexity

We are not aware of a rigorous complexity analysis of firewall queries. Let us here note that checking difference of two Deny Overrides firewall policies is NP hard if the number of columns is unbounded: Given a clause $C_i : x \vee \overline{y} \vee u$ we can associate the rule $r_i : x \in [0:0] \wedge y \in [1:1] \wedge z \in [0:1] \wedge u \in [0:0]$, so a set of clauses $C_1 \wedge \ldots \wedge C_n$ is satisfiable iff the following policies $P_1 : x \in [0:1] \wedge y \in [0:1] \wedge z \in [0:1] \wedge u \in [0:1]$ and $P_2 : r_1, \ldots, r_n$ (of allow rules) are different. The number of columns in firewalls is of course fixed, and several data-structures and related polynomial time algorithms are reported in the literature. For instance [5] compiles simple firewall rules into tries. Our approach with encoding into bit-vector logic side-steps concerns about devising domain specific efficient algorithms. Compilation into bit-vector formulas is linear and SecGuru admits arbitrary queries that can be expressed over bit-vector logic.

4. ALL BV-SAT

Given two policies $P_1(\vec{x})$ and $P_2(\vec{x})$ what is their difference? We can of course characterize the differences as $P_1(\vec{x}) \wedge \neg P_2(\vec{x})$ and $\neg P_1(\vec{x}) \wedge P_2(\vec{x})$, but this says little to a system administrator about which packets are allowed by one and not the other. We would like a way to enumerate packets that belong to the differences in a succinct way. For this purpose we develop three increasingly more sophisticated algorithms for enumerating such packets in progressively more compact form. The algorithms work on arbitrary bit-vector formulas. We will use $\varphi[\vec{x}]$ for an arbitrary bit-vector formula with free variables $\vec{x}$.

The first is Algorithm All-SAT provided in Algorithm 1. It is based on enumerating all satisfiable values for a formula φ in a straight-forward way. It is the default solution to the All-SAT [6] problem when used with modern SAT solvers. The second, Algorithm All-BVSAT in Algorithm 2, enumerates *cubes* of values that satisfy φ. A cube is a cross-product of intervals. This representation corresponds closely to how firewall rules are represented in policies. The third, Algorithm All-BVSAT$\star$ in Algorithm 4, generalizes enumeration of cubes to *multi-cubes*. A multi-cube is a cross-product of *sets of intervals*. Both cubes and multi-cubes provide readable ways to inspect properties of policies. Multi-cubes, may however, provide an exponentially more succinct representation of differences than cubes. It is also easy to read off the number of values that satisfy a cube and (a multi-cube): it is the product of values in each interval (set). Besides being useful for analyzing firewall policies our algorithms can also be used in general for counting the number of solutions to a formula. This problem is also known as #SAT [12]. The use of multi-cubes is a particular good fit for firewall policies due to the way policies are normally specified over address ranges.

4.1 All-SAT

Algorithm 1 contains a straight-forward All-SAT algorithm. It queries a given formula B in a loop. It adds a disjunction of new disequalities to B in every loop iteration (here represented as a single disequality between a vector of values and a vector of variables). SAT and SMT solvers support incrementally adding constraints, so that in each iteration only the new disequality needs to be added to the state of the solver. Don't cares in a satisfying assignment also

can be found efficiently and dropped, resulting in stronger constraints and fewer iterations of the main loop. In spite of support for adding constraints incrementally, basic All-SAT enumeration suffers in practice from degraded performance as the set of disequalities grows. The enumeration obtained from an All-SAT loop may also be overly verbose. The next method address this limitation.

Algorithm 1: ALL-SAT

Input: Formula $\varphi[\vec{x}]$.
Output: values $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_k$, s.t.
$\varphi \equiv \vec{x} = \vec{v}_1 \vee \vec{x} = \vec{v}_2 \vee \ldots \vee \vec{x} = \vec{v}_k$
$B \leftarrow \varphi$;
$k \leftarrow 0$;
while B *is satisfiable* **do**
  $k \leftarrow k + 1$;
  $\vec{v}_k \leftarrow$ a satisfying assignment to B;
  $B \leftarrow B \wedge (\vec{x} \neq \vec{v}_k)$;
end
return $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_k$;

Algorithm 2: ALL-BVSAT

Input: Formula $\varphi[\vec{x}]$.
Output: Sets $\vec{S}_1, \vec{S}_2, \ldots, \vec{S}_k$, s.t.
$\varphi \equiv \vec{x} \in \vec{S}_1 \vee \vec{x} \in \vec{S}_2 \vee \ldots \vee \vec{x} \in \vec{S}_k$
$B \leftarrow \varphi$;
$k \leftarrow 0$;
while B *is satisfiable* **do**
  $k \leftarrow k + 1$;
  $\vec{v} \leftarrow$ a satisfying assignment to B;
  $\vec{S}_k \leftarrow \{v_1\} \times \ldots \times \{v_{|\vec{v}|}\}$;
  foreach *index* $i = 1 \ldots |\vec{v}|$ **do**
    $\vec{S}_k \leftarrow$ **Min-BVSAT**$(\varphi, \vec{S}_k, i)$;
    $\vec{S}_k \leftarrow$ **Max-BVSAT**$(\varphi, \vec{S}_k, i)$;
  end
  $B \leftarrow B \wedge (\vec{x} \notin \vec{S}_k)$;
end
return $\vec{S}_1, \vec{S}_2, \ldots, \vec{S}_k$;

4.2 All-BVSAT using cubes

Enumerating one solution at a time is unsatisfactory when many solutions can be represented succinctly using cubes. Let us illustrate the idea of enumerating solutions as cubes on an example. We keep the query and values abstract to retain the generality of the algorithm.

1. Find initial $ip_0, port_0$, such that $(ip_0 = ip) \wedge (port_0 = port) \Rightarrow \varphi$.
2. Maximize interval $[lo_{ip}, hi_{ip}]$, such that $ip_0 \in [lo_{ip}, hi_{ip}]$ and $lo_{ip} \leq ip \leq hi_{ip} \wedge (port_0 = port) \wedge \neg\varphi$ is unsatisfiable.
3. Maximize next interval $[lo_{port}, hi_{port}]$, such that $port_0 \in [lo_{port}, hi_{port}]$ and $lo_{ip} \leq ip \leq hi_{ip} \wedge lo_{port} \leq port \leq hi_{port} \wedge \neg\varphi$ is unsatisfiable.
4. Produce the set of intervals $[lo_{ip}, hi_{ip}] \times [lo_{port}, hi_{port}]$ All pairs of values in these two intervals satisfy φ.
5. Update the query to $\varphi := \varphi \wedge \neg(lo_{ip} \leq ip \leq hi_{ip} \wedge lo_{port} \leq port \leq hi_{port})$ and repeat the loop.

The walk-through assumed there was some way to maximize intervals efficiently. We provide an algorithm for maximizing intervals in Algorithm 3. We rely on some notation for describing the algorithms.

We use S to range over sets of bit-vector values of a given size. The representation that will be convenient for the sets S is as a union of intervals. So for example $S := [0..3] \cup [6..7]$ is a set with the values $\{0, 1, 2, 3, 6, 7\}$. The predicate $x \in S$ expands to a disjunction $lo_1 \leq x \leq hi_1 \vee \ldots \vee lo_n \leq x \leq hi_n$, where $S = [lo_1..hi_1] \cup \ldots \cup [lo_n..hi_n]$. The predicate $\top$ is true on all values (bit-vectors). It also is used to represent the set that contains all bit-vector values for a given length, so $x \in \top$ expands to *true*. When S is a non-empty set then $\min S$ is the minimal element in S. $\vec{S}$ is a cross-product of sets $S_1 \times \cdots \times S_n$. $\vec{S}[i \mapsto S]$ is the product $S_1 \times \cdots \times S_{i-1} \times S \times S_{i+1} \times \cdots \times S_n$; in other words, it is the product $\vec{S}$ with the ith set replaced by S. The predicate $\vec{x} \in \vec{S}$ is short for $\bigwedge_{i=1}^{|\vec{x}|} x_i \in S_i$.

Algorithm 2 enumerates all solutions to a formula φ as cubes (products of intervals). In each iteration it selects some value $\vec{v}$ that is not yet covered by any of the existing sets. It then widens the value $\vec{v}$ as much as possible to intervals.

It relies on a procedure **Min-BVSAT** and a symmetric variant **Max-BVSAT** for extending a satisfying interval maximally down and up. Algorithm 3 provides an implementation of **Min-BVSAT**. It first checks if there is an evaluation to the parameters $\vec{x}$ such that the value of x_i is below $\min S_i$ and that satisfies $\neg\varphi$. As long as it is the case, it checks if there is a value still below $\min S_i$, but above the previous value. This process ensures that the post-condition is established: that all values in $\vec{S}[i \mapsto S_{lo}]$ satisfy φ and that S_{lo} is extended maximally downwards with values that maintain φ. By induction on i, this implies that the resulting cube $\vec{S}_k$ is maximum: it is not possible to extend any of the faces without loosing the property of satisfying φ.

The proposed implementation uses linear search, where the bound on the number of loop iterations is given by the size of the domain of x_i. If x_i is a 32-bit bit-vector then the potential number of iterations is in the order of 2^{32}. Nevertheless, when profiling these algorithms in the context of **All-BVSAT** for production policies we found they converged in average within ten steps for each cube. Nevertheless, we had to use a binary search based implementation of these procedures for **All-BVSAT**$\star$. Binary search is asymptotically much more efficient (linear in the number of bits instead of exponential). We observed that the average number of steps required was five.

Algorithm 3: Min-BVSAT. Extend S_i downwards.

Input: Formula $\varphi[\vec{x}]$, sets $\vec{S}$ s.t. for every $\vec{v} \in \vec{S}$, $\varphi[\vec{v}]$, and index i into $\vec{S}$.
Output: $\vec{S}[i \mapsto S_{lo}]$, such that $S_{lo} \supseteq S_i$, and for every $\vec{v} \in \vec{S}[i \mapsto S_{lo}]$, $\varphi[\vec{v}]$. If $\min S_{lo} > 0$, then there is some value $\vec{w} \in \vec{S}[i \mapsto \{\min S_{lo} - 1\}]$, such that $\varphi[\vec{w}]$ is false.
$\vec{S}' \leftarrow \vec{S}[i \mapsto \top]$;
$l \leftarrow \min S_i$;
$B \leftarrow \neg\varphi \wedge \vec{x} \in \vec{S}' \wedge x_i < l$;
while B *is satisfiable* **do**
  $l \leftarrow$ the satisfying assignment to x_i;
  $B \leftarrow B \wedge l < x_i$;
end
return $\vec{S}[i \mapsto S_i \cup [l + 1.. \min S_i]]$

4.3 All-BVSAT using multi-cubes

We can in some cases do exponentially better than enu-

Algorithm 4: ALL-BVSAT⋆

Input: Formula $\varphi[\vec{x}]$.
Output: Sets $\vec{S}_1, \vec{S}_2, \ldots, \vec{S}_k$, s.t.
$\varphi \equiv \vec{x} \in \vec{S}_1 \vee \vec{x} \in \vec{S}_2 \vee \ldots \vee \vec{x} \in \vec{S}_k$

```
B ← φ;
k ← 0;
while B is satisfiable do
    v⃗ ← a satisfying assignment to B;
    foreach index j = 1 ... k do
        S⃗_j ← Extend(S⃗_j, v⃗);
        B ← B ∧ (x⃗ ∉ S⃗_j);
    end
    if φ ∧ B ∧ x⃗ = v⃗ is still satisfiable then
        k ← k + 1;
        S⃗_k ← {v_1} × ... × {v_|v⃗|};
        foreach index i = 1 ... |v⃗| do
            S⃗_k ← Min-BVSAT(φ, S⃗_k, i);
            S⃗_k ← Max-BVSAT(φ, S⃗_k, i);
        end
        B ← B ∧ (x⃗ ∉ S⃗_k);
    end
end
return S⃗_1, S⃗_2, ..., S⃗_k;
```

merating cubes by using *multi-cubes*. Instead of enumerating product of intervals, enumerate products of sets of intervals. Policy rules may contain multi-cubes with up to four ranges corresponding to both source and destination addresses and ports. Multi-cubes can be much more succinct than cubes, for example the set

$$\underbrace{([0..3] \cup [6..7]) \times \cdots \times ([0..3] \cup [6..7])}_{N \text{ times}}$$

requires 2^N cubes to represent. The algorithm that we will describe next requires at most N multi-cubes to reconstruct the above set.

The idea behind the algorithm can be explained as follows. The algorithm extends **All-BVSAT** by trying to insert new values into previous cubes (that then become multi-cubes). It first relies on finding some value $\vec{w}$ that has not yet been included in any of the existing multi-cubes. Then, for each multi-cube S_j and each index i into $\vec{w}$ it determines whether $\vec{S}_j$ can be extended by using w_i and the existing sets of values from $\vec{S}_j$ in positions different from i. So it checks whether the vector $\vec{S}'$, where $\vec{S}'$ is obtained from $\vec{S}_j$ by replacing the i'th set by $\{w_i\}$, implies φ (or equivalently, has an empty intersection with $\neg\varphi$). If it does, then the algorithms for extending $\vec{S}'$ at the i'th index can be applied.

Algorithm 5: Extend. Extend set assignment.

Input: Set $\vec{S}$ and formula φ such that $\varphi[\vec{v}]$ for every $\vec{v} \in \vec{S}$. A vector $\vec{w}$, such that $\vec{w} \notin \vec{S}$ and $\vec{w}$ satisfies φ

```
foreach index i = 1, ..., |w⃗| do
    S⃗' ← S⃗[i ↦ {w_i}];
    if ¬φ ∧ (x⃗ ∈ S⃗') is unsatisfiable, and w_i ∉ S_i then
        S⃗' ← Min-BVSAT(φ, S⃗', i);
        S⃗' ← Max-BVSAT(φ, S⃗', i);
        S⃗ ← S⃗[i ↦ S_i ∪ S'_i];
    end
end
return S⃗;
```

We should note that our algorithm may not find the smallest multi-cube representation of the satisfying assignments to φ, but it finds one within a polynomial overhead. To see this, let $\vec{S}_1, \ldots, \vec{S}_k$ be an arbitrary decomposition into multi-cubes. Without loss of generality consider $\vec{S}_1$ and suppose it has up to n intervals in each dimension. There are up to n covers of the set of disjoint intervals. For each such cube it takes at most $\sum_i |\vec{S}_{1i}|$ (where $\vec{S}_{1i}$ is the number of intervals in the ith coordinate of $\vec{S}_1$) iterations to cover all other cubes in $\vec{S}_1$.

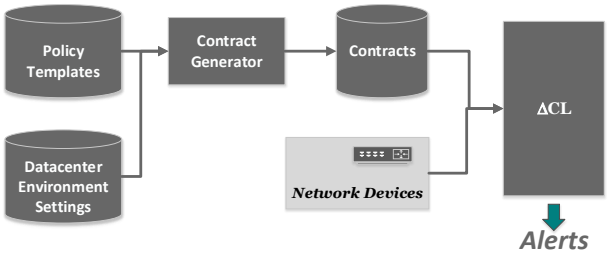

Figure 5: Workflow for creating contracts and validating policies.

5. EXPERIENCE

We now describe the different experiences and the benefits from using SecGuru in our environment.

5.1 Continuous Validation and Monitoring

Network policies in Azure are under constant flux. The high-level policies themselves do not change significantly. However, the instantiation of these policies using IP addresses changes frequently and also varies depending on the environment. For example, all management interfaces have a common restrictive network policy. However, the IP addresses of these services vary depending on the environment. In addition, when capacity is increased, there is a corresponding increase in the IP addresses assigned to the management services. All these changes result in a constant flux in the policies deployed in the various devices.

While coping with these changes, we also need to assure both the availability and security of services. The number of places where the policies are enforced dramatically increases the scale and extent of the problem. An important security policy is to make sure that private management interfaces are not exposed beyond necessary. For example, connectivity to management interfaces of network devices is tightly controlled to trusted services. If such interfaces are accidentally exposed to potentially malicious users, then they will become a target for exploitation.

Thus, Azure continuously validates network policies from the time they are initially provisioned in WaNetMon (Recall Figure 4). In our settings, policies vary depending on the type of devices. For example, routers enforce a type of policy, while a hypervisor packet filter enforces a different type of policy. The concrete policy varies depending on the environment in which the devices are deployed. Therefore, WaNetMon uses a workflow described in Figure 5 for validating the policies. WaNetMon has an inventory of contract templates that capture the security and availability properties of the policy, and creates custom contracts

based on the environment settings. WANETMON uses these custom contracts to continually validate policies.

We also leverage the same contracts as a regression test suite to ensure the sanity of routine maintenance operations for network policies.

SECGURU has had a measurable positive impact in prohibiting policy misconfigurations. There were several instances where an incorrect change was avoided from using SECGURU as a regression test suite. In addition, alerts from continuous validation of policies using SECGURU in WANETMON help us proactively repair policy deviations from the normal. Such deviations are commonly the result of administrative operations performed for debugging live-site issues.

5.2 Maintaining Complex Legacy Policies

The previous section described using SECGURU to validate policies that are well understood. However, there are situations in which complex policies that use rules covering many different objectives have to be maintained. For example, we had a legacy Edge ACL that comprised more than three thousand rules. It was an onerous challenge maintaining this ACL manually. The ACL evolved over several years through an ad hoc manual process. In addition, growth and churn in IP address allocations resulted in continuous updates to this ACL introducing further complexity. The ACL was not amenable to simple human inspection, and often updates led to misconfigurations and connectivity outages.

Because of the huge business impact of the misconfigurations, we needed an incremental method for making required updates and simplifications to the ACL. More concretely, we needed a method to ensure that changes to the policy do not violate essential security or availability properties, and that the impact of the ACL change is along the intent. For example, we may identify several redundant rules in the ACL and remove them. Alternatively, we may add rules to allow new IP address ranges. For each of these updates, we need the confidence in the newly deployed ACLs.

We leveraged SECGURU as a means of running a regression test suite for the edge ACL. In SECGURU's contract validation mode, the contracts essentially act as regression test cases for the ACL. We divided the connectivity that the Edge enables into two buckets, namely connectivity that is well understood and don't cares. For the cases, that are well understood, we created contracts. This is essentially an under-specification of the policy. ACL changes were done to enable new connectivity, or cleanup existing don't cares. Prior to making these changes, we run the regression test suite. This test gave us the confidence that we are not breaking the connectivity that is known and well understood. After making the change, we add additional contracts to the regression test cases to cover the most recent updates.

Extraction of contracts helped documenting and understanding the existing ACL, and an iterative process around this model led to massive simplifications. We were able to reduce the ACL to less than 1000 lines without any major connectivity outages or business impact.

5.3 Coping with Diverse Semantics

SECGURU's semantic and symbolic analysis have proved to be useful for coping with diverse semantics in a number of cases, and we highlight this utility using an additional example. Most network devices allow a stateless implementation of stateful ACLs. In Figure 6, $Policy_1$ allows all traffic from source address range 172.64.0.0/15. Note the additional highlighted rule in $Policy_2$. This rule additionally allows any TCP packet that is part of an ongoing communication. In the TCP protocol, all packets that are part of an ongoing communication have either the "ACK" or "RST" packet set in the TCPFlags field. The highlighted rule instructs the router to allow any packet that has either the "ACK" or "RST" flag set. Thus, $Policy_2$ allows the target to communicate with anybody, and the return traffic will be permitted per the highlighted. However, inbound connections are permitted only from 172.64.0.0/15. SECGURU models the precise semantics using bit-vector logic. Table 1 shows the drift report that lists all the additional traffic pattern that $Policy_2$ allows, and it shows that the policy additionally allows all valid TCP packets except SYN packets.

Src Addr	Src Port	Dst Addr	Dst Port	Protocol	TCP Flags
Any	*Any*	*Any*	*Any*	TCP	RST
Any	*Any*	*Any*	*Any*	TCP	ACK
Any	*Any*	*Any*	*Any*	TCP	FIN-ACK
Any	*Any*	*Any*	*Any*	TCP	PSH-ACK
Any	*Any*	*Any*	*Any*	TCP	RST-ACK
Any	*Any*	*Any*	*Any*	TCP	URG-ACK

Table 1: What is additionally allowed by $Policy_2$ in Figure 6?

6. BENCHMARKS

We evaluate SECGURU using production policies. In addition, we also created additional benchmarks to exercise SECGURU in worst-case scenarios and beyond the scale of policies we have seen in production. In the following we describe the characteristics of firewall and router policies in our benchmark. By design, our benchmarks do not add redundant rules that may be subject to trivial simplifications. We will make our benchmarks available publicly when the paper is published.

6.1 Firewall Policies

A prevalent security practice in configuring firewall policies is to follow the "default deny" [11] strategy. Default-deny policies deny everything by default and allow only traffic patterns that are explicitly allowed, and this is in line with the principle of fail-safe defaults [27]. Our benchmarks focus only on such policies.

In default-deny policies, we observe that it is common to have a combination of allow and deny rules such that an allow rule permits connectivity to a wide-range of IP addresses, and deny rules block connectivity to a smaller subset of this range for specific protocols or ports. Table 2 describes such a sequence. The first rule in the table allows unrestricted connectivity for entire range of addresses described by 128.230.0.0/16. The second rule denies TCP protocol for 4 IP addresses described by 128.230.33.42/30, and the third rule denies any connectivity on ports $100-200$ for IP addresses described in 128.230.33.64/30.

We synthetically generate these patterns as follows. The address ranges to be used for the source and destination IP addresses, the port ranges to be used for the source and destination ports, and the protocol ranges are all provided as

```
1  permit tcp 172.64.0.0/15 any
```

```
1  permit tcp any any established
2  permit tcp 172.64.0.0/15 any
```

Figure 6: Stateless implementation of a stateful ACL. $Policy_1$ (left-handside example) uses a stateless ACL, but $Policy_2$ right-handside example additionally used the highlighted rule to mimic statefulness.

Action	Src Addr	Src Port	Dst Addr	Dst Port	Protocol
ALLOW	⋆	⋆	128.230.0.0/16	⋆	⋆
DENY	⋆	⋆	128.230.33.42/30	⋆	TCP
DENY	⋆	⋆	128.230.33.64/30	100-200	⋆

Table 2: An example pattern of allow and deny rules

input. Then, an allow rule is created by picking a random traffic pattern from the allowed range. From this traffic pattern, a set of traffic patterns are picked at random to create block rules. We continue to generate such patterns until we have the desired number of rules.

We also point out that block rules whose address range does not overlap with an existing allow rule are redundant in a default-deny policy. We do not create such rules for our benchmark policies.

6.2 Router Policies

We did not come across a representative pattern of rules in router policies that could be of relevance to a benchmark. Router policies are first applicable policies, i.e., the first applicable rule in the policy overrides all other rules. First applicable policies allow an administrator to make incremental changes to the policy without having to redesign the complete policy. For example, let us consider that an administrator encounters a new traffic pattern that should be allowed (or denied) and it is currently denied (or allowed) by the policy. Then, he may append a new rule to the beginning of the policy to enable this scenario without redesigning the policy. In our production environments, we make only systematic changes to router policies and avoid such ad hoc practices.

We believe first applicable policies comprising a random sequence of allow and deny rules would be an appropriate benchmark for evaluation. Therefore, we designed such policies. Similar to our generation method for firewall policies, the address ranges to be used for the source and destination IP addresses, the port ranges to be used for the source and destination ports, and the protocol ranges are all provided as input. Then, we randomly pick traffic patterns from the allowed ranges to create allow or block rules until we have created the desired number of rules.

In first-applicable policies, rules whose address range is a subset of preceding rule are redundant. We avoid adding such redundant rules to the benchmarks.

6.3 Experimental Methodology

We performed two experiments. The first experiment evaluated the scalability of contract validation, and the second experiment evaluated the scalability of change-impact analysis. We used a computer with an Intel dual-core processor with a clock rate of 2.8GHz and 4GB RAM. For both experiments, we used both real production policies and a set of realistic synthetic policies.

We created synthetic policies with the number of rules ranging between 100 and 15000. For each policy, we created five contracts, wherein each contracts checks for connectivity to a random choice of IP address ranges. We report the average time for evaluating a contract on each policy. In the second experiment, we evaluated the scalability of change-impact analysis in SecGuru. We created pairs of synthetic policies of sizes ranging between 100 and 15000 rules. Each pair comprises two unrelated policies that do not have any intersecting traffic patterns, and represents the worst-case scenario for change-impact analysis. We report the time taken to analyze the difference between such policy pairs. Our results show that SecGuru has acceptable performance in all cases.

6.4 Results

Table 3 contains the result of our experimental evaluation. The firewall and router policies that we observed in Azure contained between a few hundred to a few thousand rules. SecGuru was efficient in analyzing these policies. For these policies, SecGuru took an average time of 0.3 seconds for contract validation, an average time of 1.5s for enumerating cubes, and an average time of 3s for enumerating multi-cubes.

The time taken by SecGuru to analyze the synthetic benchmarks quickly increases with the number of rules. This is expected because the synthetic policies are worst-case scenarios for SecGuru. However, these worst-case scenarios are less likely to occur in practice.

For contract validation, SecGuru's performance is still acceptable. For synthetic policies whose sizes are similar to our production policies, the time taken for validation is still a fraction of a second. For larger policies, the time taken is still under a minute.

For change-impact analysis on synthetic policies, SecGuru is slower when compared to production policies. This is also expected because the pair of policies in the synthetic benchmarks are totally unrelated and this scenario is the worst-case for both the cube and multi-cube enumeration. These scenarios are highly unlikely because we always compare policies that are related to one another. In addition, a number of packet filtering devices have a static limit on the number of rules that can be added. Therefore, it is very unlikely that we need scalability beyond 1000 rules.

A key strength of this approach is that we have a general purpose analysis engine that can be used to analyze all types of network connectivity restrictions irrespective of their semantics. This is particularly suited for a complex environment like Azure where we have to deal with a number of different policies with varying semantics. Once the policies are faithfully encoded as bit-vector logic formulas, we can apply the normal policy operators without having to worry about their semantics.

		Contract Validation		Change-Impact Analysis			
				Cubes		*Multicubes*	
	#Rules	*Firewall Policies*	*Router Policies*	*Firewall Policies*	*Router Policies*	*Firewall Policies*	*Router Policies*
Real Policies	200-1000	0.3s	0.3s	1.5s	1.5s	3s	3s
Synthetic policies	100	0.39s	0.305s	6s	7s	6s	8s
	500	0.483s	0.270s	36s	34s	40s	48s
	1000	0.811s	0.315s	1m 20s	1m 18s	1m 18s	1m 40s
	5000	5.272s	0.7s	7m 8s	7m 33s	7m 14s	11m 16s
	10000	15.74s	1.25s	20m 25s	21m 12s	16m 40s	37m 23s
	15000	38.48s	1.8s	32m 52s	33m 20s	28m 5s	59m 7s

Table 3: Evaluation of SecGuru symbolic analysis engine.

7. RELATED WORK

The Margrave firewall analysis engine [26] encodes firewall rules and queries into first-order logic. It uses KodKod [29] to search for finite models. We found that Margrave does not work on our scenarios: first, Margrave does not produce the complete differences between policies in a compact way like SecGuru does. Second, Margrave only supports router policies, and not the firewall policies in our benchmarks. We also observed that the current implementation of Margrave does not adequately scale for some of our large router policies. However, we hypothesize that our algorithms for enumerating solutions compactly may be valuable in the context of Margrave and related scenarios.

Conformance of firewall configurations with respect to security policies is checked in [32] using an SMT solver for the theory of integer linear arithmetic. This is similar to SecGuru's contract validation mode. SAT and QBF solvers are explored more recently [35] for checking firewall properties and optimizing firewall rules. These approaches also do not address compact enumeration of solutions.

Formal firewall conformance testing is addressed in [7]. The tool uses the high-level and powerful environment of Isabelle/HOL to synthesize test-cases from constraint satisfaction problems that are solved using Z3.

The Vantage tool [4] uses algorithms for enumerating differences between policies. Similar to SecGuru it aims at enumerating traffic patterns compactly that are blocked by one policy and not the other. Vantage enumerates what corresponds to cubes, while we introduce the more succinct representation of multi-cubes and the corresponding enumeration algorithms. Vantage [5] uses a trie-based data-structure to store sets of intervals, taking advantage of how IP address ranges are represented using a k-bit prefix followed by wild card bits. It implements specialized algorithms for computing intersection of rectangles based on the trie data-structure. These data-structures are essentially BDDs. BDDs were previously used in [14, 23, 33] for representing policies and evaluating queries. Firewall Decision Diagrams, described in [13], is a variant of BDDs that is tuned to firewalls. They support compact enumeration of cubes but not multi-cubes.

Other tools adopt simulation, such as the commercial firewall analyzer AlgoSec [2] based on Fang [25] and its sequel Lumeta [30]. The tool lets administrators answer queries involving router and firewall configurations. A query may be of the form whether a machine is accessible. Given a query, it simulates the traversal of the corresponding packets in the network, and reports the set of packets that arrived in the destination. Similarly, a structured firewall query language is proposed in [21], and custom trie-based data-structures are developed to index firewall rules and answer queries over the rules.

Verification tools for access-control policies is an active research area. One way of classifying the work is whether it deals with single or multiple states. For policies with a single fixed state, related work verifies properties of the fixed state. This category includes verification for access-control policies such as SPKI/SDSI [18] and XACML [10, 15, 16, 24]. Others consider policies that additionally have state changes [9, 17, 28]. This work considers the ARBAC model that admits state changes corresponding to administrative actions granting and/or revoking roles, and verifies the safety of policies under all sequences of allowed state changes using temportal safety properties (nothing bad happens). Our work belongs in the first fixed-state category, and is focused on network connectivity restriction policies.

There is large and growing body of work on modeling the forwarding state of network, and validating it for correctness of end-end reachability properties [19, 20, 22, 34]. These tools are focussed on detecting problems such as forwarding loops and black-holes, and not on semantically validating the correctness of network connectivity policies. The compact enumeration of drifts proposed in this paper could be useful for verifying the forwarding state as well.

Google Capirca is a system that facilitates automatic generation of network ACLs for various platforms such as Cisco and Juniper. It has an ACL checker module that facilitates syntactic analysis of the ACL payload. The module is based on basic string comparisons. It does not offer the types of semantic analysis that SecGuru provides.

8. SUMMARY

This paper described a declarative static-analysis approach for analyzing the correctness and consistency of network connectivity policies. Our approach is based on precisely encoding policies and their analysis questions as bit-vector logic formulas and solving them using the Z3 constraint solver. We implemented our approach in a tool called SecGuru. The key strength of SecGuru lies in the fact that it is a general purpose engine that can be used to analyze several types of network connectivity policies based on their precise semantics. SecGuru is deployed in Azure, where it is checking the integrity of hundreds of routers and firewalls servicing millions of machines. It has had measurable positive impact in managing network policies at scale.

9. REFERENCES

[1] Acharya, H. B., and Gouda, M. G. Linear-Time Verification of Firewalls. In *ICNP* (2009), pp. 133–140.

[2] Algorithmic Security Inc. Firewall Analyzer: Make your firewall really safe, 2006. (Whitepaper).

[3] Bellovin, S. M., and Bush, R. Configuration management and security. *IEEE Journal on Selected Areas in Communications 27* (2009), 268–274.

[4] Bhatt, S., Okita, C., and Rao, P. Fast, Cheap, and in Control: Towards Pain-Free Security. In *USENIX Systems Administration Conference* (2008), pp. 75–90.

[5] Bhatt, S., and Rao, P. Enhancements to the Vantage Firewall Analyzer. Tech. Rep. HPL-2007-154R1, HP Laboratories, 2007.

[6] Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[7] Brucker, A. D., Brügger, L., and Wolff, B. hol-TestGen/fw - An Environment for Specification-Based Firewall Conformance Testing. In *ICTAC* (2013), Z. Liu, J. Woodcock, and H. Zhu, Eds., vol. 8049 of *Lecture Notes in Computer Science*, Springer, pp. 112–121.

[8] de Moura, L., and Bjørner, N. Z3: An Efficient SMT Solver. In *TACAS 08* (2008).

[9] Ferrara, A. L., Madhusudan, P., and Parlato, G. Security Analysis of Access Control through Program Verification. In *CSF* (2012), IEEE Computer Society.

[10] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. Verification and change-impact analysis of access-control policies. In *ICSE* (2005), ACM, pp. 196–205.

[11] Garfinkel, S., and Spafford, G. *Practical UNIX and Internet security*. O'Reilly, 1996.

[12] Gomes, C. P., Sabharwal, A., and Selman, B. Model counting. In Biere et al. [6], pp. 633–654.

[13] Gouda, M. G., and Liu, A. X. Structured firewall design. *Computer Networks 51*, 4 (2007), 1106–1120.

[14] Gupta, S., LeFevre, K., and Prakash, A. SPAN: a unified framework and toolkit for querying heterogeneous access policies. In *HotSec* (2009), USENIX, pp. 5–5.

[15] Hu, H., and Ahn, G. Enabling verification and conformance testing for access control model. In *SACMAT* (2008), ACM, pp. 195–204.

[16] Hughes, G., and Bultan, T. Automated verification of access control policies using a sat solver. *Int. J. Softw. Tools Technol. Transf. 10*, 6 (2008), 503–520.

[17] Jayaraman, K., Ganesh, V., Tripunitara, M., Rinard, M., and Chapin, S. Automatic error finding in access-control policies. In *CCS* (2011), ACM, pp. 163–174.

[18] Jha, S., and Reps, T. W. Model Checking SPKI/SDSI. *J. of Computer Security 12*, 3–4 (2004), 317–353.

[19] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 9–9.

[20] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.* (Sept. 2012), 467–472.

[21] Liu, A. X., Gouda, M. G., Ma, H. H., and Ngu, A. H. H. Firewall Queries. In *International Conference On Principles Of DIstributed Systems* (2004), pp. 197–212.

[22] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM.

[23] Marmorstein, R. M., and Kearns, P. An Open Source Solution for Testing NAT'd and Nested iptables Firewalls. In *LISA* (2005), pp. 103–112.

[24] Martin, E., and Xie, T. A fault model and mutation testing of access control policies. In *WWW* (2007), ACM, pp. 667–676.

[25] Mayer, A. J., Wool, A., and Ziskind, E. Fang: A Firewall Analysis Engine. In *IEEE Symposium on Security and Privacy* (2000), pp. 177–187.

[26] Nelson, T., Barratt, C., Dougherty, D. J., Fisler, K., and Krishnamurthi, S. The margrave tool for firewall analysis. In *LISA* (Berkeley, CA, USA, 2010), USENIX Association, pp. 1–8.

[27] Saltzer, J. H., and Schroeder, M. D. The Protection of Information in Computer Systems. *Proc. of the IEEE* (1975).

[28] Stoller, S. D., Yang, P., Ramakrishnan, C. R., and Gofman, M. I. Efficient policy analysis for administrative role based access control. In *CCS* (2007), ACM, pp. 445–455.

[29] Torlak, E., and Jackson, D. Kodkod: A Relational Model Finder. In *TACAS* (2007), pp. 632–647.

[30] Wool, A. Architecting the lumeta firewall analyzer. In *USENIX Security Symposium* (2001).

[31] Wool, A. A Quantitative Study of Firewall Configuration Errors. *IEEE Computer 37* (2004), 62–67.

[32] Youssef, N. B. S. B., and Bouhoula, A. Automatic Conformance Verification of Distributed Firewalls to Security Requirements. In *IEEE ICSC* (2010), pp. 834–841.

[33] Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C.-N., and Mohapatra, P. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *SP* (2006), IEEE, pp. 199–213.

[34] Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., and Vahdat, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 87–99.

[35] Zhang, S., Mahmoud, A., Malik, S., and Narain, S. Verification and synthesis of firewalls using sat and qbf. In *ICNP* (2012), IEEE, pp. 1–6.