Policy Anomaly Detection for Distributed IPv6 Firewalls

Claas Lorenz[1,2] and Bettina Schnor[2]
[1]*genua mbh, Kirchheim, Germany*
[2]*Institute for Computational Science, Potsdam University, Potsdam, Germany*

Keywords: Security, Firewalls, IPv6, Model-Checking.

Abstract: Concerning the design of a security architecture, Firewalls play a central role to secure computer networks. Facing the migration of IPv4 to IPv6, the setup of capable firewalls and network infrastructures will be necessary. The semantic differences between IPv4 and IPv6 make misconfigurations possible that may cause a lower performance or even security problems. For example, a cycle in a firewall configuration allows an attacker to craft network packets that may result in a Denial of Service. This paper investigates model checking techniques for automated policy anomaly detection. It shows that with a few adoptions existing approaches can be extended to support the IPv6 protocol with its specialities like the tremendously larger address space or extension headers. The performance is evaluated empirically by measurements with our prototype implementation *ad6*.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

IPv6 has a continiously growing share of the traffic on the Internet (see (Google, 2015)). The slow but steady migration from IPv4 to IPv6 is a challenge for a variety of stakeholders like operators, administrators and security officials. For organizations with separated networks the change or upgrade of firewall technology is not a trivial task and can be accompanied by challenges regarding the consistency and security of the new firewall rulesets (i.e. see (Caicedo et al., 2009) p. 40, RFC 6180 (Arkko and Baker, 2011) p. 14). On the other hand, it can also be seen as a chance to get rid of historically grown and inefficient firewall policies and network configurations. For small policy sets a manual migration might be sufficient but it is prone to error and thus, the analysis of large policy sets cannot be achieved this way and enforces a fully automatic approach. Real world firewalls in larger organizations tend to have from a thousand up to several ten thousands of rules which is far beyond the capability of manual approaches.

This paper focuses on supporting this transition by verifying properties of firewalls and networks formally and thus, allows to compare the semantical state of the configuration before and after the migration.

The examined semantical properties are called *anomalies* and range from simple but clear misconfigurations to security threats. The considered anomalies are reachability, cyclicity, shadowing and cross-path. They are defined in (Al-Shaer and Hamed, 2004), (Yuan et al., 2006) and (Jeffrey and Samak, 2009) and verbosely introduced as follows:

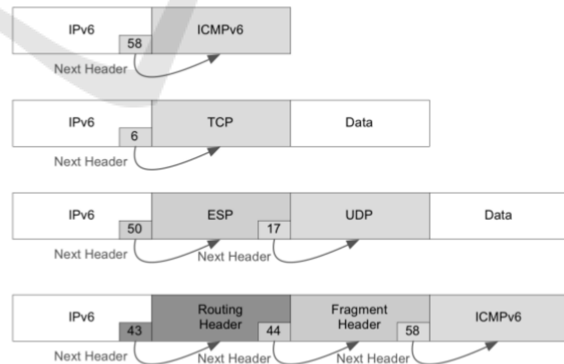

Figure 1: Examples for IPv6 Header Chains (originally inspired by (Biondi and Ebalard, 2006) p. 61).

- A *cycle* occurs if the control flow through the firewall contains a loop and there exists a packet that would be handled by this infinite path.
- A rule is *unreachable* if all possible packets are filtered by preceding rules. A simple example is given by the following IPTables script:

```
ip6tables -P OUTPUT DROP
ip6tables -A OUTPUT -j ACCEPT
```


Lorenz C. and Schnor B..
Policy Anomaly Detection for Distributed IPv6 Firewalls.
DOI: 10.5220/0005517402100219
In *Proceedings of the 12th International Conference on Security and Cryptography* (SECRYPT-2015), pages 210-219
ISBN: 978-989-758-117-5
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

```
ip6tables -A OUTPUT -p tcp \
          -j ACCEPT
```

The rule *ip6tables -A OUTPUT -j ACCEPT* matches all packets and therefore the succeeding rule cannot be reached.

- *Shadowed* rules do not have any effect on the filtering characteristics of the firewall since they are reachable but there exists no packet that can be matched by the shadowed rule. This is due to previous rules that filter all traffic matchable by the shadowed rule. The following IPTables script gives a small example:

```
ip6tables -P OUTPUT DROP
ip6tables -A OUTPUT -p tcp \
          -j ACCEPT
ip6tables -A OUTPUT -p tcp \
          --dport 80 -j ACCEPT
```

The second rule is reachable only by non-TCP packets, but cannot match them.

- The *cross-path* anomaly can only occur in network environments with multiple redundant routes to a host. If there are firewalls on these paths which handle the same traffic differently then this could result in severe security issues because an attacker would be able to send undesirable packets to his target.

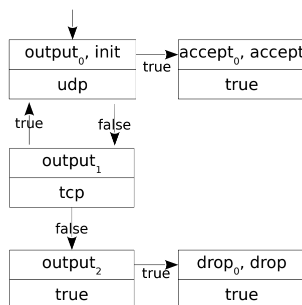

Figure 2: Example of a decision diagram with a cycle. The upper part of each node shows propositions representing meta information of the state while the lower part is its decision term.

We investigated whether these anomalies could be detected with a model checking approach also for IPv6 networks. Concerning model checking the challenges of IPv6 lay in the larger base header which has 320Bit compared to 104Bit in IPv4 (without options). The question is whether it can be handled efficiently. The same applies to the extension header chains as depicted in Figure 1 which allow chains of arbitrary length. Since these did not occur in IPv4 it must be shown that a formal approach can model them as well. Our approach investigates networks with stateless packet filters (firewalls of the first generation) that make decisions based on single packets rather than sequences.

In the next section, related work for the formal verification of firewalls is discussed. Afterwards, we propose a concept for model checking of distributed IPv6 firewalls and illustrate our approach by an example. The practical applicability is shown by performance measurements of our prototype *ad6*. The paper concludes with a summary and an outlook on future work.

2 RELATED WORK

2.1 Formal Verification with Model Checking

Model checking is a family of techniques to verify properties of a model representation of a system automatically. Typically, the properties are expressed as temporal logical formulas. Famous examples are the languages Linear Time Logic (LTL, introduced by (Pnueli, 1977)), Computational Time Logic (CTL, introduced by (Emerson and Halpern, 1986)) or the μ-Calculus (see (Kozen, 1983)). Model checking algorithms often operate on data represenations known as Kripke Structures. These are essentially directed graphs where the nodes are annotated with so called atomic propositions. For a formal introduction see (Kripke, 1963). Figure 2 shows an example of a Kripke Structure which is interpreted as a decision diagram. Each node is divided into an upper and a lower part. The upper half consists of a set of propositions that serve as meta information and mark special features given to the node. For example the first node is marked as initial node. The lower half contains a boolean formula that can be evaluated to decide which outgoing edge should be taken. If the formula *udp* is evaluated to true then the corresponding edge to the accept node is traversed. Note that both, the usage of formulas as well as boolean annotation on edges, are extensions to classical Kripke Structures. Some techniques (i.e. LTL) utilize problem encodings in SAT (see (Biere et al., 2006)) to benefit from the tremendous efforts made to optimize SAT solvers. SAT is short for the satisfiability problem of boolean logics. It is also an archetypical problem in complexity theory where it is proofed by Cook's Theorem that it is NP-complete (see (Cook, 1971)).

2.2 Tools for Policy Anomaly Detection

A couple of tools support policy anomaly detection for IPv4 firewalls. The first systematic approach to automatize policy anomaly detection with formal verification techniques was performed by Al-Shaer and Hamed (i.e. see (Al-Shaer and Hamed, 2003), (Al-Shaer and Hamed, 2004)). Al-Shaer et al. provided a classification of anomalies which is used broadly.

Generally, there are two approaches that target different scenarios. The *top-down* strategy includes a modeling phase where rules are specified in an abstract language. Afterwards they can be analyzed for anomalies and compiled into concrete rulesets that can be deployed in real firewalls. On the other hand, the *bottom-up* approach relies on existing rulesets that are parsed and analyzed.

We did not investigate on data mining approaches (as suggested in i.e. (Golnabi et al., 2006)) since statistical methods always have to deal with false positives and false negatives.

Policy Advisor. The tool Policy Advisor (PA, see (Al-Shaer and Hamed, 2004)) is a hybrid tool and uses finite automata to check all rules that lay on a path through a firewall policy pairwise. It parses a (limited) firewall language and also supports rule editing. Most notably the PA has polynomial complexity for all supported anomalies. This is achieved by utilizing a significantly less expressive rule model than the other approaches (i.e. no CIDR support). If it had the same expressiveness, it would be exponential in space.

Further, related work was provided by Abedin et al. (see (Abedin et al., 2006)) who introduced an automatic resolution algorithm for several anomalies. Kotenko and Polubelova (see (Kotenko and Polubelova, 2011)) applied LTL model checking based on the classification of Al-Shaer et altera.

FIREMAN. FIREMAN (see (Yuan et al., 2006)) is a bottom-up approach that relies on incremental updates of sets which are represented by Binary Decision Diagrams (BDD) while walking along all paths through the policies and the network configuration. The absence of cycles is a prerequisite for PA and FIREMAN because they use lists and trees as models. Since a cycle check is exponential in space or time (see (Poole and Mackworth, 2010) p. 6), they have an exponential flaw. Furthermore, the building of BDDs has an exponential space complexity.

Jeffrey and Samak. Jeffrey and Samak (see (Jeffrey and Samak, 2009)) transformed all configurations into a *Kripke Structure* and applied *SAT model checking*. Shadowing and cross-path anomalies are not supported. In their paper they used a bottom-up scenario.

In contrast to PA and FIREMAN, Jeffrey and Samak support only Reachability and Cyclicity. For a deeper analysis of PA and FIREMAN see (Lorenz, 2014). Jeffrey and Samak's approach has a very profound theoretical foundation and their model allows a very natural integration of network scenarios. The usage of SAT permits profiting from the large efforts put into solver technologies and the presented runtime results were encouraging. While this makes it a very promising candidate for an implementation of policy anomaly detection for distributed IPv6 firewalls, it did not support the detection of shadowed rules and cross-path anomalies.

FirewallBuilder. The FirewallBuilder (see (NetCitadel, 2012)) is an open source tool with a graphical user interface. It allows both, a top-down approach by modeling rulesets and networks and a bottom-up procedure by parsing existing rulesets in various languages (among others: IPTables, PIX, ASA, PF). It has a very limited policy anomaly detection support that allows the detection of shadowed rules. Contrary to its IPv4 implementation, the tool does not support address ranges for IPv6 which makes a proper definition of rules infeasable.

3 MODELLING IPv6 FIREWALLS

This section presents the modelling of IPv6 firewalls which follows the previous work of (Jeffrey and Samak, 2009) for IPv4 firewalls. They support the detection of unreachable rules and cycles. Additionally, we consider shadowing as clear misconfiguration and cross-path as potential security threat to be important.

The general approach of Jeffrey and Samak is to model the firewalls and networks as Kripke Structures first and transform them into SAT formulas afterwards. These can be solved by a SAT solver and one can extract example packets from the results which exploit a certain anomaly. Also, the results contain the associated path through the model. If the formula is not solvable, there exists no such packet and thus it does not contain the regarded anomaly. The formal verification workflow is divided into the four phases *Definition*, *Building*, *Solving* and *Reporting*. The first phase includes the definition or adjustment of the firewall policies and the network configuration. Since our motivation is the support of the migration of networks from IPv4 to IPv6 our prototype *ad6* imple-

ments a bottom-up strategy. Nevertheless, the architecture is open for top-down modelling. The building phase consists of the tasks necessary to abstract the model from the configuration as Kripke Structure and to build the SAT formulas for the policy anomaly detection. These are solved in the following *Solving* phase. Finally, the results can be used to generate a report.

During the introduction of the formal aspects we will demonstrate their outcome with a small example. The firewall configuration generated by the script

```
ip6tables -P OUTPUT DROP
ip6tables -A OUTPUT -p udp -j ACCEPT
ip6tables -A OUTPUT -p tcp -j OUTPUT
```

inherits a loop. Whenever a TCP packet is handled the first rule does not match and forwards the packet to the second rule. Since the filter is correct the packet is given to the OUTPUT chain where it is handled by the first rule again. The related decision diagram in Figure 2 shows this cycle.

3.1 Modelling a Single Firewall

A single firewall is given by the tuple

$$\mathcal{C} = (\mathcal{P}, \mathcal{S}, \gamma, \delta)$$

with

- a packet model $\mathcal{P}$,
- a state model $\mathcal{S}$,
- a decision function $\gamma : \mathcal{S} \rightarrow \mathcal{B}(\mathcal{P} \times \mathcal{S})$
- and a transition relation $\delta \subseteq \mathcal{S} \times \text{Bool} \times \mathcal{S}$.

The decision function assigns a boolean formula which represents the matching of a rule body to a state. The transition relation gives all possible transitions (s, b, t) between states. Together with the set of states they form a Kripke Structure with propositions from the packet model (see (Baier and Katoen, 2008) p.20f for a definition). In this case, the edges are labeled with boolean values which allows a deterministic transition according to some kind of decision function.

The packet-field used by the example configuration is the higher *protocol field* (the *next header field* in IPv6) which is 8Bit in size, the packet model can be defined as

$$\mathcal{P} = \{\text{proto}_i = v | i \in \{0, ..., 7\}, v \in \{0, 1\}\}$$

where each bit may have the value 0 or 1. The state model consists of five states and uses three flags to indicate the initial, the accept and the drop state

$$\mathcal{S} = \{\text{output}_0, \text{output}_1, \text{output}_2, \text{accept}_0, \text{drop}_0, \\ \text{init, accept, drop}\}$$

For the definition of the decision function it is necessary to encode the rules into a SAT representation. First, the values UDP and TCP can be encoded by using the binary representation of their protocol values. For UDP this is $17_{10} = 00010001_2$ and for TCP $6_{10} = 00000110_2$. The propositions of the packet model are connected as an AND formula for this purpose:

$$\begin{aligned}\gamma(\text{output}_0) = &\text{proto}_0 = 0 \wedge \text{proto}_1{=}0 \wedge \text{proto}_2{=}0 \wedge \\ &\text{proto}_3{=}1 \wedge \text{proto}_4{=}0 \wedge \text{proto}_5{=}0 \wedge \\ &\text{proto}_6{=}0 \wedge \text{proto}_7{=}1\end{aligned}$$

$$\begin{aligned}\gamma(\text{output}_1) = &\text{proto}_0 = 0 \wedge \text{proto}_1{=}0 \wedge \text{proto}_2{=}0 \wedge \\ &\text{proto}_3{=}0 \wedge \text{proto}_4{=}0 \wedge \text{proto}_5{=}1 \wedge \\ &\text{proto}_6{=}1 \wedge \text{proto}_7{=}0\end{aligned}$$

All other rules are trivially matching and can be represented as boolean constants:

$$\gamma(\text{output}_2) = \gamma(\text{accept}_0) = \gamma(\text{drop}_0) = \text{true}$$

Finally, the transition relation can be defined as:

$$\begin{aligned}\delta = \{ &(\text{output}_0, \text{true}, \text{accept}_0), \\ &(\text{output}_0, \text{false}, \text{output}_1), \\ &(\text{output}_1, \text{true}, \text{output}_0), \\ &(\text{output}_1, \text{false}, \text{output}_2), \\ &(\text{output}_2, \text{true}, \text{drop}_0) \\ \}&\end{aligned}$$

The model itself is easily extensible by increasing the packet or state models. Regarding the challenges for supporting the bigger IPv6 base-header and arbitrary header-chains the formalism is expressive enough. Extending to support the packet model by one bit introduces two propositions. Therefore, the packet model grows linearly with the number of bits. For extension header chains the same principle occurs. For each extension header checked by the firewall the bits that describe the header need to be introduced to the packet model. Therefore, the formalism already covers both challenges. The practical implications, especially if the larger number of propositions exploits the exponential flaws of SAT, need to be evaluated experimentally.

3.2 Modelling Networks

We did not modify the modelling of networks as proposed by Jeffrey and Samak. The basic idea is to model each network node as Kripke Structure and add states representing incoming and outgoing network interfaces. These states are connected to the node's input, output and forward tables as well as to other nodes' network interfaces. For a formal introduction please refer to (Jeffrey and Samak, 2009) p. 63. Note

that it is possible to introduce nondeterministic behaviour by connecting an outgoing network state with multiple inbound states. For example, this allows to model wifi networks.

3.3 Kripke Structure

The encoding of the model as Kripke Structure is done analogously to Jeffrey and Samak's by constructing the formula

$$\begin{aligned}\text{trans}(\mathcal{C}) = &\forall(t,c,u) \in \delta.(\\ &\quad y_{(t,c,u)} \rightarrow (\\ &\quad\quad (c \leftrightarrow \text{trans}(t,\gamma(t)))\wedge\\ &\quad\quad (\text{trans}(t,\mathit{init}) \vee \exists(s,b,t) \in \delta.y_{(s,b,t)})\\ &\quad)),\end{aligned}$$

where $y_{(t,c,u)}$ is a variable that shows whether the transition (t,c,u) is true in a run. The implicant of the formula requires that if a transition is true then it must be motivated. At first, the rule body $\gamma(t)$ must be evaluated according to the guard value c. This is done by applying the formula

$$\text{trans}(s,\mathit{true}) = \mathit{true}\text{ , trans}(s,\mathit{false}) = \mathit{false}$$
$$\text{trans}(s,B \wedge C) = \text{trans}(s,B) \wedge \text{trans}(s,C)$$
$$\text{trans}(s,B \vee C) = \text{trans}(s,B) \vee \text{trans}(s,C)$$
$$\text{trans}(s,\neg B) = \neg\text{trans}(s,B)$$
$$\text{trans}(s,\phi) = \begin{cases} \phi & \text{when } \phi \in \mathcal{P} \\ \sigma(s,\phi) & \text{when } \phi \in \mathcal{S} \end{cases}$$

(where $\sigma(s,\phi)$ is true if ϕ is a proposition of the state s) which evaluates a boolean formula over atomic propositions. Afterwards, the state t needs to be reached. This is the case if either it is an initial state which is checked by $\text{trans}(t,\mathit{init})$ or it has a predecessor with an outgoing transition (s,b,t) leading to t. A solution for $\text{trans}(\mathcal{C})$ represents a path through the model starting at an initial state.

Regarding our example the formula looks like this:

$$\begin{aligned}\text{trans}(\mathcal{C}) = &(y_{(\text{output}_0,\text{true},\text{accept}_0)} \rightarrow (\\ &\quad (\text{true} \leftrightarrow \gamma(\text{output}_0)) \wedge \text{trans}(\text{output}_0,\text{init})\\ &) \wedge (y_{(\text{output}_0,\text{false},\text{output}_1)} \rightarrow (\\ &\quad (\text{false} \leftrightarrow \gamma(\text{output}_0)) \wedge \text{trans}(\text{output}_0,\text{init})\\ &) \wedge (y_{(\text{output}_1,\text{true},\text{accept}_0)} \rightarrow (\\ &\quad (\text{true} \leftrightarrow \gamma(\text{output}_1)) \wedge y_{(\text{output}_0,\text{false},\text{output}_1)}\\ &) \wedge (y_{(\text{output}_1,\text{false},\text{output}_2)} \rightarrow (\\ &\quad (\text{false} \leftrightarrow \gamma(\text{output}_1)) \wedge y_{(\text{output}_0,\text{false},\text{output}_1)}\\ &) \wedge (y_{(\text{output}_2,\text{true},\text{drop}_0)} \rightarrow (\\ &\quad (\text{true} \leftrightarrow \gamma(\text{output}_2)) \wedge y_{(\text{output}_1,\text{false},\text{output}_2)}\\ &)\end{aligned}$$

We are now able to further constrain the set of paths through the model.

3.4 Anomalies

The anomalies discussed in this paper were introduced by different authors. Unreachability and cyclicity were supported by (Jeffrey and Samak, 2009) and shadowing comes from (Al-Shaer and Hamed, 2004). The latter anomaly was also supported by (Yuan et al., 2006) who introduced cross-path as well. In this paper, we adapt shadowing and cross-path to extend the algorithm of Jeffrey and Samak.

All anomaly encodings follow the same scheme. They build upon the Kripke Structure encoding and add the global constraints as well as an anomaly-specific constraint. All formulas have the form

$$\begin{aligned}\text{anomaly}(\mathcal{C}[,t]) = &\text{trans}(\mathcal{C})\wedge\\ &\text{anomaly_constraint}(\mathcal{C}[,t])\wedge\\ &\text{global_constraints}(\mathcal{C})\end{aligned}$$

where anomaly_constraint $\in$ {reach_constraint, cycle_constraint, shadow_constraint, cross_constraint} and the parameter t applies to reachability and shadowing.

Reachability. To detect an unreachable state $t \in \mathcal{S}$, it is sufficient to add a simple constraint

$$\text{reach_constraint}(\mathcal{C},t) = \exists(s,b,t) \in \delta.y_{(s,b,t)} \vee \text{trans}(t,\mathit{init})$$

that enforces the existence of an incoming transition. If it is an initial state, it is reachable by default. Note that it is necessary to run the detection for each state of interest. Thus, if all states should be checked the amount of runs equals the amount of states.

Cyclicity. The detection of cycles needs a constraint to force the path to be loop-shaped:

$$\begin{aligned}\text{cycle_constraint}(\mathcal{C}) = (&\\ &\forall(s,b,t) \in \delta.(y_{(s,b,t)} \rightarrow \exists(t,c,u) \in \delta.y_{(t,c,u)})\\ &) \wedge (\\ &\exists(s,b,t) \in \delta.(y_{(s,b,t)} \wedge \text{trans}(s,\mathit{init}))\\ &)\end{aligned}$$

The first part of the constraint ensures that every transition on the path has a successor which means that they form a loop. Since the empty path is a trivial solution for first part, the second part ensures that one of the initial transitions must be true and therefore enforces a meaningful solution. The original formalism (see (Jeffrey and Samak, 2009) p. 62) allowed the empty path as trivial solution for cycles.

Shadowing. The shadowing anomaly is caused by prerequisite rules that filter all packets which are relevant for a subsequent rule. The central constraint is

$$\text{shadow_constraint}(\mathcal{C},t) = \text{reach}(\mathcal{C},t) \wedge \text{trans}(t,\gamma(t)).$$

Since an unreachable state $t \in \mathcal{S}$ is automatically shadowed, this behaviour is enforced by reusing the reachability detection as first part of the constraint. The second part checks upon arrival of the state, whether there are still packets left that can be matched by the rule's body. For shadowed states the same necessity for multiple runs occurs as for reachability.

Cross-Path. If the Kripke Structure is nondeterministic, which for example can be caused by nonunique routing in WiFi networks, then a cross-path anomaly can occur. Depending on the nondeterministic routing decision the packet can be handled differently. On one path it is accepted while on another it is dropped. The constraint is the following:

$$\begin{aligned}&\text{cross_constraint}(\mathcal{C}) = \\ &\quad (\exists(s,b,t) \in \delta.(y_{(s,b,t)} \wedge \text{trans}(t, accept))) \wedge \\ &\quad (\exists(s,b,t) \in \delta.(y_{(s,b,t)} \wedge \text{trans}(t, drop)))\end{aligned}$$

The first part enforces a route to an accepting state while the second requires a drop. Introduced to the main formula the conjunction ensures that the constraint is met by one single packet.

As our example contains a cycle anomaly we want to extend the formula with the respective constraint allowing paths to be loop-shaped only:

$$\begin{aligned}&\text{cycle_constraint}\mathcal{C}) = (y_{(\text{output}_0,\text{true},\text{accept}_0)} \rightarrow \text{false}) \wedge \\ &\quad (y_{(\text{output}_0,\text{false},\text{output}_1)} \rightarrow (\\ &\quad\quad y_{(\text{output}_1,\text{true},\text{output}_0)} \vee y_{(\text{output}_1,\text{false},\text{output}_2)})) \wedge \\ &\quad (y_{(\text{output}_1,\text{true},\text{output}_0)} \rightarrow (\\ &\quad\quad y_{(\text{output}_0,\text{true},\text{accept}_0)} \vee y_{(\text{output}_0,\text{false},\text{output}_1)})) \wedge \\ &\quad (y_{(\text{output}_1,\text{false},\text{output}_2)} \rightarrow y_{(\text{output}_2,\text{true},\text{drop}_0)}) \wedge \\ &\quad (y_{(\text{output}_2,\text{true},\text{drop}_0)} \rightarrow \text{false}) \wedge \\ &\quad (y_{(\text{output}_0,\text{true},\text{accept}_0)} \vee y_{(\text{output}_0,\text{false},\text{output}_1)})\end{aligned}$$

The first five implications require every transition to have a successor. If a path satisfies this constraint, then it forms a loop. In this example the transitions from output_0 to output_1 (implication 2) and from output_1 back to output_0 (implication 3) can be set true while also satisfying their conclusion. The rest of the implicants can be set false to render their clauses true. The last line removes the empty path from the solution space. It is true in this scenario.

The last step for our example is to enforce mutual exclusion for the bits of the packet model. At this stage it would be possible to i.e. have the propositions $\text{proto}_3 = 0$ and $\text{proto}_3 = 1$ active at the same time which is not possible in a real packet. In a network scenario also the occurence of multiple initial states would need our attention.

3.5 Global Constraints

Bit propositions may occur in three different forms, namely 0, 1 or don't care. To ensure mutual exclusion for every bit proposition there must exist a global constraint. This is given by the formula:

$$\text{global_bits}(\mathcal{C}) = \forall(f = v) \in \mathcal{P}.\neg((f = 0) \wedge (f = 1))$$

In our example this constraint leads to the formula:

$$\begin{aligned}\text{global_bits}(\mathcal{C}) = &\neg(\text{proto}_0 = 0 \wedge \text{proto}_0 = 1) \wedge \\ &\neg(\text{proto}_1 = 0 \wedge \text{proto}_1 = 1) \wedge \\ &\neg(\text{proto}_2 = 0 \wedge \text{proto}_2 = 1) \wedge \\ &\neg(\text{proto}_3 = 0 \wedge \text{proto}_4 = 1) \wedge \\ &\neg(\text{proto}_4 = 0 \wedge \text{proto}_4 = 1) \wedge \\ &\neg(\text{proto}_5 = 0 \wedge \text{proto}_5 = 1) \wedge \\ &\neg(\text{proto}_6 = 0 \wedge \text{proto}_6 = 1) \wedge \\ &\neg(\text{proto}_7 = 0 \wedge \text{proto}_7 = 1)\end{aligned}$$

Additionally, networks may have undesired behaviour if the amount of active init states is not restricted to one. Otherwise multiple paths through the model may be explored simultaneously. The constraint

$$\text{global_inits}(\mathcal{C}) = \bigoplus_{s \in I}(s),$$

where $I \subseteq \mathcal{S}$ is the set of initial states, ensures the limitation of active init states to one ($\bigoplus$ is the cumulative exclusive-or operator).

By chaining these constraints with AND, they form a formula

$$\text{cycle} = \text{trans}(\mathcal{C}) \wedge \text{cycle_constraint}(\mathcal{C}) \wedge \text{global_bits}(\mathcal{C})$$

that may be transformed into a representation suitable for a SAT-solver. In many cases this is the conjunctive normal form (CNF). Table 1 shows the solution assignment for our example. The report includes the loop-shaped path

$$\text{output}_0 \rightarrow \text{output}_1 \rightarrow \text{output}_0$$

through the model and the packet exploiting this path including the bits extracted from the variable assignment. It could also be used for an automatically generated network test. Note that there might be more than one solution for cycles. Some solvers support the enumeration of results. This is not feasable for the cycle detection since the solution space is too large.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

For example filter decisions for IPv6 packets are often based on address prefixes. This leaves the bits of the postfix as don't care and thus, makes an enumeration exponential in space. All these solutions apply to the same path and therefore do not improve the knowledge about the system. A mechanism to further constrain the formula to forbid already known loops is out of scope of this paper.

Table 1: Variable assignment that solves the formula for the cycle detection example.

Variable	Value	Variable	Value
$y_{(\text{output}_0,\text{true},\text{accept}_0)}$	false	$\text{proto}_0 = 0$	true
$y_{(\text{output}_0,\text{false},\text{output}_1)}$	true	$\text{proto}_0 = 1$	false
$y_{(\text{output}_1,\text{true},\text{output}_0)}$	true	$\text{proto}_1 = 0$	true
$y_{(\text{output}_1,\text{false},\text{output}_2)}$	false	$\text{proto}_1 = 1$	false
$y_{(\text{output}_2,\text{true},\text{drop}_0)}$	false	$\text{proto}_2 = 0$	true
		$\text{proto}_2 = 1$	false
$\text{proto}_3 = 0$	true	$\text{proto}_4 = 0$	true
$\text{proto}_3 = 1$	false	$\text{proto}_4 = 1$	false
$\text{proto}_5 = 0$	false	$\text{proto}_6 = 0$	false
$\text{proto}_5 = 1$	true	$\text{proto}_6 = 1$	true
$\text{proto}_7 = 0$	true	$\text{proto}_7 = 1$	false

4 IMPLEMENTATION

Our prototype *ad6* owns a modular architecture as shown in Figure 3. The *Aggregator* uses a parser for the firewall rulesets which are connected in the next step to the network configuration. Currently, *ad6* supports the rule set of the *ip6tables* tool (see (Welte and Ayuso, 2014)). Note that the modular approach allows to replace the Aggregator with another module that allows a top-down modeling. The *Instantiator* consists of a set of libraries to build Kripke Structures from configurations and a procedure to transform them to basic SAT formulas consisting of trans($\mathcal{C}$) and the global constraints. Also the transformation to CNF is performed here. Afterwards, they are enriched with the anomaly specific part and solved by a standard SAT-solver interfaced with an adapter. Both utilized solvers, MiniSAT (see (Een and Sorenson,) and Clasp (see (Kaufmann et al., 2012), lack a native interface for Python which was the main implementation language. Thus, a workaround for an adapter including the fork of a subprocess as well as reading from and writing to files was developed.

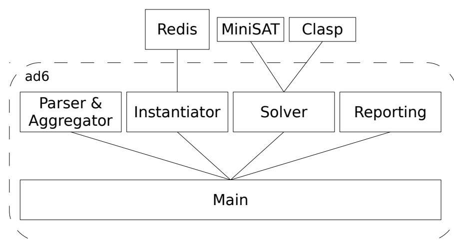

Figure 3: Architecture of ad6.

Intermediate results of the building process (i.e. transition encodings of the Kripke Structure) are stored in a database. So, if a rule is changed, (large) parts of the Kripke Structure can be reused which should have a positive effect on the performance. A scenario where this might be useful applies if an administrator changes the input configuration (i.e. by removing a rule) where most formula parts remain untouched. We utilized the Redis NoSQL-database (see (Pivotal Software, 2014)) which is an in-memory key-value store with a native Python interface.

5 EVALUATION

This section presents the evaluation of *ad6*. It was evaluated for different workloads to investigate the scalability of the presented approach. We measured the runtime of the different phases, i.e. building and solving phase, separately. The building phase which is performed by the Instantiator includes all activities necessary to provide the formulas that should be checked in the solving phase. Further, we used two different solvers, MiniSAT and Clasp.

5.1 Workload and Testbed Description

Table 2 shows the parameters of the tested workloads. The small workload represents a firewall of a single multi-service host typical for a small business server (45 rules). The medium size was inspired by our university campus and represents a gateway firewall of a large organization separating several subnets, hosts, and services (721 rules). The large configuration includes the medium and several instances of the small workload organized in subnets like a DMZ and networks of organizational subunits (2044 rules). The network graph of this scenario is shown in Figure 4.

The corresponding rule sets aimed to be free of anomalies, but included rules concerning extension headers which have a share of about 11% for the small, 0.7% for the medium and 8% for the large workload. Our firewall policies follow the exemplary ruleset of the IDSv6 project (see (IDSv6-Project, 2013)).

All measurements were performed on a single machine. Table 3 shows its technical details. Since *ad6* is implemented single threaded, there are enough cores

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

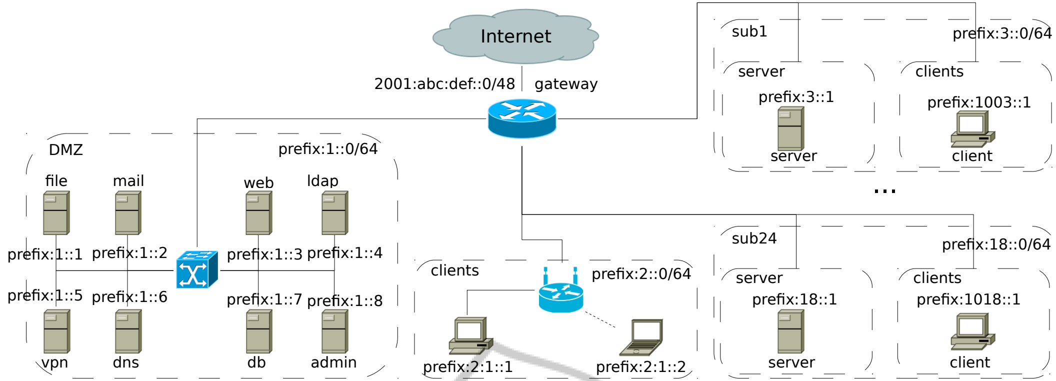

Figure 4: Network topology that forms the basis of the workload. The network prefix is 2001:abc:def::0/48

Table 2: Workload parameters.

workload	Rules	Kripke Nodes	Transitions
small	45	51	84
medium	721	732	1435
large	2044	2328	4097

Table 3: Technical details of the test machine.

CPU	Intel i7-3630QM
# of Cores	4
Frequency	2.4GHz
RAM	8GB
OS	Arch Linux
Kernel	v3.16.2
Python	v3.4.1
Redis	v2.8.17
MiniSAT	v2.2.0
Clasp	v3.0.3

left to serve the database and the operating system without interference.

5.2 Experimental Results

Figure 5 shows the total runtime results for the three workloads. For each workload, the runtimes for the different solvers MiniSAT and Clasp are given. Further, the runtime of the building phase is given titled *FirstUse*.

The intermediate results were finite and steady with a very low standard deviation. This shows that the risk of an exponential runtime due to the IPv6 challenges did not occur. Comparing the two solvers, the runtimes are very similar.

While the total runtime of *ad6* (FirstUse plus solving time) is about 3 seconds for the small workload, it is 239 seconds for the medium, and about 37 min for the large rule set. If fit to a quadratic curve $f(x) = ax^2 + bx + c$, the coefficients are

phase	a	b	c
building (FirstUse)	0.0002	-0.06	3.54
solving (Clasp)	0.0004	-0.04	2.85

where x is the number of rules in the input. For a workload of 3000 rules the estimated runtime is about 85 min with 27 min for the building and 58 min for the solving phases. Nevertheless, the performance behaves clearly superlinear but subquadratic and has further potential for improvement since the current implementation is still a prototype.

The major delay is caused by the anomalies reachability and shadowing. Contrary to cycles and cross-path anomalies which need a single run to be detected globally they need to be checked nodewise. This includes the appending of the rule specific constraint to the base formula (consisting of trans(C) and the global constraints) and a solver run. The total number of runs can be quantified by the formula $f(x) = 2x + 2$ where x is the number of Kripke Nodes. The other factor influencing the total runtime is the size of the base formula that causes every run to be longer. The formula grows with every transition introduced to the Kripke Structure. As a rule of thump one can say that most new rules introduce two transitions. One is taken when the rule matches and the other if not. Taking both factors into consideration the complexity of the algorithm is quadratic. To increase the applicability further improvements or even another approach for these time intensive anomalies are required.

Reuse of Formula: The building phase includes all activities necessary to provide the formulas that should be checked in the solving phase. A potential improvement lays in the reuse of formula parts from previous runs. This includes all formulas for transitions and rule matching in conjunctive normal form. In our tests we wanted to quantify the maximal possible gain of reusage. We enforced this by not changing any rule after the first run so all formula parts could be fetched from the database. Figure 5 shows that the

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

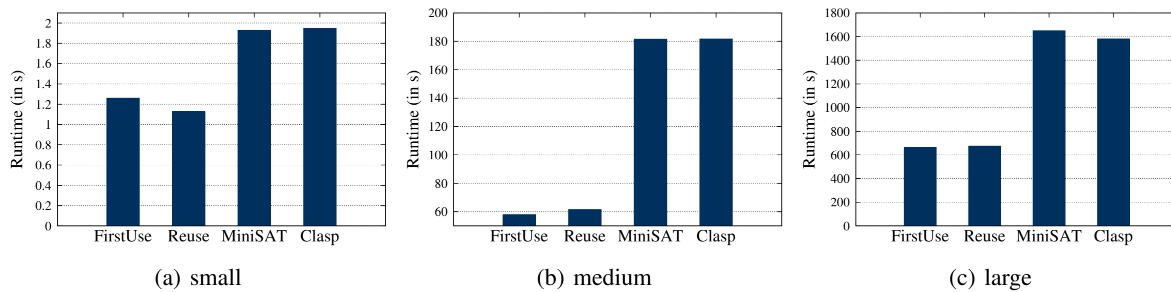

Figure 5: Total runtimes of the different phases for each workload (in s).

differences between the first use and the reuse during the building and solving phase are negligible. In that regard, the most surprising effect is the slightly worse performance of the reuse of formula parts compared to a complete rebuild. The database overhead seems to superseed the benefits. Concerning the migration from IPv4 to IPv6 this circumstance is not relevant since the policy anomaly detection is applied once after the migration.

RAM Consumption: Modern SAT solvers can handle millions of different boolean variables and are typically just limited by memory. Normally, the policy anomaly detection should not exceed this limitation since packet models are rather small depending on the size of the header chain. The other factor is the size of the Kripke Structure. Both did not exceed 10000 propositions in our workload. To quantify the memory behaviour we periodically measured the RAM consumption. It was very stable and varied between 55MB and 322MB. Therefore, the memory is not regarded as bottleneck for the overall scalability.

5.3 Discussion

The average solving time lays in the same order of magnitude like the results of Jeffrey and Samak (see (Jeffrey and Samak, 2009) p.65). Their considerably low initialization runtime includes only the creating of the model and the base instance without its conversion to CNF. While Jeffrey and Samak test only reachability and cyclicity, the presented approach is also able to test shadowing and cross-path. The difference would most likely tighten, if native interfaces for the solvers would be available for *ad6*.

The building and checking of a large configuration had a total duration of about 37min. Whether this is acceptable, depends on the use case scenario. For the migration of IPv4 networks to IPv6, the long runtime does not matter since the policy anomaly detection is only performed once. Also, classical network setups tend to be relatively static and therefore, have very few rule changes if no automatic rule generation mechanisms (i.e. for some UDP services) are present. On the other hand, very dynamic environments like Software Defined Networks (SDN) would require periodic checking.

6 CONCLUSION AND FUTURE WORK

The main contribution of this paper is the proof that model checking techniques can also be applied successfully for IPv6 firewalls to detect anomalies. The presented approach is based on the algorithm of Jeffrey and Samak to test reachability and cyclicity. Additionally, we have extended the concept to also detect shadowing and cross-path. The presented prototype, the policy anomaly detector *ad6*, is a useful tool to increase the semantical assurance of IPv6 firewalls.

The impact of larger addresses and therefore more boolean variables in the encoding were evaluated experimentally and did not show the potential exponential behaviour. Also, it was shown that IPv6 extension header chains do not break the applicability of the formalism but may cause a massive but still linear growth of propositions in the formula. This would also apply to any other extensional protocol support and, in a limited way, already occured in the original algorithm with IPv4 options.

The overall performance of the prototype *ad6* is sufficient for the network migration use case and allows a regular inspection of its sanity. For fast configuration changes, as they may appear in SDN, the checking cannot be performed rapidly or even in realtime. But there is much potential for improvements in the prototype implementation.

Apart from the enhancement of the code quality and the implementation of native interfaces for the solvers, there are further improvements intended:

- Learning from intermediate results - The time intensive task of checking for unreachability and shadowing can be improved by learning from intermediate results. So, all nodes that lay on a path to a reachable rule are reachable as well. Also,

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

unreachable rules are shadowed by default. The respective checks for these rules can be skipped. This is also true for initial rules which are always reachable and unshadowed.

- Parallelization - The overall performance can be improved by parallelizing parts of the application. Especially, the building and solving of unreachability and shadowing formulas is completely independent and can be processed in parallel. Also, parts of the building process may be parallelized as well.
- Expressiveness - There are further interesting features for the policy anomaly detection like the support for stateful firewalling or the consideration of effects introduced by VPN-tunnels.

REFERENCES

Abedin, M., Nessa, S., Khan, L., and Thuraisingham, B. M. (2006). Detection and Resolution of Anomalies in Firewall Policy Rules. In *Data and Applications Security, 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security 2006, Proceedings*, pages 15–29.

Al-Shaer, E. S. and Hamed, H. H. (2003). Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In *Integrated Network Management VII, Managing It All, IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003)*, pages 17–30.

Al-Shaer, E. S. and Hamed, H. H. (2004). Discovery of Policy Anomalies in Distributed Firewalls. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2605–2616.

Arkko, J. and Baker, F. (2011). Guidelines for Using IPv6 Transition Mechanisms during IPv6 Deployment. RFC 6180.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. The MIT Press.

Biere, A., Heljanko, K., Junttila, T. A., Latvala, T., and Schuppan, V. (2006). Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science*, 2(5).

Biondi, P. and Ebalard, A. (2006). Scapy and IPv6 networking. Slides from http://www.secdev.org/conf/scapy-IPv6_HITB06.pdf.

Caicedo, C. E., Joshi, J. B., and Tuladhar, S. R. (2009). IPv6 Security Challenges. *Computer*, 42(2):36–42.

Cook, S. A. (1971). The Complexity of Theorem-Proving Procedures. Technical report, University of Toronto.

Een, N. and Sorensson, N. A minimalistic and high-performance SAT solver. https://github.com/niklasso/minisat.

Emerson, E. A. and Halpern, J. Y. (1986). "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *Journal of the Association for Computing Machinery (JACM)*, 33(1):151–178.

Golnabi, K., Min, R., Khan, L., and Al-Shaer, E. (2006). Analysis of Firewall Policy Rules Using Data Mining Techniques. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 305–315.

Google (2015). Google IPv6 - Statistics. https://www.google.com/intl/en/ipv6/statistics.html.

IDSv6-Project (2013). Exemplary ip6tables init script. http://www.idsv6.de/Downloads/iptables_ruleset.sh.

Jeffrey, A. and Samak, T. (2009). Model Checking Firewall Policy Configurations. In *POLICY*, pages 60–67. IEEE Computer Society.

Kaufmann, B., Schaub, T., and et. al. (2012). A conflict-driven nogood learning answer set solver. http://www.cs.uni-potsdam.de/clasp/.

Kotenko, I. and Polubelova, O. (2011). Verification of security policy filtering rules by Model Checking. In *IEEE 6th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2011, Prague, Czech Republic, September 15-17, 2011, Volume 2*, pages 706–710. IEEE.

Kozen, D. (1983). Results on the Propositional mu-Calculus. *Theor. Comput. Sci.*, 27:333–354.

Kripke, S. (1963). Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94.

Lorenz, C. (2014). Paper Discussion: Policy Advisor and FIREMAN. Technical report, University of Potsdam. http://www.cs.uni-potsdam.de/bs/research/docs/techreports/2014/l14.pdf.

NetCitadel (2012). FirewallBuilder. www.fwbuilder.org.

Pivotal Software (2014). Redis Documentation. http://redis.io/documentation.

Pnueli, A. (1977). The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society.

Poole, D. and Mackworth, A. (2010). Lecture 3.2 on Artificial Intelligence. Slides from http://artint.info/slides/ch03/lect2.pdf.

Welte, H. and Ayuso, P. N. (2014). The netfilter.org "iptables" project. http://www.netfilter.org/projects/iptables/.

Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C.-N., and Mohapatra, P. (2006). FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy*, pages 199–213. IEEE Computer Society.