Presentation Outline

- **Introduction and Motivation**
  - What is a **Firewall**
  - Their **configuration** are **difficult** to manage

- **Transcompilation Pipeline**
  - A language-based **Solution**
  - FireWall Synthesizer (**FWS**)

- **Function-Based Redefinition** (Master Thesis)
  - from Firewalls to Functions and Back
  - Composition
  - Function Representation

- **Ongoing and Future Work**
  - Tag System
  - Networks of Firewalls
What is a Firewall?

**Inspects the traffic:** for each packet
- accepts or drops it
- possibly modifying it (NAT)

Based on a **configuration**
- List of rules
- Possibly using **tags**
- **Control-flow** constructs
- **Complex Interaction** among rules (Shadowing)
- **Different** configuration languages
- **Low level** details

Difficult and **error** prone:
- Configuration
- Cross-system porting
- Test
- Verification
Our Goal

Old Legacy Technology

iptables
ipfw
pf
Our Goal

High level representation

decomposition

Old Legacy Technology

iptables
ipfw
pf
Our Goal

High level representation

update
test
generate
verify
inspect

Old Legacy Technology

iptables
ipfw
pf
Our Goal

High level representation

compilation

Old Legacy Technology

iptables
ipfw
pf

High Level Management of Firewall Configurations
Transcompilation Pipeline between firewall languages

- Supports iptables, pf, ipfw and (partially) CISCO-ios
- General approach
- Supports NAT
- Formal semantics
- tool: FireWall Synthesizer
Each firewall system

- Has its own configuration language
- Makes different evaluation steps to process packets
- Lots of low level details
  - First do the NAT, than filtering or vice-versa?
  - How to express complex conditions (disjunction and negation)?
Each firewall system

- Has its own configuration language
- Makes different evaluation steps to process packets
- Lots of low level details
  - First do the NAT, than filtering or vice-versa?
  - How to express complex conditions (disjunction and negation)?

General Model

Firewall = set of rules + the evaluating procedure
**Firewall = set of rules + the evaluating procedure**

**Configuration**

Assigns a rulesets to each node

**Ruleset**: list of rules \( r = (\phi, a) \)

- \( \phi(p) \): condition
- \( a \): action
  - ACCEPT
  - DROP
  - NAT\((d_n, s_n)\)
  - MARK\((m)\)
  - \textcolor{red}{\textsc{Goto}(R)}
  - \textcolor{red}{\textsc{Call}(R)}
  - RETURN

**Control Diagram**

\( S \) are the addresses of the firewall
Transcompilation Pipeline

High Level Management of Firewall Configurations
Transcompilation Pipeline

tool: FWS
FireWall Synthesizer
From Firewalls to Functions and Back: The Idea

Previous implementation of the pipeline synthesis:

- **Associate two predicates with a configuration:** its meaning on pairs $p, p'$ when $p$ is accepted as $p'$ or on discarded $p$
- **Compute the models of a predicate** (SAT-solver)
  Black-box approach (no fine tuning)

Change of domain:

**Function-based redefinition of the pipeline**

(Firewalls $\rightarrow$ Functions) :
  
  source configuration $\mapsto$ function representing its meaning

(Firewalls $\leftarrow$ Functions) :
  
  functional representation $\mapsto$ target configuration
\[ \tau : \mathcal{P} \to \mathcal{T}(\mathcal{P}) \cup \{\perp\} \quad \text{where} \]

- \( \mathcal{P} \): network packets
- \( \mathcal{T}(\mathcal{P}) \): transformations possibly applied to packets
- \( \perp \): discard of a packet

New pipeline stages:

- **ruleset synthesis**: rulesets became functions
- **composition**: computes the semantics of the firewall
- **generation**: assign functions to the target nodes

Why:

- **Parametric** w.r.t. IFCL specification
- Support **minimal control diagrams** and MARK
- Translation from IFCL to target language is trivial
Function Representation

Functions $\tau : \mathcal{P} \rightarrow \mathcal{T}(\mathcal{P}) \cup \{\bot\}$ as sets of pairs $(P, t)$

- $t$ is a transformation
- $P$ is a multi-cube of packets

**Cube**:
Cartesian product of one segment for each dimension

**Multi-cube**:
Cartesian product of one union of segments for each dimension

- **succinct** representation
- sets of packets verifying **rule conditions**
- sets of packets verifying **arc conditions**
- closed under **transformations**
We **scan** the ruleset rule-by-rule, **keeping track** of

- $P$ packets still to process
- $t$ transformation assigned to $P$

$$P = \begin{cases} 
  P_s & \text{packets that verify the rule condition} \\
  P_n & \text{packets that do not – managed by the other rules} 
\end{cases}$$

**if** the action accept/rejects the packet **then** $(P_s, t')$, where $t'$ updates $t$

**else** processing continues with the other rules on $P_s$ (updating $t$ to $t'$)
Composition

Ideally, for each $p \in \mathbb{P}$

- compute $t$ in the first node
- compute $p'$:
  (how $p$ is when exits node $q$)
- check $\psi(p')$ ... if it does then
  - compute $t'$ in the second node
  - Overall: $p \mapsto t$ updated by $t'$

Composition Algorithm:

The same, but with Multi-cubes ...

(... with additional details)
Example from ipfw to pf: formalization

```
ipfw -q nat 1 config ip 151.15.185.183
ipfw -q nat 2 config redirect_port tcp 9.9.8.8:17 17
ipfw -q add 0010 nat 1 tcp from 192.168.0.0/24 to not 192.168.0.0/24
ipfw -q add 0020 nat 2 tcp from 151.15.185.183 to not 192.168.0.0/24 17
ipfw -q add 0030 allow tcp from 151.15.185.183 to not 192.168.0.0/24 out
ipfw -q add 0040 deny all from any to any
```
Example from ipfw to pf: ruleset synthesis

\[ R_0 : (sIP \in 192.168.0.0/24 \land dIP \notin 192.168.0.0/24, \text{NAT}(\ast : \ast, 151.15.15.183 : \ast)); \\
(sIP = 151.15.15.183 \land dIP \notin 192.168.0.0/24 \land dPort = 17, \text{NAT}(9.9.8.8 : \ast, \ast : \ast)); \\
(true, \text{DROP}) \]

<table>
<thead>
<tr>
<th>Received packets</th>
<th>Accepted packets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>source</strong></td>
<td><strong>destination</strong></td>
</tr>
<tr>
<td>192.168.0.0/24</td>
<td>*</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>*</td>
</tr>
</tbody>
</table>
Example from ipfw to pf: composition

### $\tau_0$

<table>
<thead>
<tr>
<th>Received packets</th>
<th>Accepted packets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>source</strong></td>
<td><strong>destination</strong></td>
</tr>
<tr>
<td>192.168.0.0/24</td>
<td>*</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>*</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
</tr>
</tbody>
</table>

### $\tau_1$

<table>
<thead>
<tr>
<th>Received packets</th>
<th>Accepted packets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>source</strong></td>
<td><strong>destination</strong></td>
</tr>
<tr>
<td>192.168.0.0/24</td>
<td>*</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>*</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>*</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
</tr>
</tbody>
</table>

### Received packets

<table>
<thead>
<tr>
<th><strong>source</strong></th>
<th><strong>destination</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>151.15.185.183</td>
<td>*</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>* { 151.15.185.183 192.168.0.0/24 }</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>* { 127.0.0.1 192.168.0.0/24 }</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>* { 127.0.0.1 151.15.185.183 192.168.0.0/24 }</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>* { 127.0.0.1 151.15.185.183 192.168.0.0/24 }</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>* { 192.168.0.0/24 }</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>17</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>17</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>* { 127.0.0.1 151.15.185.183 192.168.0.0/24 }</td>
</tr>
<tr>
<td>151.15.185.183</td>
<td>17</td>
</tr>
</tbody>
</table>
Generation
How to generate functions

Problem: not every ruleset can be assigned to each node!

- Assign **Labels** to nodes
  - DROP
  - SNAT
  - DNAT
- Different expressive power

**Algorithm**
- **For each** pair \((P, t)\) with \(t \neq \perp\)
  - Find the **path**
  - **For each** node \(q\)
    - Preceding nodes \(\rightarrow P_q\)
    - Labels in \(q \rightarrow t_q\)
  - **Special management for** DROP pairs \((P, \perp)\)
Management of DROP pairs

Special management for DROP pairs \((P, \perp)\)

- For each node: packets **still not managed**
- **Drop as many as possible**

Legend:
- \(\perp\): remaining
- \(\square\): will be dropped
- \(\bigotimes\): dropped

\[ q \quad \{SNAT\} \quad q' \quad \{DROP\} \]
Recap

This transcompilation approach

- Is \textbf{parametric} w.r.t. the IFCL specification
- Supports the use of \textbf{tags} in IFCL
- Supports firewalls with \textbf{minimal control diagram}
- Preserves the \textbf{NAT}
- Reveals \textbf{different expressive power} of firewall languages
Ongoing and Future Work
Objectives

- Preserve the structure of the original configuration: Refactoring
- Reduce the gap between real languages and IFCL
- Fully support of tag system in real languages
- Handle networks with many firewalls
- Port configurations to Software Defined Networks
Problem with tags: \textit{pf}

**PF:**
- Rules \textbf{read top-down}
- \textbf{Last matching} rule is applied
- \textbf{Tag} is applied \textbf{immediately} (evaluation continues)
- \textbf{Quick rules} are applied immediately (evaluation stops)

**IFCL:**
- Rules read top-down and \textbf{applied immediately}
- Tags never stop the evaluation
Basic solution

Just rewrite **bottom-up** the same list of rules (prepending quick rules)

**Example:**

\[
\begin{align*}
&(true, \ \text{DROP}) \\
&(src = 1.2.3.4, \ \text{ACCEPT}) \\
&(dst = 5.6.7.8, \ \text{NAT}(1.6.3.8, *)) \\
&(src = 8.8.8.8, \ \text{DROP})
\end{align*}
\]
Basic solution

Just rewrite **bottom-up** the same list of rules (prepending quick rules)

**Example:**

\[
\text{(true, DROP)} \\
\text{(src = 1.2.3.4, ACCEPT)} \\
\text{(dst = 5.6.7.8, NAT(1.6.3.8,*))} \\
\text{(src = 8.8.8.8, DROP)}
\]

become

\[
\text{(dst = 5.6.7.8, NAT(1.6.3.8,*))} \\
\text{(src = 8.8.8.8, DROP)} \\
\text{(src = 1.2.3.4, ACCEPT)} \\
\text{(true, DROP)}
\]
Basic solution: tag

**Divide** each rule $r$ into

- **quick part**: $r'$ ($\not\in$ + tag)
- **slow part**: $r''$ (everything else)

**Example:**

$$R = \begin{cases} 
(r_1) \\
(r_2) \\
\ldots \\
(r_n) 
\end{cases}$$
Basic solution: tag

**Divide** each rule \( r \) into

- **quick part**: \( r' (\not\not + \text{tag}) \)
- **slow part**: \( r'' \) (everything else)

**Example:**

\[
R = \begin{cases} 
(r_1) \\
(r_2) \\
\ldots \\
(r_n) 
\end{cases}
\]

\[
R' = \begin{cases} 
(r'_1) \\
(r'_2) \\
\ldots \\
(r'_n) 
\end{cases}
\]

\[
\text{reverse}(R'') = \begin{cases} 
(r''_n) \\
\ldots \\
(r''_2) \\
(r''_1) 
\end{cases}
\]
**Basic solution: tag**

**Divide** each rule \( r \) into

- **quick part**: \( r' \) (\( \not\in \) + tag)
- **slow part**: \( r'' \) (everything else)

**Example:**

\[
R = \begin{cases} 
(r_1) \\
(r_2) \\
\vdots \\
(r_n) 
\end{cases} 
\]

\[
R' = \begin{cases} 
(r'_1) \\
(r'_2) \\
\vdots \\
(r'_n) 
\end{cases} 
\]

\[
\text{reverse}(R'') = \begin{cases} 
(r''_n) \\
\vdots \\
(r''_2) \\
(r''_1) 
\end{cases} 
\]

*The devil is in the detail*
Problem with tags: Example

\((true, \text{ DROP})\)

\((src = 1.2.3.4 \land tag = a, \ tag \leftarrow b; \text{ ACCEPT})\)

\((dst = 5.6.7.8 \land tag = b, \text{ NAT}(1.6.3.8, \star))\)
\((true, \ DROP)\)

\((src = 1.2.3.4 \land tag = a, \ tag \leftarrow b; \ ACCEPT)\)

\((dst = 5.6.7.8 \land tag = b, \ NAT(1.6.3.8, \star))\)

\(\downarrow\)

\((src = 1.2.3.4 \land tag = a, \ tag \leftarrow b)\)

\((dst = 5.6.7.8 \land tag = b, \ NAT(1.6.3.8, \star))\)

\((src = 1.2.3.4 \land tag = a, \ ACCEPT)\)

\((true, \ DROP)\)
(true, DROP)
(src = 1.2.3.4 ∧ tag = a, tag ← b; ACCEPT)
(dst = 5.6.7.8 ∧ tag = b, NAT(1.6.3.8, ⋆))

↓

(src = 1.2.3.4 ∧ tag = a, tag ← b)
(dst = 5.6.7.8 ∧ tag = b, NAT(1.6.3.8, ⋆))
(src = 1.2.3.4 ∧ tag = b, ACCEPT)
(true, DROP)
(true, DROP)
(src = 1.2.3.4 ∧ tag = a, tag ← b; ACCEPT)
(src = 1.2.3.4 ∧ tag = c, tag ← b; NAT(⋆,5.2.7.4))
(dst = 5.6.7.8 ∧ tag = b, NAT(1.6.3.8,⋆))
(true, DROP)

(src = 1.2.3.4 ∧ tag = a, tag ← b; ACCEPT)

(src = 1.2.3.4 ∧ tag = c, tag ← b; NAT(⋆, 5.2.7.4))

(dst = 5.6.7.8 ∧ tag = b, NAT(1.6.3.8, ⋆))
(true, DROP)
(src = 1.2.3.4 ∧ tag = a, tag ← b; ACCEPT)
(src = 1.2.3.4 ∧ tag = c, tag ← b; NAT(⋆, 5.2.7.4))
(dst = 5.6.7.8 ∧ tag = b, NAT(1.6.3.8, ⋆))

↓

(src = 1.2.3.4 ∧ tag = a, tag ← b1)
(src = 1.2.3.4 ∧ tag = c, tag ← b2)
(dst = 5.6.7.8 ∧ tag = b1, tag ← b; NAT(1.6.3.8, ⋆))
(dst = 5.6.7.8 ∧ tag = b2, tag ← b; NAT(1.6.3.8, ⋆))
(src = 1.2.3.4 ∧ tag = b2, tag ← b; NAT(⋆, 5.2.7.4))
(src = 1.2.3.4 ∧ tag = b1, tag ← b; ACCEPT)
(true, DROP)
Programming network behaviour at high level

**NetKAT**: Kleene Algebra with Tests for Networks

Kleene Algebra for reasoning about network structure

Boolean Algebra for reasoning about switch behaviour

Packet Algebra for reasoning about packets

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>.</th>
<th>¬</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>action (policy)</td>
<td>choice</td>
<td>composition</td>
<td>fail</td>
<td>skip</td>
<td></td>
</tr>
<tr>
<td>test (predicate)</td>
<td>disjunction</td>
<td>conjunction</td>
<td>negation</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

$f = n$ (test on a packet field)   \[ f ← n \] (modification of a packet field)
Programming network behaviour at high level

Network topology : a NetKAT formula
Each Firewall configuration : NetKAT formula
Code Motion & Refactoring : Equational theory
Security property : NetKAT formula
Property verification : Equational theory
Compilation from real firewall languages to NetKAT

From IFCL to NetKAT is quite simple:

- **Ruleset**: a NetKAT formula (a syntactic translation)
- **Control Diagram**: as Network topology
- **Non-propagation of Tags**: explicitly set to empty in ruleset

\[
[(\phi, t); R] = \begin{cases} 
(\phi) \cdot (t) + (\neg \phi) \cdot [R] & \text{if } t \in \{\text{ACCEPT, NAT}\} \\
(\neg \phi) \cdot [R] & \text{if } t = \text{DROP} \\
(\phi) \cdot (t) \cdot [R] + (\neg \phi) \cdot [R] & \text{if } t = \text{MARK}(m) \\
(\phi) \cdot [R'] \cdot [R] + (\neg \phi) \cdot [R] & \text{if } t = \text{GOTO}(R') \\
(\phi) \cdot [R'] \cdot [R] + (\neg \phi) \cdot [R] & \text{if } t = \text{CALL}(R') \\
(\phi) + (\neg \phi) \cdot [R] & \text{if } t = \text{RETURN}
\end{cases}
\]
Compilation from NetKAT to real firewall languages

NetKAT for configuring traditional firewalls: NetKAT $\rightarrow$ specific language

- Each language corresponds to a normal form
- Equational reduction to the specific normal form
- Compilation from normal form of NetKAT to target language
- Preserve the structure of the original configuration for free
Our NEW Goal
Our NEW Goal

High level representation

decompilation

High Level Management of Firewall Configurations
Our NEW Goal

High level representation

update
test
generate
verify
inspect

High Level Management of Firewall Configurations
Our NEW Goal

High level representation

Compilation

ipfw  pf  iptables