Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

A Stream-based Approach to Network Intrusion Detection

submitted by
Florian Kohn

submitted on
September 30th, 2019

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisor
Hazem Torfah, M.Sc.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Christian Rossow

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ________________________ ______________________________
(Datum/Date) (Unterschrift/Signature)

Abstract

Automatic approaches to network intrusion detection have become indispensable for the recognition of malicious activities within a network. With the help of network intrusion detection systems (NIDS), software applications that monitor a network for violations, network administrators can monitor the network against predefined attacks. With the rising complexity of modern day cyber attacks, there is a demand for more expressive specification languages, that allow us to specify complex attack patterns. In this thesis, we introduce a stream-based approach to network intrusion detection, based on an extension of the real-time stream language RTLola with parameterization. In contrast to most state of the art network intrusion detection systems, RTLola can express state-based properties. We address the interesting challenges posed by this strictly more expressive approach and demonstrate the features of RTLola using different real-world examples.

Acknowledgement

I want to thank Prof. Bernd Finkbeiner for supporting my work on the research topic of my choice. Moreover, i am grateful for all the help and support of my advisor Hazem Torfah, who always had a minute to spare for my questions. Additionally, I want to thank Prof. Bernd Finkbeiner and Prof. Christian Rossow for reviewing this thesis. Last but not least, a sincere thank you to my family, friends and all the people in the lab, who gave me all the support I needed to finish this thesis.

Contents

Chapter 1

Introduction

As cyber attacks nowadays pose a serious threat to our society, automated detection mechanisms like network intrusion detection systems have become an essential part of computer networks. A network intrusion detection system (NIDS) monitors a network for violations of a given specification. With the rising complexity of modern cyberattacks the need for more expressive specification languages has risen.
Current state-of-the-art network intrusion detection systems like Snort [22] feature a rule-based specification language with an extensive set of keywords to give easy access to the distinct properties of a network packet. In this rule-based approach each network packet is matched individually against the rules of the specification. This allows for an efficient implementation of the monitor as it can be highly parallelized for instance in the multi-threaded successor of Snort Suricata [1]. The inherent problem with the rule-based approach is that it is not possible to specify state-based properties in it. To specify such properties a more expressive specification language is needed.
To give an example why it is important to be able to monitor state-based properties, one can imagine a scenario where a hosting provider of FTP servers wants to track the failed login attempts per user and raise an alarm if a user fails to login more than five times within a given time window. To monitor such a specification one has to track the failed login attempts per user and then count all the failed request in the time frame.
In this thesis we present an extension of the stream-based specification language RTLola that is capable of expressing such stateful properties in a readable and convenient manner.

1.1 Stream-based Specification Languages

Stream-based specification languages use streams as their underlying computation model. A stream can be described as a continuously growing array of values. The current value of a stream is determined by an expression that may

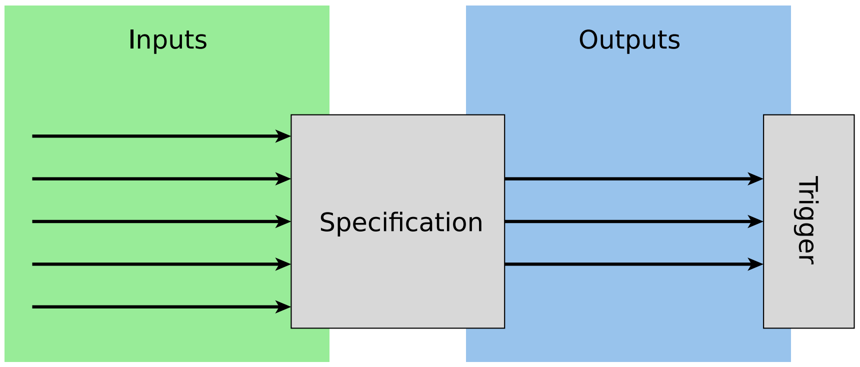


Figure 1.1: Anatomy of a Stream-based specification Language

depend on other streams. As illustrated in Figure 1.1 there are two classes of streams. Input streams capture the observations made of the system under test. Output streams represent deductions based on the input streams or other output streams. These output streams can influence the triggers. A trigger is a special kind of boolean output stream that raises an alarm to the user when its value becomes true.
A language is called synchronous, if all streams in the specification are extended at the same time and asynchronous, if every stream can progress at its own pace. Furthermore, stream-based languages differ in the way they allow a stream to be accessed. Most commonly a stream is accessed with a discrete offset based on the current position of the stream. Some languages have real-time capabilities, which for example, allow for an access of all values of a stream computed within the last minute.
If these ideas are interpreted in a network monitoring context input streams become streams of network packet properties like the source IP address or the destination port number. Every input stream describes a different packet property and is extended whenever a new packet arrives at the network monitor. Output streams model the behavior of an attacker. These output streams can then be used to compute a series of triggers that notify the administrator that his system is under attack.
The first stream-based specification language was Lola [10]. Lola is designed for synchronous monitoring of hardware circuits and uses a discrete time model. As this approach is quite limited RTLola [13], the real-time successor of Lola, was introduced. In RTLola every stream advances at its own pace determined either by the streams it depends on, or at fixed rate set by the specifier. This asynchronous model is combined with aggregations that can be computed over real-time sliding windows. Another variant of Lola, Lola 2.0 [11], introduces parameterization similar to the one known from quantified event automata [3] but lacks the asynchronous real-time nature of RTLola.
We need all the above features to specify the behavior mentioned in the exam-

ple. We first use the concept of parametrized stream templates, introduced in Lola 2.0, to sort the incoming packets into sets for the same destination and then use a real-time window, introduced by RTLola, to count these packets.

1.2 Overview

In Section 1.3 a summary of the related work done in the field of monitoring is presented. In Section 2.1 we extend the RTLola specification language with the concept of parameterization by presenting the modified syntax. In Section 2.2 an evaluation model for the newly described language is presented. In Chapter 3 the language features of RTLola are showcased using different real-world example specifications. In Chapter 4 experiments are conducted to evaluate our approach using the extended monitoring tool for RTLola. In Chapter 5 the results of this thesis are summarized.

1.3 Related Work

The class of Network Intrusion Detection Systems can be divided into two groups, namely signature based ones and anomaly based ones. Signature based Intrusion Detection Systems use a set of signatures which classify known malicious behavior. Opposed to that, anomaly based systems often compute statistics or use other machine learning techniques over properties of the network traffic to distinguish between normal and abnormal traffic. This has the inherent downside that such a NIDS can only detect abnormal not malicious behavior. This implies a higher false positive rate of such systems. Signature based NIDS on the other hand can only detect previously known attacks.
An elaborate signature based NIDS is Bro [21]. It features its own full sized script language which enables a specifier to express even the most advanced attack patterns. The downside of this fully featured script language is its complexity, which can lead to unmaintainable specifications that lack any kind of formal guarantee.
An example for an anomaly based NIDS is PHAD [18]. PHAD stands for *Packet Header Anomaly Detection* and builds a non-stationary machine learning model of the network without previously knowing which features of a packet header are relevant. During a test on the DARPA IDS evaluation data set [17], PHAD detected 70 out of 180 attacks while producing 100 false alarms.
In the following, two approaches to signature based intrusion detection systems are distinguished: The rule-based approach and the stream-based approach. In the rule-based approach signatures are given as rules. Each rule describes one attack pattern and is matched individually against every packet. An example for such a NIDS is Snort [22]. While Snort, and especially its multithreaded successor Suricata [1], feature a rich set of commands to access nearly every network packet property across the network stack, it misses the ability to keep state between two arriving packets.

To circumvent this limitation, one could use a temporal logic for monitoring a network, like Roger and Goubault-Larrecq proposed in their paper [23]. Tools based on this idea are for example Orchids [20], which uses an existential fragment of a first-order temporal logic. It can handle data through variable binding and parameterized constraints. Nevertheless, the tool is limited to eventuality based properties. Other examples for temporal logic based intrusion detection systems are Monid [19] and TeStID [2]. Monid uses the temporal logic Eagel, which features parametric recursive equations, while TeStID combines the temporal logic MSFOMTL (Many Sorted First Order Metric Temporal Logic) with stream data processing. There are also other real-time capable tools. For instance R2U2 [24], which uses bayesian reasoning together with the temporal logic MTL as a specification language. With its FPGA based monitor it is aimed at unmanned aerial system monitoring. Another monitor implementation for MTL (and MDL) is Arial [5], which provides space guarantees logarithmic in the event rate. Related to this is MONPOLY [4] which was presented by D. Basin et al. and uses MFOTL (metric first-order temporal logic) as its specification language. Both MONPOLY and Arial are limited to boolean verdicts.
Vigna and Kemmerer took a different approach to state-based NIDS. In their paper [26] they use state transition diagrams together with a network fact base modeling the network environment to specify attack scenarios. As the network environment is known to the system, it is additionally able to distribute monitoring instances throughout the network without loosing functionality.
In the stream-based approach to signature based network intrusion detection a signature is expressed using a stream-based specification language. An example for such a specification language is TeSSLa [15]. TeSSLa operates on pice-wise constant signals and is build upon an asynchronous stream model. Nevertheless, TeSSLa misses some language features that are crucial for a network monitoring purpose like parameterization. Gorostiaga and Sánchez presented the stream-based language Striver [14] which shares the real-time nature of RTLola, but lacks the ability to aggregate over stream values. The most recent work on RTLola includes the development of a monitor in the programming language Rust [12]. Additionally, there has been an effort [6] to create an FPGA based monitor for RTLola. Last but not least, Schwenger [25] introduced a formal semantics for RTLola and showed how it can be used to monitor cyber physical systems.

Chapter 2

RTLola

2.1 Syntax

In this section the syntax of RTLola as it is currently implemented in the StreamLab [12] runtime monitoring tool is presented. The syntax is based on the one of Lola 2.0 [11] but includes features like sliding windows and variable-rate input streams. In general a RTLola specification is a set of stream equations over strongly typed stream variables of the following form:

$$\begin{array}{lll}
\textbf{input} & i_1 : T_1 & \\
& \vdots & \\
\textbf{input} & i_n : T_n & \\
\textbf{output} & o_{n+1} & \langle \textbf{@}act_{n+1} \rangle : T_{n+1} \\
& & \langle \textbf{filter: } f_{n+1} \rangle \\
& & := e_{n+1} \\
& \vdots & \\
\textbf{output} & o_{n+m} & \langle \textbf{@}act_{n+m} \rangle : T_{n+m} \\
& & \langle \textbf{filter: } f_{n+m} \rangle \\
& & := e_{n+m} \\
& & \\
\textbf{output} & s_{n+m+1} & \langle (p_1, ..., p_l) : (V_{p_1} \times ... \times V_{p_l}) \rangle \langle \textbf{@}act_{n+m+1} \rangle : T_{n+m+1} \\
& & \langle \textbf{filter: } f_{n+m+1} \rangle \\
& & := e_{n+m+1} \\
& \vdots & \\
\textbf{output} & s_{n+m+k} & \langle (p_1, ..., p_l) : (V_{p_1} \times ... \times V_{p_l}) \rangle \langle \textbf{@}act_{n+m+k} \rangle : T_{n+m+k} \\
& & \langle \textbf{filter: } f_{n+m+k} \rangle \\
& & := e_{n+m+k}
\end{array}$$

In the following we call $i_1, ..., i_n$ independent stream variables as they refer to input streams which represent the inputs to the system. Furthermore, we call $o_{n+1}, ..., o_{n+m}$ dependent stream variables as they refer to output streams whose value depends on other stream variables. Finally, we introduce template stream variables $s_{n+m+1}, ..., s_{s+m+k}$ to RTLola. A template stream variable is defined over a set of parameters $p_1, ..., p_l$, such that an instance of a template stream is created by selecting the values for these parameters. Intuitively, a stream template variable refers to a set of stream instances, where each instance of the template has a different assignment of the parameters. This concept of parameterization is similar to the one introduced by Quantified Event Automata [3]. Output streams and template streams are equipped with an activation condition $act_{n+1}, ..., act_{n+m+k}$ respectively. The activation condition determines when a stream should be evaluated. There are two possibilities for the activation condition: It could either be a frequency at which the stream is evaluated or a boolean expression over stream names. This boolean expression can also be inferred automatically as the conjunction of all dependent streams.
Last but not least, $f_{n+1}, ..., f_{n+m+k}$ are optional boolean filter streams. A filter stream imposes an additional constraint to the activation of a stream, meaning a stream is only evaluated iff its filter stream evaluates to true at the same point in time.

2.1.1 The Type System

A type T_i in RTLola consists of two orthogonal types, a value type V_i and a stream type S_i. The value type is out of the set of all value types:

$$\mathcal{V} \ni v_1, v_2 ::= \mathit{Bool} \mid \mathit{String} \mid \mathit{Int} \mid \mathit{Float} \mid v_1 \times v_2$$

Furthermore, the symbol # is added as a value to indicate a non existing one. The value type indicates the usual semantics of a value. This includes, for example, the amount of memory needed to store such a value. For this reason, the fixed size variants of the above listed types are often used. The stream type indicates when a new value should be calculated. It can either be *independent*, meaning the point of time at which a value arrives cannot be influenced, *periodic*(n), meaning a new value is calculated periodically, where n describes the frequency in Hz, or *event-based*(ϕ), where ϕ is a boolean expression over stream variables. An event-based stream is extended whenever ϕ can be evaluated to $\top$ when assigning all occurring stream variables to $\top$ if the corresponding stream was extend or to $\bot$ otherwise. Note that, although RTLola is a strongly typed language, the types of output and template streams can often be inferred automatically and are thus optional.

2.1.2 Defining Stream Expressions

Each stream expression $e_{n+1}, ..., e_{n+m}$ is defined over a set of independent stream variables $i_1, ..., i_n$, output stream variables $o_{n+1}, ..., o_{n+m}$ and template stream variables $s_{n+m+1}, ..., s_{n+m+k}$. A stream expression $e_{n+1}, ..., e_{n+m}$ is defined recursively as follows:

- Let c be a constant of value type V, then c is a stream expression of value type V.
- Let i_p for $1 \leq p \leq n$ be an input stream variable with value type V_p, o_j for $n+1 \leq j \leq n+m$ be an output stream variable with value type V_j and s_l for $n+m+1 \leq l \leq n+m+k$ be a template stream variable with value type V_l and parameter type α_l, then i_p is a stream expression of value type V_p, o_j is a stream expression of value type V_j and $s_l(\beta)$ is a stream expression of value type V_l, if β is of value type α_l.
- If x_i for $1 \leq i \leq n+m$ is a stream variable with value type V, then $x_i.\mathit{offset}(\mathit{by} : y)$ and $x_i.\mathit{get}()$ are stream expressions of value type V for all $y \in \mathbb{N}$.
- If s_i for $n+m+1 \leq i \leq n+m+k$ is a template stream variable with value type V_i and parameter types $V_{p_1}, ..., V_{p_l}$ then $s_i(v_1, ..., v_l).\mathit{offset}(\mathit{by} : p)$ and $s_i(v_1, ..., v_l).\mathit{get}()$ are stream expressions of type V_i for $p \in \mathbb{N}$ and $v_1 \in V_{p_1}, ..., v_l \in V_{p_l}$.
- If f is a k-ary function of type $V_1 \times ... \times V_k \rightarrow V$ and $e_1, ..., e_k$ are stream expressions of value type $V_1, ..., V_k$ respectively, then $f(e_1, ..., e_k)$ is an expression of value type V.
- If b is a stream expression of value type bool and e_1, e_2 are stream expressions of value type V, then *if* b *then* e_1 *else* e_2 is a stream expression of value type V
- If v is a value of type V and e is a stream expression of value type V, then $e.\mathit{defaults}(\mathit{to} : v)$ is a stream expression of value type V.
- Let f be an aggregation function of type $V^* \rightarrow V'$ and let x_i for $1 \leq i \leq n+m$ be a stream variable with value type V, then $x_i.\mathit{aggregate}(\mathit{over} : d, \mathit{using} : f)$ is a stream expression of value type V' for $d \in \mathbb{R}_{\geq 0}$.
- Let f be an aggregation function of type $V^* \rightarrow V'$ and let s_i for $n+m+1 \leq i \leq n+m+k$ be a template stream variable with value type V and parameter types $V_{p_1}, ..., V_{p_l}$, then $s_i(v_1, ..., v_l).\mathit{aggregate}(\mathit{over} : d, \mathit{using} : f)$ is a stream expression of value type V' for $d \in \mathbb{R}_{\geq 0}$ and $v_1 \in V_{p_1}, ..., v_l \in V_{p_l}$.
- Let s_i be stream template variable of value type V and f be an aggregation function of type $\{s_{n+m+1}, ..., s_{n+m+k}\} \rightarrow V^* \rightarrow V'$, then $f(s_i)$ is a stream expression of type value V'. An example for such a function f is *count*. It evaluates to total number of instances of the given stream template variable. Note that the second argument of f is supplied at runtime as the set of current values of all stream instances corresponding to that stream variable.

The stream expressions $e_{n+m+1}, ..., e_{n+m+k}$ are additionally defined over the parameter names of their corresponding stream and are therefore defined in the same way as above with the addition:

- If p_i for $i \in 1, ..., l$ is a parameter of value type V_{p_i}, then p_i is a stream expression of type V_{p_i}.

Furthermore, RTLola specifications often contain one ore more triggers of the form:

$$\textbf{trigger } b$$

Where b is a stream expression of value type bool. If the value of b becomes true, the monitor raises an alert.

2.1.3 An Example Specification

```
input protocol: String
input IPv4::destination: (UInt8, UInt8, UInt8, UInt8)
input TCP::source: UInt16
input payload: String
input direction: String

output ftp: Bool := protocol="TCP" & TCP::source=21 & direction="Outgoing"
output failed: Bool := matches(payload, "/530\s+(Login|User|Failed|Not)/smi")

output FTPBruteforce(dst: (UInt8, UInt8, UInt8, UInt8)): Bool
                             filter (IPv4::destination=dst & ftp & failed)
                          := True

trigger FTPBruteforce(IPv4::destination).aggregate(over: 5s, using: count) > 5
```

Figure 2.1: A RTLola specification to detect an FTP bruteforce attack.

This section illustrates how to use RTLola to specify unwanted network behavior. Therefore, consider the example from the beginning, in which a hosting provider wants to protect his FTP server from too many login attempts by a single client. In practice, this indicates a bruteforce attack on the users password. During such an attack an adversary tries to guess the password of the user by trying every possible password one after another. This inevitably leads to an unusual high amount of failed login requests which can be detected by a NIDS. The RTLola specification in Figure 2.1 describes this behavior in the following way:
The input streams at the top of the specification represent the different attributes of a network packet. The input stream *protocol* represents the protocol used in the packet, while *payload* contains the actual payload bytes of the packet as a string. The *direction* input stream either takes the value *Incoming*, meaning the packet is coming into the local network, or *Outgoing*, meaning the packet is leaving the local network. The *IPv4::destination* stream contains the IPv4

address of the packet destination and the *TCP:source* stream contains the TCP source port of the packet. Note that the input streams are extended whenever a new network packet is captured.
To describe the attack itself, we first define an output stream called *ftp* which contains the value *True* iff a FTP packet is leaving the local network. This is achieved by comparing different packet fields with values defined by the protocol. For example, a FTP packet is always send over TCP port 21. The *failed* output stream is set to the value *True* if the packet is a response to a failed login attempt. This is done by matching on the payload of the packet using a regular expression. The declaration below is a stream template declaration which is parametrized over the destination IP address *dst*. Therefore, every instance of this template handles exactly one network client, namely the one identified by the destination IP address. The value of each instance is always *True*, but only at those positions where a new failed login request from the same network client is received. This is handled by the filter condition.
In the next step, we define when the monitor should raise an alarm by defining a trigger. In the condition of the trigger the previously defined template stream *FTPBruteforce* is used, referring to the instance corresponding to the current destination IP address. Next all positions of that instance which have been added during the last five seconds are counted by applying a count aggregation. Last but not least, the resulting value is compared to our defined threshold of five.

2.1.4 Parameterization as a Key Feature

The above example demonstrates how the addition of parametrization adds a key concept for network monitoring to RTLola. Parameterization allows the specifier to first divide the incoming events into sets with equal properties and then specify a behavior per set rather than for all incoming events. In the example the stream of packets is divided into sets in which each packet has the same destination address and then specify the malicious behavior for each network destination. This enables us to count the failed login requests per user rather than total number of failed requests. Therefore, this features allows the specification to scale with the number of users in the network as the given threshold does not depend on the number of users. Nevertheless, this plus of expressiveness does not come without some downsides: As the number of total stream instances is only known at runtime it is no longer possible to compute a memory bound for a specification a priori. Furthermore, the runtime of the monitor is proportional to the number of stream instances it has to manage. An especially expensive task is computing an aggregation over the instances of a stream template as the monitor has to track which instances influence the value of the aggregation and then has to update this mapping every time an instance is created or deleted. It follows that keeping an intermediate value of the aggregation computation that is only updated when an instance produces a new value is not possible in most cases anymore.

2.2 RTLola Evaluation

In this section we introduce a formal evaluation model for RTLola. Given a RTLola specification containing the following streams:

$$\begin{aligned}
\text{Input Streams: } & \mathbb{I} := i_1, ..., i_n \text{ of types: } T_1, ..., T_n \\
\text{Output Streams: } & \mathbb{O} := o_{n+1}, ..., o_{n+m} \text{ of types: } T_{n+1}, ..., T_{n+m} \\
\text{Template Streams: } & \mathbb{Y} := s^{\alpha_1}_{n+m+1}, ..., s^{\alpha_k}_{n+m+k} \text{ of types: } T_{n+m+1}, ..., T_{n+m+k}
\end{aligned}$$

Where $\alpha_i := V^{\alpha_i}_1 \times ... \times V^{\alpha_i}_l$ for $V^{\alpha_i}_1, ..., V^{\alpha_i}_l \in \mathcal{V}$ describes the parameter type of a template stream variable $s^{\alpha_i}_i$. In the following, let τ_i be the stream in the evaluation model corresponding to the input stream variable i_i and σ_i be the stream corresponding to the the output stream variable o_i. Furthermore, let ϵ^{β}_i describe an instance of the corresponding template stream variable $s^{\alpha_i}_i$, where β is of parameter type α_i. The following function is used to refer to this mapping:

$$\begin{aligned}
&\text{stream: } \mathbb{I} \cup \mathbb{O} \cup \mathbb{Y} \to \mathbb{P}(\mathbb{S}) \\
&\quad \text{stream}(i_i) := \{\tau_i\} \\
&\quad \text{stream}(o_i) := \{\sigma_i\} \\
&\quad \text{stream}(s^{\alpha_i}_i) := \{\epsilon^{\beta}_i \mid \beta \in \alpha_i\}
\end{aligned}$$

We can now define an evaluation model Γ of RTLola as a function from a continuous time $\mathbb{T} := \mathbb{R}_{\geq 0}$ into the power-set $\mathbb{P}(\mathbb{S})$ of $\mathbb{S}$, where $\mathbb{S}$ is the set of all possible streams in the evaluation model at a given time and defined as follows:

$$\mathbb{S} := \{\tau_i \mid 1 \leq i \leq n\} \cup \{\sigma_i \mid 1 \leq i \leq m\} \cup \{\epsilon^{\beta}_i | 1 \leq i \leq k \land \beta \in \alpha_i\}$$

Each stream π_i in $\mathbb{S}$ is a function from the time $t \in \mathbb{T}$ to a value $v \in V_i \cup \{\#\}$. Note, that the symbol # is used to denote a non existing value. To define the value of an output or template stream x_i at time t, we refer to its stream expression using e_i. This mapping is made explicit by defining a function *exp* which maps an output or template stream to its corresponding stream expression as follows:

$$\begin{aligned}
&exp(\sigma_i) = e_i \\
&exp(\epsilon^{\beta}_i) = e_i[p^i_1 \backslash v_1, ..., p^i_l \backslash v_l] \text{ for } \beta = (v_1, ..., v_l)
\end{aligned}$$

Where $p^i_1, ..., p^i_l$ refer to the parameter names of the stream template variable $s^{\alpha_i}_i$ and $[p^i_1 \backslash v_1, ..., p^i_l \backslash v_l]$ denotes the substitution of these parameter names with their corresponding assigned values.
We now pose three conditions on an evaluation model for RTLola:

(1) The evaluation model is populated with a sufficient amount of streams.

(2) A stream produces a value at the correct time.

(3) The value of a stream is computed correctly.

In the following, we will show how these conditions are ensured by defining a series of predicates.

2.2.1 Stream Aliveness

We define a boolean predicate *alive* that defines whether a semantic stream should be included in the evaluation model given time t. Intuitively, input streams and output streams are always alive and are therefore always included in $\Gamma(t)$. In contrast, a template stream instance is only alive iff it has been accessed before. First, let $uses(e, \epsilon_i^\beta)$ be a predicate that is true iff the evaluation of e depends on ϵ_i^β then *alive* can be formalized as follows:

$$\begin{aligned}
alive :& \mathbb{S} \times \mathbb{T} \to \mathbb{B} \\
alive(\tau_i, t) =& \top \\
alive(\sigma_i, t) =& \top \\
alive(\epsilon_i^\beta, t) =& \exists t' \leq t. \exists \pi_j \in \Gamma(t'). uses(exp(\pi_j), \epsilon_i^\beta)
\end{aligned}$$

To guarantee condition (1) it is required that $alive(\pi_i, t)$ holds for every stream $\pi_i \in \Gamma(t)$.

2.2.2 Stream Activation

To guarantee condition (2), we introduce a set of boolean predicates such that a stream π_i should only produce a value at time t iff $active(\pi_i, t)$ holds. For any stream $\pi_i \in \mathbb{S}$ with stream type S_i $active(\pi_i, t)$ is defined as follows:

$$\begin{aligned}
\text{active: } & \mathbb{S} \times \mathbb{T} \to \mathbb{B} \\
\text{active}(\tau_i, t) :=& \tau_i(t) \neq \# \\
\text{active}(\sigma_i, t) :=& \begin{cases} t = k * \frac{1}{f} & \text{if } S_i = \mathit{periodic}(f) \land k \in \mathbb{N} \\ \mathit{activated}(\phi, t) \land \mathit{filter\_active}(\sigma_i, t) & \text{if } S_i = \mathit{event\text{-}based}(\phi) \\ \bot & \text{otherwise} \end{cases} \\
\text{active}(\epsilon_i^\beta, t) :=& \begin{cases} t = k * \frac{1}{f} & \text{if } S_i = \mathit{periodic}(f) \land k \in \mathbb{N} \\ \mathit{activated}(\phi, t) \land \mathit{filter\_active}(\epsilon_i^\beta, t) & \text{if } S_i = \mathit{event\text{-}based}(\phi) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

Intuitively, input streams are independent which means that their activation cannot be influenced, hence they are active iff they receive a new value. Both, an output stream and a template stream instance may be periodic meaning that they only produce a new value iff the time t is a multiple of the streams period. In the event driven case a stream is activated iff its activation condition is satisfied and its corresponding filter stream is alive, active and contains the value $\top$ at the same time t. These properties are captured using the *filter_active* and *activated* predicates respectively. Therefore, let $vars(\phi)$ denote the set of all variables in a boolean expression ϕ out of the set of all boolean expressions $\mathbb{X}$ and $eval(\phi, \gamma)$ describe the evaluation of a boolean expression ϕ for a variable assignment $\gamma : vars(\phi) \times \mathbb{B}$. Furthermore, let *filter* : $\mathbb{S} \to \mathbb{S}$ return the filter stream of a stream. If a stream has no filter a constant $\top$ stream shall be returned.

$$
\begin{aligned}
\text{filter\_active: } & \mathbb{S} \times \mathbb{T} \to \mathbb{B} \\
\text{filter\_active}(\pi_i, t) = & \mathit{alive}(\pi_i, t) \wedge \mathit{active}(\mathit{filter}(\pi_i), t) \wedge \mathit{filter}(\pi_i)(t)
\end{aligned}
$$

$$
\begin{aligned}
\text{activated: } & \mathbb{X} \times \mathbb{T} \to \mathbb{B} \\
\text{activated}(\phi, t) = & \mathit{eval}(\phi, \{(x, b) \mid x \in \mathit{vars}(\phi) \wedge b = \bigvee_{\pi_j \in \mathit{stream}(x)} \mathit{active}(\pi_j, t)\})
\end{aligned}
$$

Note that the assignment is constructed such that every stream name is assigned to $\top$, iff there is a stream instance corresponding to that name, that is currently active and $\bot$ otherwise.

2.2.3 Expression Evaluation

To satisfy the third and last condition we define the value of stream expressions such that it holds for all $\pi_i \in \Gamma(t)$ that $\pi_i(t) = \mathit{val}(\mathit{exp}(\pi_i), t)$ iff $\mathit{active}(\pi_i, t)$ holds. Therefore, let π_i be any stream in $\mathbb{S}$ then $\mathit{val}(\mathit{exp}(\pi_i), t)$ is defined recursively as follows:

$$
\begin{aligned}
\mathit{val}(c, t) &= c \\
\mathit{val}(i_i, t) &= \tau_i(t) \\
\mathit{val}(o_i, t) &= \sigma_i(t) \\
\mathit{val}(s_i(\beta), t) &= \epsilon_i^{\beta}(t) & \text{for } \beta \in \alpha_i
\end{aligned}
$$

$$
\begin{aligned}
\mathit{val}(f(e_1, ..., e_h), t) &= f(\mathit{val}(e_1, t), ..., \mathit{val}(e_h, t)) \\
\mathit{val}(\text{if } b \text{ then } e_1 \text{ else } e_2, t) &= \begin{cases} \mathit{val}(e_1, t) & \text{if } \mathit{val}(b, t) = \mathit{True} \\ \mathit{val}(e_2, t) & \text{otherwise} \end{cases} \\
\mathit{val}(e_1.\text{defaults}(\mathit{to} : v'), t) &= \begin{cases} \mathit{val}(e_1, t) & \text{if } \mathit{val}(e_1, t) \neq \# \\ v' & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{val}(x_i.\text{offset}(\mathit{by} : y), t) &= \begin{cases} \mathit{val}(x_i.\text{offset}(\mathit{by} : y - 1), t') & \\ \qquad \text{if } (v, t') \in \mathit{sample}(\pi_i, t) \wedge y > 0 & \\ v \qquad \text{if } (v, t') \in \mathit{sample}(\pi_i, t) \wedge y = 0 & \\ \# \qquad \text{otherwise} & \end{cases} \\
\mathit{val}(x_i.\mathit{get}(), t) &= \pi_i(t)
\end{aligned}
$$

Let $s_i^{\alpha_i}$ be a template stream and $\beta \in \alpha_i$.

$$val(s_i(\beta).\text{offset}(by: y), t) = \begin{cases} val(s_i(\beta).\text{offset}(by: y-1), t') & \\ \quad \text{if } (v, t') \in sample(\epsilon_i^\beta, t) \land y > 0 \\ v \quad \text{if } (v, t') \in sample(\epsilon_i^\beta, t) \land y = 0 \\ \# \quad \text{otherwise} \end{cases}$$

$$val(s_i(\beta).get(), t) = \epsilon_i^\beta(t)$$

Aggregation over instances of a stream template variable s_i are defined as follows:

$$val(f(s_i), t) = f(s_i)(\{\epsilon_i^\beta(t) \mid \epsilon_i^\beta \in \Gamma(t)\})$$

Aggregation over a real-time window of size d using an aggregation function f is defined as follows for an input or output stream variable x_i and the corresponding stream π_i in the evaluation model and a template stream $s_i^{\alpha_i}$ for $\beta \in \alpha_i$:

$$\begin{aligned} val(x_i.\text{aggregate}(over: d, using: f), t) = \\ f(\{\pi_i(t') \mid \forall t' : max(t-d, 0) \leq t' \leq t \land \pi_i(t') \neq \#\}),\ \text{for } d \in \mathbb{T} \\ val(s_i(\beta).\text{aggregate}(over: d, using: f), t) = \\ f(\{\epsilon_i^\beta(t') \mid \forall t' : max(t-d, 0) \leq t' \leq t \land \epsilon_i^\beta(t') \neq \#\}),\ \text{for } d \in \mathbb{T} \end{aligned}$$

Where $sample(\pi_i, t)$ describes the last value and the corresponding timestamp of a stream π_i and is defined as follows:

$$\begin{aligned} & sample(\pi_i, t) := \\ & \{(\pi_i(t'), t') \mid\ 0 \leq t' < t \land \pi_i(t') \neq \# \land \nexists t^* : t' < t^* < t \land \pi_i(t^*) \neq \#\} \end{aligned}$$

2.2.4 Correct Evaluation Model

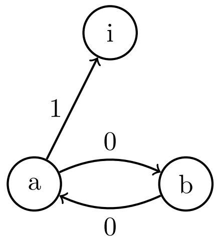
The predicates for conditions (1), (2) and (3) can be combined into one definition of a correct evaluation model:

Definition: Correct Evaluation Model

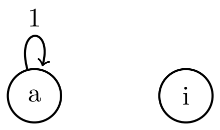
Let $\Gamma : \mathbb{T} \to \mathbb{P}(\mathbb{S})$ be an evaluation model for RTLola. We call Γ correct, iff it holds for a fixed time $\mathcal{T}$:

$$\forall 0 \leq t \leq \mathcal{T}. \forall \pi_i \in \Gamma(t). alive(\pi_i, t) \land \pi_i(t) = \begin{cases} val(exp(\pi_i), t) & \text{if } active(\pi_i, t) \\ \# & \text{otherwise} \end{cases}$$

2.2.5 Well-defined specifications



(a) A dependency graph with a zero weight cycle



(b) A dependency graph with a negative self reference.

Figure 2.2

Similar to the definition given for Lola 2.0 [11] we define what it means for a specification to be well-defined: A specification Φ consisting of a set of well-typed streams is well-defined, iff it has a unique evaluation model for a given set of input traces of length N. This condition does not immediately hold for any specification. For example, a self reference with offset 0 leads to the non existence of an evaluation model, because the *active* predicate can not be evaluated for such stream. As this semantic property is expensive to validate a notion of *well-formedness* is introduced, which implies *well-definedness*, but can be checked on a syntactic layer by analyzing the dependency graph of a specification.
The dependency graph of a specification is a weighted directed multi-graph (V, E), where the set of vertices is defined as follows:

$$V := \{i_1, ..., i_n, o_{n+1}, ..., o_{n+m}, s^{\alpha_i}_{n+m+1}, ..., s^{\alpha_k}_{n+m+k}\}$$

For any $v_i \in V \backslash \{i_1, ..., i_n\}$ an edge is added according to the following rules:

- If the stream expression of v_i contains the subexpression v_j.get(), then an edge $(v_i, v_j, 0)$ is added.
- If the stream expression of v_i contains the subexpression: v_j.offset($by : w$), then an edge (v_i, v_j, w) is added.
- If the stream expression of v_i contains a subexpression of the form v_j.aggregate($over : x, using : f$), then an edge $(v_i, v_j, 0)$ is added.
- If the activation condition of v_i contains the stream variables A_i then an edge $(v_i, v_j, 0)$ is added for all $v_j \in A_i$
- If v_i has a filter stream v_j, then the edge $(v_i, v_j, 0)$ is added.

Intuitively, an edge (v_i, v_j, w) represents the fact, that the stream v_i depends on the stream v_j with an offset w. A path in the graph is a sequence $v_1 \xrightarrow{w_1}$

$v_2 ... v_{k-1} \xrightarrow{w_{k-1}} v_k$, such that it holds: $\forall (v_i, v_{i+1}, w_i) \in E$. A cycle is a path such that it holds: $v_1 = v_k$. The weight of a cycle is defined as the sum over all weights along the path. A specification is called *well-formed*, iff the following two properties are met:

1. The dependency graph does not contain any zero weight cycle.
2. For any node $v_i \in V$ there should exist a path $v_1 \xrightarrow{w_1} v_2 ... v_{k-1} \xrightarrow{w_{k-1}} v_k$, such that $v_1 = v_i$ and v_k is an input stream or a periodic stream.

We now present two specifications, where one of the above properties is violated respectively. First observe the following specification:

```
input i: Int8
output a := i.offset(by: 1).defaults(to: 0) + b.get()
output b := a.get()
```

Intuitively, this specification cannot be evaluated, as the stream a depends on the stream b with offset 0 and the other way around. Therefore, we can never calculate a value for a or b. The dependency graph of the specification is depicted in Figure 2.2 (a). One can easily see, that the graph contains a zero weight cycle $a \xrightarrow{0} b \xrightarrow{0} a$. It follows, that the first property for well-formedness is violated. Next, consider the following example where the second property is violated:

```
input i: Int8
output a := a.offset(by: 1).defaults(to: 0) + 1
```

The dependency graph of this specification is shown in Figure 2.2 (b). It again contains a cycle, but this time the cycle has weight 1, which means, that the first property is not violated. Intuitively, the stream a describes an increasing counter starting at 0. The problem arises when defining the value of the stream at a fixed time t which cannot be done. When looking at the inferred activation condition of a the problem becomes more obvious as it is a boolean expression containing just the variable a which would translate to: The value of the stream a is computed whenever there is a new value for the stream a. Therefore, it is impossible to determine the points in time when a should be evaluated. Looking at the dependency graph one can see that it does not contain a path from node a to the input i which means that the second property is invalidated.

Chapter 3

Specifying Security Properties

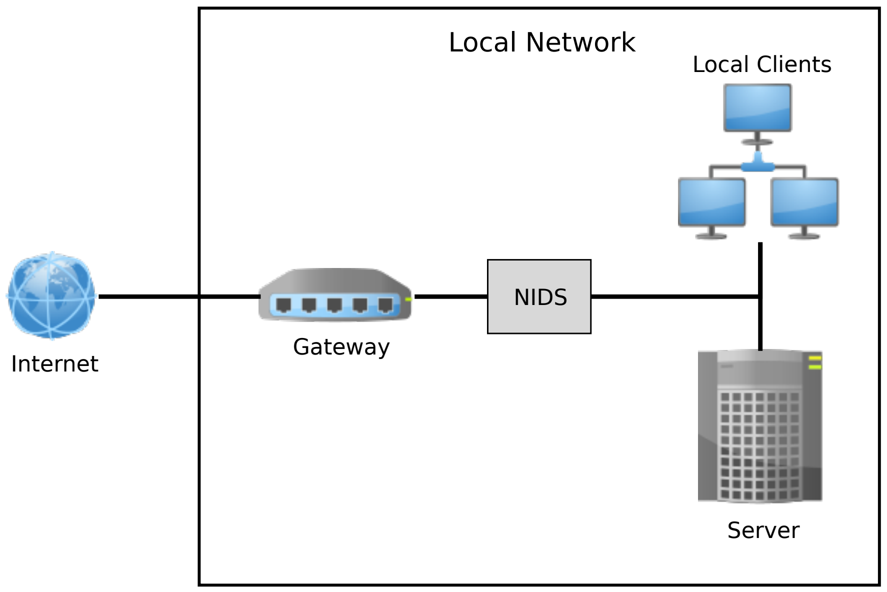


Figure 3.1: The network topology assuming for the examples.

In this section a series of examples is shown, that demonstrate the expressiveness of the presented specification language. The functionality of each specification is described and explained. For our examples assume a network consisting of a local server together with some local clients. Both are connected to the internet through a gateway such that all the network traffic first has to pass through our

NIDS. This topology is depicted in Figure 3.1. For better readability the input streams needed for the following examples are now stated and omitted later.

```
input protocol: String
input payload: String
input direction: String
input timestamp: UInt64

input IPv4::source: (UInt8, UInt8, UInt8, UInt8)
input IPv4::destination: (UInt8, UInt8, UInt8, UInt8)
input IPv4::flags::df: Bool
input IPv4::flags::mf: Bool

input TCP::source: UInt16
input TCP::destination: UInt16
input TCP::ack_number: UInt32
input TCP::window_size: UInt16
input TCP::flags::ack: Bool
input TCP::flags::syn: Bool
input TCP::flags::fin: Bool

input UDP::source: UInt16
input UDP::destination: UInt16
```

The input *protocol* is derived as the hightest detected protocol. For instance, if the payload of the network packet contains a TCP header the *protocol* input would state *TCP* as its value. In contrast, the *payload* input stream contains all the data that could not be parsed as any known protocol header. The value of the *direction* input stream depends on the source IP address of the packet and a description of the local network IP address range and represents the direction of the packet, meaning whether it is leaving or coming into the local network. The *timestamp* stream represents the UNIX timestamp of the current packet. The *IPv4::source* and *IPv4::destination* input streams contain the source and destination IPv4 address of the current packet respectively while *IPv4::flags::df* refers to the *Don't Fragment* flag of the IP header and *IPv4::flags::mf* refers to the *More Fragments* flag. The next seven input streams refer to fields in the TCP header. The streams *TCP::source* and *TCP::destination* represents the TCP source and destination port respectively while *TCP::ack_number* has the acknowledgment number of the current TCP packet as it's value. Furthermore, the *TCP::window_size* stream captures the *window size* field of the TCP header that encodes the number of bytes the sender of the packet is willing to receive. The access to the *ACK*, *SYN* and *FIN* TCP flags is provided using the input streams *TCP::flags::ack*, *TCP::flags::syn* and *TCP::flags::fin*. Last but not least, the UDP source and destination port are referenced using the streams *UDP::source* and *UDP::destination*. Note, that all available input streams are documented in appendix A.

3.1 Non State-based Specifications

We begin with specifying some simple non state-based properties. The first step for an attacker, when wanting to infiltrate a system, often is to run a port scan on the victim machine. A port scan describes the technique of sending tcp syn packets to different (possibly all) ports of the target machine. By observing from which ports he receives an answer the attacker can conclude which services are running on the target. Such a pattern can easily be specified in RTLola. The following specifications detects such packets and is especially tailored towards port scans executed with the tool *NMap*.

```
output TCPPortScan := protocol="TCP" & direction="Incoming" &
     TCP::ack_number=0 & !IPv4::flags::df & payload="" & TCP::flags::syn &
     TCP::window_size = 1024

trigger TCPPortScan
```

We first define a boolean verdict *TCPPortScan* which captures the properties of a TCP packet used by NMap for a port scan. Next, the trigger is defined to raise an alert whenever this boolean verdict becomes *True*. Note, that the output stream declaration can be omitted and the trigger definition set directly to the expression of *TCPPortScan*.
Another typical use case for a NIDS is to check for an attack signature corresponding to a specific software security flaw. For instance, there was a SQL injection vulnerability discovered in the *j_photo* component of Joomla. Joomla is an open source content management system for web development, which is widely used across the internet. The specification below can be used to detect this specific vulnerability.

```
output SQLInjection := protocol="TCP" & direction="Incoming" &
     TCP::destination=80 & matches(payload,
"/GET /index\.php\?option=com_jphoto&.*view=category&.*Id=INSERT.+INTO/Ui")

trigger SQLInjection
```

Again, we define a boolean verdict that becomes *True* if a HTTP requests to the local server includes a maliciously formed url. This is checked using the *matches* function which matches a string against a given regular expression. The trigger is then set to alert whenever this boolean verdict becomes *True*.

3.2 Denial of Service Attacks

In the following we present two different approaches to detecting a denial of service attack, or short DoS attack. During a denial of service attack an adversary tries to exhaust the resources of the target machine. Such resources are memory, network bandwidth or computational power. For the case of network bandwidth this is usually done by sending exceptionally many packets to the target until his local network infrastructure collapses under this load. These kind of attacks have proven to be especially hard to detect as there are also valid reasons like flash crowds why a network is hit with an unusually high amount of traffic. The next specification aims to detect a denial of service attack against a VoIP (voice over IP) phone in the local network. The VoIP standard is based on UDP and require an *INVITE* message to be send at the beginning of a call. If an attacker repeatedly sends this message he can inhibit the VoIP device from functioning at all.

```
output voip := protocol="UDP" & UDP::destination=5060 & direction="Incoming"
output invite := matches(payload, "/INVITE/smi")

output VoIPInviteFlood(src: (UInt8, UInt8, UInt8, UInt8)): Bool
                      filter: IPv4::source=src & voip & invite
                   := True

trigger VoIPInviteFlood(IPv4::source).aggregate(over: 60s, using: count) > 100
```

The *voip* output stream is *True*, iff a VoIP specific packet enters the local network while the *invite* stream is *True*, iff the packet contains a VoIP *INVITE* message. This is again achieved by matching the packet payload against a regular expression. To detect the attack all VoIP *INVITE* message of a user are counted and an alarm is raised if this count exceeds 100 during the period of 60 seconds. Therefore, we first declare a template stream *VoIPInviteFlood* that is parameterized over the source IP address and is always *True* at those positions, where another VoIP *INVITE* message from the same user is received. This is expressed using the filter condition of the stream. Note, the similarities to the FTP bruteforce example in Figure 2.1. The trigger of the specification refers to the *VoIPInviteFlood* instance corresponding to the source IP address of the current packet with the expression `VoIPInviteFlood(IPv4::source)`. Note, that if this instance does not already exist it is created. In the next step the real-time capabilities of RTLola are put to use for the aggregation over all values of the last 60 seconds of this instance using the function *count* which counts the number of values in this interval. This yields the number of VoIP *INVITE* messages from the current source IP address of the last 60 seconds. Last but not least, it is checked whether this number exceeds 100 and raise an alert if it does.
A DDoS, or distributed denial of service attack is an advanced DoS attack, where an adversary uses multiple machines under his control to amplify the attacks' efficiency. The detection mechanisms for DDoS attacks differ substantially from the ones of DoS attacks, as the attack is no longer limited to one network client. Therefore, a correlation between the behaviour of different users has to be detected. One approach, which was proposed by Bhuyan, Bhattacharyya and Kalita [8], is to compute the generalized entropy over the source IP addresses which is expected to drop during a DDos attack as more packets from the same IP addresses are expected to arrive . Note, that the generalized entropy of order α for a discrete probability distribution X is defined as follows:

$$H_\alpha(X) = \frac{1}{1-\alpha} log_2 \left(\sum_{p \in X} p^\alpha \right)$$

We now present a specification that computes this metric to detect a DDos

attack step by step:
First of all, let the order of the generalized entropy be defined using a constant stream that produces the value 2 at a frequency of $1Hz$:

```
output alpha @1Hz := 2
```

The next streams compute the discrete probability of all source IP addresses of the last five minutes again at a frequency of $1Hz$:

```
output packetsPerIp(src: (UInt8, UInt8, UInt8, UInt8)):Bool
    filter: IPv4::source = src
    := True

output totalCount: UInt64
    := IPv4::source.aggregate(over: 5min, using: count)

output sourceProbability(src: (UInt8, UInt8, UInt8, UInt8)) @1Hz: Float64
    := packetsPerIp(src).aggregate(over: 5min, using: count).defaults(to: 0)/
                                totalCount.offset(by: 0).defaults(to: 1)
```

First, all packets of one source IP address are accumulated using the template stream *packetsPerIp*. Then the total number of packets that arrived during the last five minutes is computed using an aggregation over the *IPv4::source* stream as the *totalCount* stream. Finally, define the *sourceProbability* stream that computes the desired probability by first counting the packets per IP using an aggregate and dividing this by the total number of packets in that time period. Note, that we have to specify appropriate default values and offsets as the *sourceProbability* stream is periodic while the *packetsPerIp* stream and the *totalCount* stream are event based. Therefore, it cannot be assume that the *packetsPerIp* stream and the *totalCount* stream currently have a value when the *sourceProbability* stream is evaluated. The next streams finally compute the generalized entropy:

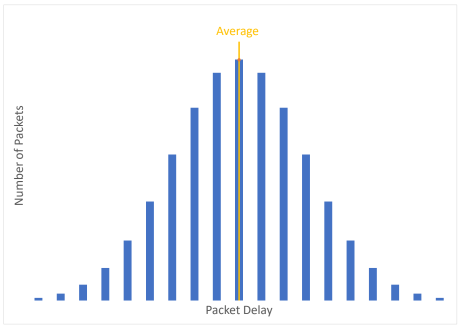
```
output sourceProbabilityAlpha(src: (UInt8, UInt8, UInt8, UInt8)): Float64
    := pow(sourceProbability(src), alpha)

output sourceEntropy: Float64
    := 1/(1 - alpha) * log2(sum(sourceProbabilityAlpha))

output invoke: Float64
    := sourceProbabilityAlpha(IPv4::source)
```

The stream *sourceProbabilityAlpha* exponentiates the computed probability of each source address with the constant value of alpha such that the stream *sourceEntropy* can compute the generalized entropy by using an aggregation over the stream instances of *sourceProbabilityAlpha* to compute the sum. Notice the stream *invoke* which is defined to create the instances of *sourceProbabilityAlpha*. Last but not least, the trigger is defined:

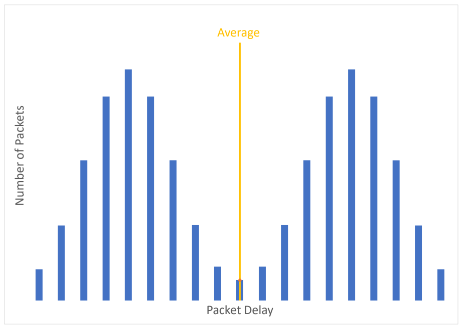
```
output avgEntropy: Float64
    := sourceEntropy.aggregate(over:1h, using: avg)

trigger abs(sourceEntropy - avgEntropy) > 5
```

To obtain a baseline for a comparison the average entropy of the last hour is computed and compared to the current entropy. A significant difference between these two values is a strong indication of a DDoS attack. This specification showcases another RTLola feature, namely periodic streams. As the source address probabilities are defined to be computed at a rate of 1 Hz every dependent stream, including the trigger, is evaluated at that rate as well. This allows for the metric to be evaluated independent from the rate of the incoming packets which is essentials as during a DDoS attacks network packets arrive at much higher frequencies which could lead to the exhaustion of the monitors computational resources.

3.3 Covert Channel



(a) Packet delay histogram for normal network traffic.



(b) Packet delay histogram if a delay based covert channel is present.

Figure 3.2: Packet Delay Histograms

As a third example we present specifications for the detection of covert channels. A covert channel is a communication channel that tries to hide the fact that communication is happening. It can be used by an attacker to, for instance, exfiltrate sensitive data out of the local network without being noticed. Two types of covert channel can be distinguished in a computer network: A storage based covert channel and a delay based covert channel. A storage based covert channel hides the data in unused portions of protocol headers. For example, the IPv4 *protocol* header field is of eight bit width, but is only defined for values from 0 to 142 which means all values above 142 can be used by an attacker to encode data. To detect a storage based covert channel one has to check the value range of such fields. A specification for the IPv4 *protocol* header field is depicted below:

```
trigger IPv4::protocol > 142
```

A delay based covert channel functions differently. Under the assumption that an adversary can control the time when a network packet is send by the host

the adversary can, for example, encode a logical one as a long delay between two packets and a logical 0 as a short delay. This way he transfers bits without actively sending any network packet.
Berk, Giani and Cybenko showed [7] that a delay based covert channel can be detected by analyzing the histogram that accumulates the number of packets in a delay interval. During the absence of a delay based covert channel the histogram should depict a normal distribution of the packets over their delays. Such a histogram is shown in Figure 3.2 (a). In particular it holds in this case that the most packets arrive with the average delay. If a covert channel is present, we see two local maxima in the histogram around the delays that the adversary chooses for the encoding of a bit. This is depicted in Figure 3.2 (b). The difference in the two histograms can be used to detect the presence of a covert channel by comparing the maximum number of packets in an interval to the number of packets in the average interval. If they mismatch a delay based covert channel is likely. As shown in Figure 3.2 (b) for the case that a delay based covert channel is present the average interval is between the two local maxima which means that the average interval cannot contain the maximum number of packets. In the following a specification that computes this histogram for every network client at runtime is presented and a trigger is defined such that an alert is raised when a delay based covert channel is likely. To improve the readability of the specifications let expressions of the form `let x = e` are introduced as a syntactic sugar. Let expressions can be resolved by replacing all occurrences of x with the expression e in all stream expressions declared after the let expression until the end of a stream declaration is reached.

```
output TimestampBySource(src: (UInt8, UInt8, UInt8, UInt8)):UInt32
    filter: IPv4::source = src
    := timestamp

output InterPacketDelay(src: (UInt8, UInt8, UInt8, UInt8)):Float64
    filter: IPv4::source = src
    := TimestampBySource(src) - TimestampBySource(src).offset(by: 1)

output InterPacketDelayAvg(src: (UInt8, UInt8, UInt8, UInt8)):Float64
    filter: IPv4::source = src
    := InterPacketDelay(src).aggregate(over: 5min, using: avg)
```

The specification starts by accumulating all timestamps corresponding to one network client using the template stream *TimestampBySource*. The stream *InterPacketDelay* calculates the current packet delay for every user while *InterPacketDelayAvg* computes an average over these values. The following two streams are concerned with discretizing these two values by computing the upper interval bound for the values:

```
output CurrentIntervalBound(src: (UInt8, UInt8, UInt8, UInt8)):UInt8
    filter: IPv4::source = src
    let delay = InterPacketDelay(src)
    := if delay >= 0 & delay < 10 then 10 else
       if delay >= 10 & delay < 20 then 20 else
       if delay >= 20 & delay < 30 then 30 else
       if delay >= 30 & delay < 40 then 40 else
```

```
        if delay >= 40 & delay < 50 then 50 else
        if delay >= 50 & delay < 60 then 60 else
        if delay >= 60 & delay < 70 then 70 else
        if delay >= 70 & delay < 80 then 80 else
        if delay >= 80 & delay < 90 then 90 else
        if delay >= 90 & delay < 100 then 100 else 110

output AverageIntervalBound(src: (UInt8, UInt8, UInt8, UInt8)):UInt8
    filter: IPv4::source = src
    let delay = InterPacketDelayAvg(src)
    := if delay >= 0 & delay < 10 then 10 else
       if delay >= 10 & delay < 20 then 20 else
       if delay >= 20 & delay < 30 then 30 else
       if delay >= 30 & delay < 40 then 40 else
       if delay >= 40 & delay < 50 then 50 else
       if delay >= 50 & delay < 60 then 60 else
       if delay >= 60 & delay < 70 then 70 else
       if delay >= 70 & delay < 80 then 80 else
       if delay >= 80 & delay < 90 then 90 else
       if delay >= 90 & delay < 100 then 100 else 110
```

These discrete interval bounds are now used to define the stream representing the histogram as follows:

```
output Histogram(src: (UInt8, UInt8, UInt8, UInt8), bound: UInt8):UInt32
    let intervalBound = CurrentIntervalBound(src)
    filter: IPv4::source = src & bound=intervalBound
    := Histogram(src, bound).offset(by: 1).defaults(to: 0) + 1

output invoke: UInt32
    := Histogram(IPv4::source, CurrentIntervalBound(IPv4::source))
```

The *Histogram* stream is parameterized over a source IP address and a bound. It follows that each instance of the stream represents a single interval of a network client. The value of an instance expresses the number of packets in this interval. Therefore the value is increased whenever a packet with the same source IP address arrives with a delay falling into this interval. To create new instances of the *Histogram* stream an additional *invoke* stream is denoted. Finally, the probability for a covert channel is calculated and a trigger defined:

```
output ProbCovChannel(src: (UInt8, UInt8, UInt8, UInt8)):Float64
    let averageBound = AverageIntervalBound(src)
    let averageCount = Histogram(src, averageBound)
    let maxCount = max(Histogram, src)
    filter: IPv4::source = src
    := 1 - (averageCount / maxCount)

trigger ProbCovChannel(IPv4::source) > 0.8
```

To calculate the probability the number of packets in the average interval is divided by the maximum number of packets in an interval. This maximum is calculated by aggregating over all instances of the *Histogram* stream while fixing the source parameter using the so defined aggregation function *max*. This specification illustrates the expressiveness of RTLola and shows that although RTLola formally classifies as a specification language for signature based NIDS it is possible to compute statistics in real time to detect anomalies.

Chapter 4

Experiments

In this section the performance of the presented network intrusion detection approach is evaluated. For this purpose, the existing monitoring tool for RTLola called StreamLab [1] was extended with a networking interface A. The tool is implemented in the programming language Rust and is capable of handling recorded network traffic in the well-known *pcap* file format, as well as directly parsing network packets when they are received by the network interface.
For the experiments we use the DARPA network intrusion detection evaluation dataset from 1999 [17]. The dataset aims to model the traffic and attacks seen at a military base. It includes five weeks worth of captured data with a two hour maintenance break every day. For the experiments the capture file from the Tuesday of the second week is used. The following specification is monitored to compare the performance of the tool to the rule-based NIDS Suricata [1]:

```
input protocol: String
input TCP::ack_number: UInt64
input IPv4::flags::df: Bool
input TCP::flags::syn: Bool
input IPv4::length: UInt64
input IPv4::ihl: UInt64
input TCP::data_offset: UInt64

output payloadLength := IPv4::length - IPv4::ihl * 4 - TCP::data_offset * 4

output TCPPortScan := if protocol="TCP" & TCP::ack_number=0 &
     !IPv4::flags::df & payloadLength=0 & TCP::flags::syn then 1 else 0

output threshold @1min := TCPPortScan.aggregate(over: 1min, using: sum) > 10

trigger threshold
```

The specification aims to detect TCP syn port scans. During such a port scan a high amount of specially crafted TCP syn packets is send, which can be detected by a NIDS. The above specification is tested against the following Suricata rule

[1] https://www.react.uni-saarland.de/tools/streamlab/

taken from the emerging threats open ruleset [2]:

```
alert tcp any any -> any any (msg:"ET SCAN NMAP -sS window 1024";
   fragbits:!D;
   dsize:0;
   flags:S,12;
   ack:0;
   threshold: type both, track by_dst, count 10, seconds 60;
   reference:url,doc.emergingthreats.net/2009582;
   classtype:attempted-recon; sid:2009582; rev:3;
   metadata:created_at 2010_07_30, updated_at 2010_07_30;)
```

All experiments were run on a hexa-core machine with a 3.7 GHz AMD Phantom II processor equipped with 12 Gb of RAM. The results are shown in the table below:

# Packets	Suricata Alerts	RTLola Triggers	Suricata Time in Sec.	RTLola Time in Sec.
300000	261	150	2.492	0.921
600000	473	265	3.705	1.810
900000	837	381	6.437	2.864
1200000	1191	472	8.647	3.803
1585120	1630	721	11.671	6.139

This outcome demonstrates that the presented approach to network intrusion detection is indeed efficient with being about two times faster as its competitor Suricata. The higher number of Suricata alerts is due to the fact that Suricata raises multiple alerts for the same TCP syn scan attack. The RTLola specification reduces this effect by using a periodic *threshold* stream that aggregates the packets connected to a single TCP syn scan before firing a trigger.
To evaluate the online monitoring performance of the tool a local network was monitored over the duration of 1 hour for a DoS attack using the following specification:

```
input IPv4::source: (UInt64, UInt64, UInt64, UInt64)

output CurrentPacketCount @1Hz := IPv4::source.aggregate(over: 1min, using:
    count)

output AveragePacketCount @1Hz := CurrentPacketCount.aggregate(over: 10min,
    using: avg).defaults(to: 0)

trigger CurrentPacketCount > AveragePacketCount * 10
```

During this period of time we measured the CPU usage of the monitor, while the network was put under load through online browsing and the use of streaming services. Furthermore, the network was hit by a three DoS attack during this period of time. All attacks were successfully detected by the NIDS. During these DoS attacks the CPU usage of the monitor peaked at 17% while its overall average usage was 5%. This confirms the performance results from the previous experiment.

[2]`https://rules.emergingthreats.net/open/suricata-4.0/emerging-all.rules`

Chapter 5

Conclusion

In this thesis, we presented a stream-based approach to network monitoring based on an extension of RTLola with parameterized output templates. We demonstrated the expressiveness of this approach using different real world examples. Furthermore, we implemented a networking interface for the existing monitoring tool for RTLola such that it can be used as a network intrusion detection system and validated its performance by conducting a series of experiments.

In the future, we aim to increase the performance of the monitor by analysing how and to what extend parameterized RTLola specifications can be parallelized. As stream instances of the same template are independent from each other one can think of spreading stream instances over multiple threads. Furthermore, it remains to be investigated whether a technique similar to the one presented for the efficient evaluation of sliding windows [16] can be applied to aggregations over stream instances as well. This is especially interesting if these instances are distributed across several threads.

To further improve the performance of the NIDS one could introduce a distributed monitoring approach to RTLola. Similar to the approach shown in [26] one could use a static analysis of the specification to determine nodes in the network at which a monitor should be placed. This would additionally require to modify the monitor such that a state could be shared between multiple running instances.

As shown with the example of the delay based covert channel in Section 3.3 RTLola is already capable of computing statistical values. In the future these capabilities can be extended by adding a separate anomaly based component to the monitor such that for example RTLola computes the features for the anomaly based component. A bidirectional communication approach between the two components can be desirable as well. This way RTLola could be used to detect nearly any network threat imaginable.

Another interesting concept to explore in more detail is to allow the expression of dependencies between stream instances of a single template. This would ultimately allow for the expression of hyperproperties in RTLola. Hyperproperties

as introduced by Clarkson and Schneider [9] allow for example the specification of information flow policies.
Future work may also be considered with developing a network focused syntax variant of RTLola such that specifying network properties becomes easier and less verbose. This may include the introduction of new keywords that help to specify that a template stream instance should only be extended iff its parameter values match the values of given expressions. Furthermore, the handling of binary data like the payload of a packet can be improved by introducing convenient methods to match this data against other binary data. All this would contribute to making RTLola even more suitable for network monitoring applications.

Bibliography

[1] Suricata open source ids. `https://suricata-ids.org`. Accessed: 2019-06-24.

[2] Abdulbasit Ahmed, Alexei Lisitsa, and Clare Dixon. Testid: a high performance temporal intrusion detection system. *Proceedings of the ICIMP*, pages 20–26, 2013.

[3] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 68–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[4] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monpoly: Monitoring usage-control policies. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 360–364, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[5] David A Basin, Srdjan Krstic, and Dmitriy Traytel. Aerial: Almost event-rate independent algorithms for monitoring metric regular properties. In *RV-CuBES*, pages 29–36, 2017.

[6] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. FPGA stream-monitoring of real-time properties. In *ESWEEK-TECS special issue, International Conference on Embedded Software EMSOFT 2019, New York, USA, October 13 - 18*, 2019.

[7] Vincent Berk, Annarita Giani, George Cybenko, and N Hanover. Detection of covert channel encoding in network packet delays. *Rapport technique TR536, de lUniversité de Dartmouth*, 19, 2005.

[8] Monowar H Bhuyan, DK Bhattacharyya, and Jugal K Kalita. An empirical evaluation of information metrics for low-rate and high-rate ddos attack detection. *Pattern Recognition Letters*, 51:1–7, 2015.

[9] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[10] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE, 2005.

[11] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *International Conference on Runtime Verification*, pages 152–168. Springer, 2016.

[12] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: Stream-based monitoring of cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 421–431. Springer, 2019.

[13] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. *CoRR*, abs/1711.03829, 2017.

[14] Felipe Gorostiaga and César Sánchez. Striver: stream runtime verification for real-time event-streams. In *International Conference on Runtime Verification*, pages 282–298. Springer, 2018.

[15] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: runtime verification of non-synchronized real-time streams. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1925–1933. ACM, 2018.

[16] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[17] Richard Lippmann, Joshua W Haines, David J Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Computer networks*, 34(4):579–595, 2000.

[18] Matthew V. Mahoney and Philip K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 376–385, New York, NY, USA, 2002. ACM.

[19] Prasad Naldurg, Koushik Sen, and Prasanna Thati. A temporal logic based framework for intrusion detection. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 359–376. Springer, 2004.

[20] Julien Olivain and Jean Goubault-Larrecq. The orchids intrusion detection tool. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer*

Aided Verification, pages 286–290, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[21] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23):2435 – 2463, 1999.

[22] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[23] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model-checking. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 220–, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. R2u2: Monitoring and diagnosis of security threats for unmanned aerial systems. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 233–249, Cham, 2015. Springer International Publishing.

[25] Maximilian Schwenger. Let's not trust experience blindly: Formal monitoring of humans and other cps. Master's thesis, Saarland University, 09 2019.

[26] Giovanni Vigna and Richard A Kemmerer. Netstat: A network-based intrusion detection approach. In *Proceedings 14th Annual Computer Security Applications Conference (Cat. No. 98EX217)*, pages 25–34. IEEE, 1998.

Appendices

Appendix A

Streamlab IDS Usage

To use streamlab as an intrusion detection system the subcommand *ids* is used. It requires the input of lola specification file, a description the local network space and either an pcap file for offline monitoring or a network interface to listen on. For more information we refer the reader to the help page.

Network Input Stream Structure

Input streams are declared as usual in the specification file. Their names are composed as follows:

```
<Protocol>::<Header Field>
```

For example:

```
input Ethernet::source: (UInt8, UInt8, UInt8, UInt8, UInt8, UInt8)
```

will provide the input stream with the current source MAC addresses. Header flags can be accessed as follows:

```
<Protocol>::flags::<Flag name>
```

For example:

```
input IPv4::flags::df: Bool
```

Available Header Fields

Ethernet

```
destination: (UInt8, UInt8, UInt8, UInt8, UInt8, UInt8)
source: (UInt8, UInt8, UInt8, UInt8, UInt8, UInt8)
etype: UInt16
```

IPv4

```
source: (UInt8, UInt8, UInt8, UInt8)
destination: (UInt8, UInt8, UInt8, UInt8)
```

```
ihl: UInt8
dscp: UInt8
ecn: UInt8
length: UInt16
identification: UInt16
fragment_offset: UInt16
ttl: UInt8
protocol: UInt8
checksum: UInt16
```

IPv4 flags:

```
df: Bool
mf: Bool
```

IPv6

```
source: (UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8,
    UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8)
destination: (UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8,
          UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8, UInt8)
traffic_class: UInt8
flow_label: UInt32
length: UInt16
hop_limit: UInt8
```

TCP

```
source: UInt16
destination: UInt16
seq_number: UInt32
ack_number: UInt32
data_offset: UInt8
window_size: UInt16
checksum: UInt16
urgent_pointer: UInt16
```

TCP flags

```
ns: bool
cwr: bool
ece: bool
urg: bool
ack: bool
psh: bool
rst: bool
syn: bool
fin: bool
```

UDP

```
source: UInt16
destination: UInt16
length: UInt16
checksum: UInt16
```

Auxiliary Input Streams

```
payload: String
```

The payload stream contains the payload of each packet as an UTF-8 encoded string.

```
direction: String
```

The direction stream can either take the value *Incoming* or *Outgoing*, depending on whether the destination IP of a packet belongs to the given local network.

```
protocol: String
```

The protocol stream contains the name of the highest level recognized protocol. The possible values are:

- TCP
- UDP
- IPv4
- IPv6
- Ethernet2
- Unknown