A Stream-based Specification Language for Network Monitoring*

Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah

Saarland University

Abstract. We introduce Lola 2.0, a stream-based specification language for the precise description of complex security properties in network traffic. The language extends the specification language Lola with two new features: template stream expressions, which allow input data to be carried along the stream, and dynamic stream generation, where new monitors can be invoked during the monitoring process for the monitoring of new subtasks on their own time scale. Lola 2.0 is simple and expressive: it combines the ease-of-use of rule-based specification languages like Snort with the expressiveness of heavy-weight scripting languages or temporal logics previously needed for the description of complex stateful dependencies and statistical measures. Lola 2.0 specifications are monitored by incrementally constructing output streams from input streams, while maintaining a store of partially evaluated expressions. We demonstrate the flexibility and expressivity of Lola 2.0 using a prototype implementation on several practical examples.

Keywords: Runtime verification, Monitoring, Network intrusion detection

1 Introduction

Automatic support for the monitoring of network traffic has become essential in order to cope with the massive exchange of data over high-speed networks and the constantly rising number of attacks. With the help of *network intrusion detection systems* (NIDS), system administrators check the network against predefined malicious patterns and identify previously unknown attack patterns based on irregularities observed in the network traffic. For instance, to check whether a server is subject to a denial of service attack, one observes whether a large number of connections are established to the server in a short period of time from external IP addresses.

Traditionally, monitoring tasks in telecommunication networks have been specified in powerful scripting languages, such as the N-Code language in the Network Flight Recorder (NFR) [14]. Intrusion detection systems implemented in such languages extract from the network traffic a complex combination of temporal patterns and statistical measures that distinguish intrusions from normal

* This work was partially supported by the German Research Foundation (DFG) in the Collaborative Research Center 1223 and by the Deutsche Telekom Foundation.

network traffic. Such heavy-weight solutions are, however, expensive to develop and maintain, since specification and monitoring algorithm are typically not separated and dependencies on future behavior have to be explicitly encoded.

Descriptive specification languages allow us to naturally express future protocol behavior in concise and readable specifications. One such language is the stream-based specification language Lola [7], which describes complex temporal patterns with references into the past and the future in a simple way and can both monitor correctness properties and compute statistical measures. Lola specifications resemble programs in a synchronous programming language like Lustre [12], Esterel [5], or Signal [10], but may include formulas that refer to future values of streams. For network monitoring tasks, however, specifying properties for individual connections of a network stream is cumbersome, because every possible connection would need to be defined in a separate stream.

The contribution of the paper is to introduce new language features into the Lola language that allow us to run each stream on an *individual slice of the incoming data* and on an *individual time scale*. In this way, inexpensive patterns can be used as *filters* that produce streams that run at slower speeds with less data, and can therefore be analyzed against more expensive patterns.

To illustrate the need for the new language features of Lola, consider the classic Lola specification

```
input bool loginSuccess
output int attempts := ite(loginSuccess, 0, attempts[+1,0] + 1)
```

which computes, from a given position in the stream, the number of overall failed login attempts *until* that future point of time where either the login attempt succeeds or the stream ends. Now, to distinguish the login attempts of individual users, Lola 2.0 extends the streams of Lola to parameterized *stream templates*. The instantiation of a template, as well as the speed in which an instance runs, is determined dynamically by auxiliary *invocation* and *extension* streams. If, for example, we wish to count the number of failed login attempts *per user*, we might introduce a stream template

```
input bool loginSuccess
input String uid
output int attempts<user> : inv: uid; ext: useraction :=
                            ite(loginSuccess,0,
                            attempts(user)[+1,0]+1)
```

where `uid` and `useraction` are auxiliary streams: the `uid` stream contains the id of the user who is currently logging in, and causes an invocation of the instance of the `attempts` stream corresponding to that user if that instance does not exist already, i.e., during the *first* login attempt of that user; the `useraction` stream extends the `attempts` stream whenever that user attempts *another* login:

```
output bool useraction<u> := (uid=u)
```

As a result, the `attempts(u)` stream of a particular user `u` consists of those positions, and only those positions, where the user `u` tries to log on. The condition

under which the monitor should raise an alarm is indicated in Lola with the keyword `trigger`. In Lola 2.0, the trigger condition might involve an aggregation over the instances of a stream template, as in the following example:

```
output bool bruteforce<user> : inv: uid; ext: useraction :=
                               attempts(user)>3
trigger any(bruteforce)
```

The alarm is triggered if there exists a user who attempts more than three failed login attempts. Note that the only expensive part of the monitoring happens in the `attempts(user)` streams, which each run very slowly, at the pace of an individual user, and deal with very little data. While many instances of the `attempts` template might be active at the same time, this does not constitute a performance bottleneck as the instances could easily be distributed over several parallel machines.

Outline In Section 2, we relate Lola 2.0 to other specification mechanisms for network monitoring. In Section 3 we discuss the syntax of Lola 2.0. In Section 4 we provide some illustrative examples of the application of Lola 2.0 to network intrusion detection. In Section 5 we turn to describing the semantics of Lola 2.0 in more detail. In Section 6, we present a monitoring algorithm for Lola 2.0 specifications. We report on experimental results in Section 7 and conclude in Section 8.

2 Related Work

Approaches to network intrusion detection are broadly classified into *signature-based* [19, 1, 16, 17, 11] and *anomaly-based* [8, 15] approaches. Signature-based approaches monitor for known patterns of attacks, while anomaly-based approaches detect deviations from the usual behavior. Typically, signature-based approaches run the risk of missing attacks that do not follow a known pattern, anomaly-based approaches can recognize previously unseen attacks, but often produce false alarms. While our approach belongs to the category of signature-based approaches, it can, to some extent, emulate an anomaly-based approach by computing certain statistics and raising an alarm if the values fall out of the normal range.

Within the signature-based approaches, a wide range of specification languages has been proposed that differ significantly in expressiveness and ease-of-use. One of the most common NIDS is the Snort system [19]. Specifications for Snort are based on a simple rule-based model language that describes per packet tests and actions. Snort rules can define statistical anomaly patterns over packets and collect traffic based on data contained in the payload of the packets. Suricata[1], is a more recent implementation using the same rule-based input language as Snort. The focus on individual packets, rather than the relation between multiple packets, is the key weakness of light-weight specification

[1] https://suricata-ids.org

approaches like Snort and Suricata rules. On the other end of the expressivity spectrum, systems like Bro [18], which use an event-based scripting language as a specification mechanism, fall into the category of heavy-weight specification mechanisms, which have the full power of a programming language.

As first pointed out by Roger and Goubault-Larrecq [20], the temporal patterns in the relations of multiple packets can naturally be expressed in a temporal logic. Approaches to intrusion detection based on temporal logic include the ORCHIDS [17], TeStID [2], and MONID [16] tools. ORCHIDS uses a specialized temporal logic tailored towards eventuality properties and employs an expressive underlying rule-based language as well as the capability to spawn monitors for individual instances of monitoring tasks. In comparison to Lola 2.0, non-determinism in the specification has to be handled explicitly. TeStID uses Many Sorted First Order Metric Temporal Logic (MSFOMTL). MONID uses the temporal logic EAGLE, which is based on parameterized recursive equations. A simpler, and often more efficient version of EAGLE is the rule-based specification language RuleR [4].

Our approach is based on the stream-based specification language Lola [7]. The definition of Lola output streams in terms of other streams resembles synchronous programming languages (notably Lustre [12], Esterel [5], and Signal [10]). Unlike these languages, Lola is not, however, an *executable* programming language, but a *descriptive* specification language. Lola subsumes many other specification languages, such as the temporal logics, and has been shown to provide natural encodings for both the temporal and the statistical measures needed to monitor industrial hardware designs. More theoretical work on Lola concerns the complexity, expressiveness, succinctness, and closure properties of Boolean streams [6]. The new version of Lola presented in this paper extends the original language with the concepts of *parameterization* and *multiple temporal time scales*.

Parameterization is a common concept in specification languages for runtime verification. In parametric temporal logic [9], parameters refer to quantitative measures, such as the number of steps until an eventuality is fulfilled. While this type of parametric specification can also be encoded in Lola, the purpose of the parameterization in Lola 2.0 is to run individual streams on small slices of the incoming data stream. This type of parameterization is similar to the parameterization in QEA (Quantified Event Automata) [3], an approach based on state machines, where a given trace is sliced into separate projections for different parameter values. Both types of parameterization appear in rule-based specification languages like LogFire [13], where a set of facts $F(v_1, \ldots, v_n)$ for some name F and parameter values $v_1, \ldots, v_n$ is maintained. A generic approach to add parameterization to an existing specification language was presented by Rosu and Chen [21]. The parameterization in Lola 2.0 extends these approaches with the dynamic creation and termination of streams and the aggregation of statistics over the instances of a stream template.

3 Stream-based Specifications

We introduce the syntax of stream-based Lola specifications in two steps. We begin with "standard" Lola, as introduced in [7], where specifications are given by equations over stream variables. In the second step, we introduce Lola 2.0, by generalizing such stream equations to stream equation templates.

Lola 1.0 A *Lola specification* is a system of equations of stream expressions over typed *stream variables* of the following form:

$$\begin{aligned}
&\textbf{input } T_1\ t_1\\
&\qquad\vdots\\
&\textbf{input } T_m\ t_m\\
&\textbf{output } T_{m+1}\ s_1 := e_1(t_1,\ldots,t_m,s_1,\ldots s_n)\\
&\qquad\vdots\\
&\textbf{output } T_{m+n}\ s_n := e_n(t_1,\ldots,t_m,s_1,\ldots s_n)
\end{aligned}$$

Each stream expression $e_i(t_1,\ldots,t_m,s_1,\ldots s_n)$, for $1 \leq i \leq n$ is defined over a set of *independent* stream variables $t_1,\ldots,t_m$ and *dependent* stream variables $s_1,\ldots,s_n$. Independent stream variables refer to input stream values, and dependent stream variables refer to output stream values computed over the values of all streams. All stream variables are typed: the type of an independent stream variable t_i is T_i, the type of an dependent stream variable s_i is T_{m+i}.

A *stream expression* $e(t_1,\ldots,t_m,s_1,\ldots s_n)$ is defined recursively as follows:

- Let c be a constant of type T and let s_i for $1 \leq i \leq n$ be a stream variable of type T', then both $e = c$ and $e = s_i$ are atomic stream expressions of type T and T' respectively.
- Let $f : T_1 \times T_2 \times \ldots T_k \to T$ be a k-ary function, then for stream expressions $e_1,\ldots e_k$ of type $T_1,\ldots,T_k$, the expression $e = f(e_1,\ldots,e_k)$ is a stream expression of type T.
- Let b be a boolean stream expression and e_1, e_2 stream expressions of type T, then $e = \textbf{ite}(b, e_1, e_2)$ is a stream expression of type T. The expression evaluates to e_1 when b is *true* and to e_2 when b is *false*.
- Let e' be a stream expression of type T, d a constant of type T, and i an integer, then $e = e'[i, d]$ is a stream expression of type T. The stream expression $e'[i, d]$ refers to the value of expression e' offset i positions from the current position. If such a position is not defined, then the value of the stream is the default value d.

In addition to the stream equations, Lola specifications often contain a list of *triggers*

$$\textbf{trigger } \varphi_1, \varphi_2, \ldots, \varphi_k$$

where $\varphi_1, \varphi_2, \ldots, \varphi_k$ are expressions of type boolean over the stream variables. Triggers generate notifications when their value becomes *true*.

Lola 2.0 Lola 2.0 extends Lola with *stream equation templates* of the following form:

$$\begin{aligned}\textbf{output } T\ s\langle p_1 : T_1, \ldots, p_l : T_l\rangle : \ & \textbf{inv: } s_{inv};\\ & \textbf{ext: } s_{ext};\\ & \textbf{ter: } s_{ter} :=\\ & e(t_1, \ldots, t_m, s_1, \ldots s_n, p_1, \ldots, p_l)\end{aligned}$$

Each such stream equation template introduces a *template variable* s of type T that depends on *parameters* $p_1, \ldots, p_l$ of types $T_{p_1}, \ldots T_{p_l}$, respectively. For given values $v_1, \ldots, v_l$ of matching types $T_{p_1}, \ldots T_{p_l}$ we call

$$s\langle v_1, \ldots, v_l\rangle = e(t_1, \ldots, t_m, s_1, \ldots s_n, p_1, \ldots, p_l)[p_1/v_1, \ldots p_l/v_l]$$

an *instance* of s. The template variables s_{inv}, s_{ext}, and s_{ter} indicate the following *auxiliary streams*:

- s_{inv} is the *invocation* template stream variable of s and has type $T_{p_1} \times \cdots \times T_{p_l}$. If some instance of s_{inv} has value $(v_1, \ldots v_l)$, then an instance $s\langle v_1, \ldots, v_l\rangle$ of s is invoked.
- s_{ext} is the *extension* template stream variable of s and has type *bool* and parameter of type $T_{p_1}, \ldots T_{p_l}$. If s is invoked with parameter values $\alpha = (v_1, ..., v_l)$, then an extension stream s^{α}_{ext} is invoked with the same parameter values. If s^{α}_{ext} is *true*, then the value of the output stream $s\langle v_1, \ldots, v_l\rangle$ is computed at the position.
- s_{ter} is the *termination* template stream variable of s and has type *bool* and parameters of type $T_{p_1}, \ldots T_{p_l}$. If s is invoked with parameter values $\alpha = (v_1, ..., v_l)$, then a terminate stream s^{α}_{ter} is invoked with the same parameter values. If s^{α}_{ter} is *true*, then the output stream $s\langle v_1, \ldots, v_l\rangle$ is terminated and not extended until it is invoked again.

A *template stream expression* $e(t_1, \ldots, t_m, s_1, \ldots s_n, p_1, \ldots, p_l)$ is defined like a stream expression in Lola 1.0, with the following additions:

- Let p_i for $i \in \{1, \ldots l\}$ be a parameter. Then p_i is a template stream expression of type T_i.
- Let s be a template variable, and Op be an *aggregation operator* of type T. For example, `any` is an aggregation operator of type `bool`, `count` is an aggregation operator of type `int`. Then $Op(s)$ is a template stream expression of type T.

If a stream equation template has no parameters, we omit the empty parameter tuple $\langle\rangle$. We also permit that any of the auxiliary streams may be omitted, in which case the invocation stream is set to the default stream σ_0, which is the constant stream that produces the empty tuple () in every position; the extension template stream is set to the constant stream that produces *true* in every position, and the termination template stream is set to the constant stream that produces *false* in every position. Note that in this way, Lola 1.0 stream equations are special cases of Lola 2.0 stream equation templates. The same also holds for independent stream variables. If omitted from the declaration, the invocation, extension and termination streams are set to the default values.

4 Example Specifications

In this section we show how we can employ Lola 2.0 to define properties over network traffic. Consider the Lola 2.0 specification given in Figure 1. The specification defines a pattern for detecting a web application fingerprinting attack. In such an attack a hostile client sends arbitrary HTTP requests and awaits the responses from the server, which contain a HTTP response header with information about the server software vendor, its version, and more. Such information allows the client to determine known vulnerabilities according to the type of the server. The attacker mostly requests access to random URLs, which may lead in many cases to an HTTP response declaring either a bad HTTP request or a page not found message. One way to observe such an attack is to observe server responses containing either "Bad HTTP request" or "Page not found" messages and then check whether the IP address, which initiated the request, continues sending random requests to the server.

In the specification, the stream `webApplicationFingerprinting` is invoked for a pair of source and destination addresses every time the invocation stream `badHttpRequestInvoke` is extended with a new pair of addresses. Such a pair is recorded whenever a bad request or no page found response is sent out, as defined by the extension stream of `badHttpRequestInvoke`[2].

Once an instance of `webApplicationFingerprinting` is invoked it tracks the number of bad requests, using the extension stream `badHttpRequestExtend` which is invoked simultaneously with the same pair of addresses. If at some point the status code OK was returned then the instance is terminated via the termination stream `webApplicationFingerprintingTerminate`. This allows the monitoring process to discard many instances of the template that otherwise would cause many false positive alerts. If an instance is not terminated and its value exceeds a certain threshold, then the monitoring algorithm alerts about a potential web application fingerprinting threat. The latter is defined by the keywords `trigger` and `any`.

We consider another example involving denial of service attacks (DoS). One way of checking whether a server is subject to a DoS attack, is to observe whether a large number of connections are established to the server in a short period of time from external IP addresses. Consider a client that is trying to perform a DoS attack via a TCP-SYN scan. The hostile client sends a SYN request to the server to initiate a three-way handshake, upon which the server responds with a SYN/ACK packet including the port number it was sent from. The malicious client then sends no ACK packet to acknowledge the reception of the SYN/ACK package, or might even request a reset of the communication, which leaves the port and connection data structure open and thus leads to eventual resource exhaustion. One way to monitor such an attack is to check whether a large number of uncompleted handshakes are observed in the traffic.

[2] In Figure 1 the extension stream of `badHttpRequestInvoke` is defined explicitly in the output stream. This could also have been defined separately by a declaration of another boolean output stream with the same condition.

```
input string Protocol, RequestMethod, ResponsePhrase, Source, Destination

output (string, string) badHttpRequestInvoke;
ext: Protocol="HTTP" & (ResponsePhrase="Bad Request" | "Not Found")
:= (Source, Destination)

output bool badHttpRequestExtend<src, dst>:
inv: badHttpRequestInvoke;
:= src=Source & dst=Destination &
ResponsePhrase = "Bad Request" | "Not Found"

output bool webApplicationFingerprintingTerminate<src,dst>:
inv: badHttpRequestInvoke;
:= src=Source & dst=Destination & ResponsePhrase = "OK"

output int webApplicationFingerprinting<src, dst>:
inv: badHttpRequestInvoke;
ext: badHttpRequestExtend;
ter: webApplicationFingerprintingTerminate
:= webApplicationFingerprinting(src, dst)[-1,0]+1

trigger any(webApplicationFingerprinting > threshold)
```

Fig. 1. A Lola 2.0 specification for a web application fingerprinting pattern

Figure 2 shows a specification for checking whether the number of open TCP requests exceeds a given threshold using the stream template `tcpSynScan`. Whenever there is a TCP request from a client to the server, the monitor waits for an acknowledgment from the client. This is determined by the specification `waitForAck` which is invoked by the stream `incompleteHandshakeInvoke` for a pair of addresses. At the same time the stream `incompleteHandshakeInvoke` also invokes an instance of the template `tcpSynInvoke`. If a certain time passes without seeing an acknowledgement, then the instance is extended by the pair of source and destination addresses and an instance of `tcpSynScan` is invoked to monitor a potential TCP SYN scan attack for this pair of IP addresses. From this position on the monitor keeps track of how many TCP requests are received from an IP address or whether Syn requests keep being sent from one address without acknowledgements. When one of the thresholds `threshold2` and `threshold3` is exceeded, the monitor triggers an alert. This is achieved using the keywords `trigger`, `any` and `count`.

5 Lola 2.0 Semantics

We now give a formal definition of the Lola 2.0 semantics. Let Φ be a specification with independent stream variables $t_1, \ldots, t_m$ of type $T_1, \ldots T_m$, respectively, and template stream variables $s_1, \ldots, s_n$ of types $T_{m+1}, \ldots T_{m+n}$, respectively.

```
input string Protocol, Syn, Ack, Source, Destination

output (string,string) incompleteHandshakeInvoke:
ext: Protocol= "TCP" & Syn="Set" & Ack="Not Set";
:= (Source,Destination)

output bool incompleteHandshakeTerminate<src, dst>:
inv: incompleteHandshakeInvoke;
=Source=src & Destination=dst & Syn="Not Set" & Ack="Set"

output int waitForAck<src,dst>:
inv: incompleteHandshakeInvoke;
ter: incompleteHandshakeTerminate
= waitForAck(src, dst)[-1,0]+1

output (string,string) tcpSynInvoke<src, dst>:
inv: incompleteHandshakeInvoke;
ext: waitForAck(src,dst)[0,0] > threshold
ter: waitForAck(src,dst)[0,0] > threshold
= (src,dst)

output bool tcpSynExtend<src,dst>:
inv:tcpSynInvoke;
= src = Source & dst = Destination & Syn = "Set"

output bool tcpSynTerminate<src,dst>:
inv:tcpSynInvoke;
= src = Source & dst = Destination & Syn = "Not Set" & Ack="Set"

output int tcpSynScan<src,dst>:
inv:tcpSynInvoke;
ext:tcpSynExtend;
ter:tcpSynTerminate;
=tcpSynScan(src,dst)[-1,0] +1

trigger count(tcpSynScan) > threshold2
trigger any(tcpSynScan > threshold3)
```

Fig. 2. A specification of a TCP SYN scan pattern

We fix a natural number $N \geq 0$ as the length of the traces. An *evaluation model* of Φ is a set Γ of streams of length N, where each stream has type $T_{m+i} \cup \{\#\}$ for $1 \leq i \leq n$. The symbol # is added to the types to indicate that the stream does not exist yet at a particular position, for example if the stream has not been invoked yet. In the following, we use s_i^α to refer to the instance of a template variable s_i with parameter values α, and σ_i^α to refer to a corresponding stream in Γ.

We now pose several conditions that evaluation models must satisfy. Intuitively, the conditions concern the two mutually dependent requirements that (1) the evaluation model is populated with a sufficiently large set of streams, and that (2) each stream actually produces the right values. To guarantee requirement (1), we describe the elements of Γ inductively as follows:

- $\sigma_0 \in \Gamma$, where σ_0 is the constant stream that produces the empty tuple () in every position.
- For each template stream variable s_i, we consider the associated invocation stream variable s_{inv}. If Γ contains some stream σ^{α}_{inv} for some parameter values $\alpha \in T^{inv}_1 \times \cdots \times T^{inv}_{l_i}$, then Γ must also contain a stream for every instance of s_i invoked by σ^{α}_{inv} at some position; i.e., for all $j < N$ where $\sigma^{\alpha}_{inv}(j) \neq \#$, there must exist some stream $\sigma^{\beta}_i \in \Gamma$ for the instance of s_i given by the parameter values $\beta = \sigma^{\alpha}_{inv}(j)$.

To guarantee condition (2), that each stream actually produces the right values, we first characterize the positions in which the stream exists.

Let $\mathit{alive}(s_i, (v_1, \ldots, v_{l_i}), j)$ be *true* for some stream position j if the stream was actually invoked, i.e., there is a stream $\sigma^{\beta}_{inv} \in \Gamma$ for some instance of the associated invocation stream variable s_{inv} (with arbitrary parameter values β) and an earlier stream position $j' < j$ such that $\sigma^{\beta}_{inv}(j') = (v_1, \ldots, v_l)$, and the stream was not terminated in the meantime, i.e., for $j' < j'' \leq j$ we have $\sigma^{\beta'}_{ter}(j'') = \mathit{false}$ for all instances of the termination stream variable with $\beta' = (v_1, \ldots, v_l)$.

If a stream exists in some position, we determine its value by evaluating the corresponding stream expression. For each stream $\sigma^{\alpha}_i \in \Gamma$ for the instance $\alpha = (v_1, \ldots, v_{l_i})$ of some stream template variable s_i,

$$\sigma^{\alpha}_i(j) = \begin{cases} \mathit{val}(e_i[p_1/v_1, \ldots, p_{l_i}/v_{l_i}], j) & \text{if } \mathit{alive}(s_i, (v_1, \ldots, v_{l_i}), j) \\ \# & \text{otherwise} \end{cases}$$

where the evaluation function $\mathit{val}(e_i[p_1/v_1, \ldots, p_{l_i}/v_{l_i}], j)$ is defined as follows:

- if $\sigma^{\alpha}_{ext}(j) = \mathit{true}$, where σ^{α}_{ext} is the extension stream of σ^{α}_i, then $\mathit{val}(e, j)$ is defined as follows:
  - $\mathit{val}(c)(j) = c$
  - $\mathit{val}(t_h)(j) = \tau_h(j) \quad$ for $1 \leq h \leq m$
  - $\mathit{val}(f(e_1, \ldots, e_h))(j) = f(\mathit{val}(e_1)(j), \ldots, \mathit{val}(e_h)(j))$
  - $\mathit{val}(\mathbf{ite}(b, e_1, e_2))(j) = \text{if } \mathit{val}(b)(j) \text{ then } \mathit{val}(e_1)(j) \text{ else } \mathit{val}(e_2)(j)$
  - $\mathit{val}(s^{\beta}_h[0, d])(j) = \begin{cases} \sigma^{\beta}_h(j) & \mathit{alive}(s_h, \beta, j) \\ d & \text{otherwise} \end{cases}$
  - $\mathit{val}(s^{\beta}_h[k, d])(j) = \begin{cases} d & \text{if } j \geq N \text{ or } j < 0 \\ \mathit{val}(e[k-1, d])(j+1) & \text{if } k > 0, \sigma^{\beta}_{ext}(j) = \mathit{true} \\ \mathit{val}(e[k+1, d])(j-1) & \text{if } k < 0, \sigma^{\beta}_{ext}(j) = \mathit{true} \\ \mathit{val}(e[k, d])(j+1) & \text{if } k > 0 \\ \mathit{val}(e[k, d])(j-1) & \text{otherwise} \end{cases}$

– otherwise $val(e_i[p_1/v_1, \ldots, p_{l_i}/v_{l_i}], j) = \#$

Intuitively, the extend stream defines a local clock for every template variable. Unlike in Lola, where all streams follow the same one clock, streams in Lola 2.0 follow several clocks depending on their invocation time and the extension pace. The invoke stream starts new instances of the template output stream whenever it evaluates to a fresh parameter instantiation. The extend stream is evaluated for all instances which are active on a current stream. Whenever it evaluates to *true*, the template output stream instance advances on its timeline. A template output stream instance is terminated whenever its terminate stream evaluates to true for its parameter instantiation. The clocks can be inductively defined on top of the clock of stream σ_0, which we call the base clock.

Well-defined specifications. We say a specification is *well-defined*, if for any set of appropriately typed input streams of length N for the independent stream variables, it has a unique evaluation model. In general, specifications need not be well-defined, for example through self-references without offsets in stream expressions or circular offsets via multiple stream variables, which lead to the non-existence of evaluation models or lead to infinitely many evaluation models for a given set of input streams.

Since well-definedness is a semantic condition and expensive to check, we give a syntactic criterion, called *well-formedness*, which implies well-definedness and can be checked by a simple check on the dependency graph. For a specification Φ, its associated *dependency graph* is a weighted and directed multi-graph $G = \langle V, E \rangle$ with $V = \{s_1, \ldots, s_n, t_1, \ldots, t_m\}$. We add an edge $e \in E$ where $e = \langle s_i, s_k, w \rangle$ from s_i to s_k with weight w iff the stream expression of s_i contains the subexpression $s_k[w, d]$ for some default value d. Edges leading to t_k are added analogously. Thus, the edges represent the fact that expression s_i depends on s_k at (positive or negative) offset w. Since each stream may be used more than once with different offsets in an expression, the graph may contain multiple edges between vertices. A *cycle* in the graph is a sequence $v_1 \xrightarrow{e_1,w_1} v_2 \ldots v_k \xrightarrow{e_k,w_k} v_{k+1}$ such that all $e_i = \langle v_i, v_{i+1}, w_i \rangle \in E$, and $v_1 = v_{k+1}$. The total weight of the cycle is the sum of all weights w_i along the cycle. A specification is *well-formed*, iff it does not contain a zero-weight cycle. Well-formed specifications are guaranteed to be well-defined.

6 The Monitoring Algorithm

We now describe a monitoring algorithm for the evaluation of a given Lola 2.0 specification on a set of input streams for the independent stream variables. The streams become available *online*, i.e., one position at a time. The length of the streams is *a-priori* unknown and the full streams may be too large to store in memory.

The central data structure of the algorithm is the *equation store*, which consists of the following parts: A store S, in which we keep a set of the currently

active instances of template stream variables; a store of *resolved* equations R, which are fully evaluated but may still be used by other streams, and a store of *unresolved* equations U, which are not yet fully evaluated.

For each position, we begin the evaluation by adding the input stream values at the current position to the store R. As we are adding resolved equations to R, we always check whether they start new invocations for any of the template stream expressions. Should this happen, we add these to the store S and add corresponding unresolved equations to the store U. We then continue by simplifying the equations in U by function applications, rewriting rules for conditionals and resolving stream access and offsets by the equations from store R. The invocation check and the simplification step are repeated until nothing new is added to R and no new streams are invoked. The number of repetitions depends on the structure and dependencies of the specification. Equations are removed from the store R whenever they are not needed anymore.

To simplify the presentation of the algorithm, we assume that all extension and termination streams are locally determined, i.e., their value at every position can be calculated just from the values of the input streams at the same or earlier positions. Let, for each independent stream variable t_i, the corresponding input stream be denoted by τ_i. Starting at position $j = 0$ with the empty equation stores, the algorithm performs the following steps for each position:

1. For each input stream t_i, add $\tau_i(j) = c$ to store R.
2. Add $\sigma_0(j) = ()$ to R.
3. Initialize the set of active stream valuations: For all template streams s_i, and valuations α such that $\alpha \in S(s_i, j-1)$, if $\sigma^{\alpha}_{\text{ext}} = \text{true}$ and $\sigma^{\alpha}_{\text{ter}} = \text{false}$ then $\alpha \in S(s_i, j)$.

Then repeat the following steps until a fixpoint is reached:

1. Simplify all equations in U, if any expression is now constant, add to R.
2. Check for new invocations, extensions and terminations by the additions to R.
3. If for some stream template s_i, and any position k, $\sigma^{\alpha}_{inv}(k) = \beta$ is added to R, then $S(s_i, k) = S(s_i, k) \cup \beta$ and we add $\sigma^{\beta}_{i}(k) = e$ to U.
4. If for some stream template s_i, and any position k, $\sigma^{\alpha}_{ext}(k) = \mathit{true}$ is added to R, we add $\sigma^{\alpha}_{i}(k) = e$ to U.

The equations in U are simplified according to the following rules:

- Function application: e.g. $0 + x \to x$, ...
- Rewriting for conditionals: $\mathbf{ite}(\mathit{true}, e_1, e_2) \to e_1$, $\mathbf{ite}(\mathit{false}, e_1, e_2) \to e_2$.
- Resolve stream access: If $\sigma_{i,\alpha}(j) = c$ in R, replace every occurence of $\sigma_{i,\alpha}(j)$ by c in U.
- Resolve stream offsets: If some $\sigma_{i,\alpha}(j) = e_i$ in U contains a subexpression $\sigma_{i,\alpha}(j)[k, d]$, $\sigma^{\alpha}_{ext}(j) = c$ is in R for $c \in \{\mathit{true}, \mathit{false}\}$, and $\sigma^{\alpha}_{ter}(j) = \mathit{false}$ then

$$\sigma_{i,\alpha}(j)[k,d] \to \begin{cases} \sigma_{i,\alpha}(j) & \text{if } k = 0, \sigma^{\alpha}_{ext}(j) = \mathit{true} \\ \sigma_{i,\alpha}(j+1)[k-1,d] & \text{if } k > 0, \sigma^{\alpha}_{ext}(j) = \mathit{true} \\ \sigma_{i,\alpha}(j-1)[k+1,d] & \text{if } k < 0, \sigma^{\alpha}_{ext}(j) = \mathit{true} \\ \sigma_{i,\alpha}(j+1)[k,d] & \text{if } k > 0 \\ \sigma_{i,\alpha}(j-1)[k,d] & \text{if } k \geq 0, j > 0 \\ d & \text{otherwise} \end{cases}$$

– Resolve nonliving stream offsets: If some $\sigma_{i,\alpha}(j) = e_i$ in U contains a subexpression $\sigma_{i,\alpha}(j)[k,d]$, and $\alpha \notin S(s_i, j)$, then $\sigma_{i,\alpha}(j)[k,d] \to d$.

During the monitoring, we use a garbage collection process to remove entries from store R that are no longer needed. For each template stream expression s_i, we initially calculate the cutoff vector, which determines when a resolved stream expression can be cleared from store R. The vector records the usage of the stream expression within the definition of other streams and the maximal offset value. The vector contains one entry for every other stream, with default value 0. If there exits a reverse path in the dependency graph from stream s_k to stream s_i, we use the path with the smallest negative weight occuring on the edge originating in s_k on the path as the value. This yields the longest time we keep a value for s_i in memory for any dependency of s_k.

In an extra *garbage collection* store GC, we keep track of the current vectors of stream extensions which need to occur before a value can be eliminated. Whenever a new stream is invoked, we initialize $GC(s_i, \alpha, j) = (c_1, \ldots, c_n)$ with the cutoff vector $(c_1, \ldots, c_n)$. Whenever a stream s_k is extended, we increment the corresponding component c_k of all vectors in GC. If a vector in GC for any α and any s_i reaches strictly positive values in all elements at position j, we can safely remove $\sigma_{i,\alpha}(j)$ from R.

Once the stream has terminated, we replace all open offset expressions beyond the end of the stream with the specified default value and compute the fixpoint once again.

Efficiently monitorable specifications A specification is called *efficiently monitorable* if its memory consumption is constant in the length of the input streams. In Lola 1.0, a specification is guaranteed to be efficiently monitorable, if the value of every stream depends, at every position, only on values of other streams up to a bounded number of steps into the future [7]. A corresponding result for Lola 2.0 does not hold, because we do not know how many streams are invoked during run-time. Thus, the memory needed for a Lola 2.0 specification therefore might grow with the length of generated traces. In practice, it is, however, often possible to bound the number of instances invoked during the monitoring process. This additional assumption in fact allows us to syntactically characterize a class of efficiently monitorable specifications. The restriction from Lola 1.0 that future dependencies are bounded is, however, not strong enough for Lola 2.0. The reason is that, even when a reference in a Lola 2.0 specification looks only a constant number of steps into the future, the actual occurrence of these future events might be delayed indefinitely by the extension stream. To

obtain an efficiently monitorable fragment for Lola 2.0, we must therefore forbid all future references. Arbitrary references into the past remain allowed.

7 Experimental Results

We have implemented the monitoring algorithm for the efficiently monitorable fragment of Lola 2.0 as a command-line tool in C. As an input, it takes pre-processed network capture files (using the tool Wireshark[3]), which contain only the relevant input data defined by the input streams in the specification, and a Lola specification and produces output streams and statistics according to the specification.

Our experiments use network capture files from the Malware Capture Facility Project[4]. The network capture files range from 0.9 million up to 2.4 million packets and capture the traffic in a time frame of 24 hours. All experiments were run on a single quad-core machine with an 3.6 GHz Intel Xeon processor with 32 GB RAM. The input stream files were stored on an internal SSD drive.

Table 1 shows the result of the monitoring tool on the specification in Figure 2. We computed the number of `count` triggers, whose task was to observe the number of open handshake communications that have been waiting for more than 500 packets for an acknowledgment. We also observed the `any` trigger which checked whether any TCP request was not acknowledged after 600 packets. We compare the results of our specification with a Snort specification that checks whether the number of TCP Syn requests exceeds a threshold of 100 requests per 60 seconds. The results show that the specification in Figure 2 never triggered, and therefore all Syn-requests were acknowledged eventually. In comparison, a large number of Snort alerts were issued on the trace files for the Snort specification. The reason for that is that Lola 2.0 allows an intermediate step using the templates `waitForAck` and `tcpSynScan`, where the monitor waits for the acknowledgment for a pair of IP-addresses before triggering. The high number of `waitForAck` invocations in comparison to the number of `tcpSynScan` invocations shows that Lola 2.0 allows to filter many TCP communications before starting the check for a possible TCP-Syn scan.

However, on the trace files used in the experiment of Table 1, Snort was able to return all the alerts in less than a minute. Since the following manual inspection of the Snort alerts is necessary to evaluate the potential TCP Syn Scan attack, this points to an interesting trade-off between the expressiveness of the specification mechanism and the time needed to analyze large trace files.

8 Conclusion

We have extended the stream-based specification language Lola with stream templates. Lola 2.0 is a descriptive language that subsumes many other specification languages and we showed how one can provide natural encodings for

[3] http://www.wireshark.org
[4] http://mcfp.weebly.com

Table 1. A comparison between our Lola 2.0 prototype and the rule-based language Snort for detecting a simple pattern of TCP SYN scans.

#Packets	Snort alerts	Invocation		Count Trigger	Any Trigger	Time
		Wait	Scan			(sec)
901710	613	53654	340	323	0	2550.31
1710372	472	95983	260	254	0	6279.87
1857752	1699	107721	280	274	0	6786.06
1954427	2428	115787	379	369	0	7160.27
2419006	2036	146748	869	835	0	10347.96

properties and attack patterns over network traffic. The extended language provides a bridge between more light-weight approaches and monitoring techniques based on expensive formalisms such as the temporal logics, combining both simplicity and expressiveness. During runtime, each template can be instantiated dynamically to obtain new streams. This allows each stream to run on an individual slice of the incoming data. In this way, Lola 2.0 can combine specifications that run on widely varying amount of data, and with widely varying speed. Inexpensive patterns can be used as filters that produce streams that run with less data, which can subsequently be analyzed against more expensive patterns.

Even though our prototype is an online monitoring tool, we have evaluated the tool on previously recorded pcap log data. In future work, we plan to deploy the monitor directly in the network. Since Lola specifications can easily be parallelized, such an implementation will likely consist of several connected nodes, placed at strategically chosen positions within the network. Further investigating the trade-off between the expressiveness and efficiency in descriptive, stream-based approaches for network monitoring remains an interesting topic for future work.

References

1. Ahmed, A., Lisitsa, A., Dixon, C.: A misuse-based network intrusion detection system using temporal logic and stream processing. In: Network and System Security (NSS), 2011 5th International Conference on. pp. 1–8 (Sept 2011)
2. Ahmed, A., Lisitsa, A., Dixon, C.: Testid: a high performance temporal intrusion detection system. Proceedings of the ICIMP pp. 20–26 (2013)
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Fm 2012: Formal methods: 18th international symposium, paris, france, august 27-31, 2012. proceedings. pp. 68–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruler. J. Log. Comput. 20(3), 675–706 (2010), http://dx.doi.org/10.1093/logcom/exn076
5. Berry, G.: Proof, language, and interaction: essays in honour of Robin Milner, chap. The foundations of Esterel, pp. 425–454. MIT Press (2000)

6. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 64–79. Springer (2014), http://dx.doi.org/10.1007/978-3-319-11164-3_6
7. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05). pp. 166–174. IEEE Computer Society Press (June 2005)
8. Debar, H., Becker, M., Siboni, D.: A neural network component for an intrusion detection system. In: Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on. pp. 240–250 (May 1992)
9. Faymonville, P., Finkbeiner, B., Peled, D.: Monitoring parametric temporal logic. In: McMillan, K.L., Rival, X. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 8318, pp. 357–375. Springer (2014)
10. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Proceedings Conference on Functional Programming Languages and Computer Architecture. pp. 257–277. Springer-Verlag (1987)
11. Goubault-Larrecq, J., Olivain, J.: A smell of orchids. In: Leucker, M. (ed.) Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers. Lecture Notes in Computer Science, vol. 5289, pp. 1–20. Springer (2008)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (September 1991), citeseer.ist.psu.edu/halbwachs91synchronous.html
13. Havelund, K.: Rule-based runtime verification revisited. International Journal on Software Tools for Technology Transfer 17(2), 143–170 (2015), http://dx.doi.org/10.1007/s10009-014-0309-2
14. Lee, W., Park, C.T., Stolfo, S.J.: Automated intrusion detection using NFR: methods and experiences. In: Proceedings of the Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, USA, April 9-12, 1999. pp. 63–72. USENIX (1999), http://www.usenix.org/publications/library/proceedings/detection99/lee.html
15. Lee, W., Stolfo, S.J., Mok, K.W.: A data mining framework for building intrusion detection models. In: Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on. pp. 120–132 (1999)
16. Naldurg, P., Sen, K., Thati, P.: Formal techniques for networked and distributed systems – forte 2004: 24th ifip wg 6.1 international conference, madrid spain, september 27-30, 2004. proceedings. pp. 359–376. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
17. Olivain, J., Goubault-Larrecq, J.: Computer aided verification: 17th international conference, cav 2005, edinburgh, scotland, uk, july 6-10, 2005. proceedings. pp. 286–290. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
18. Paxson, V.: Bro: A system for detecting network intruders in real-time. Comput. Netw. 31(23-24), 2435–2463 (Dec 1999), http://dx.doi.org/10.1016/S1389-1286(99)00112-7
19. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration. pp. 229–238. LISA '99, USENIX Association, Berkeley, CA, USA (1999), http://dl.acm.org/citation.cfm?id=1039834.1039864

20. Roger, M., Goubault-Larrecq, J.: Log auditing through model-checking. Computer Security Foundations Workshop, IEEE 0, 0220 (2001)
21. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Logical Methods in Computer Science 8(1) (2012), http://dx.doi.org/10.2168/LMCS-8(1:9)2012