embeddedworld2016
Exhibition&Conference
...it´s a smarter world

Practical Event Monitoring in the LogicGuard Framework*

Wolfgang Schreiner, David Cerna, Temur Kutsia
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at
David.Cerna@risc.jku.at
Temur.Kutsia@risc.jku.at

Michael Krieger, Bashar Ahmad
RISC Software GmbH
Hagenberg, Austria
Michael.Krieger@risc-software.at
Bashar.Ahmad@risc-software.at

Helmut Otto, Martin Rummerstorfer, Thomas Gössl
SecureGUARD GmbH
Linz, Austria
hotto@secureguard.at
mrummerstorfer@secureguard.at
tgoessl@secureguard.at

Abstract*—We describe further progress on the previously introduced LogicGuard specification language and execution framework. This framework generates from a high-level logic specification of a desired property of a stream of events an executable program that observes the stream in real time for violations of the property. While previous presentations were based on an early and incomplete prototype, we are now able to report on some practical applications of the operational framework in the context of network security. As a startup example, we present the "Rogue DHCP" scenario where a device illicitly poses as a DHCP server in order to feed newly connected devices with wrong connectivity information; the monitor detects this attack by looking for duplicate offers to the same DHCP client, of which one is from the attacker. Our main scenario is "Dynamic DNS (DDNS) Cache Poisoining" where an attacker poses as a DDNS client and feeds the DDNS server with wrong DNS update information; the monitor detects this attack by learning about the frequency of legitimate DDNS updates and reporting updates that occur significantly earlier than expected.

****Supported by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program by the project 846003 "LogicGuard II".***

***Keywords*—runtime monitoring; event streams; network security; predicate logic.**

I. Introduction

The work presented in this paper originates in the problem of monitoring network traffic in order to detect illegitimate or problematic actions, e.g., attacks of external intruders or more generally activities that deviate from a specified norm, e.g. (as presented in [8]), violations of safety properties in production processes. In a more abstract form, we wish to monitor (potentially infinite) streams of "events" for violations of specified properties. Rather than coding these monitors manually (which is tedious, error-prone, and results in programs that are difficult to maintain), the goal is to generate these monitors from high-level declarative formal specifications of the desired stream properties.

The LogicGuard framework [10] addresses this problem by a specification language that is based on the well-known foundations of predicate logic (more specifically a subset called monadic logic [4]) and set theory. Properties are expressed by quantified formulas whose variables denote positions in a stream; using nested quantification and an ordering predicate on stream positions, complex properties can be formulated. Furthermore, from a given stream of "low-level" events, virtual streams of "high-level" events may be defined by a notation analogous to set-builder notation; monitors may thus operate on much higher levels of abstraction than provided by the external streams.

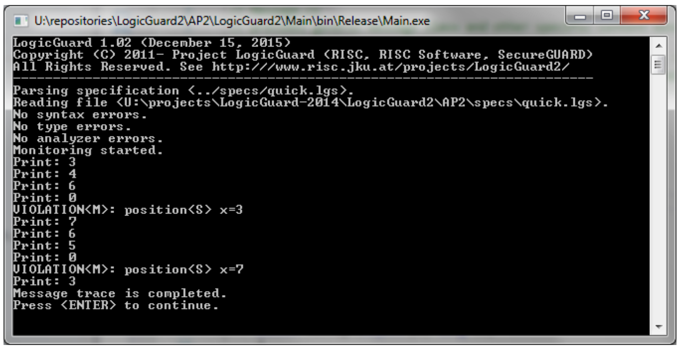
This work is embedded in the general context of *runtime verification* that aims to detect violations of formally specified system behaviors, not by analyzing the construction of the system (as is done in static verification), but by observing the execution of the system. Most approaches to runtime verification rely on specialized formalisms such as linear temporal logic, regular expressions, context-free grammars, or rule systems [1][2][3][5] which may be translated to automata such that it is comparatively easy to generate executable monitors. However, the expressiveness of these calculi is limited such that it may be hard or impossible to formulate the properties of interest.

On the contrary, the LogicGuard specification language is very rich which simplifies the specification of monitors but makes their efficient execution challenging. In particular, they may not be able to cope with a bounded amount of memory, i.e., they may require to preserve an unbounded amount of "stream history" or to keep track of an unbounded amount of "formula instances". To address the first problem, in [7] a static analysis is devised that only accepts specifications that can be executed with a finite amount of history and passes the corresponding information to the runtime system which may subsequently "prune" the history buffer to that amount. To address the second problem, work has started on a static analysis to determine an upper bound for the number of formula instances required for the execution of the monitor [6]. The first analysis is already integrated in the framework, the second one is currently under implementation.

The LogicGuard specification language is documented in a tutorial and reference manual [9]; while the source code of the framework (which has been implemented in C# and F# on the basis of Microsoft .NET technology) is proprietary, an executable binary is publically available for non-commercial purposes [10].

The LogicGuard framework, while representing a research prototype, is stable and ready for experimental use; we have therefore started to use it for practical application scenarios in the context of online network monitoring and post-execution analysis of network traffic captured in trace files. The goal of this paper is to demonstrate two such applications taken from the context of network security. As a startup example, we present the "Rogue DHCP" scenario where a device illicitly poses as a DHCP server in order to feed newly connected devices with wrong connectivity information. Our main scenario is "Dynamic DNS (DDNS) Cache Poisoining" where an attacker poses as an DDNS client and feeds the DDNS server with wrong DNS update information. By these scenarios, also typical "patterns" of LogicGuard specifications are demonstrated.

The remainder of this paper is organized as follows: in Section II, we give a short overview on the LogicGuard framework and its specification language by some small examples. In Section III, we present as a first practical application the "Rogue DHCP" attack scenario that can be easily addressed by the presented language mechanisms. In Section IV, we discuss the "Dynamic DNS Cache Poisoning" attack scenario, which is substantially more challenging; we develop for this scenario multiple solutions, starting with a simple but incomplete one, proceeding to a complete one that mimics a "global state machine" which however mainly relies on external code, and finally deriving a solution that mimics a collection of "local state machines"; we then generalize the solution to discover not only attacks but also failures of the legitimate clients.


U:\repositories\LogicGuard2\AP2\LogicGuard2\Main\bin\Release\Main.exe

```
LogicGuard 1.02 (December 15, 2015)
Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD)
All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard2/
------------------------------------------------------------------------
Parsing specification <../specs/quick.lgs>.
Reading file <U:\projects\LogicGuard-2014\LogicGuard2\AP2\specs\quick.lgs>.
No syntax errors.
No type errors.
No analyzer errors.
Monitoring started.
Print: 3
Print: 4
Print: 6
Print: 0
VIOLATION<M>: position<S> x=3
Print: 7
Print: 6
Print: 5
Print: 0
VIOLATION<M>: position<S> x=7
Print: 3
Message trace is completed.
Press <ENTER> to continue.
```

Fig. 1: The LogicGuard software.

II. The LogicGuard Framework

In order to make this paper self-contained, we give in this section (partially based on material from [8]) an overview on the LogicGuard framework. The framework essentially consists of a specification language, a compiler, a static analyzer, and a runtime system. The compiler translates a declarative specification in the LogicGuard language into an executable monitor. The static analyzer ensures that the resulting monitor can operate with a finite amount of history, i.e., a finite buffer of past messages that are preserved in the monitor during its execution and passes the corresponding information to the runtime system.

The runtime system repeatedly invokes the monitor with messages that are provided from external sources (e.g. from a network interface for the live monitoring of network traffic or from a file that holds captured traffic for offline analysis). Every such invocation lets the monitor update its state and potentially report the positions of some messages (the current or some previous ones) that violate the specification. Fig. 1 shows a screenshot for a run of the system with a simple specification that analyzes a small file and reports two violations for the messages at positions 3 and 7 (and also displays some informative output).

A specification in the LogicGuard language consists of a sequence of declarations respectively definitions:

- *External streams*: these are the "real" streams whose messages are provided by the runtime system from external sources, e.g., network interfaces; they are just declared in the specification.
- *Internal streams*: these are "virtual" streams whose messages are constructed from other (external or internal) streams to raise the level of abstraction of the specification; these streams are therefore defined in the specification by some stream terms.
- *Monitors*: these are descriptions of properties that certain (external or internal) streams shall satisfy; monitors are therefore defined by logical formulas that formally express these properties.

A specification may contain declarations of external functions and predicates for the use in the specification; for these operations, the runtime system dynamically links external .NET code to the monitor. The main role of the specification language (which itself has no computational capabililities) is thus to coordinate the execution of these

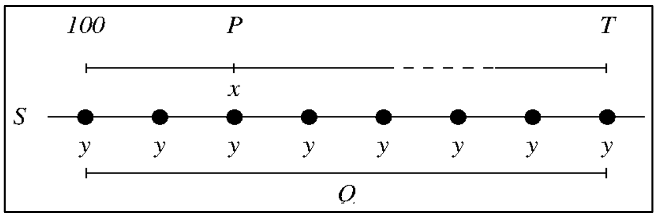


Fig. 2. Monitoring a stream.

operations which may be implemented in arbitrary .NET languages.

We are now going to sketch these elements by illustrative examples; for the full language, see the manual [9].

A. Monitors

Monitors are defined by formulas that may involve the usual propositional connectives for negation, conjunction, disjunction, implication, and equivalence (denoted by the operators `!`, `&&`, `||`, `=>`, and `<=>`) but also quantified formulas of the form *tag variable body* where

- *tag* indicates the kind of quantification (e.g., `forall` and `exists` for universal and existential quantification),
- *variable* is of the form *stream identifier constraints* where *identifier* is introduced as a local variable that runs over all positions in *stream* allowed by *constraints*;
- *body* is a formula that is evaluated for all assignments to *identifier*.

For instance, the definition

```
monitor<S> M =
  monitor<S> x : P(@x) =>
    forall<S> y
      with x-100 <= _ until T(@y) :
      Q(@x,@y)
```

introduces a monitor *M* over some stream *S*. It is defined by the quantified phrase of form `monitor<`*S*`>` *x* `:` *F* which assigns to variable *x* every position of stream *S*, evaluates formula *F* for this assignment, and reports every position assigned to *x* for which *F* is not true as a violation of the specification.

In more detail, *F* asks, for every message at position *x* that satisfies property *P*, whether property *Q* is true for every message at every position *y* in *S* that occurs not earlier than 100 time units before *x*; for every message at *x* the monitoring stops with the first message at *y* for which property *T* holds. This relationship is illustrated in Fig. 2.

It should be thus noted that the fact whether position *x* violates the specified property depends on certain messages before *x* (which requires to keep all messages of age less than or equal 100 in a buffer) as well as on certain messages after *x* (which requires to preserve the obligation to monitor *x* until property *T* is observed).

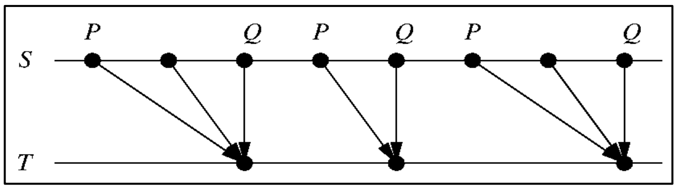


Fig. 3. Constructing a virtual stream.

B. Streams

A declaration

```
stream<Packet> IP;
```

introduces an external stream *IP* whose values have some (previously declared) type `Packet`; the LogicGuard runtime system delivers to this stream all packets that it captures from some designated network interface.

On the other side, a definition

```
stream<M> T =
  stream<S> x satisfying P(@x) :
    value[s,b,f]<S>
    y with x <= _ until Q(@y): @y
```

introduces an internal stream *T* by a quantified term of form

```
stream<S> x … : f(@x)
```

which denotes the stream $f(x_1), f(x_2), \ldots$ constructed from the messages $x_1, x_2, \ldots$ on stream *S*; the notation for stream construction mimics the classical set builder notation $\{f(x) \mid x \in S\}$.

In above example, each $f(x_i)$ is represented by a quantified term of form

`value[`s, b, f`]` *variable term*

which evaluates for all assignments of positions to the variable introduced by *variable* the denoted *term* yielding together with the base value *b* a non-empty sequence of values; by application of a binary function *f* these values are gradually combined to a single value that denotes the result of the term; the tag *s* indicates whether the evaluation must proceed in sequence or whether (because f is a commutative and associative operation) the order of combinations may be arbitrary.

In above definition, the stream consists of a sequence of values $v_1, v_2, \ldots$ of type *M* where each v_i is constructed from some value x_j on stream *S* that satisfies predicate *P*: the function *f* combines in v_i the values $b, x_j, x_{j+1}, \ldots, x_{j+n}$ where x_{j+n} is the first message for which property *Q* holds. This relationship between the original messages in *S* and the constructed messages in *T* is also illustrated in Fig. 3.

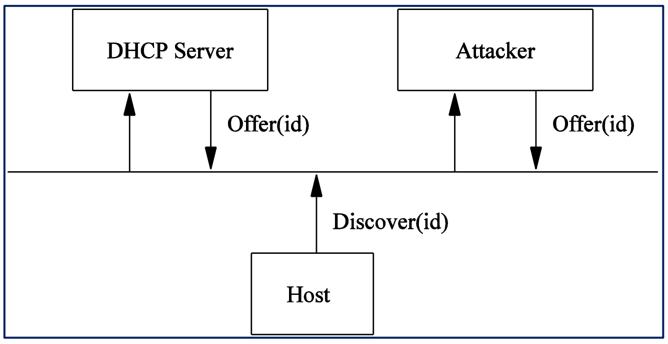


Fig. 5: Rogue DHCP (Scenario).

III. ROGUE DHCP

A. *Scenario*

We present as a first security-related application scenario of the LogicGuard framework "Rogue DHCP". DHCP (Dynamic Host Configuration Protocol) is responsible for providing every client that is freshly connected to a network with essential connectivity information, e.g. its own IP address, the address of the gateway, and the address of the DNS server. When a client is connected to the network, it broadcasts a "DHCP discover" message with a unique transaction ID. The DHCP server connected to the network then replies with a "DHCP offer" message that contains the ID and the connectivity information.

In a rogue DHCP attack (see also Fig. 5), a malicious device is connected to the network that poses as a DHCP server and replies to the request with wrong information (which may e.g. redirect DNS requests to a wrong DNS server). The goal is to detect such attacks by observing that there are two DHCP offers to the same discover message (one of which is from an attacker).

B. *Solution*

A simple solution to above scenario is represented by a specification with the following core (see also Fig. 4):

```
stream<Packet> IP;

monitor<IP> M =
  monitor<IP> x: IsDhcpOffer(@x) =>
    forall<IP> y with x < _ <=# x+T
      while !(IsDhcpDiscover(@y) &&
              SameId(@x,@y)):
      !(IsDhcpOffer(@y) && SameId(@x,@y));
```

For every position *x* in stream *IP* that denotes a DHCP offer message, a monitor instance is created that waits at most *T* time units for a position *y* that also denotes a DHCP offer with the same transaction ID (if during this time another discover message with the same transaction ID is detected, the instance terminates, because any subsequent offer may refer to the new request). If such an offer is detected, the position *x* is reported to violate the specification. The time bound *T* is chosen to ensure that every monitor instance eventually terminates.

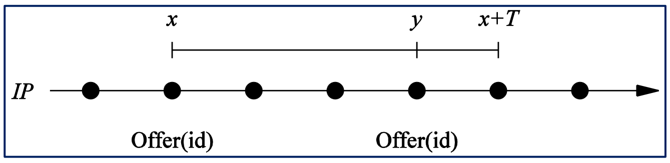


Fig. 4: Rogue DHCP (Monitoring).

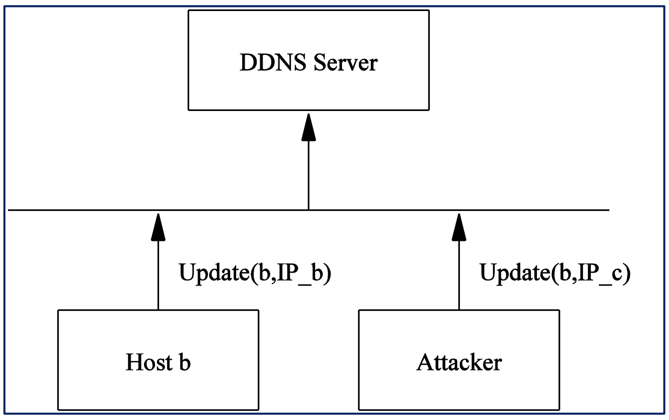


Fig. 6: DDNS Cache Posoning (Scenario).

IV. DYNAMIC DNS CACHE POISONING

A. *Scenario*

A more complex security-relevant application scenario for the LogicGuard framework is "DNS cache poisoning" (also called "DNS spoofing"). DNS (domain name system) is responsible for translating domain names into Internet Protocol (IP) addresses. Every host *a* that wants to contact another host *b* first sends a DNS request with the domain name of *b* to a DNS server which responds to *a* with the IP address of *b*; to this IP address *a* then sends its message. Dynamic DNS (DDNS) is a variant of DNS that supports changes of IP addresses by letting host *b* sending in regular intervals via DNS update messages its current IP address to the DDNS server.

In a DNS cache poisoning attack an attacker inserts into the DNS server for *b* the address of another host *c* such that *a* unknowingly contacts *c* rather than *b*. In the case of DDNS, an attacker may poison the cache by impersonating as *b* and sending a DNS update message for *b* with the IP address of *c*. This scenario is depicted in Fig. 6.

The goal is to develop a monitor that detects such attacks based on its expectation when the next DDNS update message for some host is expected: if a DDNS update message comes significantly earlier than expected, the corresponding message shall be reported as a potential attack. Furthermore, the set of hosts registered at a DDNS server is not fixed in advance but has to be determined by the observations of the monitor.

B. *Fixed Time Interval*

For illustrative purposes, we start with a simple specification that reports an attack, if the interval between two successive DNS update messages for a host, is smaller than *T* time units (for some fixed constant *T*). The core of this specification is as follows:

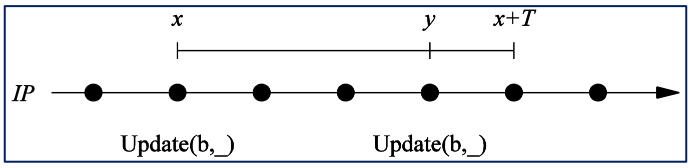


Fig. 7: DDNS Cache Poisoning (Fixed Time Interval)

```
stream<Packet> IP;

monitor<IP> M =
  monitor<IP> x: IsDnsUpdate(@x) =>
    !(exists <IP> y with x < _ <# x+T:
        IsDnsUpdate(@y) && IsSameHost(@x,@y));
```

For every position *x* in the stream of IP packets, the monitor *M* checks by invocation of some external predicate whether the message at *x* is a DNS update message; if not, position *x* trivially satisfies the specification. However, if it is indeed such a message, an instance of the monitor is generated that monitors every message at any position *y* that occurs after *x* but less than *T* time units afterwards: if this message is also a DNS update message for the same host as in the message at position *x* (as determined by some external predicates), the monitor reports *x* as a violating position.

This specification illustrated in Fig. 7 is too simplistic because it depends on a fixed constant *T*; the following specifications will determine appropriate attack criteria dynamically and specifically for each individual host.

C. Global State Machine

Our next attempt to address the scenario mimics a "state machine" that maintains a table that records for every host for which some DNS update operation has been observed some statistics about all DNS updates observed so far. This mimicry proceeds by constructing a stream *GS* whose values represent the states of such a "global state machine" (see also Fig. 8):

```
stream<Packet> IP;

stream<GlobalState> GS =
  stream[seq,NewStateG(),UpdateStateG]<IP> x
    satisfying IsDnsUpdate(@x):
      @x;
```

The first value of *GS* is determined by a call *NewStateG() of* some external function, every subsequent state is constructed by arrival of a DNS update message on stream *IP* by applying the external function *UpdateStateG* to the previous state and the message. The monitor processes this stream of states:

```
monitor<GS> M =
  monitor<GS> x:
    IsAttackG(@x) =>[seq] PrintG(@x);
```

Every state that indicates the observation of an attack for some host (as indicated by the external predicate *IsAttackG*) is

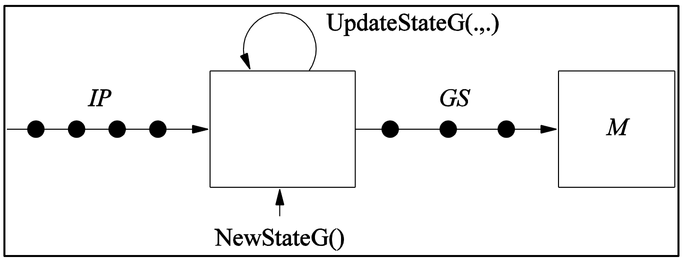


Fig. 8: DDNS Cache Poisoning (Global State Machine)

passed to an external predicate that denotes the truth value "false" (such that the position of the state is reported as a violation) and prints as a side effect the information about the attacked hosts.

In this specification, all the strategy about the collection of the statistical information and the decision to report an attack is delegated to the external data type *GlobalState* and the functions *UpdateStateG* and *IsAttackG*; the role of the monitor is reduced to drive these functions in the form of a single global state engine. Since the external entities become quite complex, we strive for a more "fine-grained" solution.

D. Local State Machines

Our final attempt to address the scenario starts by constructing (again in a "state-machine" like fashion) from the external stream *IP* a virtual stream *U* that only contains the DNS update messages and marks the first update message of every new host observed (see also Fig. 9 where the gray bullets denote marked messages):

```
stream<Packet> IP;

stream<DNSUpdate> U =
  stream[seq,NewUpdate(),NextUpdate]<IP> x
    satisfying IsDnsUpdate(@x): @x;
```

Here every "DNSUpdate" value actually not only contains the current (potentially marked) DNS update but also the set of all hosts observed so far. From this set, the external function *NextUpdate* determines the marking of the next DNS update message and the new host set.

From this stream, we construct the following stream that merges the states of a number of "local" state engines, one for each observed host:

```
stream<LocalState> LS =
  merge<U> x satisfying IsNewHost(@x):
    stream[seq,NewStateL(@x),UpdateStateL]
      <U> y with x < _
      satisfying IsSameHostU(@x,@y):
        @y;
```

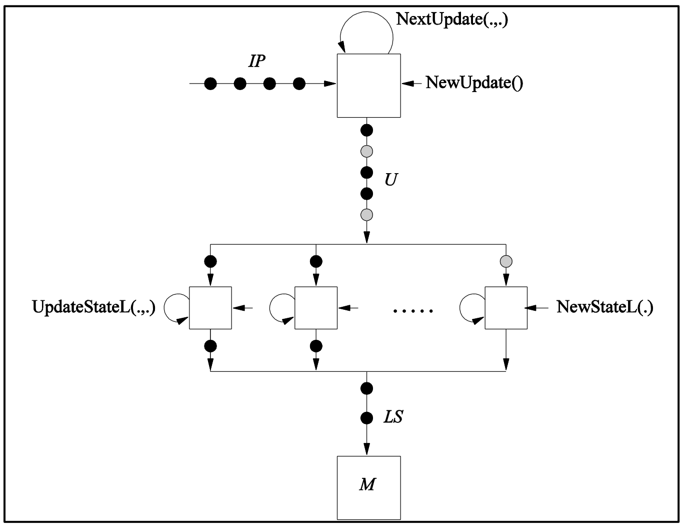


Fig. 9: DDNS Cache Posoning (Local State Machines)

For every position *x* in the stream *U* that denotes an update message of a not yet observed host (as determined by the predicate *IsNewHost*), the `stream` construct creates a new "local" state engine that starts with the initial state constructed by the function *NewStateL* from this initial update message: for every subsequent update message (at a position *y* bigger than *x*) that refers to the same host, the function *UpdateState* creates a new state that collects all statistics for this individual host. The `merge` construct finally merges the contents of all streams, thus constructing a stream *LS* of local states. The merging proceeds time-based: whenever some of the local engines produces a new state, this state is immediately propagated to *LS* which is fed to the following monitor:

```
monitor<LS> M =
  monitor<LS> x:
    IsAttackL(@x) =>[seq] PrintL(@x);
```

Monitor *M* checks every local state whether it indicates an attack for a particular host; if yes, the position *x* of that stream is reported to violate the specification and the information about the violating host is printed.

While this specification seems more complex than in the "global state machine" abroach, it actually considerably simplifies the implementation of the external data type *LocalState* and the functions *UpdateStateL* and *IsAttackL* operating on this type: they only have to consider the statistics of a single host, rather than that of all hosts observed so far. This specification thus represents our preferred solution.

E. Timeouts

The scenario presented so far can be extended to not only consider DHCP update messages that arrive earlier than expected (indicating potential attacks) but also messages that do *not* arrive at the expected time (indicating potential host failures). In order to deal with such "timeout" scenarios where the failure is not triggered by the arrival of a message but by the lack of an arrival, the LogicGuard runtime system can be started in a mode where, if no message has arrived for a certain amount of time from the network interface, the monitor is invoked with an artificially generated "null" message. We then revise the specification as follows:

```
stream<Packet> IP;

stream<DNSUpdate> U =
  stream[seq,NewUpdate(),NextUpdate]<IP> x
    satisfying IsNull(@x) || IsDnsUpdate(@x):
      @x;
```

Here also null messages are passed to the "state engine" generating the stream *U*; for every such arriving message, the engine generates a correspondingly tagged message in stream *U*. We then also revise the construction of the "local state" stream as follows:

```
stream<LocalState> LS =
  merge<U> x satisfying IsNewHost(@x):
    stream[seq,NewStateL(@x),UpdateStateL]
      <U> y with x < _
      satisfying IsNullU(@y)
              || IsSameHostU(@x,@y):
        @y;
```

The null message arriving to stream *U* is propagated to each of the "local" state engines which can then update their local statistics also according to the lack of an expected message. By extending the monitor to

```
monitor<LS> M =
  monitor<LS> x:
    IsAttackL(@x) || IsDownL(@x) =>[seq]
    PrintL(@x);
```

it can then report not only attacks but also timeouts.

V. Conclusions

The LogicGuard stream monitor specification framework originally introduced in [8] is now ready for its practical application in various event monitoring scenarios. While the focus has been *security*-related applications, such as the online monitoring of network traffic, we also envision *safety*-related applications scenarios where the deviation of activities from some specified norm is to be detected. Furthermore, by an extension of the runtime system that allows for the generation of null events if no other activity is observed, also the expected *progress* of operation can be monitored. The source code of the software is proprietary, but an executable binary of the framework is publically available for non-commercial purposes together with a tutorial and reference manual [9][10].

Current work proceeds along two directions: on the one side, we are elaborating more and more application scenarios and experimentally evaluate the runtime behavior (performance and memory consumption) of the corresponding monitors. On the other hand, we work on the static analysis of specifications in order to give a priori (compile time) information about the expected runtime behavior of the monitors generated from these [6]. A prototypical implementation of the analysis will soon become part of the software. Our long-term goal is to devise transformation and advanced compilation techniques in order to generate from high-level declarative LogicGuard specifications monitors whose performance can compete with those developed by low-level manual coding.

REFERENCES

[1] A.M. Ahmed, „Online network intrusion detection system using temporal logic and stream data processing," Ph.D. thesis, University of Liverpool, UK, 2013.

[2] H. Barringer, A. Goldberg, K. Havelund, K. Sen, "Program monitoring with LTL in Eagle," IPDPS'04, 18th International Parallel and Distributed Processing Symposium — Workshop 16 PADTAD. Santa Fe, NM, USA, April 30, 2004.

[3] H. Barringer, D. Rydeheard, K. Havelund, "Rule systems for run-time monitoring: from Eagle to RuleR," Journal of Logic and Comput. 20(3), pp. 675–706, 2010.

[4] C. Büchi, "Weak second-order arithmetic and finite automata," Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6, pp. 66–92, 1960.

[5] F. Chen, G. Rosu, "MOP: an efficient and generic runtime verification framework," 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07). pp. 569–588. ACM, New York, 2007.

[6] D. Cerna, W. Schreiner, and T. Kutsia, "Space Analysis of a Predicate Logic Fragment for the Specification of Stream Monitors", SCSS 2016, 7th International Symposium on Symbolic Computation in Software Science, Tokyo, Japan, March 28-31, 2016. EasyChair Proceedings in Computing (EPiC), in press.

[7] T. Kutsia, W. Schreiner, "Verifying the soundness of resource analysis for LogicGuard monitors (revised version)", Technical Report 14-08, RISC, Johannes Kepler University, Linz, Austria, September 2014.

[8] W. Schreiner, T. Kutsia, M. Krieger, B. Ahmad, H. Otto, and M. Rummerstorfer, "Securing device communication by predicate logic specifications," embedded world conference 2015, February 24-26 2015, Nürnberg, Germany, Matthias Sturm et al. (ed), Design&Elektronik, Haar, Germany.

[9] W. Schreiner, T. Kutsia, D. Cerna, M. Krieger, B. Ahmad, H. Otto, M. Rummerstorfer, and T. Gössl, "The LogicGuard stream monitor specification language: tutorial and reference manual," Technical Report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. October 2015. http://www.risc.jku.at/projects/LogicGuard2/software/Manual.pdf.

[10] The LogicGuard Software, 2016. http://www.risc.jku.at/projects/LogicGuard2/software.