A Stream Runtime Verification Tool with Nested and Retroactive Parametrization[⋆]

Paloma Pedregal[1,2(✉)], Felipe Gorostiaga[1,3(✉)], and César Sánchez[1]

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid (UPM), Spain
[3] CIFASIS, Argentina

Abstract. In online monitoring, a monitor is synthesized from a formal specification, which later runs in tandem with the system under study. In offline monitoring the trace is logged as the system progresses to later do post-mortem analysis after the system has finished executing.
In this tool paper we demonstrate the use of *retroactive dynamic parametrization*, a technique that allows an online monitor to revisit the past log as it progresses. This feature enables new monitors to be incorporated into an already running system and to revisit the past for particular behaviors, based on new information discovered. Retroactive parametrization also allows a monitor to lazily ignore events and revisit and process them later, when the monitor discovers that it should have processed those events. We showcase the use of retroactive dynamic parametrization to perform network monitor denial of service attacks.

1 Introduction

Runtime verification (RV) [2,18] is a lightweight formal dynamic verification technique that analyzes a single trace of execution using a monitor derived from a specification. The initial specification languages to describe monitors in RV where borrowed from property languages for static verification, including linear temporal logic (LTL) [23], adapted to finite traces [3, 8, 19]. Most RV languages describe a monolithic monitor that later processes the events received. Dynamic parametrization (also known as parametric trace slicing) allows quantifying over objects and spawning new monitors that follow independently objects as they are discovered, like in Quantified Event Automata (QEA) [1].

Stream runtime verification [4, 11, 22](SRV), pioneered by Lola [7] defines monitors by declaring the dependencies between output streams and input streams. The initial application domain of Lola was the testing of synchronous hardware. Temporal testers [24] were later proposed as a monitoring technique for LTL based on Boolean streams. Copilot [11, 21, 22] is a DSL similar to Lola, which declares dependencies between streams in a Haskell-based style, and then generates C monitors. Lola2.0 [9] extends Lola allowing dynamically parametrized streams, similarly to QEA. Stream

[⋆] This work was funded in part by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR, and by a research grant from Nomadic Labs and the Tezos Foundation.

runtime verification has also been extended recently to asynchronous and real-time systems [6, 10, 12, 17]. HLola [5, 13, 15] is an implementation of Lola as an embedded DSL in Haskell, which allows borrowing datatypes from Haskell directly as Lola expressions, and using features like higher-order functions to ease the development of specifications and the runtime system. In this paper we use HLola and extend it with capabilities for retroactive dynamic parametrization.

In practice it is common to monitor properties that are defined after the system starts running, and we cannot or do not wish to stop the system. Therefore, the monitor will receive online new events after being installed. Then, one can (1) *ignore* that the monitor is started in the middle of the computation and pretend that the history starts after the monitor is installed, (2) *encode the lack of knowledge* of the monitor in the specification, or, (3) if the beginning of the trace was stored in an accessible *log*, allow the monitor to collaborate with the log to process the missing past events and then continue to process the future events online. The first option is the most natural and in many cases an acceptable solution, while the second option has been explored in [16], but these two options neglect the beginning of the trace which can sometimes affect the monitoring task. The third option requires a novel combination of *offline and online monitoring* offering the possibility of accessing the past of the trace. Moreover, enriching an SRV monitor with the ability of accessing the past allows the description of properties that revisit the past exploiting information discovered at a later time.

In this tool paper we demonstrate an extension of the tool HLola[4] which enables a novel dynamic instantiation of monitors called *retroactive dynamic parametrization*. The new tool offers dynamic parametrization and extends it with the ability to revisit the past of a live stream of events, effectively combining online and offline runtime verification. A longer version of this paper is available at [20].

2 Overview

Preliminaries. Stream Runtime Verification (SRV) generalizes monitoring algorithms to arbitrary data, by preserving the temporal dependencies and generalizing the datatypes using multi-sorted data theories. HLola is an extensible implementation of Lola [7] developed as an embedded DSL in Haskell, from which HLola borrows datatypes as data theories. HLola also allows the easy implementation of new powerful features as libraries with no changes to the core engine. The tool described in this paper incorporates retroactive parametrization to HLola.

A Lola specification $\langle I, O, E\rangle$ consists of (1) a set of typed input stream variables I, which correspond to the inputs observed by the monitor; (2) a set of typed output stream variables O which represent the outputs of the monitor as well as intermediate observations; and (3) defining equations, which associate every output $y \in O$ with a stream expression E_y that describes declaratively the intended values of y (in terms of the values of inputs and output streams). The set of *stream expressions* of a given type is built from constants and function symbols as constructors (as usual), and also from *offset expressions* of the form s**[now]**, or s[k|d] where s is a stream variable,

[4] Available at `https://github.com/imdea-software/hlola/`

k is an integer number and d is a value of the type of s (used as default). For example, `altitude[now]` represents the value of stream `altitude` in the current instant, and `altitude[-1|0.0]` represents the value of stream `altitude` in the previous instant, with `0.0` as the value used at the first instant. HLola can be efficiently monitored, meaning that (most programs) can be monitored in constant space and in constant time per event (trace-length independent monitoring [5]).

As a byproduct of its design, HLola allows *static parametrization* in stream definitions, this is, streams that abstract away some concrete values, which are later instantiated by the compiler. Even though static parametrization is very useful to define libraries and clean specifications, parameters are expanded at static time before the monitor starts running, and parametric streams cannot be spawned with a value that is discovered at runtime. The keystone of the design of HLola is to use datatypes and functions from Haskell as the data theories of Lola. In turn, HLola also allows using Lola specifications as datatypes, via the function $runSpec$ that executes a specification over the input trace and produces a value of the result type. This allows Lola specifications to be used as data theories within Lola, a feature called *nested monitoring* [14]. Nested monitors allow writing functions on streams as SRV specifications, creating and executing these specifications dynamically. In [14] nested monitors are created and destroyed within an instant and their final states are lost. Also, nested monitors cannot access the past of the trace before the beginning of the sub-trace they receive. In this tool paper we introduce novel features by relaxing these restrictions, gaining the ability to *combine offline and online runtime verification*. We allow nested monitors to be created dynamically and continue executing alongside their parent monitor. The states of the dynamically created nested monitors are carried on to the next instant, and they can inspect the full past of the system.

We introduce the following kinds of nested monitors:

1. Retroactive Nested Monitors, which can access events and trigger a finer analysis of the past of the trace when necessary. For example, consider monitoring network traffic, where the monitor receives (1) the source and destination of each IP packet, and (2) the packets per second in the last hundred instants. We want to detect whether an address has received too many packets in the last hundred instants, which can be specified as follows: *if the packet flow is low, then there is no attack, but when the flow rate is above a predefined threshold (`threshold_pps`) we have to inspect the last hundred packets and check if a given address is under attack.* We can define a specification which only observes the packets per second in the last hundred instants, ignoring the source and destination of the IP packets. If the packets per second exceed `threshold_pps`, this triggers the creation of a *retroactive nested monitor*, which will retrieve the past of the trace using the new keyword `withTrace` and do the more expensive analysis of detecting if an address is in fact under attack. Note how this specification detects an attack at most one hundred instants after it happens. Also note that the nested monitors in this example are created, executed and destroyed at every instant.

2. (Forward) Dynamic Parametrization, which let us instantiate a parametric stream using dynamically discovered values, via the new keyword **`over`**. The `over` operator takes a parametric stream `strm` of type `S` with a parameter of type `P`, and a stream `params` of sets of values of type `P`, and creates an expression of type `Map P S`, whose keys at any given instant are the values in `params[now]`, and where the value associated to each key is the instantiation of `s` over the key. Using the `over` operator we can dynamically instantiate a parametric stream over a set of parameters that are discovered while processing the trace of input.

Consider a scenario where we are monitoring network traffic, and following every TCP 3-way handshake in which (1) the source sends a packet `SYN`, then (2) the destination sends `SYN/ACK` and then (3) the source sends `ACK`. We define a parametric stream which receives a pair of addresses and generates a value, which can be `Valid` or `Error`, depending on whether the handshake is correct or not. We cannot know statically for which pair of addresses we have to instantiate the parametrized stream, and therefore, the monitor has as parameter a pairs of addresses to follow. At every instant, the monitor can add a new pair of source and destination addresses. In this manner we can use a parametric stream over dynamic values using the `over` operator. Every time a new value is incorporated to the set of active parameters, we spawn a new monitor parametrized with the new value. Then, we preserve the state of this monitor between instants in an auxiliary stream, executing the nested monitor alongside the outer monitor until the auxiliary monitor is no longer needed, that is, until its associated parameter is removed from the set.

3. Retroactive Dynamic Parametrization, which allows revisiting the past of a trace every time the monitor discovers a new parameter to instantiate the parametric stream. The new parametrized stream continues to monitor in an online manner. The static parametrization already present in HLola is too limited to implement this feature because the monitor cannot know the state of a parametric stream over an arbitrary parameter in the middle of the trace unless the parameter was determined statically. Using static parametrization to implement dynamic parametrization is only feasible for small parameter sets, like Booleans or a small enumerated type, but it becomes unfeasible when the space of potential parameters is large.

To implement retroactive dynamic parametrization we add a new clause `withInit` to the `over` operator to describe an *initializer*, which indicates how the nested monitor can take its state up to the current instant. An initializer will typically call an external program with the corresponding arguments that indicate how to efficiently retrieve the elements in the past of the trace that are relevant to the parameter.

Consider the case of monitoring a file system assessing whether every time a file is read or written, it had been created previously. One way is to use *forward dynamic parametrization* following all the files as they are opened. With *retroactive* monitoring, we can start following a file id just when it is read or written, and only then (calling an external program) retrieve the past of the trace for that parameter. The external program can use an index to efficiently retrieve only the events relevant to a file id or even only the `open` events.

3 HLola with Dynamic Parametrization

We have implemented retroactive nested parametrization, forward dynamic parametrization and retroactive dynamic parametrization in HLola.

Dynamically mapping a parametric stream `strm` with a stream of set of parameters `params` of type `Set P` creates an auxiliary stream `x_over_params` of type `Map P MonitorState` that associates, at every instant, each parameter p in `params` with the state of the nested monitor corresponding to `s<`p`>`. The value of `x 'over' params` is simply the projection of the parametrized streams in the monitors of `x_over_params[now]`. There are three possibilities for the behavior of the auxiliary stream for given p:

(1) $p \in$ `params[-1|∅]` $\setminus$ `params[now]`: the parameter was in the set in the previous instant, but it is no longer in the set in the current instant. In this case, p and its associated value are deleted from the map `x_over_params[-1|∅]`.

(2) $p \in$ `params[-1|∅]` $\cap$ `params[now]`: the parameter was in the set and it is still in the set now. In this case, we feed the current event to the monitor associated with p and let it progress one step. Then, the value of the parametrized stream in the nested monitor is associated with p in the returned map.

(3) $p \in$ `params[now]`$\setminus$`params[-1|∅]`: the parameter was not in the set, but it is now. The monitor for p is installed, executing the initializer (possibly revisiting the past) to get the monitor up to date and ready to continue online. After installing the monitor, the new event is injected, and the value of `s<`p`>` is associated to p the returned map. Note that the past is only revisited when a new parameter is discovered. Once the stream is instantiated with the parameter, its corresponding nested monitor will continue executing over the future of the trace online.

Since we want HLola to support initialization from different sources (e.g. a DBMS, a blockchain node, or plain log files) the initializer of the internal monitors typically invokes an external program. This external program, called *adapter*, is in charge of recovering the trace and formatting it adequately for the monitor.

4 A Network Traffic Case Study and Empirical Evaluation

We report in this section an empirical evaluation of retroactive dynamic parametrization, implemented in our tool, that extends HLola [13]. We use our tool to implement monitors that describe several algorithms for the detection of distributed denial of service attacks (DDOS). All the experiments were executed on a Linux machine with 256GB of RAM and 72 virtual cores (Xeon Gold 6154 @3GHz) running Ubuntu 20.04. For conciseness we use RP to refer to retroactive parametrization, and non-RP to implementations that do not use retroactive parametrization (but use dynamic parametrization). We evaluate empirically the following hypotheses:

(H1) RP is functionally equivalent to non-RP.

(H2) RP and non-RP run at similar speeds, particularly when most dynamic instantiations turn out to be irrelevant.

(H3) RP consumes significantly less memory than non-RP, particularly when most instantiations are irrelevant.

(H4) Aggregated RP—where the monitor receives summaries of the trace that indicate if further processing is necessary—is functionally equivalent to RP.

(H5) Aggregated RP is much more efficient than RP and non-RP without aggregation.

The datasets for the experiments are (anonymized) samples of real network traffic collected by a Juniper MX408 router that routes the traffic of an academic network used by several tens of thousand of users (students and researchers) simultaneously, routing approximately 15 Gbps of traffic on average. The sampling ratio provided by the routers was 1 to 100 flows[5]. Each flow contains the metadata of the traffic sampled, with information such as source and destination ports and addresses, protocols and timestamps, but does not carry information about the contents of the packets. These flows are stored in aggregated batches of 5 minutes encoded in the **netflow** format.

Our monitors implement fourteen known DDOS network attacks detection algorithms. An attack is detected if the volume of connections to a destination address surpasses a fixed attack-specific threshold, and those connections come from a sufficiently large number of different attackers, identified by source IP address. The number of different source addresses communicating with a destination is known as the *entropy* of the destination. Each attack is concerned with a different port and protocol and considers a different entropy as potentially dangerous.

In order to process the network data needed by the monitors, we developed a Python adapter that uses `nfdump`, a toolset to collect and process netflow data. The tool `nfdump` can be used to obtain all the flows in a batch, optionally applying some simple filters, or to obtain summarized information about all the flows in the batch. For example, `nfdump` can provide all the flows received, filtered by a protocol or address, as well as the volume of traffic to the IP address that has received the most connections of a specific kind.

Our empirical evaluation consists of four datasets in which we knew whether each attack was present:

(D1) A batch of network flows with an attack based on malformed UDP packets (UDP packets with destination port 0). This batch contains 419938 flows, with less than 1% malformed UDP packets. The threshold for this attack is 2000 packets per second, which is surpassed in this batch for one single address, for which the entropy of 5 is exceeded.
(D2) A batch of network flows with no attack, containing 361867 flows, of which only 66 are malformed UDP packets (roughly, 0.001%).
(D3) A batch of network flows with no attack, but with many origin IP addresses and 100 destination addresses.
(D4) Intervals with several batches, where only one batch has an attack based on malformed UDP packets.

The monitors in our experiments follow the same attack description: *In a batch of 5 minutes of flow records, an address is under attack if it receives more than t_0 packets per second or bits per second from more than t_1 different source addresses (where t_0 and t_1 depend on the attack).*

We have implemented our monitors in three different ways[6]:

(S1) Brute force: Using *(forward) dynamic parametrization*, the monitor calculates the number of packets and bits per second (which we call "volume") for all potential

[5] Most detection systems use a much slower sampling of 1 to 1000 or even less.

[6] The specifications for **(S1)**, **(S2)** and **(S3)** as well as the instructions and dataset to execute them are available in a dedicated branch of the repository, at `https://github.com/imdea-software/hlola/tree/RV2023`.

target IP addresses. It also computes the entropy for each potential target address and for each attack. For every flow, the monitor internally updates the information about the source address, destination and volume.

```
input String fileId
input Flow flow

define Int flowCounter = flowCounter[-1|0] + 1
define Bool firstFlow = fileId[now] /= fileId [-1|""]
define Bool lastFlow = fileId[now] /= fileId[1|""]

output [String] attacked_IPs = map detect attacks
  where detect atk = (attack_detection atk)[now]

define String attack_detection <AttackData atk> =
  if (markerRate atk)[now] > threshold atk then
    if (ipEntropy atk)[now] > maxEntropy atk then
      (maxDestAddress atk)[now]
    else "Over threshold but not entropy"
  else "No attack"

define Int markerRate <AttackData atk> = ...
define String maxDestAddress <AttackData atk> = ...

define AddrInfo addrInfo <AttackData atk> =
  insertWith updt destAddr[now] (extractInfo atk flow[now]) prev
  where
    prev = if firstFlow[now] then empty else (addrInfo atk) [-1|empty]
    updt (p,b,ts,te) (p',b',ts',te') = (p+p',b+b',min ts ts',max te te')

define Histogram attackHist <AttackData atk> = let
  hist = if firstFlow[now] then empty else (attackHist atk) [-1|empty]
  in insertWith (+) destAddr[now] 1 hist
```

The specification uses the flowCounter to perform retroactive dynamic parametrization. The stream attacked_IPs maps the parametric stream attack_detection over the list of attacks. The stream attack_detection checks that the marker (bits per seconds or packets per second) of the attack and the IP entropy of any address do not exceed the thresholds. If the thresholds are exceeded, the IP address most accessed (which is calculated in maxDestAddress) is considered to be under attack. The stream markerRate calculates the bits per seconds or packets per second of an attack, while the stream maxDestAddress calculates the most accessed address. The stream addrInfo keeps a map of the packets, bits, start time and endtime per destination address. Similarly, the stream attackHist keeps a map of the number of accesses per destination address. In this scenario, we calculate the ipEntropy (not shown in the monitor above) of every address at all times, and we simply return the size of the set of different origin IP addresses of the most accesseed IP.

(S2) Retroactive: In this implementation, the monitor also analyzes all flows, calculating the volume of packets for each address, but in this case the monitor lazily avoids calculating the entropy, using *retroactive dynamic parametrization*. The monitor only calculates the entropy when the volume of traffic for an address surpasses the threshold. The monitor uses the over operator to revisit the past flows of the batch filtered by that attack, using the Python adapter which produces the subset of the flows required to compute the entropy. The monitoring then continues calculating the entropy until the end of the batch. The specification for this implementation is the same as in **(S1)**, but with a different implementation of the IP entropy calculation:

```
define Int ipEntropy <AttackData atk> =
  (maybe 0 size . listToMaybe . elems) mset
  where
    mset = setSrcForDestAddr atk ‘over‘ maybeAddress atk
           ‘withInit‘ initer atk fileId[now] flowCounter[now]
define (Set String) setSrcForDestAddr <AttackData atk> <String dst> = let
  prevSet = if firstFlow[now] then empty
            else (setSrcForDestAddr atk) [-1|empty]
  in insert srcAddr[now] prevSet
define (Set String) maybeAddress <AttackData atk> =
  if (attack_detection atk)[now] then singleton (maxDestAddress atk)[now]
  else empty
```

In this case, we define a parametric stream `setSrcForDestAddr` that calculates the set of different origin IPs of a destination address. We define an auxiliary stream `maybeAddress` that contains the most accessed address, if it exceeds the threshold. The definition of `ipEntropy` will instantiate dynamically the stream `setSrcForDestAddr` with the most accessed address once it exceeds the threshold, with an initializer specific to the suspected attack and address. We compose different functions to retrieve the values of the map (which is at most one), and get the size of the corresponding set, if it exists, using 0 as the default value.

(S3) Aggregated: This specification uses *retroactive nested monitors with retroactive dynamic parametrization* to analyze summaries of batches of flows, executing a nested specification over the current batch and the suspected attack, when one of the markers is above a predefined threshold: the monitor receives a summary of a five minute batch of network data, as a single event containing fourteen attack markers. The monitor is based on the ability of the backend to pre-process batches using `nfdump` to obtain—for each attack and for the whole batch—the maximum volume of traffic for any IP address. If an attack marker is over the threshold, the monitor spawns a nested monitor which retrieves a subset of the flows for that batch and attack, and analyzes those flows in a more detailed way. This second nested monitor behaves like the retroactive parametrization in **(S2)**. The aggregation of data provides a first, coarse overview serving as a necessary condition to spawn the expensive nested monitor. This is particularly useful because attacks are infrequent and the ratio of false positives of the summary detection is relatively low.

There are two great advantages to this implementation, in comparison to the implementation in **(S2)**: the finer analysis of the flows will only be performed when the aggregated data indicates a possble attack, instead of all the time and for all flows; and when the nested monitor *is* triggered, it will be triggered with two parameters, a specific batch and attack, that will be used to filter the flows before processing them. Being able to filter the flows by the characteristics of a specific attack greatly reduces the amount of flows of the batch to a small percentage (for example, in the dataset **(D1)**, which is a batch with an attack, less than 1% of the flows of the batch were part of the attack).

The nested specification `flowAnalyzer` is triggered by a marker which indicates a possible attack. This specification analyzes individual flows, and it is exactly the specification described in **(S2)**, but it will be used to analyze all flows in a batch only if the aggregated marker is positive.

```
use innerspec flowAnalyzer
input String fileId
input Int marker <AttackData atk>
output [String] attacked_IPs = map detect attacks
  where detect atk = (attack_detection atk)[now]
define String attack_detection <AttackData atk> =
  if (marker atk)[now] > threshold atk then
    runSpec (flowAnalyzer atk (flowRetriever atk fileId[now]))
  else "No attack"
```

The constant `attacks` is a list of the attack data of the fourteen different attacks that the monitor can detect. The nested specification `flowAnalyzer` analyzes individual flows, and it can use **retroactive dynamic parametrization**, or (the less efficient) **non-retroactive dynamic parametrization**.

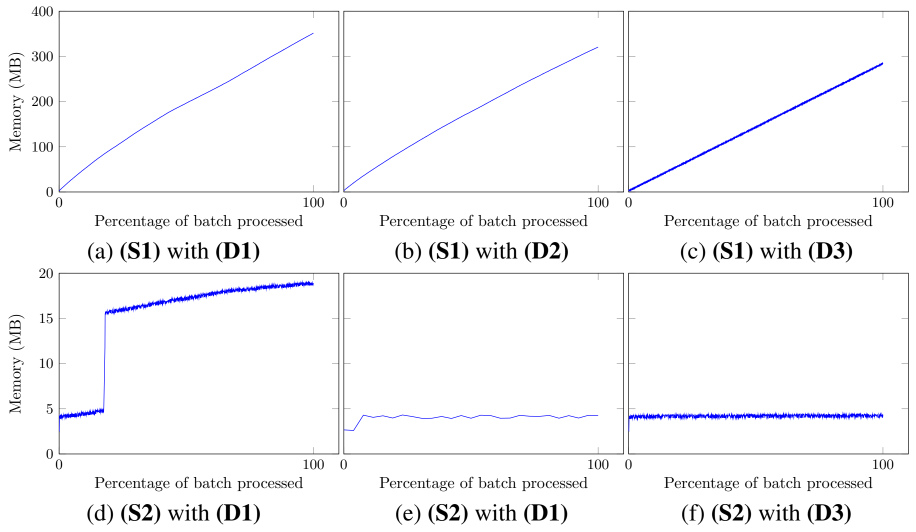
Results: In the first experiment we run the three implementations against dataset **(D4)**. In this interval of multiple batches, only one of which contains an attack, all three implementations identify the batch with the attack and correctly detect the kind of attack and target address. This confirms empirically hypotheses **(H1)** and **(H4)**.

In the second experiment we run specifications **(S1)** and **(S2)** against datasets **(D1)**, **(D2)** and **(D3)**. The results are reported in the following table:

	(D1) (Attack)	**(D2)** (No Attack)	**(D3)** (No Attack)
(S1) (Brute force)	18m12.146s	15m51.599s	16m34.795s
(S2) (Retroactive)	20m43.921s	17m19.844s	19m30.518s
(S3) (Aggregated)	0m16.208s	0m2.109s	0m2.115s

We can see that the running times for the brute force and retroactive implementations are similar, while the aggregated implementation is extremely fast in comparison, which empirically confirms **(H2)** and **(H5)**. This is because **(S3)** exploits the summarized information, and does not find any marker over the threshold for the datasets **(D2)** and **(D3)** so the flows within the batch are never individually processed. For dataset **(D1)**, a nested monitor will be executed because one of the markers (for the attack with malformed UDP packets) is over the threshold, but it will only try to detect the attack corresponding to that marker, and it will only receive a small subset of the flows (less than 1% of the flows of the batch are relevant for the attack). If all the markers for all the attacks were over their threshold and all the flows were implicated in the attacks, the time required would be closer to the retroactive implementation. The ad-hoc aggregation of data by the external tool is very efficient, as is the verification of this data by the monitor, so this implementation is especially advantageous when the positives (or false positives) are expected to be infrequent, and when most of the data can be filtered out before executing the nested monitor.

In a third experiment we run a version of specifications **(S1)** and **(S2)**—instrumented with capabilities to measure memory consumption—on **(D1)**, **(D2)** and **(D3)**. The re-



(a) **(S1)** with **(D1)** (b) **(S1)** with **(D2)** (c) **(S1)** with **(D3)**

(d) **(S2)** with **(D1)** (e) **(S2)** with **(D1)** (f) **(S2)** with **(D3)**

Fig. 1. Memory usage of the brute force (a), (b), (c) and retroactive (d), (e), (f).

sults, reported in Fig. 1, empirically confirm **(H3)**. For the three datasets, the memory used by the brute force approach increases linearly over time, as it has to keep track of the volume and IP entropy for every attack and every potential target address. On the other hand, the memory usage of the retroactive implementation remains close to constant, with a sudden increase when an attack is detected and the past flows have to be retrieved and processed.

5 Conclusions

In this paper we have introduced the concept of retroactive dynamic parametrization. In dynamic parametrization, proposed in QEA and Lola2.0, a new monitor (which is an instance of a generic monitor) is instantiated the first time a parameter is discovered. In retroactive dynamic parametrization the decision to instantiate a dynamic parametric monitor can be taken later in the future, for example when a given parameter is discovered to be interesting.

Effectively implementing retroactive parametrization requires the ability to revisit the history of the computation, a task that can be efficiently implemented with a logging system. Therefore, retroactive parametrization allows a fruitful combination of offline and online monitoring. Retroactive parametrization also allows monitors to be created in the middle of an execution without requiring to process the whole trace from the beginning.

We have implemented this technique in HLola and empirically evaluated its efficiency, illustrating that it can efficiently detect distributed denial of service attacks in realistic network traffic.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Proc of the 18th Int'l Symp. on Formal Methods (FM'12). LNCS, vol. 7436, pp. 68–84. Springer (2012)
2. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5, https://doi.org/10.1007/978-3-319-75632-5
3. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Proc. of the 26th Int'l Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06). LNCS, vol. 4337, pp. 260–272. Springer (2006). https://doi.org/10.1007/11944836_25
4. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. Theoretical Computer Science **631**, 118–138 (2016). https://doi.org/10.1016/j.tcs.2016.04.019, http://dx.doi.org/10.1016/j.tcs.2016.04.019
5. Ceresa, M., Gorostiaga, F., Sánchez, C.: Declarative stream runtime verification (hLola). In: Proc. of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20). LNCS, vol. 12470, pp. 25–43. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_2
6. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: Proc. of the 21th Brazilian Symp. on Formal Methods (SBMF'18). LNCS, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
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: Proc. of the 12th Int'l Symp. of Temporal Representation and Reasoning (TIME'05). pp. 166–174. IEEE CS Press (2005). https://doi.org/10.1109/TIME.2005.26
8. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Proc. of the 15th Int'l Conf. on Computer Aided Verification (CAV'03). LNCS, vol. 2725, pp. 27–39. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_3
9. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16). LNCS, vol. 10012, pp. 152–168. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_10
10. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR **abs/1711.03829** (2017)
11. Goodloe, A.E., Pike, L.: Monitoring distributed real-time systems: A survey and future directions. Tech. rep., NASA Langley Research Center (2010)
12. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18). LNCS, vol. 11237, pp. 282–298. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_16
13. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Proc. of the 27th Int'l Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21). Part II. pp. 349–356. LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_18
14. Gorostiaga, F., Sánchez, C.: Nested monitors: Monitors as expressions to build monitors. In: Feng, L., Fisman, D. (eds.) Runtime Verification. pp. 164–183. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_9

15. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the Striver language. International Journal on Software Tools for Technology Transfer **23**, 157–183 (2021). https://doi.org/10.1007/s10009-021-00605-3, `https://link.springer.com/article/10.1007/s10009-021-00605-3`
16. Gorostiaga, F., Sánchez, C.: Monitorability of expressive verdicts. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods. pp. 693–712. Springer International Publishing, Cham (2022)
17. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: Runtime verification of non-synchronized real-time streams. In: Proc. of the 33rd ACM/SIGAPP Symposium on Applied Computing (SAC'17). pp. 1925–1933. ACM Press (2018). https://doi.org/10.1145/3167132.3167338, `https://dl.acm.org/doi/10.1145/3167132.3167338`, track on Software Verification and Testing Track (SVT)
18. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)
19. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, New York (1992)
20. Pedregal, P., Gorostiaga, F., Sánchez, C.: Retroactive parametrized monitoring (2023). https://doi.org/10.48550/arXiv.2307.06763, `https://doi.org/10.48550/arXiv.2307.06763`
21. Perez, I., Dedden, F., Goodloe, A.: Copilot 3. Tech. Rep. NASA/TM–2020–220587, NASA Langley Research Center (April 2020)
22. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Proc. of the 1st Int'l Conf. on Runtime Verification (RV'10). LNCS, vol. 6418, pp. 345–359. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_26
23. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77),. pp. 46–67. IEEE CS Press (1977)
24. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: 14th International Symposium on Formal Methods (FM'06). LNCS, vol. 4085, pp. 573–586. Springer-Verlag (2006)