LarvaStat: Monitoring of Statistical Properties

Christian Colombo, Andrew Gauci, and Gordon J. Pace

Department of Computer Science, University of Malta, Malta

Abstract. Execution paths expose non-functional information such as system reliability and performance, which can be collected using runtime verification techniques. Statistics gathering and evaluation can be very useful for processing such information for areas ranging from performance profiling to user modelling and intrusion detection. In this paper, we give an overview of LarvaStat — a runtime verification tool extending LARVA [2] with the ability to straightforwardly specify real-time related statistical properties. Being automaton-based, LarvaStat also makes explicit the overhead induced by monitoring.

1 Introduction

Runtime verification tools mainly focus on the analysis of system traces for the verification of functional aspects of the system. However, system executions are also rich in information related to non-functional system properties, such as system security, dependability and performance. LarvaStat extends the runtime verification tool LARVA [2] with the capability of collecting statistical information, and verifying non-functional requirements based on such statistics. Taking security as an example application area, LarvaStat allows for the characterisation of suspicious user behaviour through statistical evaluation, which can subsequently be used to blacklist users deemed suspicious. This mechanism has been applied to develop an intrusion detection system based on techniques presented in [4] and an integrated system profiler used for measuring system performance.

LarvaStat's statistical constructs are based on the notion of *incrementally computable statistics* [5], characterising a class of statistics which can be efficiently evaluated in both time and space. An incrementally computable statistic involves (i) storing the current statistic's valuation, and (ii) executing an update function when a new value is to be added to the input data set. Many statistics such as the count, average, maximum, minimum and variance all admit incrementally computable behaviour, although others, such as the median, do not.

All statistics LarvaStat collects are themselves exposed to the monitoring tool as *statistical events* — exposing the latest statistic valuation upon each update. This allows for (i) writing specifications based on these events (such as blocking users upon the statistic valuation exceeding a certain threshold); and (ii) the specification of *multilayered statistics* — statistics over statistics, such as the mean of the maximum download file size.

Moreover, it is often the case that statistics are required to be collected only for certain subtraces. For example, a statistic intent on counting the number of bytes sent during some communication is only interested from the moment of opening to that of

G. Rosu and O. Sokolsky (Eds.): RV 2010, LNCS 6418, pp. 480–484, 2010.

closing of a communication channel. LarvaStat allows for the specification of *intervals of interest* for statistics.

2 LarvaStat

LarvaStat is an event-driven runtime verification framework, and is concerned with interpreting observed event information. Parametrised events can be either observable system actions (such as a method call), timer events, automata-generated events or a combination thereof.

Definition 1. *Given a set* basicevent *of basic events (parametrised over a set of values V) and set* timer *of timer variables, we define a composite parametrised event* event *as either (i) a basic event, (ii) a timeout on a timer (over $\mathbb{R}$), (iii) a choice between events through the general choice operator $\sum$, or (iv) the complement of an event ($\overline{e}$).*

$$\text{event} ::= \text{basicevent} \mid \text{timer} \ @ \ \delta \mid \sum 2^{\text{event}} \mid \overline{\text{event}}$$

The first statistical construct is the *statistic aggregator*, and is defined as (i) an initial statistic valuation (eg. initialising the count), and (ii) an update rule (eg. incrementing the count upon the occurrence of an event).

Definition 2. *A statistical aggregator ranging over Γ is defined through (i) the initial memory value $\gamma_0 \in \Gamma$; and (ii) the update rule entailing a parametrised event (triggering the update), a condition (acting as an event filter), an update function on the memory, and an event on which to signal the updated value. We assume that values over Γ can be mapped to V to pass the value over the output event.*

$$event \times V \rightarrow (\text{cond} \times (\Gamma \rightarrow \Gamma)) \times event$$

Note that cond *stands for a condition on the system state and timer configuration.*

Given a sequence of timestamps, basic events and system states $(t_i, e_i(v_i), \theta_i)$ (with i ranging from 1 to n) and statistical aggregator with initial memory γ_0 and update action (in, s, out), statistical events would be triggered at each point in the trace where a basic event $e_i(v_i)$ triggers in and such that the condition is triggered — c holds, where $(c, u) = s(v_i)$. At each such position, γ (starting with value γ_0) is updated to $u(\gamma)$, with the result being output as an event: $out(u(\gamma))$. Formal definitions of trace semantics of event triggering are given in [2].

Example 1. A statistical aggregator counting the number of bytes sent requires memory storage containing the current amount, and is initialised to 0. The statistic is updated on each basic even *send(v)* (v represents the number of bytes sent), with the update action defined as: $(send, \lambda v.(\lambda x.true, \lambda n.n + v), result)$.

See Fig. 1(a) for an example *point statistic* written in LarvaStat, specifying a statistic aggregator for counting the number of successful user logins.

LarvaStat also supports statistics evaluated over intervals of interest, defined as a statistic aggregator and an interval characterisation. This interval dictates which system trace subsequence is relevant to the specified statistic aggregator. Intervals are characterised by identifying the opening and closing events (eg. an interval specifying the

opening and closing of a connection channel). Through the use of timers, one can use this approach to define intervals by giving the opening event and the duration of time during which to calculate the statistic (eg. a statistic counting the number of user downloads in the first thirty minutes of logging in).

Definition 3. *Statistics aggregation over an interval of interest is defined as (i) a statistic aggregator; (ii) the event and condition marking the interval opening* event $\times V \rightarrow$ cond*; and (iii) the event and condition marking the interval closing* $V \rightarrow$ (event $\times V \rightarrow$ cond).

Note that the closing event is also parametrised by the parameter given to the opening event. Every time an opening event (satisfying the condition) is triggered, a new statistic aggregator is created and initialised, which continues calculating the value until the closing event appears.

Example 2. A statistical aggregator over interval of interest evaluating the number of bytes sent on a per connection basis is defined through (i) the statistic aggregator defined in example 1, (ii) interval opening *(openConnection,*λport.true*)*, and (iii) interval closing $\lambda port_0.(closeConnection, \lambda port_1.port_0 = port_1)$. Note that it is assumed that both *openConnection* and *closeConnection* are parametrised by the port number.

See Fig. 1(b) for an example *interval statistic* written in LarvaStat, specifying the statistic aggregator over interval of interest defined above (ignoring port number to simplify presentation).

3 Case Study

LarvaStat has been used for implementing a probabilistic intrusion detection and integrated system profiler sitting above an ftpd server implemented in Java[1]. The system profiler is responsible for quantifying system performance, whereas the intrusion detection system observes user behaviour, with the aim of capturing suspicious behaviour through the use of misuse detection and anomaly detection techniques [4]. Moreover, given that the monitoring of users is expensive, an additional mechanism has been implemented for the probabilistic choice of users to monitor. This choice is dependent on two parameters: *user risk factor* and *system load*. Both parameters are extrapolated from statistical information collected by LarvaStat.

System profiling is carried out by quantifying the current system load (assuming that the server's performance is tightly bound to bandwidth usage and the current count of logged in users), and analysing system load history for predictive purposes. For example, counting the number of currently logged in users is specified through three statistics, as seen in Fig. 1(a). *UsersLoggedIn* counts the number of user logins, *UsersLoggedOut* counts the number of log out events, while *CurrentUserCount* is a layered statistic which listens to the previous two statistics.

The intrusion detection uses various techniques, (i) a Markov chain analysing the user's command sequence, with each ftpd command being related to a risk factor, and

[1] http://www.anomic.de/AnomicFTPServer

marking the user as suspicious if the command sequence exceeds a threshold; and (ii) the use of statistical moments for the characterisation of abnormal user behaviour, monitoring each user's download and upload behaviour patterns, and assuming a statistically predictable pattern.

```
POINTSTAT UsersLoggedIn : Integer {
  INIT {UsersLoggedIn.setValue(new Integer(0));}
  EVENTS {successfulLogin()}
  UPDATE {UsersLoggedIn.setValue(
          UsersLoggedIn.getValue() + 1);}
}
POINTSTAT UsersLoggedOut : Integer {...}
POINTSTAT CurrentUserCount : Integer {
    INIT{CurrentUserCount.setValue(
                        new Integer(0));}
    EVENTS{ UsersLoggedIn_Event |
           UsersLoggedOut_Event }
    UPDATE{ CurrentUserCount.setValue(
        UsersLoggedIn.getValue() -
        UsersLoggedOut.getValue()); }}
```

```
INTERVALSTAT byteCount : Integer {
 INIT{byteCount
        .setValue(new Integer(0));}
 EVENTS{sendInfo}
 CONDITION{ }
 INTERVAL {
   OPEN  [ downloadStarting ]
   CLOSE [ downloadComplete ]
 }
 UPDATE{ byteCount.setValue(
           byteCount.getValue()
                   + bufferSize); }}
```

Fig. 1. LarvaStat statistic construct examples (a) and (b)

Fig. 2 shows automata (which are processed by LARVA) which are automatically generated by LarvaStat to calculate the statistics *UsersLoggedIn* and *byteCount* from the description in Fig. 1. Transitions are tagged by the event which fires them, the event which they fire, and the action to update the statistic. The initial state is tagged with the action to initialise the statistic. LarvaStat does not extend LARVA's expressivity, but rather is a syntactic sugar for the intuitive high-level specification of statistical properties.

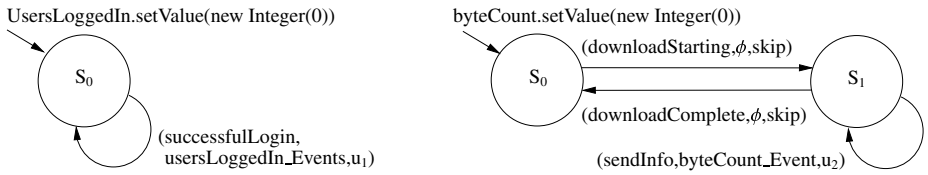

Fig. 2. DATEs executing statistical constructs (a) and (b)

The case study contains the specification of twenty statistics, some of which are evaluated on the system, whereas others are evaluated on a per user basis. All statistics are incrementally computable and intuitively defined, while also being implemented without altering a line of the underlying ftpd system code.

System overhead was measured by simulating multiple users logging in concurrently and exhibiting varied download and upload behaviour patterns. This setting was run multiple times with and without the additional intrusion detection system, whereby the system on average exhibited approximately a 9% processing overhead while being monitored.

4 Related Work and Conclusions

Three existing related approaches have been identified. The approach in [5] specifies a framework focusing on the asynchronous collection of statistics over runtime executions. This is achieved by presenting a Linear Temporal Logic extension focused on evaluating numerical queries on the trace, and admits a tractable evaluation strategy given complete trace knowledge. Lola [3] is another tool, and presents a functional stream computation language allowing for the expression of system properties, numerical queries as well as guaranteeing bounded memory requirements. EAGLE [1] is a third approach, whose use of meta operators allows for the encoding of multiple formalisms such as interval temporal logics, finite state automata, as well as logics for the expression of numerical queries. Our approach supports real-time related statistics collection, and enables interval masking over traces. The automaton-based approach, also makes explicit the overhead induced by monitoring over and above that due to statistics state storage and update.

LarvaStat shows the potential of applying runtime verification techniques for the collection of non-functional metrics about the system being monitored, which can then be used to verify properties over these metrics. By extending an existing runtime verification tool, the resulting framework is able to both collect statistics over system executions, as well as monitoring system properties quantified through statistical queries. The ftpd case study, shows the applicability of the approach, by adding probabilistic intrusion detection and a system profiler to an existing tool.

References

1. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
2. Colombo, C., Pace, G., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009)
3. 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), pp. 166–174 (2005)
4. Denning, D.E.: An intrusion-detection model. IEEE Transactions on Software Engineering 13, 222–232 (1987)
5. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.B.: Collecting statistics over runtime executions. Electr. Notes Theor. Comput. Sci. 70(4), 36–55 (2002)