View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio della ricerca- Università di Roma La Sapienza

Automatic Invariant Selection for Online Anomaly Detection

Leonardo Aniello[1], Claudio Ciccotelli[1], Marcello Cinque[2], Flavio Frattini[2,3], Leonardo Querzoni[1], and Stefano Russo[2]

[1] Università di Roma Sapienza - Rome, Italy
{aniello, ciccotelli, querzoni}@dis.uniroma1.it
[2] Università degli Studi di Napoli Federico II - Naples, Italy
{macinque, sterusso}@unina.it
[3] RisLab - Research and Innovation for Security Lab - Naples, Italy
flavio.frattini@rislab.it

Abstract. Invariants are stable relationships among system metrics expected to hold during normal operating conditions. The violation of such relationships can be used to detect anomalies at runtime. However, this approach does not scale to large systems, as the number of invariants quickly grows with the number of considered metrics. The resulting "background noise" for the invariant-based detection system hinders its effectiveness. In this paper we propose a general and automatic approach for identifying a subset of mined invariants that properly model system runtime behavior with a reduced amount of background noise. This translates into better overall performance (i.e., less false positives).

1 Introduction

Anomaly detection techniques based on the usage of invariants have long been introduced to discover anomalous behaviors in processing systems [1, 2]. An invariant is a property of a system that is expected to hold while the system runs correctly. The idea of invariant-based anomaly detection is that it is possible to automatically analyze the evolution of the system at runtime to identify stable correlations among some monitored metrics. Such a detection process involves an initial training phase to learn invariants representing the correct behavior of the system. Then, whenever an invariant is *broken* or *violated* during operation—i.e. the underlying correlation between metrics is lost—it is considered a sign of a probable malfunction in the system.

The use of invariants is gaining interest in the field of systems where faults may have severe impacts. These systems are characterized by a great complexity that, on one side, increases the possibility of malfunctioning and, on the other side, hampers the adoption of classic fault detection techniques based on design-time modeling of normal operation conditions [3]. The practical adoption of invariants for anomaly detection is limited by their sensitivity to the number and quality of monitored system variables, however. A moderately complex system may expose hundreds of invariants, and only a subset of them stably captures

its correct behavior, while a large fraction are either useless (because not linked to malfunctions) or excessively unstable (e.g., they are easily broken even if the system behaves correctly). Blindly monitoring all mined invariants introduces noise and fluctuations in the detection output, which create false positives; [4] this hampers the practical usability of this technique. Currently, there is no clear approach to *"filter"* the invariants space to get rid of such unwanted effects.

In this paper we present a practical and repeatable approach to analyze a (possibly very large) set of mined invariants to automatically select a *core subset* of them that properly captures the correct runtime system behavior, while showing a good degree of insensitivity to exogenous factors not linked to malfunctions. The approach exploits information provided by both the correct and the anomalous behavior of the system.

The approach is evaluated in a testbed equipped with a real web-based application (e.g., a web-banking portal) where we inject faults (from a defined fault model), to force the anomalous behavior. Results show the effectiveness of the proposed invariant selection approach, especially in reducing false positives.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 introduces our invariant selection approach; Sections 4 and 5 present a case study based on a real application scenario and discuss the results of applying our approach to it. Finally, Section 6 concludes our work.

2 Related Work

A.B. Sharma et al. [4] proposed to use invariants to detect faults in distributed systems: a mining tool is described, and mined invariants are used for the detection. Their application can then be extended to support log analysis [2, 5]. In [6] automatically mined invariants are used for online anomaly detection in a cloud-based processing system.

Invariants can be classified [7] in control-flow, execution-flow, and value-based. In this paper, we focus on an extension of value-based invariants, known as *flow intensity invariants*. They have been introduced to measure the intensity with which internal monitoring data, treated as *time series*, react to the volume of user requests. In general, time series may be mined from system/application logs and resources utilization data through common monitoring tools. Hence, the approach does not depend on the particular system under monitoring.

A flow-intensity invariant is commonly selected among all the combinations of the collected metrics by estimating its ability in describing a phenomenon. As an example, in [1] an invariant is built when two measurements are available; then, it is incrementally validated when new observations are available. If, after a certain number of measurements, a confidence score of the model is less than a threshold, the invariant is discarded. However, this approach does not scale with

[4] A *false positive* is an error in the detection, in which an anomaly is reported when no anomalies occurred. A *false negative* is an omission of the detector, which does not report an occurred anomaly.

the size of the system, and generates a "background noise" of broken invariants that undermines its efficiency for anomaly detection.

For these reasons, [8] introduced a filtering stage where it is estimated the probabilitythat an invariant would have been mined when considering a random input: if this probability is larger than a certain threshold, the invariant is selected, otherwise it is filtered out. Another approach consists in considering the number of times an invariant is violated [2]. However, as discussed in the remainder, also invariants broken too often should be treated with care: if an invariant is easily violated, it may be useful for detection completeness, but it may also generate many false positives, thus negatively affecting the accuracy.

Differently from such approaches, we introduce an *automatic* filtering stage, identified as *filtering 2* hereafter. It is based on both correct and anomalous runs of the system, instead of only considering correct executions, as for the commonly adopted selection procedure, which we identify as *filtering 1*.

3 Approach

The invariant-based approach we propose (Figure 1) is based on three steps: 1) Mining, 2) Automatic Filtering, and 3) Detection. The *invariants mining* step consists in the analysis of data characterizing the *correct system behavior* to identify invariant relationships between pairs of observed variables. Step 2 consists in the automatic *filtering* of found invariants in order to extract a subset of them that can be usefully exploited in step 3 for the *detection* of anomalies.

Existing invariant-based detection approaches, such as the ones proposed in [5] and [6], only consider a training dataset representing the correct behavior of a system to be used for the mining and the filtering is based on goodness of fit (*filtering 1*). This way, many invariants are mined. In this paper, we introduce a further filtering step (*filtering 2*), which also considers known faulty behaviors. To identify the invariants that are potentially good symptoms of anomalies, we consider a further dataset representative of the system when faults are activated. Thus, given the fault model for the considered system, the idea is to inject instances of such faults in the system (for details, see Section 4) in order to

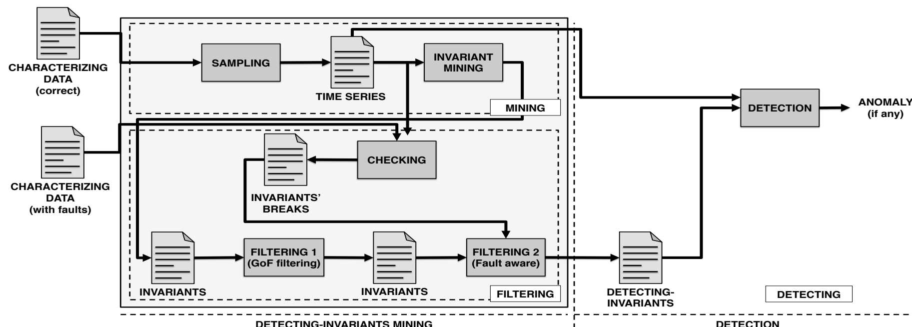

Fig. 1: Approach for invariant mining and filtering for anomaly detection.

collect the data that characterize the faulty behavior and that are then used to check when invariants are actually violated. The following sections describe the details of the three steps.

3.1 Invariant mining

The first step consists in sampling the available data in order to have time series for mining invariants. As characterizing data, we consider data collected from the monitoring of the processing system, i.e., related to resources' utilization, such as CPU use, memory use, network packets, etc. This makes the approach generic and not dependent on the specific system workload. A time series is a sequence of values corresponding to measurements of parameters, uniformly spaced, with a certain sampling time, over a time interval. Thus, a time series is a function f over a domain of real numbers $\mathbb{R}$ and of a discrete time argument $t \in T$, $f{:}T \rightarrow \mathbb{R}$. Collected data may require some manipulation in order to have all the time series with the same sampling time. The selection of the sampling time is important for the results of the detection [5]. In our case, examples of considered time series are $f_1(t) =$*cpu_system_metric* and $f_2(t) =$*proc_run_metric* representing the use of the CPU in non user mode and the number of running processes, respectively. Observations of the time series at different times results in a relation as $f_2(t) + a_1 f_2(t-1) + \cdots + a_n f_2(t-n) = b_0 f_1(t-k) + \cdots + b_m f_1(t-k-m)$; by considering the vectors of coefficients and samples $\theta = [a_1, \ldots, a_n, b_0, \ldots, b_m]^T$ and $\varphi(t) = [-f_2(t-1), \ldots, -f_2(t-n), f_1(t-k), \ldots, f_1(t-k-m)]^T$, we have $f_2(t) = \varphi(t)^T \theta$. For the parameters estimation, and thus for the mining process, we use the least squares method, as described in [5].

3.2 Automatic Filtering

Filtering operations aim to improve the detection by removing redundant and/or inaccurate invariants, e.g., the ones that break either too often, leading to a large number of false positives, or too seldom, generating false negatives. They consist of three phases: *checking*, *filtering 1*, and *filtering 2*, described in the following.

Checking phase is used to verify when invariants are broken if faults are injected. Thus, apart from monitored data related to the correct behavior, also collected data of anomalous behaviors are used. In this case, as during the operational phase of the system, a set of time series is used as input. The assessment of broken invariants is discussed in Section 3.3.

The second part of the filtering step is made up of two phases. The goal is to filter those invariants that are not actually able to detect anomalies. Such filtering operations are performed on vectors associated to each invariant and reporting their behavior in an observation period. Considering the time lapse $t_0 \ldots t_{n-1}$, for each invariant i_j, we consider a vector v_j of size n, where n are the instants of observation, and

$$v_j[k] = \begin{cases} 0, & \text{if } i_j \text{ is not broken at } t_k \\ 1, & \text{otherwise} \end{cases}$$

Filtering 1 phase is based on the *Goodness of Fit* test. It considers only the correct behavior of the system and is also used in [5, 6]. The basic idea is to remove invariants with a goodness of fit (GoF) outside a specific range. Clearly, invariants with a low GoF are invariants that do not provide a good modeling, i.e., are not able to properly describe system behavior. Conversely, invariants with a too large GoF are likely mined from too similar time series and are thus meaningless. Consider, as an instance, an invariant relating CPU use time *CPU_util* and CPU idle time *CPU_idle* (*CPU_util*=1-*CPU_idle*). The two time series are linearly dependent, thus, the resulting invariant has a very large GoF and is always verified, but it is useless. To discard invariants by testing the quality of their fitting, we use the Coefficient of Determination R^2, which represents the percentage of the variation that can be explained by the model. The closer the value of R^2 to 1, the better the regression.

Filtering 2 phase uses a dataset representative of the system when anomalies occur, unlike the existing approaches based only on the system correct behavior. This dataset is obtained by means of fault injection (see Section 4). This is the main novelty of the invariant selection strategy we propose, aiming at removing invariants that may provoke erroneous evaluations, i.e., false positives and false negatives. This phase includes the following five filtering operations.

1) Never-broken invariants filtering. We remove invariants that are never broken. As a matter of fact, there could be a relation between two time series that is also able to well describe the variance of the system, but the relation always holds. Thus, it is not useful to detect anomalous events.

2) Correlated invariants filtering ($Corr_{th}$). When applying the GoF filtering, invariants relating to similar time series are removed. There could still be invariants representing similar relations and that are broken at the same time. Previously, we considered the example of *CPU_util* and *CPU_idle* measurements; the invariant relating them is filtered; but, let now also consider memory utilization *Mem_util*; if there is an invariant $Mem_util = \alpha CPU_util$, there will also be an invariant $Mem_util = \alpha' CPU_idle$. Nevertheless, those invariants provide the same information, and considering both of them would be redundant. By considering the vectors v_j associated to all the invariants, we consider all the possible pairs of invariants $\langle i_i, i_j \rangle$ and compute the correlation by means of the Pearson's correlation index. If the correlation is larger than a threshold $Corr_{th}$, invariant i_j is removed and only invariant i_i is considered.

3) Too-often broken invariants filtering (Oft_{all}). Some invariants may be too weak, i.e., they are broken too often and, similarly to never-broken invariants, are not useful to detect different-than-common behaviors that are likely related to anomalies' occurrence. We filter invariants that are broken more than Oft_{all} times the average number of times all the invariants are broken, in the examined lapse of time.

4) Invariants broken before injection filtering. Some invariants may be broken before a fault is injected, when the system behavior is expected to be correct. Consider an invariant that is not broken too often (thus, it is not filtered at *step 3*) but it is always broken before an injected fault is activated, i.e., before

the anomaly occurs. This invariant would report not occurred anomalies, thus generating false positives, so we filter all the invariants that are broken when no anomaly is occurring in the system.

5) Seldom broken invariants fault-specific filtering (Sld_{spec}). Invariants that are seldom broken for a specific fault are filtered. Even though at *step 3* invariants broken too often are filtered, the remaining ones should break often enough to detect the forced anomaly and avoid false negatives. We remove invariants that are broken less than Sld_{spec} times the average number of times the invariants are broken, in the examined lapse of time, for a specific injected fault.

3.3 Detection

A detector implements a distance function δ, which evaluates at runtime the distance of the actual system behavior from the expected one, and consider a threshold τ that, when exceeded by the value of δ, triggers an alarm.

At a time t, with respect to a specific system parameter $\hat{\theta}$ and an input $f_1(t)$, the detector has to compute the distance of the actual response of the system $f_2(t)$ from the estimated response $\hat{f}_2(t|\hat{\theta})$. As in [6], we adopt the residual function as a distance function: $R_{f_1,f_2}(t) = |f_2(t) - \hat{f}_2(t|\hat{\theta})|$. Thus, an invariant is broken at time t, if $R_{f_1,f_2}(t) > \tau$, where τ represents the tolerance of the detection system.

On the selection of the value of τ heavily depends the results of the detection, as discussed in [5]. As a matter of fact, if considering a detector with $\tau = 0$, invariants would be broken too often, generating many false positives; on the contrary, a too large τ would reduce too much the number of breaks, and too false negatives would take place. In [5], it is shown that when adopting a threshold τ which adapts to the specific prediction, the detection appears both complete and accurate. Specifically, we consider the prediction interval (p.i.) of the output with respect to the provided input [9]: the invariant is considered broken if the actual output of the system is outside the p.i. of the model's output. This happens if, for a certain value $f_1(t)$, the difference between the actual value $f_2(t)$ of the system and the estimated value $\hat{f}_2(t|\hat{\theta})$ is larger than the standard deviation of $\hat{f}_2(t|\hat{\theta})$. The standard deviation is computed as:

$$\sigma = S_{err}[1 + \frac{1}{n} + \frac{(f_{1_p} - \bar{f}_1)^2}{\sum f_1^2 - n\bar{f}_1^{\,2}}]^{1/2} \tag{1}$$

where S_{err} is the standard deviation of the model error (square root of the mean squared error), $\bar{f}_1$ is the sample mean of the predictor variable (the input of the model), and f_{1_p} a specific value of f_1.

4 Case Study

In order to evaluate the feasibility and performance of the proposed approach for invariant selection we set-up a testbed and deploy on top of it a web-based

application with the aim of mimiking a typical online service offered, for example, by a bank to its customers. We monitor the system to collect time series related to several metrics. Different workloads are submitted to the system, to collect data from different load conditions. The testbed is used to produce both a *training set*, to be used for invariant mining, and a *training-test set*, for the assessment. Time series are collected for both *correct executions*, i.e., executions where no anomalies occur, and *faulty* ones, where one or more anomalies take place as a consequence of injected faults.

Testbed — The testbed is composed of 4 servers, each equipped with an Intel Xeon X5560 Quad-Core CPU clocked at 2.28 GHz and with 24 GB of RAM. The system presents a standard 3-tier architecture with one of the servers (master) hosting a centralized load balancer based on the Apache 2 Web Server and mod_cluster 2.6.0 module. The business tier hosts a JBoss AS 7.1.1.Final cluster running an instance on each machine, one master co-located with the Web Server and three slaves in execution on each of the other servers. The storage layer is based on a single instance of MySQL 5.5.38 running on the master node. On top of the JBoss cluster, we deployed a web application [10] working on both the business tier, with a front-end web application, and on the storage tier, interacting with the database. The testbed is monitored by means of the Ganglia monitoring system.

Workload — We generate workloads from a fifth machine running Tsung 1.5.0 [11]. The load consists in the number of requests per second sent to the web application. Each request involves the generation of ∼400 packets in the testbed. We consider three load levels, in order to cover several operational conditions: *low*, *medium*, and *high*. By means of a preliminary analysis, we identify the *high* level, which uses almost all the resources of the system. *medium* and *low* levels are selected by considering a load that is 2/3 and 1/3 of the *high* level, respectively. The three workload levels are generated from a normal distribution by varying the mean (μ) and the standard deviation (σ) of the *connections per second*. Specifically, we consider *low* with $\mu = 5$, $\sigma = 1$, *medium* with $\mu = 10$, $\sigma = 2$ and *high* with $\mu = 15$, $\sigma = 2$

Faultload — Candidate faults for injection are selected to include those that *(i)* are often the cause of problems in real settings, especially after changes to the deployment setup (e.g. where the deployment of a new application version or the reconfiguration of an existing one may trigger some of such faults), and *(ii)* that can not be easily detected through basic monitoring tools (e.g., a crashing process that leaves a debug trace in some log). We consider faults related to actual anomalies that can occur in processing systems and identify them on the basis of both our direct experience on real operational datacenters, and information drawn from scientific literature and online resources. The defined fault model is also compliant to the well known and widely adopted taxonomy defined by Avizienis et al. in [12]. In particular we considered the following faults:

– *Misconfiguration faults* that derive from human errors caused by the wrong configuration of a system. Configuration errors are both common and highly detrimental, and detecting them is desirable [13]:

Table 1: Invariant filtering parameters considered as *factors*.

Factor	**Level 1**	**2**	**3**	**4**	**5**	**6**	**...**	**11**	**12**
Oft_{all}	0.1	0.2	0.4	0.6	0.8	1.0	...	2.0	4.0
Sld_{spec}	0.1	0.2	0.4	0.6	0.8	1.0	...	2.0	4.0
$Corr_{th}$	0.70	0.75	0.80	0.85	0.90	0.95			

  - *SQL misconfiguration*: we reduce significantly the connection pool used by the application server to connect with the DB.
  - *AJP-long misconfiguration*: we reduce the thread pool for the AJP protocol (which allows the communication between the Apache web server and the JBoss slaves) to a very small size.
  - *AJP-short misconfiguration*: same as AJP-long misconfiguration, but we also reduced the length of the queue associated with the thread pool.
- *Reconfiguration faults*, representing changes of configuration during maintenance that cause unexpected failures [14, 15]:
  - *Write permissions*: we revoke write permissions to one of the JBoss instances on its working directory.
- *Denials of service faults*, either malicious or not, that cause the system unavailability due to the saturation of some hardware resources:
  - *CPU stress*: we impose an abnormal CPU load on the target machine by running a strongly CPU-intensive task.
  - *Memory stress*: we impose an abnormal level of memory activity that causes high memory contention on the target machine.
  - *Disk stress*: we cause an abnormal disk access activity on the server hosting the SQL server.
  - *Full partition*: we cause the disk partition on the machine hosting Apache and the SQL server to become full.
- *Development faults* that typically produce erratic output or software aging phenomena [16]:
  - *Memory Leak*: we run a process affected by memory leak that causes the memory of the target machine to progressively saturate. The memory exhaustion in turns triggers the thrashing phenomenon.

Plan of experiments — The implemented testbed and the planned injections allow us to obtain both *correct executions*, i.e., executions where no anomaly occur, and *faulty executions*, where, from a certain time t_a, one or more of the considered faults are injected. These executions are used to produce both a **training set**, to be used for mining and filtering the invariants, and a **test set** to be used for assessing performance. Each execution for the training set lasted on average $\sim$ 9 minutes (time needed to reach a steady state, where metrics can be collected while excluding impact from any transient effect). For the test phase, we produced a single 90 minutes long test set were the system transitions among all the possible combinations of workloads and faults.

Filtering operations are performed on the training set by considering several values for the filtering parameters introduced in Section 3, considered as *factors*:

$Corr_{th}$, for filtering correlated invariants, Oft_{all}, for removing too-often broken invariants, and Sld_{spec}, for seldom broken invariants fault-specific filtering. Table 1 shows the values used in our tests. We consider a design of experiments (DoE) [9], where the factors are the filtering parameters. The *levels* (i.e., the values assigned to the factors) are in a wide range to cover several cases. As response variables, we consider common metrics for detection assessment. *Coverage* (Cov) is the portion of kinds of anomalies that are found by a detector. If n kinds of anomalies occur, and the detector finds r of them, $Cov = r/n$. *Completeness* (Cpl) is the portion of anomalies that are found by the detector over the occurred anomalies. If o anomalies occur, the detector may find p of such anomalies, with $p \leq o$; $Cpl = p/o$. *Accuracy* (Acc) is the portion of anomalies correctly reported by a detector. A concrete detector finds s anomalies over $s' \leq s$ actual anomalies. $Acc = s'/s$. *Detection latency* (Lat) is the time required by the detector to report the occurrence of an anomaly.

5 Results

In this section, we present results related to the application of the approach to the training set and to the test set, and compare them to the common approach of invariant mining that only filters by considering the GoF (*Filtering - Goodness of Fit filtering*, discussed in Section 3.2). Also, we discuss how the configuration of the filtering influences detection performance.

5.1 Training

The mining and filtering steps have been applied on the training set in order to identify the invariants and find the best configuration of filtering parameters for the system at hands. As a result of the plan discussed in Section 4, the experimentation produces 1,176 outputs for each response variable; to evaluate the anomaly detection performance, we also consider the F-measure (F), defined as the harmonic mean of completeness and accuracy: $F\text{-}measure = (2 \cdot Cpl \cdot Acc)/(Cpl + Acc)$. The larger the completeness and accuracy (ideally, $Cpl = 1$ and $Acc = 1$), the better the detection quality of the detector, since it avoids false positives and false negatives.

To identify the values to be used for the filtering parameters $Corr_{th}$, Oft_{all}, and Sld_{spec}, we use the Pareto multi-objective optimization algorithm [17]. The algorithm returns a Pareto front with 16 combinations of the configuration parameters. Among such configurations, we consider the one allowing the detector to identify all the anomalies occurring in the system. Invariant-based detection approaches, in fact, are expected to have a large completeness given the large number of invariants that can be violated when anomalies occur [5]. Results with this combination are reported in Table 2. They are achieved for $Corr_{th} = 0.85$, $Oft_{all} = 1.8$, and $Sld_{spec} = 3$.

On the training set, the invariant based approach detects all the kinds of anomalies that occur in the system, and over all the anomalies, of all the kinds, all

Table 2: Results of the training related to the best combination of the filtering parameters. $Corr_{th} = 0.85$, $Oft_{all} = 1.8$, $Sld_{spec} = 3$.

Cpl	*Acc*	*Lat*	*F-m*	# inv
1.00	0.80	33.33	0.89	25

Table 3: Results of the tuned detector applied to the test set. First row: results of the proposed approach. Second row: results without the proposed filtering.

Cpl	*Acc*	*Lat*	*F-m*	# inv
0.99	0.76	65.55	0.86	25
1.00	0.57	47.78	0.73	265

are detected. The accuracy is 80%; thus, there are few false positives. Anomalies are detected within 33 seconds.

We also observe a large reduction of the number of used invariants (around 90% of reduction) after filtering. 78 metrics are monitored, thus, the possible invariants (considering all the combinations) are $3,003$. The GoF filtering selects 265 of these invariants, which involve all the 78 metrics. The proposed filtering reduces the number of used invariants by about 90%, which involves only 19 metrics out of the 78 monitored. In practical terms, this implies a significant reduction of the monitoring overhead.

Due to space limitation, we report as examples of the mined invariants the ones that are often violated when an anomaly occurs: one relates the use of the CPU in non user mode (*cpu_system_metric*) to the number of running processes (*proc_run_metric*), another one relates the average size of incoming packets (*avg_packet_size_metric*) and CPU usage (*cpu_idle_metric*).

5.2 Test

The configuration of the filtering defines the invariants to be used by the detection module that we run on the test set. This allows us to assess the behavior of the detector in the operational stage, when ground truth is available. Achieved results are compared to the ones of the detector using not filtered invariants, and reported in Table 3. The first row of the table reports results related to the detector based on the proposed filtering approach. Results in the second row are related to the detector without filtering.

The comparison of the two detectors, with and without the *fault aware filtering*, shows that the proposed approach outperforms the detector not using the introduced filtering. When no filtering is done, coverage and completeness are maximum. In fact, having a large number of invariants implies there is a large chance that there is a violated invariant, and the anomaly is detected, whatever its kind is. Latency is small due to similar observations: the larger the number of invariants, the sooner one is violated and the anomaly detected. The chance that an anomaly is erroneously signaled is large, however. Among all the invariants,

one may be broken even if there is no anomaly, generating false positives and, then, reducing the accuracy. A result of 0.57 for the accuracy implies that out of 100 anomalies reported by the detector, only 57 have been caused by faults, while the remaining 43 represent false positives.

The adoption of the proposed approach significantly improves the performance of the detector, making the approach practicable in our experimental settings. While the completeness remains high, as expected for invariant-based approaches (as also shown in [6]), the filtering approach proposed in this paper, by selecting the right subset of invariants, improves the accuracy pushing it to 0.76, i.e. reducing false positive to 24% of the reported anomalies. Thus, the performance improves even if less than 10% of the original invariants are adopted (from 265 to 25), hence reducing the overall monitoring overhead. On the other hand, the use of a reduced set of invariants slightly increases the latency to 65s, i.e., the anomaly is detected within one minute from the activation of the fault causing it. Note that, since the effects of the fault may affect the system several seconds after its injection, this is an upper bound of the latency.

Analysis of variance is then used to figure out which filtering parameters mostly impact the results of the detection. Results show that detection mainly depends on Oft_{all} and Sld_{spec} parameters, while the impact of $Corr_{th}$ is not statistically significant. Specifically, the variance of coverage and completeness is explained by Oft_{all} for 55% and by Sld_{spec} for 45%. variance on accuracy is explained by Oft_{all} for 89%.

6 Discussion and Conclusion

Invariant-based detectors discussed in the scientific literature present a number of false positives, given the high chance of invariants being violated, when also negligible conditions change in the system. Presented results demonstrated that the proposed invariants' filtering approach improves the performance of common invariant-based detectors, which remove invariants by only considering their capacity of properly modeling the correct system behavior. The proposed approach exploits knowledge on the faulty behavior of the system to select those invariants that are sensible enough to be violated in the case of anomaly, thus not causing false negatives, but not weak enough to break also when the system is correctly behaving, producing false positives.

The achieved detector outperforms the detector not using the introduced filtering. It covers all the anomalies of the injected kinds, and, over all the occurring anomalies, reveals 99% of them, with an accuracy of 76%. Clearly, reported figures are specific to the case study system, but the proposed approach is general enough to be applied to a wide range of systems. Moreover, while many invariant mining approaches consider application-specific monitored data, this one uses resources' usage information common to every processing system and collectable with any of the existing, free and open source, monitoring tools.

Acknowledgments

This work has been supported by the TENACE PRIN Project (n. 20103P34XC) funded by MIUR. The work by Cinque and Russo has also been partially supported by EU under Marie Curie IAPP grant n.324334 CECRIS (CErtification of CRItical Systems).

References

1. G. Jiang, H. Chen, and K. Yoshihira, “Discovering likely invariants of distributed transaction systems for autonomic system management,” *Cluster Computing*, vol. 9, no. 4, pp. 385–399, Oct. 2006.
2. J.-G. Lou et al., “Mining invariants from console logs for system problem detection,” in *Proc. of the USENIX annual technical conference*, 2010.
3. X. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun, “Pod-diagnosis: Error diagnosis of sporadic operations on cloud applications,” in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, 2014.
4. A.B. Sharma et al., “Fault detection and localization in distributed systems using invariant relationships,” in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, 2013.
5. S. Sarkar, R. Ganesan, M. Cinque, F. Frattini, S. Russo, and A. Savignano, “Mining invariants from saas application logs,” in *Tenth European Dependable Computing Conference (EDCC 2014)*, May 2014.
6. F. Frattini, S. Sarkar, J. Khasnabish, and S. Russo, “Using invariants for anomaly detection: The case study of a saas application,” in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, 2014.
7. S.K. Sahoo et al., “Using likely program invariants to detect hardware errors,” in *IEEE Int.l Conference on Dependable Systems and Networks (DSN)*, 2008.
8. M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, 2001.
9. R. Jain, *The Art of Computer Systems Performance Analysis.* Wiley-Interscience, NY, 1991.
10. Ticket Monster. http://www.jboss.org/ticket-monster/.
11. Tsung. http://tsung.erlang-projects.org/.
12. A. Avizienis et al., “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.
13. J. Zhang et al., “Encore: Exploiting system environment and correlation information for misconfiguration detection,” *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 687–700, Feb. 2014.
14. Rice University - Division of Information Technology, “Why Are My Jobs Not Running?” April 2013, http://rcsg.rice. edu/rcsg/shared/scheduling.html.
15. IGI - Italian Grid Infrastructure, “Troubleshooting guide for CREAM,” April 2013, https://wiki.italiangrid. it/twiki/bin/view/CREAM/ TroubleshootingGuide.
16. A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, “Workload characterization for software aging analysis,” in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, 2011.
17. D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.