An Efficient and Scalable Intrusion Detection System on Logs of Distributed Applications

David Lanoe, Michel Hurfin, Eric Totel, Carlos Maziero

An Efficient and Scalable Intrusion Detection System on Logs of Distributed Applications [⋆]

David Lanoë[1,2], Michel Hurfin[1], Eric Totel[2], and Carlos Maziero[1,3]

[1] Univ Rennes, INRIA, CNRS, IRISA, 35000 Rennes, France
`michel.hurfin@inria.fr`
[2] CentraleSupélec, INRIA, CNRS, IRISA, 35000 Rennes, France
`eric.totel@centralesupelec.fr, david.lanoe@centralesupelec.fr`
[3] Univ Federal Paraná, 81531-980 Curitiba, Brazil
`maziero@inf.ufpr.br`

Abstract. Although security issues are now addressed during the development process of distributed applications, an attack may still affect the provided services or allow access to confidential data. To detect intrusions, we consider an anomaly detection mechanism which relies on a model of the monitored application's normal behavior. During a model construction phase, the application is run multiple times to observe some of its correct behaviors. Each gathered trace enables the identification of significant events and their causality relationships, without requiring the existence of a global clock. The constructed model is dual: an automaton plus a list of likely invariants. The redundancy between the two sub-models decreases when generalization techniques are applied on the automaton. Solutions already proposed suffer from scalability issues. In particular, the time needed to build the model is important and its size impacts the duration of the detection phase. The proposed solutions address these problems, while keeping a good accuracy during the detection phase, in terms of false positive and false negative rates. To evaluate them, a real distributed application and several attacks against the service are considered.

Keywords: Anomaly detection · Distributed application · Models.

1 Introduction

Sensitive applications such as e-commerce or file systems are now running on large scale distributed systems, being prime targets for attackers who can misuse the service, steal information, compromise service availability or data integrity. The usual approach to detect incorrect behaviors in distributed systems consists in monitoring each process with a local intrusion detection system (IDS). Events generated locally are stamped with a global clock and sent to a central correlation engine. This engine orders all events using their timestamps and analyzes their sequence, to detect predefined patterns corresponding to attack scenarios [8]. This approach can only detect known attacks, for which a "signature" of events

[⋆] This work is partially funded by the French Ministry of Defense (DGA).

can be specified. In addition, a global clock is needed to build the total order over all the events, which may be unfeasible in large distributed applications.

Anomaly-based intrusion detection systems look for deviations from a normal behavior reference model; any deviation is interpreted as an observable consequence of an attack. No previous knowledge about the possible attacks is required. In addition, recent works [15] shown that it is possible to build effective models without relying on a precise observation of the events' order. Instead of a total event ordering, weaker but easier to observe partial order relations, like Lamport's "happened before" [7], may be used. In a previous paper [15], we proposed a method to build a behavior model for a distributed application. This method takes as input an execution trace to build a lattice of consistent cuts [4], which is then used for deducing two complementary sub-models: a list of likely invariants and an automaton (a state machine). A single execution of the distributed application may not capture all its correct behaviors; those not observed could be futurely rejected by the model as false positives, during the detection phase. To face this problem, several executions traces are used, generating distinct sub-models of the same application. They may then be merged [1] and generalized [15, 10], to produce models accepting additional behaviors that have not been observed. The main challenge is to build an effective model while addressing scalability (both during the construction of the model and the detection phases). As our ultimate goal is to detect anomalies, effectiveness can be measured in terms of false positive and false negative rates. The time needed to build and update the model is important, specially for longer execution traces. Finally, smaller models enable faster attack detection, which is often a major requirement.

The contribution of this paper is fourfold: 1) As the lattice of consistent cuts grows exponentially in the context of large distributed systems [5], we propose solutions that reduce the computing time and generate a more compact initial model. A judicious selection of a few sequences allows us to obtain more quickly a model that is not less efficient; 2) Existing solutions usually build a huge and complex automaton-based model and then reduce it, during a final generalization phase. We propose an iterative approach where the model under construction is repeatedly merged and generalized; 3) To analyze their impacts on the performance but also on the quality of the detection, the proposed solutions were implemented and evaluated using a real distributed file system and some real attacks against it; 4) finally, we show that the two sub-models (the invariant list and the automaton) are complementary. The paper is organized as follows: Section 2 presents the state of the art and discusses some related work. Section 3 describes our contributions to reduce the time complexity of the construction phase without sacrificing effectiveness. Finally, Section 4 assesses our proposal.

2 State of the Art

It is usual to model the behavior of a single process by the sequence of systems calls it issues or, more generally, by the sequence of actions performed during its computation. In some distributed systems, the existence of a global clock allows us to keep a model based on a single sequence of actions. Otherwise, the analysis has to focus on causal dependency relations between events. Inferring a model of the correct behaviors of an application from a set of traces is an old research topic. Using such a model to design an anomaly detection system is a more recent research area. The behavior model is usually either an automaton [9, 11] or a set of temporal properties [2]. As a model learned from a finite set of traces is possibly incomplete, related work includes also generalization techniques [3, 10]. Scalability issues are important and addressed in [5, 16, 6] in ways different from what is proposed in this paper. Rather than providing separate descriptions of incompatible work (*i.e.* conducted with different objectives), we describe in this section a uniform framework based on the concept of lattice of the consistent cuts [4]. Thus, this framework is closely related to two studies [1, 15]. During our experiments, an implementation of this framework is used as a reference point.

In an anomaly-based detection approach, a first phase (called the construction phase) is executed once to build a reference model that specifies correct behaviors. Then, this model is used during a detection phase to monitor an execution potentially targeted by attacks. Figure 1 illustrates this global process, already adopted in [15]. Three main modules (grey boxes in Fig. 1) are in charge of 1) generating an intermediate model for each learned execution, 2) merging the intermediate models and generalizing the obtained model, and 3) detecting an anomaly. They are described in the following sub-sections.

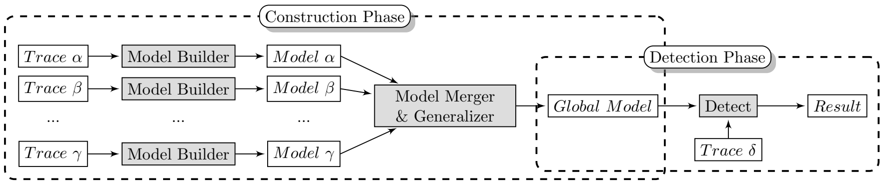

Fig. 1. Overview of the approach

2.1 Building an Intermediate Model from an Execution Trace

During the construction phase, the application is executed a finite number of times (executions α, β, ..., γ) and produces one trace (E^α, E^β, ..., E^γ) per execution. We assume that no attack could occur during the construction phase. The module *Model builder* (see Fig. 1) analyzes each trace separately and generates an intermediate model for each execution. A distributed application consists of n processes (p_1, ..., p_n) running on one or more physical nodes. The number of interacting processes may change from one execution of the application to the other. Therefore, n denotes the maximal number of processes involved during any observed execution. During a computation α, each process produces a sequence of events where an event corresponds to either an internal action or a communication action, namely the sending of a message m on a channel c (denoted $c!m$) or the delivery of a message m received on channel c (denoted $c?m$). The n processes are instrumented to generate log files. During an execution α, each process p_i logs information about its local events in a local log file E_i^α. Thus a trace E^α is a collection of n logs. While events of a same log are totally ordered, events of a same trace are only partially ordered if we consider the well known *happened-before* relation [7], denoted $\prec^\alpha$. We assume that the logs are rich enough to deduce if two events of the trace are causally dependent or not. Either all logged events are stamped with a vector clock or enough information about the communication events is stored, to allow matching the corresponding sending/receiving events in the trace [15]. Based on the causality relation, *consistent cuts* can be defined [4]. A cut is a subset of events. A consistent cut corresponds to a global state that may be reached during the computation.

Definition 1. *$C \subseteq E^\alpha$ is a consistent cut of the distributed computation α if and only if $\forall e \in C, \forall f \in E^\alpha, f \prec^\alpha e \Rightarrow f \in C$.*

The set of consistent cuts of a distributed computation forms a lattice. Let us consider the example of an application that involves two processes p_1 and p_2. In Fig. 2, during an execution α, the space-time diagram shows that 7 events occurred in the following order: $< a;\ b;\ c_2!m;\ c_1!m;\ c_1?m;\ c_2?m;\ b >$. But this order is not observable without a global clock. The corresponding lattice is depicted in Fig. 2. Each point (white circle) corresponds to a consistent cut. A path from the empty set of events (the initial cut at the bottom left) to the whole set of events E^α (the final global state at the top right) corresponds to a total order compatible with the observed partial order. There exist 18 different paths and one of them corresponds to the total order in which the events occurred.

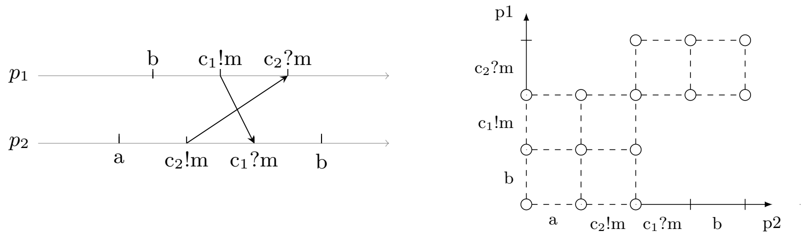

Fig. 2. Space-time diagram of the execution α and its corresponding lattice

An intermediate model, specific to the execution α, is created using the trace E^α. During the construction of the model, each intermediate model (as well as the final one) is dual (*i.e.*, composed of two sub-models). A first sub-model is a list of likely invariants. Assuming that each event has a type (denoted a, b, ...), usual invariants [2, 15] are considered: "a always followed by b" (noted $a \to b$), "b always preceded by a" ($a \leftarrow b$), and "b never followed by a" ($b \nrightarrow a$). To compute the list of invariants, we adopt a technique proposed in a different context [12]. During the construction of the lattice, for each type of event a, we compute

$min(a)$ (respectively $max(a)$) the set of cuts where an event of type a appears for the first time (respectively for the last time) along a path. Each set contains at most n elements and is necessarily a singleton if a is a type of event that can be generated by only one process. For example, during the execution α, $min(c_1!m)$ is equal to $\{\{b,\ c_1!m\}\}$ and $max(c_1!m)$ is equal to $\{\{b,\ c_1!m,\ a,\ c_2!m\}\}$. To obtain the list, we check the following constraints for each pair (a,b). The coordinates of the cuts in the lattice are used to test the strict inclusion conditions.

$$a \rightarrow b : \forall C_x \in max(a), \exists C_y \in max(b) \text{ such that } C_x \subset C_y$$
$$a \leftarrow b : \forall C_y \in min(b), \exists C_x \in min(a) \text{ such that } C_x \subset C_y$$
$$b \nrightarrow a : \forall C_y \in min(b), \nexists C_x \in max(a) \text{ such that } C_y \subset C_x$$

The second sub-model is an automaton that is trivially deduced from the constructed lattice (see Fig. 3). At this stage, the two sub-models are redundant. Note that scalability issues arise during the construction of each intermediate model: the size of the lattice may grow exponentially [5].

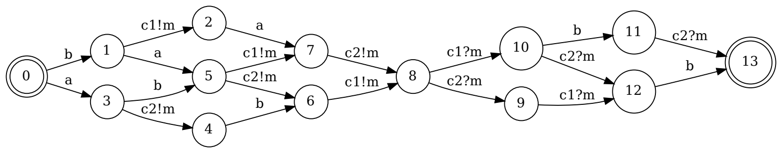

Fig. 3. The intermediate automaton sub-model of the execution α

2.2 Building a Global Model from Multiple Intermediate Models

First the *Model Merger & Generalizer* module (Fig. 1) aims at merging the constructed intermediate models. The final list of invariants must contain only the invariants that have been satisfied during each of the learned executions. Despite the fact that some events are not observed in all executions, merging these lists is rather trivial. Merging the different automata is even easier: it suffices to merge the initial states of the learned automata. At this stage, the obtained automaton accepts only the learned behaviors. As a distributed application may exhibit an infinity of behaviors, a model that captures a finite number of them will necessarily generate false positive alerts during the detection process: correct behaviors that have not been learned will be rejected. The interest of a generalization algorithm is twofold: first, it reduces the size of the model and, second, it extends the model with new behaviors. In fact, generalization algorithms usually introduce loops in the automaton, which allow it to accept infinite behaviors. The *Ktail* algorithm used in [1, 15] is a well-known generalization algorithm. It merges states with the same k futures (root subgraphs that are exactly the same up to a distance k).

In Fig. 4, we apply the *Ktail* algorithm on a model learned during the single execution α (see the automaton depicted in Fig. 3). The states 4 and 12 are

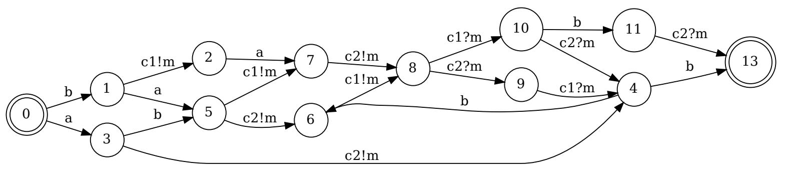

Fig. 4. Generalization of the automaton of execution α with $k = 1$

merged and the resulting automaton now integrates loops and accepts an infinity of behaviors. Among the new behaviors introduced, some are abnormal (For example, in $< a; c2!m; b; c1!m; c2?m; c1?m; b; c1!m; c2?m; c1?m; b >$ three messages are sent and four are received). The automaton sub-model becomes permissive and, during the detection phase, false negative may occur: an anomaly occurs but it is not detected. This explains the interest of keeping two sub-models. Even if the automaton may accept an incorrect behavior, the invariants are still there to possibly reject the behavior and secure the application (In our example, the incorrect behavior described above does not verify $c1!m \nrightarrow c1!m$).

The generalization phase suffers also from a scalability problem. As it occurs after the merge of all the intermediate automata, the number of states is huge.

2.3 Use of the Global Model during a Detection Phase

The *Detection* module (Fig. 1) checks whether the trace generated by a running application is accepted or not by the learned model. Consider that the monitored execution is called δ. Again, the trace E^δ is characterized by several total orders that are compatible with the observed partial order $\prec^\delta$. Like in [15], we use a depth-first search strategy to test all the candidate sequences, until one of them is accepted by the automaton and is in conformity with the set of invariants. An alert is raised if no sequence satisfies both requirements (other strategies may also be considered [13]). Sometimes, another verification can be done during the consumption of the trace E^δ: no receiving event can occur as long as the corresponding sending event has not been executed before. This invariant (denoted `InvCom`) characterizes the communications and cannot be expressed using invariants between types of events (*i.e.*, $\rightarrow$, $\leftarrow$, and $\nrightarrow$). It may discard candidate sequences (the verifications with the two sub-models are bypassed).

3 Solving Scalability Issues Without Sacrificing Efficiency

3.1 Focusing on a Subset of Specific Sequences in the Lattice

The first proposal modifies the *Model Builder* component (Fig. 1). In [15], for each execution, the entire lattice is built. To face this exponential time and space complexities, we propose to consider only a small subset of x sequences. At both extremes, the n local computations can be carried out sequentially or concurrently. Thus, in the whole lattice, the total number of sequences is between 1 and $\mathcal{S}!/\mathcal{P}$, where $\mathcal{S} = \sum_{i=1}^{n} |E_i^\alpha| = |E^\alpha|$ and $\mathcal{P} = \prod_{i=1}^{n}(|E_i^\alpha|!)$. Moreover, the length of any sequence is equal to $\mathcal{S}$. Among all the possible sequences, a subset of at least one and at most $n!$ sequences defines the concave hull of the lattice (*i.e.*, the envelop corresponding to perimeter of the lattice's geometrical representation). Static priorities between the n processes can be used to determine these sequences. Herein, the notation $p_i \rhd p_j$ indicates that p_i has priority over p_j. Among n processes, $n!$ permutations can be defined. Each permutation is a selection rule that identifies a sequence: given a prefix of the selected sequence (*i.e.* a consistent cut), the rule identifies the next event of the trace that will extend this prefix to obtain the next consistent cut. This deterministic process selects only sequences in the concave hull. It is more appropriate than a random selection of x sequences in the whole lattice because it limits the risk of selecting quite similar sequences. In this work, to reduce again the number of sequences, we consider only a subset of n cyclic permutations where no process appears twice at the same rank in two selected combinations. With $x = n$ selection rules (rather than $x = n!$) we obtain a good approximation of the envelop. In Section 4, we show that this drastic reduction in the number of sequences allows to scale without sacrificing efficiency.

The consistent cuts contained in the x selected sequences are elements of the original lattice. As each cut is identified by its coordinates in this lattice, the construction of the automaton (through the enumeration of transitions between cuts) requires no additional cost and can be done separately for each selected sequence. These selected sequences share common consistent cuts (at least the initial and final global states). Thus, in general, thanks to these interleaving points, the number of sequences contained in the built intermediate model is much greater than x. Consider the execution α. If the two selection rules are $p_1 \rhd p_2$ and $p_2 \rhd p_1$ then $< b;\ c_1!m;\ a;\ c_2!m;\ c_2?m;\ c_1?m;\ b >$ and $< a;\ c_2!m;\ b;\ c_1!m;\ c_1?m;\ b;\ c_2?m >$ are the selected sequences. As shown in Fig. 5, these sequences share three consistent cuts. Thus, the automaton accepts two additional valid sequences: $< b;\ c_1!m;\ a;\ c_2!m;\ c_1?m;\ b;\ c_2?m >$ and $< a;\ c_2!m;\ b;\ c_1!m;\ c_2?m;\ c_1?m;\ b >$.

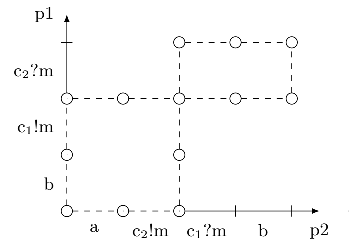

Fig. 5. Two selected sequences (with $p_1 \rhd p_2$ and $p_2 \rhd p_1$) & four sequences in the model

As a subset of sequences is now considered, the size of the automaton (number of states and transitions) can only decrease. The number of likely invariants may increase, but the new list necessarily includes the list corresponding to the whole lattice. Indeed, when some paths are no longer considered, the list of detected invariants can only increase due to the absence of some counterexamples.

3.2 Model Merging and Generalization

The second proposal suggests to call the *Model Merger & Generalizer* module not only once at the end of the construction phase but repeatedly during this phase. Generalization techniques such as the *Ktail* algorithm require to compare all the pairs of states in the automaton. Even if some optimizations are possible, this is a major time consuming activity, especially when the automaton is the result of learning several long-lasting executions. In the case of an automaton composed of y states, up to $\frac{y(y-1)}{2}$ comparisons may be necessary. The idea is to benefit from the fact that this kind of algorithm takes an automaton as input and outputs an automaton which size is much smaller. We hope to reduce the computation time by calling the generalization algorithm more often but each time with an input automaton of reasonable size (see Fig. 6). A single model under construction is maintained and combined step by step with each new created intermediate model: if z executions are learned, the *Model Merger & Generalizer* module is called $z-1$ times. The final global automaton may change if the order in which the z executions are learned is modified. Yet, this non-determinism is not a problem.

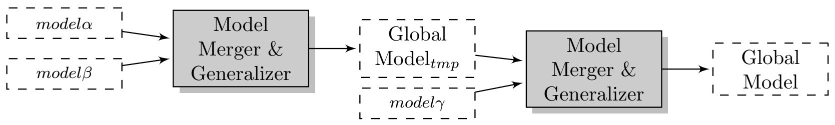

Fig. 6. Repeated calls (R) of the module

4 Assessment of the Approach

We evaluate our approach, using realistic data gathered from real executions of a well-known application which is representative of other distributed applications.

4.1 A Protocol for Conducting Experiments on Traces of XtreemFS

We consider XtreemFS [14], an open source project that provides a general purpose fault-tolerant distributed and replicated file system that can be deployed on cloud infrastructures. The file system service is built by four components: the *Directory Service* (DIR) maintains a database of services and volumes available; the *Metadata and Replica Catalog(s)* (MRC) store and manage files' metadata; the *Object Storage Device(s)* (OSD) store the actual file contents, which may be split and replicated; finally, *clients* request file system operations (volume creation/mount, file creation, etc.).

Multiple execution traces representing the correct behaviors were gathered, varying the behavior of the client and/or the duration of the execution, from 1 minute to 5 hours. Traces are used to build models and to evaluate the quality of the detection process. The set of traces is divided into v subsets; the construction of the model uses $v-1$ subsets, while the evaluation relies on the whole set of correct traces. During our experiments, the value of v is set to 5. Since there are five possible choices to exclude one subset of traces, we built - for each experiment - five different models. Thus each presented result is an average value obtained from five models that are instances of a given learning strategy. As a strategy is characterized by three parameters, we identify the different models using a naming convention composed of 4 fields separated by the character "`-`". The first field indicates which model we refer to (`Mod` stands for the whole model, `Aut` for the Automaton sub-model, and `Inv` for the Invariant sub-model). The second field corresponds to the number x of selected sequences (or `A`, when all the sequences of each lattice are selected). The third field indicates how the merging and generalization process has been performed (`S`: Single call, `R`: Repeated calls). The last field indicates the numerical value of k used in *Ktail*.

To evaluate the attack detection capability of each model, we deployed five known attacks against the integrity of a non-secured version of XtreemFS: the *NewFile* attack consists in adding the metadata of a file to the MRC server without adding its content into an OSD server; *DeleteFile* aims at deleting the file metadata on the MRC while keeping the file content on the OSD; the *OsdChange* attack changes an OSD IP address on the DIR database while its IP address did not change; *Chmod* modifies the file access policy on the MRC server without having the permission to do it; finally, the *Chown* attack modifies the file owner in the MRC metadata. A trace was collected for each attack. Regarding the quality of the detection process, we notice that the execution context and the moment the attack occurs are just as important as the nature of the attack. We perform each attack in four different contexts: (c1) while no client is active, (c2) before the client actions, (c3) after the client actions, and (c4) conducted from a source that is not the client address.

4.2 Scalability Issues during the Construction Phase

The first proposed solution (*i.e.* selecting a subset of x sequences) aims to reduce both the time required to build an intermediate automaton and its size. To check its effectiveness, we consider traces with different sizes. Data corresponding to independent constructions of the intermediate models are analyzed. Fig. 7 shows the size of the created automaton (number of transitions) and its construction time (seconds) in three cases: 1) all the sequences of the lattice are considered; 2) n of them are selected (here $n = 5 : 1$ DIR + 1 MRC + 2 OSD + 1 client); and 3) only one of them is selected. The rules applied to select sequences in case 2 are based on the following static priorities between processes: $p_1 \rhd p_2 \rhd p_3 \rhd p_4 \rhd p_5$, $p_2 \rhd p_3 \rhd p_4 \rhd p_5 \rhd p_1$, $p_3 \rhd p_4 \rhd p_5 \rhd p_1 \rhd p_2$, $p_4 \rhd p_5 \rhd p_1 \rhd p_2 \rhd p_3$, and $p_5 \rhd p_1 \rhd p_2 \rhd p_3 \rhd p_4$. Selecting n sequences avoids the exponential time and space complexity of an approach based on the whole lattice.

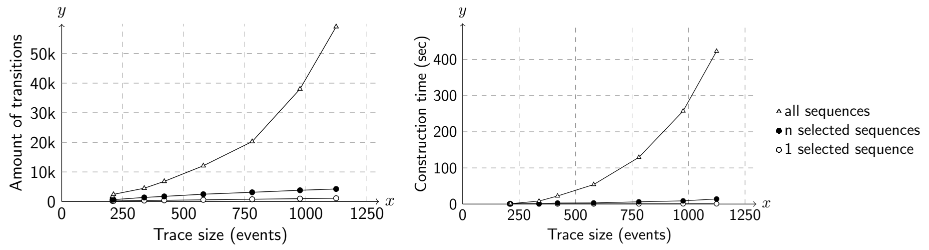

Fig. 7. Construction of an intermediate automaton: space(left)/time(right) complexity

Let us now consider the impact on the list of invariants. The number of recognized invariants depends on the number of selected sequences. Table 1 shows the number of invariants discovered during the construction of the corresponding intermediate model, considering the same cases and the same trace sizes as in Fig. 7. When a single sequence is selected ($x = 1$) the model is more restrictive, because more invariants are satisfied by the single selected sequence. Yet the error remains relatively small: the invariant list is always less than 5% bigger whatever the trace size. When only $n = 5$ sequences are selected, the detection is as precise as when considering the whole lattice (the corresponding lines in Table 1 are equal). This non-intuitive and very positive result validates the choice of using a limited number of selected sequences (and appropriate selection rules).

Table 1. Size of an intermediate invariant list

Trace size (events)	212	338	420	581	778	976	1124
1 selected sequence	3226	4721	6389	12230	16332	17295	17392
n selected sequences	3118	4643	6231	12082	16149	17084	17108
all sequences	3118	4643	6231	12082	16149	17084	17108

The second proposed solution (*i.e.* executing repeated calls to the *Model Merger & Generalizer* module rather than a single call) aims at reducing the computation time when either the number of learned executions and/or their duration increase. In Fig. 8, we compare two approaches `Mod-n-R-1` and `Mod-n-S-1`, where `R` stands for "Repeated calls" and `S` means "Single call". For distinct numbers of traces, we indicate the time required to compute the whole model. The proposed solution allows us to obtain better results when the number of learned executions increases. The curve corresponding to a single call follows more or less the number of states comparisons $\frac{y(y-1)}{2}$ that are performed during each unique call. In Fig. 8, each intermediate model is built using only $n = 5$ selected sequences. When the whole lattice is considered, the difference (between `Mod-A-R-1` and `Mod-A-S-1`) is even more impressive: taking into account 80 traces requires about 400 seconds for a single call and only 80 seconds in the case of repeated calls. Multiple calls are thus far less costly than a single, long lasting call.

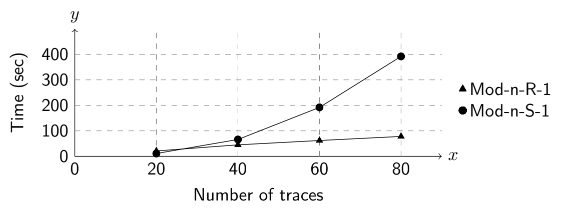

Fig. 8. Merge and generalization: single call versus repeated calls

4.3 Evaluation of the Accuracy of the Detection Phase

We want to solve scalability issues without impairing detection accuracy.

Table 2 shows detection results for the 5 attacks and 4 contexts described in Section 4.1, considering the strategies strategies `Mod-A-S-1` and `Mod-n-R-1`. Sub-models `Aut` and `Inv` are described separately, to show their complementarity and to validate the choice of keeping both. Even if it cannot always be implemented, we indicate with a check mark whether the invariant on communications (see Section 2.3) is able to detect the attack. For each strategy, as five models are created, a result $d/5$ means that d models out of five detected the attack ("-" cells mean 0/5). The results show that invariants are useful to detect attacks in contexts c1 and c2, while automata perform better in context c4. The proposed solutions to improve the scalability have no effect on the detection capabilities of the invariants, but they slightly degrade those of the automata. However, our model is dual: an alert is raised when the analyzed trace is rejected by either the `Inv` sub-model or the `Aut` sub-model. The detection based on invariants masks the possible errors of the automata sub-model in 8 out of the 20 cases. In the remaining 12 cases, the results provided by the sub-models `Aut-A-S-1` and `Aut-n-R-1` are the same, except in 2 cases (namely, the *DeleteFile* attack in context c3 and the *Chown* attack in context c2). In these two particular cases, the quality of the detection is quite similar: the scores are 0/5 and 1/5. Thus, in terms of false negative, the proposed solutions have nearly no impact. Differences are small or masked by the fact that the model is dual.

Table 2. Detection of attacks in various contexts using different sub-models

Attack	*NewFile*				*DeleteFile*				*OsdChange*				*Chmod*				*Chown*			
Context	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4
`Aut-A-S-1`	-	2/5	-	5/5	-	3/5	1/5	5/5	2/5	5/5	4/5	5/5	-	1/5	-	5/5	-	1/5	-	-
`Aut-n-R-1`	-	-	-	5/5	-	1/5	-	5/5	4/5	4/5	4/5	5/5	-	-	-	5/5	-	-	-	-
`Inv-A-S-1`	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	-	-	-	-
`Inv-n-R-1`	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	-	-	-	-
`InvCom`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	-

Table 3. Detection of attacks with different strategies for building the automata

#	Model	*NewFile*	*DeleteFile*	*OsdChange*	*Chmod*	*Chown*
1	`Aut-A-S-5`	100%	100%	100%	100%	100%
2	`Aut-A-S-1`	35%	45%	80%	30%	5%
3	`Aut-A-R-1`	30%	35%	50%	30%	5%
4	`Aut-n-S-5`	100%	100%	100%	100%	100%
5	`Aut-n-S-1`	25%	25%	85%	25%	0%
6	`Aut-n-R-1`	25%	30%	85%	25%	0%

Table 3 shows results from 6 distinct strategies to build the automaton sub-model. Each percentage indicates the proportion of attacks detected, knowing that 4 contexts and 5 models were used to test each strategy. As we focus only on the automaton sub-model, the percentages are much lower than those corresponding to the dual model (recall that the invariant sub-model may detect up to half of the first four attacks). When no generalization is done (*i.e.* for a high value of k in *Ktail*), no false negative occur (see line 1 and 4 in Table 3) but any correct behavior that has not been learned raises an alert (false positive). This strategy is not scalable and thus adopting it is unrealistic. In lines 2 and 3 (resp. 5 and 6) an intermediate model is built using the whole lattice (resp. using n sequences).

The model accuracy should be assessed in terms of false negatives but also false positives. Here we focus on the impact of the selection of x sequences on the false positive rate. Fig. 9 shows a well-known result: the smaller the value of k in *Ktail*, the better the result. This highlights the importance of the generalization phase. For a given value of k, we observe also differences depending on the number x of selected sequences. When x is too small (equal to 1), less behaviors are accepted. But the best results are obtained when n sequences are selected and not when the whole lattice is used. This good result is again in favor of our proposed solution. It may be explained by the fact that, when $x = n$, the automata that are provided as an input for the merge and generalization phase are neither too simple (not just x paths) nor too complex (the whole lattice). While keeping a wealth of behaviors, these automata have often less nodes and especially less transitions. As a consequence, the *Ktail* algorithm may merge more states and generate a final automaton accepting more behaviors.

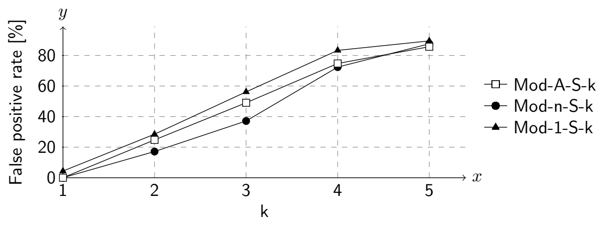

Fig. 9. False positive rate for different values of k and x

4.4 Short Traces and Long Traces

Using short traces, we have shown that a model (`Mod-n-R-k`) can be produced in a scalable manner without having a significant impact on the detection capabilities. Now we consider longer executions. When the construction phase relies on several short traces (at least 32), long traces are also accepted by the model. So it seems possible to have a learning strategy based mainly on short traces.

Table 4 shown average values allowing to compare the performance of our proposal for short (5 minutes) and long (5 hours) execution traces as inputs, for each phase (construction and merging/generalization). As for the construction phase, the results in the first column roughly correspond to the ones described in Fig. 7 and Tab. 1. The trace contains more events when the execution lasts 5 hours. Thus, the time to construct an intermediate model and the number of transitions are both bigger. Note that the ratios for results in the construction phase are around 35, except for the number of invariants, which is quite similar with short or long traces. This is due to the fact that the activity performed by the tester was similar in both executions: the same types of events were observed.

Table 4. Performance values for each phase

Phase	construction			merging & generalization		
Execution duration	5 min	5 hr	*ratio*	5 min	5 hr	*ratio*
Input trace size (events)	754	26178	34.7	754	26178	34.7
Processing time (seconds)	5	178	35.6	8	386	48.2
Number of transitions	2502	87266	34.9	386	247	0.6
Invariant list size	16643	15239	0.9	2125	2058	0.9
Reduced invariant list size	-	-	-	1190	986	0.8

Regarding the merge & generalization phase, the time needed to merge and generalize the short traces roughly correspond to that indicated in Fig. 8. Of course, it requires much more time to merge and generalize intermediate models obtained after a long execution (due to the bigger number of nodes and transitions). Yet we observe that the sizes of the result automaton and the invariant list are comparable in both cases. This is also due to the fact that, during our tests, the longer executions do not reveal new behaviors. The last line of Tab. 4 indicates the reduced size of the invariant list after a simple optimization (based on transitivity relations) that does not affect the accuracy of the `Inv-n-R-k` sub-model: for example, when a list contains the invariants $a \to b$, $a \to c$, and $b \to c$, we only have to keep $a \to b$, and $b \to c$.

5 Conclusion

We presented a scalable approach to build a behavior model of a distributed application and use it for intrusion detection. This dual model is composed of an automaton and a list of invariants, both learned from traces. Two original propositions have been made in order to scale better in time and space during the construction phase. They were evaluated using a real distributed application and a set of real attacks performed in different contexts. The evaluation showed the interest of the different proposed solutions. These solutions clearly address the scalability issues without having a significant impact on the detection capabilities.

References

1. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A.: Inferring models of concurrent systems from logs of their behavior with CSight. In: Int. Conf. on Software Engineering. pp. 468–479 (2014)
2. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A., Anderson, T.E.: Mining temporal invariants from partially ordered logs. ACM SIGOPS Operating Systems Review **45**(3), 39–46 (2012)
3. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE transactions on Computers **100**(6), 592–597 (1972)
4. Garg, V.K.: Principles of Distributed Systems. Kluwer Academic Publishers (1996)
5. Garg, V.K.: Lattice completion algorithms for distributed computations. In: Int. Conf. On Principles Of Distributed Systems. pp. 166–180 (2012)
6. Grant, S., Cech, H., Beschastnikh, I.: Inferring and asserting distributed system invariants. In: Int. Conf. on Software Engineering. pp. 1149–1159 (2018)
7. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
8. Lanoe, D., Hurfin, M., Totel, E.: A scalable and efficient correlation engine to detect multi-step attacks in distributed systems. In: 37th Int. SRDS (2018)
9. Lo, D., Mariani, L., Santoro, M.: Learning extended FSA from software: An empirical assessment. Journal of Systems and Software **85**(9), 2063–2076 (2012)
10. Lorenzoli, D., Mariani, L., Pezzè, M.: Inferring state-based behavior models. In: Int. workshop on Dynamic systems analysis. pp. 25–32 (2006)
11. Mukund, M., Kumar, K.N., Sohoni, M.: Synthesizing distributed finite-state systems from MSCs. In: Int. Conf. on Concurrency Theory. pp. 521–535 (2000)
12. Pfaltz, J.L.: Using concept lattices to uncover causal dependencies in software. In: Formal Concept Analysis. pp. 233–247 (2006)
13. Raguenet, I., Maziero, C.: A fuzzy model for the composition of intrusion detectors. In: Jajodia, S., Samarati, P., Cimato, S. (eds.) 23rd IFIP Intl Information Security Conference (SEC). pp. 237–251. Springer US, Boston, MA (2008)
14. Stender, J., Berlin, M., Reinefeld, A.: XtreemFS: A file system for the cloud. In: Data intensive storage services for cloud environments, pp. 267–285. IGI Global (2013)
15. Totel, E., Hkimi, M., Hurfin, M., Leslous, M., Labiche, Y.: Inferring a distributed application behavior model for anomaly based intrusion detection. In: European Dependable Computing Conference (EDCC). pp. 53–64 (2016)
16. Yabandeh, M., Anand, A., Canini, M., Kostic, D.: Finding almost-invariants in distributed systems. In: Symp. on Reliable Distributed Systems (SRDS) (2011)