Distributed Security Policy Conformance

Mirko Montanari, Ellick Chan, Kevin Larson,
Wucherl Yoo, and Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
{mmontan2,emchan,klarson5,wyoo5,rhc}@illinois.edu

Abstract. Security policy conformance is a crucial issue in large-scale critical cyber-infrastructure. The complexity of these systems, insider attacks, and the possible speed of an attack on a system necessitate an automated approach to assure a basic level of protection.

This paper presents Odessa, a resilient system for monitoring and validating compliance of networked systems to complex policies. To manage the scale of infrastructure systems and to avoid single points of failure or attack, Odessa distributes policy validation across many network nodes. Partial delegation enables the validation of component policies and of liveness at the edge nodes of the network using redundancy to increase security. Redundant distributed servers aggregate data to validate more complex policies. Our practical implementation of Odessa resists Byzantine failure of monitoring using an architecture that significantly increases scalability and attack resistance.

1 Introduction

Security management and policy compliance are critical issues in modern infrastructure systems. Regulatory and security organizations introduce policies and best practices to raise the minimal level of security required for power grid systems, government systems, and airport systems. We have studied industrial security policies [11, 12] that have complex challenging compliance and auditing concerns at the network level at the scale of the system concerned. Manual attempts to audit these systems are tedious, error prone, and potentially vulnerable to insider attacks or credential theft. Therefore a more principled solution to this problem is required.

The formalization of security policies and the use of hardened automated systems that validate compliance can improve the quality and efficiency of this auditing process. Although previous approaches analyzed the representation of these policies [1] and described centralized systems for collecting network information and analyzing it [6, 16], neither has adequately addressed the issue of scaling to networks of thousands of nodes or of resilience to attacks.

To address these issues, we have implemented and evaluated our policy compliance monitoring system Odessa. Our approach addresses the scaling problem by decomposing policies and distributing the validation process. Each of the

J. Camenisch et al. (Eds.): SEC 2011, IFIP AICT 354, pp. 210–222, 2011.
© IFIP International Federation for Information Processing 2011

complex rules that define the compliant and non-compliant states of the system is decomposed into local components and an aggregate component. We securely delegate the validation of local components to secure agents installed on hosts. These agents are able to reliably monitor the state of the system using virtual machine introspection. Using this information, we partition the validation of aggregate components across several distributed servers. Resilience toward attacks aimed at compromising the validation process uses Byzantine failure resistant, redundant information acquisition employing multiple agents and independent critical policy validation in multiple server style monitors.

The contributions of this paper include:

1. An algorithm for determining which portion of each policy can be validated on devices.
2. A resilient tree-based architecture that distributes to multiple servers the validation of the aggregate components of the policies and that delegates to several hosts the load of monitoring for the liveness of each device.
3. An evaluation of the scalability of our solution.

The rest of the paper is structured as follows. Section 2 describes related work in the area. Section 3 defines policy compliance and presents several examples of policies. Section 4 describes the Odessa architecture. Section 5 presents our algorithm for distributing policy evaluation. Section 6 describes our experimental evaluation. Finally, Section 7 summarizes our contributions and results.

2 Related Work

Several agent-based systems have been introduced for monitoring the security configurations of systems. NetQuery [16] and the DMTF Web Based Enterprise Management (WBEM) framework[1] provide a unified view of the configuration of a system and create notifications in case of changes in the state. However, none of these approaches provide automatic methods for distributing the evaluation of policies or decentralized mechanisms for detecting the failure of hosts.

Other non-agent based systems have been proposed for performing specific security assessments. Network scanners and security assessment tools such as TVA [6], or MulVAL [13] acquire information about the configuration of the system by using port scans or direct access to hosts. These systems have several limitations. First, changes to host configurations are detected with considerable delay because of the polling approach. Second, their architecture is centralized: the evaluation of policy compliance is performed in a central host. For very large networks, this can become both a bottleneck and a vulnerability as a single supervisory node audits, monitors, and checks remote operations that may impact integrity. ConfigAssure [10] takes a top-down approach and synthesizes network configurations from high-level specifications. However, the top-down approach is

[1] http://www.dmtf.org

not always applicable, as the organizational network is often managed by different divisions and it is not always possible to centralize the control into a single entity. Additionally, this paper focuses on policy-compliance validation. Previous work [7] discusses hardening techniques.

3 Policy Compliance

Policy compliance is a basic security and regulatory requirement for infrastructure systems. Although policy compliance cannot guarantee security, it still offers a minimal level of assurance against attacks that would be avoidable had proper security measures been taken. These policies can be specified as constraints created from regulatory requirements, or from the formalization of organization-specific security requirements. Policies are often posed as high-level specifications, and they are implemented in the system using a set of processes or rules that specify constraints on the states of the system. We focus on a set of rules that are in place to protect the industrial infrastructure against a wide-range of known types of attacks. For example, NIST specifies a set of best practices in their Security Content Automation Protocol (SCAP) [12] for regulating the configurations of government systems, and the North American Electric Reliability Corporation (NERC) provides policies about the configuration of power grid systems. For example, a policy might require all remote access points to the system to be monitored and logged at all times, or that all critical systems need to be placed within an enclosed electronic security perimeter. Changes in the configuration of the system or failures could create violations to such security policies and open it to potential attacks.

Many of these policies can be translated into rules and formalized in a logic language [1]. Odessa detects violations of these rules by representing configuration information and state information using Datalog statements [13]. Using Datalog, configurations are expressed as sets of ground statements (i.e., statements without variables). Without loss of generality, we represent configurations using the Resource Description Framework (RDF) language[2] [8]. Following RDF convention, statements are represented as subject-predicate-object triples (s, p, o). A set of statements connected with conjunctions is a knowledge base (KB). For example, we can represent the fact that a server *time.gov* provides the service *ntp* using the following KB: $(time.gov, istype, server)$, $(ntp, istype, service)$, $(time.gov, provides, ntp)$. Statements indicate conditions on the state of a host, and KBs integrate statements to reason about policy violations.

Datalog rules represent implications defined over the state of the infrastructure. The conditions of these implications are specified by a conjunction of *statement patterns*, i.e., statements that have variables as subject or object. Statement patterns are matched (i.e., *unified*) against the statements representing the state, and if the condition is true a new statement is added to the KB. Uppercase characters indicate variables and lowercase characters represent

[2] http://www.w3.org/TR/rdf-concepts/

resources. For example, we can consider a simple rule which specifies that critical servers should not run applications with known vulnerabilities without an exception. By acquiring information about the running program on each machine, annotations about the importance of each server, and information about vulnerabilities, we represent this rule by specifying that we have a violation if a critical server provides a vulnerable service as following: $(S, istype, server)$, $(A, istype, service)$, $(S, provides, A)$, $(S, criticality, high)$, $(A,\ hasvuln, V)$, $\neg (S, hasexception, E) \rightarrow (r_1, violation, V)$. The last statement, called the *consequence* of the rule, is specified by a statement pattern which have variables that appear in the condition (*body*) of the rule. The consequence can represent a violation as in our example, or it can represent a statement which can be used by other rules for validating compliance.

4 The ODESSA System

The objective of Odessa is to check if the state of the infrastructure as a whole is compliant to all policies defined by the organization and represented as Datalog rules. The architecture of the system was designed to distribute the evaluation of these rules in a scalable manner. To achieve this, Odessa uses a distributed evaluation protocol which includes the following elements: a set of *monitoring agents* that monitor the configuration of hosts and validate local portions of the organizations' policies; a set of distributed *verifiers* that validate global rules, and a set of *predicate groups* which are distributed index structures that provide a scalable support for the communication between verifiers and monitoring agents.

Agents represent the state of hosts using Datalog statements and share statements relevant to global policies with verifiers. Distributed verifiers integrate information across multiple hosts and perform the validation of policies with configuration information gathered from multiple machines. For a subset of policies critical to the security of the infrastructure, we require configuration information to be acquired independently from multiple agents and we replicate the policy validation on several verifiers which use Byzantine fault tolerance [5] to reach an agreement. By virtue of using FreePastry[3], our system inherits secured SSL communications and heartbeat-based liveness measurement. The architecture of Odessa is depicted in Figure 1.

Monitoring Agents. Monitoring agents run in a secure environment on each host. We developed agents running in the `Dom0` VM of a Xen system [2]. Virtual machines running on the same physical hosts are monitored using VM introspection [14]. Traditional hosts are monitored remotely by one of the agents using standard protocols. TPM can be used with SSL to provide a root of trust and validate the authenticity of the agent. The set of policy violations that our system detects depends on the type of information that monitoring agents can acquire from the systems.

[3] http://www.freepastry.org

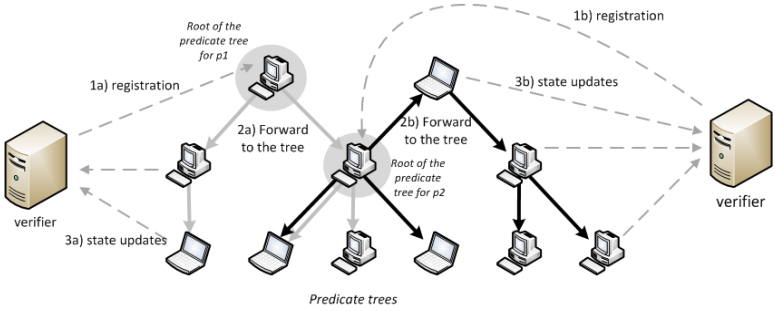

Fig. 1. Architecture of Odessa. End-hosts are organized in predicate groups. Verifiers register to the root of the group.

Verifiers. Verifiers are hosts that securely collect information shared by monitoring agents to validate rules. Each verifier manages a subset of the rules of the organization. The set of rules is partitioned across verifiers to minimize the overlap between the information required by each verifier. Critical rules are analyzed redundantly by multiple verifiers.

Predicate Group. To link monitoring agents and verifiers, we use predicate groups. Each predicate group connects monitoring agents that share a specific type of configuration statements and, hence, participate in the evaluation of the same rules. These groups distribute the processes of distributing information about new verifiers, integrating new hosts, and monitoring their liveness. Monitoring liveness is required because the state of failed hosts needs to be promptly removed from the state of the verifiers to detect correctly policy violations.

A predicate group is formed for every predicate p in the system. Membership in the group is distributed to several hosts by organizing agents into trees: each host maintains knowledge about a few other agents and monitors their liveness. The processes of constructing and maintaining these trees are inspired by the Scribe dissemination trees [4]. Communications are built on a Pastry Distributed Hash Table (DHT) [15] system, and the agent assigned to the DHT key $H(p)$ is the root of the tree for predicate p. When a new verifier starts monitoring compliance to a rule, it contacts the roots of the trees for the predicates involved in the rule to distribute its registration to all agents in the groups.

Resilience. Odessa has several design features that increase the difficulty of attacks when compared to centralized solutions. Attacks can target monitoring agents to compromise the information used for policy validation. To protect them, we run the monitoring agent in a separated secure environment, and we validate critical policies using redundant information. The separation isolates monitoring agents from possible compromises of hosts. In our implementation, we use VM-based separation but other techniques such as hardware based protections can be used without affecting the rest of the architecture. By running only monitoring agents in a separate VM, we provide a smaller attack surface for agents. For these reasons, we assume that agents behave according to our

protocol. However, while techniques have been developed for introspecting the state of machines without malware mediation [14], clever attackers could employ anti-forensics techniques and potentially conceal malicious processes. As an additional level of protection, we use redundant information acquired from independent sources in the validation of critical policies. For example, network traffic can be used to infer the presence of services on a specific machine, and multiple voltage sensors on a power line provide redundant information about the line's state. By acquiring information from multiple agents, an attacker would need to compromise several agents to thwart the validation process.

Attacks can target verifiers to conceal existing policy violations or to insert fictitious violations. We handle these cases by replicating the verification of the same policy on multiple verifiers. We use Byzantine fault tolerance for the validation of critical policies to reach an agreement even when a small subset of the verifiers is compromised. Attacks targeting predicate groups can compromise the infrastructure for distributing new verifiers registration, or for delaying the detection of failed and of new hosts. Even if agents are separated from the monitored hosts, attackers might still able to perform DoS attacks that affect one or more entire physical hosts. However, the DHT infrastructure automatically reconfigures itself to the new situation and notifies verifiers about failed hosts for triggering rule violations. Even when malicious users target the roots of the predicate groups, the DHT reassigns the failed agent role to other agents. Such attack only delays the registration of new verifiers, and it is easily detectable.

5 Rule Decomposition and Validation

To be scalable, our policy validation process detects policy violations through the coordination of monitoring agents and verifiers. We use our rule decomposition algorithm (RDA) to transform the organization's rules into an equivalent set of rules which takes advantage of information locality. This process allows Odessa to push a partial execution of the rules to each monitoring agent and hence, reduces configuration information transferred between machines.

The intuition behind the algorithm is to use information about the source of configuration statements (i.e., which agents can generate a particular configuration statement) for limiting the places where possible configurations that can trigger the body of a rule can be found. For example, if we are checking for the presence of a particular application on a host h_1, we know that information about running applications is generated only by host h_1. Using this locality rationale, we identify a portion of each rule. The execution of this portion that considers only local statements on each agent is equivalent to an execution that considers all statements in the system. Such a portion is executed independently on each agent, and only the results are shared with the rest of the system.

Our validation process is composed of two phases: decomposition and execution. The decomposition phase uses the RDA algorithm to integrate information about the locality of agents' configuration statements with the rules of the organization. The result of this process is a decomposition of each rule into *local sub-rules* and

aggregate sub-rules. In the execution phase, monitoring agents use local sub-rules to perform partial processing of the rule and use predicate groups to send processed state information. Verifiers use aggregate sub-rules to control the process of aggregating information acquired from multiple agents. A more detailed description of the algorithm can be found in the extended version of this paper [9].

5.1 Decomposition

The decomposition phase takes a rule and information about the statements generated by agents to decompose the rule into local and aggregate sub-rules. This process uses an RDF-graph representation of the rules which is more suitable for our analysis. Each rule is transformed in a *rule graph*, a labeled directed graph describing the explicit relationship between variables, resources, and predicates used in the rule. The graph has a node for each resource or variable and an edge from subject to object for every statement pattern. The statement pattern defined in the rule head is marked as the *head edge*.

Locality. For each agent, we say that a statement pattern is *local* if all its potential matching statements are always found locally, independently from the current state of the system. For identifying the local portion of the rule, we formalize our knowledge about the locality of the statement patterns using a RDF graph we call the *locality graph*. One of the nodes of the graph, called the *anchor*, identifies a specific type of agent as the information source (e.g., Host). Each undirected path starting from the root represents a conjunction of statement patterns: all the statements matching such combination of patterns are found locally on each agent. For example, we can consider a path with two statement patterns $(H, hasnetworkinterface, C)$,$(C, connectedTo, N)$. Statements matching these conditions represent the list of network interfaces and networks at which the host H is connected. For a specific host $H = h_1$, the only statements that can match these conditions are generated by h_1. The locality graph depends on the semantics of the statements that are generated by the agent. Statements used in the validation of critical policies should not be part of the local graph.

Using the locality graph we can identify subgraphs of the rule graph which can be processed locally. For clarity, we consider only one type of anchor, `Host`. We generate a different subgraph for each node of type `Host` in the rule graph. We include in this subgraph all edges and nodes that are connected to it which match edges and nodes in the locality graph. We recursively consider all paths in the locality graph. We call these subgraphs *agent-local* subgraphs. Agent-local subgraphs could have anchors which are not local for an agent. For example, given a locality graph (H, p, A) and a rule $(h_2, p, A) \rightarrow (h_2, violation, A)$, the subgraph is local only for host h_2. Without loss of generality, for every agent we choose one of the agent-local subgraph to be *local*.

Transformation into sub-rules. Once the local subgraph is identified, we generate local and aggregate sub-rules to use for the distributed rule processing. These sub-rules specify the location of the computation for the validation of rules and the structure of the communication between agents and verifiers.

A sub-rule is a pair $< \beta \rightarrow \eta, \mu >$ formed by a rule $\beta \rightarrow \eta$ (β is the body, η the conclusion) and a query μ. For local sub-rules, the rule $\beta \rightarrow \eta$ represents a portion of the original rule, and the query μ identifies the statements generated by local processing which are sent to verifiers. For aggregate sub-rules, the query identifies the information received by the agents, and the rule identifies the remaining portion of the policy validation.

Local sub-rules. For each rule graph we consider its local subgraphs. There are several cases. (i) If the local subgraph covers the entire rule $\beta \rightarrow \eta$, then we create a sub-rule $< \beta \rightarrow \eta, \emptyset >$. In this case, the entire processing of the rule can be local and the agent only needs to raise an alarm for violations. (ii) If only the head η statement of the rule graph is not part of the local subgraph, we create a local sub-rule $< \beta \rightarrow \eta, \eta >$. i.e., we locally compute the entire rule, and we share with the verifiers only the consequences. (iii) If the local subgraph covers only a portion of the rule, then we create several local sub-rules. For each edge π of the rule graph not in the local subgraph, we create a local sub-rule $< \emptyset, \pi >$, and we generate a single local sub-rule $< \beta_l \rightarrow \eta_l, \mu_l >$ for the entire local subgraph as follows.

The body of the rule β_l is constructed by taking all edges in the local subgraph and converting them into a conjunctive query of predicate patterns. Because all edges are part of the local subgraph, all statements that match the body of the rule are found locally. For example, each match of a rule body $(A, p_1, B), (B, p_2, C)$ creates a tuple (A, B, C) containing the values of the variables. These tuples need to be shared with the verifiers to complete the validation of the rule. However, we do not need to share the entire set of variables, but only the variables which are used in statement patterns outside the local subgraph. Their variables are identified by considering the nodes at the boundary of the local subgraph (i.e., nodes which have an edge in the local subgraph and an edge outside it). For example, if only the variables A and C are used in statements outside the local subgraph, we only need to share the tuple (A, C).

This tuple, which represents the information to share, is used as the head η_l of the rule. However, because the size of the tuple can change depending on the number of variables in the body, we represent the head using a variable number of statements which share the same RDF blank node as the subject. We can think of blank nodes as resources with a unique name generated automatically by the system. The object of each statement is one of the variables in the body, and the name of the predicate depends on the rule and on the name of the variable. We call these statements *rulematch* statement patterns. For example, the rulematch statements that we define for the body in the example are $(_ : k, rm_{r,'A'}, A), (_ : k, rm_{r,'C'}, C)$. The blank node $_ : k$ is substituted with the same random string for all statements, r is a unique identifier of the rule and $'A'$ and $'C'$ are strings representing the variable names. By combining body and head of the example, we have the rule $(A, p_1, B), (B, p_2, C) \rightarrow (_ : k, rm_{r,'A'}, A), (_ : k, rm_{r,'C'}, C)$. The last piece of the local sub-rule, the query μ_l, selects these rule match statements. An example of rule graph and local sub-rules is shown in Figure 2.

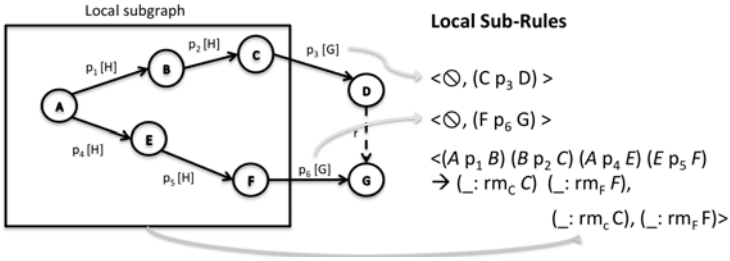

Fig. 2. Example of the conversion of a rulegraph into a set of local sub-rules

Aggregate Sub-Rules. The analysis for the generation of aggregate sub-rules is similar to the generation of local sub-rules. Even if aggregate sub-rules are executed on verifiers, we still use the concept of "locality" as locality for the agents. For edges $\pi = (A, p, B)$ not in the local subgraph we create an aggregate sub-rule with only a query $< \emptyset, \pi >$. This aggregate sub-rule specifies that all statements matching this pattern should be delivered to the verifier. If the rule graph ρ does not have a local subgraph, we add an aggregate sub-rule $< \rho, \emptyset >$ which introduces the rule in the verifier's KB. Hence, for rules with no local subgraphs, the verifiers simply collect all statements matching any of the statement patterns of the rules and perform local reasoning.

For rule graphs with a local subgraph, we need to account for the partial processing performed on the agents. We create an aggregate sub-rule with a rule ρ' where the local subgraph edges have been substituted with rulematch statements, and we create a set of queries that collects such statements.

5.2 Execution

The execution is driven by local sub-rules and aggregate sub-rules. For each aggregate sub-rule $< \rho, \mu >$ the verifier adds the rule ρ to its local KB. Considering $\mu = (A, p, B)$, it sends a message to the root of the predicate group of p, $H(p)$. The message contains the query μ which the root nodes disseminates to all agents in the group. On the agents, for each local sub-rule $< \rho, \mu >$ we add the rule ρ to the local KB and we select all statements matching μ. Assuming $\mu = (A, p, B)$, the agent sends a message toward $H(p)$ to register itself as part of the predicate group. In the DHT, the path toward such a node travels through multiple other monitoring agents. At each step, agents on the path keep track of the subscription and form a new branch of the tree used to maintain the predicate group. When an agent that is already part of the same tree is found, the branch is connected as a child of such agent. The agent receives from the new parent the registered verifiers and sends them its configurations.

For the validation of critical policies, we require verifiers to collect statements from a minimum number of different agents. A Byzantine agreement on the result is reached by broadcasting the result of local validations to other verifiers.

The DHT infrastructure supports the monitoring for liveness. Each monitoring agent is registered to several predicate groups, and the agents periodically send heartbeat messages to their neighbors. When the failure of an agent is detected, a message is sent to the registered verifiers so that all statements that had been generated by the failed agent are removed from the verifiers' state. As each host is registered to several trees, even the failures of all hosts in a branch of the tree are detected by the parent hosts in other trees.

6 Implementation and Evaluation

We implemented the components of the Odessa system using a combination of C and Java. The communication between monitoring agents and verifiers is implemented on the FreePastry system. Inference is performed by using the rule-based inference system Jena [3]. The monitoring agents run in the `Dom0` virtual machine of a Xen installation. They monitor guest VMs by accessing the host state using an extension of XenAccess [14]. A Linux kernel module is installed on guest VMs to provide additional information about the state of the system which is not easily accessible using XenAccess.

We ran the system on a real network and we validated a set of test rules which include (i) checking the presence of specific programs, (ii) checking NFS authorization for access control misconfigurations that give unprivileged users access to restricted files, (iii) and validating that critical machines are protected from external attacks. Our system was able to delegate the validation of rules (i) and (ii) to each host, and it was able to decompose rule (iii) into a local portion and a global portion. The local portion shares statements about the host address and about vulnerable programs running on the system, which are identified using the National Vulnerability Database (NVD)[4]. The global portion integrates this information across the network and using logic attack graphs it computes if a specific host can be compromised by an external attacker. We use our prototype to measure the possible delay in the verification that an attacker can introduce by performing DoS on predicate group roots before a new verifier is registered. We found that the FreePastry implementation already provides a delay limited to an average in the order of tens of seconds. The tradeoff between message frequency and delay in the detection of failures is shown in Figure 3. The parameter p represents the number of communication attempts made before declaring an agent dead. The results are an average of 20 executions.

To measure the scalability characteristics of Odessa, we performed several simulations using random models of large-scale systems. The first experiments focus on the ability of Odessa of distributing rule validations, and the second on the scalability of the system. We use the number of statements shared by monitoring agents as a metric for measuring the distribution of rule validation. We create a synthetic configuration with a structure similar to the configuration data found in real hosts (e.g., process list and associated sockets, network file system data). The configuration is composed of a constant number of statements organized in

[4] NVD: National Vulnerability Database V2.2 http://nvd.nist.gov/

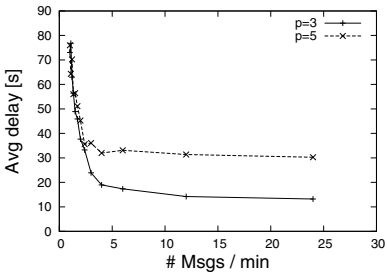

Fig. 3. Delay in the detection of agent failures

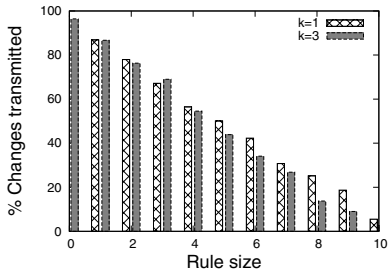

Fig. 4. Agents' statements transferred as consequences of configuration changes

tree structures where the object of the parent statement is the subject of the children. We vary the number of children at each level with a parameter k, and we vary the number of levels. Random statements have constants as objects. We consider a rule body (A_1, p_1, A_2), ..., (A_i, p_i, A_{i+1}), ..., (A_m, p_m, A_{m+1}) and we changed the local sub-rule by varying the index i to represent different types of configurations and to represent the use of critical policies (which decrease the local portion of the rule). We consider a system where agent configurations change periodically by randomly adding or removing statements and we measure the effects of the size of the sub-rule in the number of statements transmitted. We found that the number of statements decreases linearly with the increase of the local portion of the rule, as shown in Figure 4.

The next set of experiments shows that, independent from the advantages of delegating rule processing, the use of predicate groups significantly reduces the load on the verifiers for monitoring the liveness of hosts and, hence, enables an increased scalability of the system. To quantify this gain, we perform simulations to compare our architecture with an agent-based solution which relies on a central server for integrating data. We set the parameters of the two solutions (e.g., frequency of messages used for keep-alive purposes) to obtain an equivalent delay in the detection of failed hosts, and we consider both a static network (`odessa-p50-static`) and a network where a host is added or removed

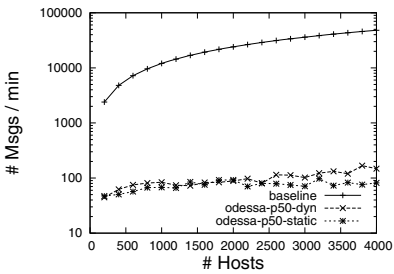

Fig. 5. Maximum number of messages sent and received by any hosts (log y)

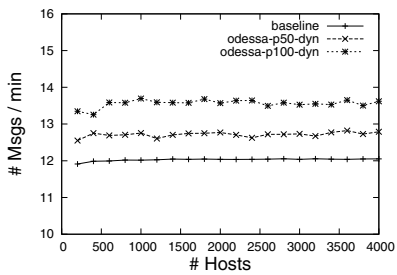

Fig. 6. Average number of messages sent by each single host

from the system so that 20% of the hosts change every hour (`odessa-p50-dyn`). We measure the maximum amount of messages sent and received by each host. We find that our solution reduces by orders of magnitude the maximum load on any single host (shown as in Figure 5) and has a limited effect on the average load of each single host (shown in Figure 6). We also find that the number of predicate groups at which hosts are connected does not significantly affect the average number of messages exchanged. In the figures, `odessa-p100-dyn` represents a network where each host is connected to 100 predicate groups, while in `odessa-p50-dyn` each host is connected to 50 predicate groups.

7 Concluding Remarks

This paper shows that resilient and large-scale policy validation is possible by introducing an architecture and an algorithm for decomposing policies and distributing their validation across multiple machines. We assess that our technique is viable and practical for deployment in large infrastructure systems.

In our future work we are planning to employ reactive agents that can harden the host according to security policies to reduce the time window of vulnerability of the system. Our approach focuses on rules for which violations have a long lifespan. Short-lived false negatives from message reordering pose a limited threat to security because they already have a small time window for attack. However, we are planning to address these issues for a more general monitoring system.

References

1. Anwar, Z., Campbell, R.H.: Automated Assessment of Critical Infrastructures for Compliance to CIP Best Practices. In: Second IFIP WG 11.10 International Conference on Critical Infrastructure Protection. IFIP (2008)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP. ACM, New York (2003)
3. Carroll, J., Reynolds, D., Dickinson, I., Seaborne, A., Dollin, C., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW. ACM, New York (2004)
4. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communications (2002)
5. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) (1982)
6. Jajodia, S., Noel, S., Berry, B.: Topological analysis of network attack vulnerability. In: Managing Cyber Threats: Issues, Approaches and Challenges (2005)
7. Johnson, C., Montanari, M., Campbell, R.H.: Automatic Management of Logging Infrastructur. In: CAE Workshop on Insider Threat. CAE (2010)
8. Montanari, M., Campbell, R.H.: Multi-Aspect Security Configuration Assessment. In: SafeConfig Workshop. ACM, New York (2009)

9. Montanari, M., Chan, E., Larson, K., Yoo, W., Campbell, R.H.: Distributed Security Policy Conformance. Technical Report, University of Illinois (February 2011)
10. Narain, S., Levin, G., Malik, S., Kaul, V.: Declarative Infrastructure Configuration Synthesis and Debugging. Journal of Network and Systems Management (2008)
11. North American Electric Reliability Corporation, Critical Infrastructure Protection Standard, CIP-001 to CIP-009 (2010)
12. NIST. SP800-126: The Technical Specification for the Security Content Automation Protocol (SCAP) (2009)
13. Ou, X., Boyer, W., McQueen, M.: A scalable approach to attack graph generation. In: CCS. ACM, New York (2006)
14. Payne, B.D., Carbone, M., Lee, W.: Secure and Flexible Monitoring of Virtual Machines. In: ACSAC. IEEE, Los Alamitos (2007)
15. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Liu, H. (ed.) Middleware 2001. LNCS, vol. 2218, p. 329. Springer, Heidelberg (2001)
16. Shieh, A., Kennedy, O., Sirer, E., Schneider, F.: NetQuery: A General-Purpose Channel for Reasoning about Network Properties. In: OSDI. USENIX (2008)