Evidence of Log Integrity in Policy-based Security Monitoring

Mirko Montanari, Jun Ho Huh, Derek Dagit, Rakesh B. Bobba, Roy H. Campbell
Information Trust Institute
University of Illinois at Urbana-Champaign
{mmontan2, jhhuh, dagit, rbobba, rhc}@illinois.edu

***Abstract*—Monitoring systems are commonly used by many organizations to collect information about their system and network operations. Typically, SNMP, IDS, or software agents generate log data and store them in a centralized monitoring system for analysis. However, malicious employees, attackers, or even organizations themselves can modify such data to hide malicious activities or to avoid expensive non-compliance fines.**

This paper proposes a cloud-based framework for verifying the trustworthiness of the logs based on a small amount of evidence data. A simple Cloud Security Monitoring (CSM) API, made available on the cloud services, allows organizations operating on the cloud to collect additional "evidence" about their systems. Such evidence is used to verify system compliance against the policies set by security managers or regulatory authorities. We present a strategy for randomly auditing and verifying resource compliance, and propose an architecture that allows the organizations to prove compliance to an external auditing agency.

I. Introduction

Even when running on the cloud, the infrastructure of an organization is subject to the requirements set by security managers and regulations like the Payment Card Industry Data Security Standard (PCI-DSS) [1] or the Federal Security Management Act (FISMA) [2]. Organizations often monitor their compliance using policy-based monitoring systems—these detect violation of security requirements and regulations. However, most of the information generated by user-operated monitoring tools such as Nagios[1] or SNMP daemons can be manipulated by tampering with the programs that collect the logs or by injecting false information into the monitoring system. Such compromised data may be used to hide malicious problems or to hide system configuration problems and avoid expensive non-compliance fines.

In this paper we focus on policy-based monitoring systems and propose a framework that uses information generated by cloud providers to verify trustworthiness of the reported monitoring information (logs). Cloud providers collect a partial view of the users' systems by monitoring their infrastructure with methods that are independent from the user-operated monitoring system. We show that a simple Cloud Security Monitoring (CSM) API available from the cloud provider enables organizations to "query" such a partial view and collect additional "evidence" about the operations of their systems—this evidence is used as an additional source of information to raise the bar against attacks, and to help with the compliance checks. The CSM API provides the users with ability to sample the state of their hosts and verify the trustworthiness (integrity) of the information that is provided to the monitoring system.

We rely on existing techniques like Virtual Machine Introspection (VMI) [10] and host virtual machine based network monitoring to acquire the additional evidence. Because such monitoring techniques typically provide only partial views on a user's system and incur performance overheads, our approach uses a novel *resource-based* sampling that takes advantage of the knowledge about policies to select the events that need to be collected. We identify the minimal sets of events needed to construct proofs using only evidence gathered from the cloud provider in order to demonstrate compliance of specified system resources. When an external monitoring entity, e.g. Monitoring-as-a-Service (MaaS), performs the sampling, the collected evidence could also demonstrate security policy compliance to third-party auditors.

Our contribution includes the following:

1) we introduce a security monitoring API to collect additional evidence about the user system state;
2) our sampling strategies allow evidence of compliance to be constructed from a small set of evidence;
3) we show that our API can work with well-known monitoring techniques, providing evidence for validating part of the network management and security policies defined in PCI-DSS [1].

The rest of the paper is structured as follows. Section II presents our policy compliance framework. Section III presents strategies for performing random policy compliance validations. Our initial evaluation is in Section IV. Section V compares our approach against the current state of the art. Finally, we conclude and discuss future work in Section VI.

II. Framework for Policy Compliance Monitoring

The goal of our framework is to provide cloud users the ability of collecting evidence that supports compliance of their systems to policies. We consider the case in which the user's infrastructure is provided by a Infrastructure-as-a-Service (IaaS) provider, and we assume that the policies specified by the user are formally represented as *rules* in an event-based monitoring system (e.g., [4]). These rules identify policy violations and are specified by indicating sequences of

[1]www.nagios.org

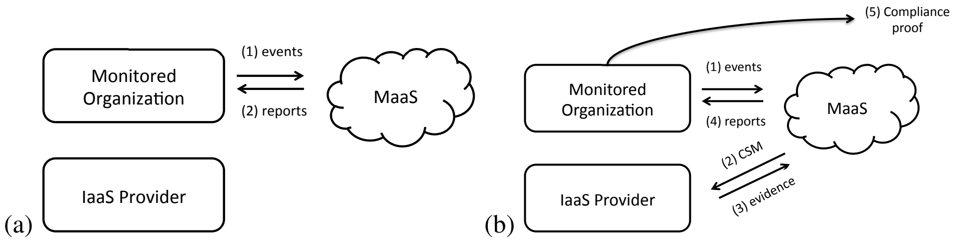

Fig. 1. (a) Current monitoring-as-a-service architecture. (b) The CSM API allows the MaaS provider to acquire evidence about the compliance of the infrastructure and prove compliance to a third-party auditor.

events that are not expected to occur in the system. In a basic monitoring system, system changes are represented as events and collected in a centralized location; these are used to detect sequences of events that violate policies. Consider a security requirement which specifies that all active hosts should be running an anti-virus software, unless specifically exempt. This requirement can be validated by representing information about active hosts, running processes, and exemptions as events.

There are several companies (e.g., monitor.us) providing third-party monitoring solutions (Monitoring-as-a-Service, or MaaS), which can be used by organizations to manage and analyze their logs. Generally, these solutions rely on acquiring log data and other events from Syslog, routers, or monitoring agents running on end-hosts. However, while these techniques can increase the confidence that an organization has on its own compliance, the information collected is subject to two problems. The first problem arises when compliance needs to be checked by external entities. As the event data are generated by the organization, the auditors have no guarantee that the data corresponds to the actual system configurations. Second, compromised systems can manipulate events reported to the monitoring system and hide non-compliant behaviors. When the infrastructure runs on a cloud infrastructure, our architecture addresses these problems by using the CSM API to collect additional evidence of compliance.

A. Rule Definition

We define event-based rules by representing events and rules in logic. Events are characterized by the type, set of parameters, and two (start and end) timestamps. We write events as $type(P_1, \ldots, P_n)_{t_s,t_e}$. We represent policies as rules in Datalog¬ (Datalog with negation) extended with time constraints [5]. We assume the safeness of variables (all variables appearing in the body and head of a rule need to appear at least in a non-negated predicate) and stratified negation (a common condition on the definition of recursion over negated predicates). We omit $_{t_s,t_e}$ if the rule is checking for a set of events that are true at current time $now > t_s \wedge now < t_e$.

Consider a policy specifying that all active machines should run an antivirus software unless they are explicitly exempt from the requirement. We use event $activedevice(M)$ to indicate that a machine M is active, and event $exempt_{av}(M)$ to indicate that M is exempt from this requirement. Event $runs(M, P)$ indicates that a machine M is running program P, and $hash(M, P, D)$ indicates that the hash of the code pages of program P running on M is D. $antivirus(D)$ is a long-lived event indicating that the hash D corresponds to an anti-virus software. Using these events we define the security requirement as follows:

$$\begin{aligned} violation \leftarrow \quad & activedevice(M), \neg runs_{av}(M), \\ & \neg exempt_{av}(M). \\ runs_{av}(M) \leftarrow \quad & runs(M, P), hash(M, P, D), \\ & antivirus(D). \end{aligned} \tag{1}$$

The first rule defines that the security requirements is violated if there is an active device at current time that is not running an antivirus software ($\neg runs_{av}$) and that it is not exempt from the antivirus requirement $\neg exempt_{av}(M)$. We consider a program an anti-virus software if the code of the running program P ($hash(M, P)$) matches the code of a known anti-virus software ($antivirus(D)$). For each policy a set of *primary resources* is identified, indicating the resources for which the policy should be verified. In this example, the primary resource is M as we are interested in verifying the described property for all hosts machines.

B. Architecture

Our architecture considers the case in which the organization uses an external MaaS to store and analyze the system logs. The CSM API, when used directly by the organization, will be useful for evaluating potential errors in the monitoring system but cannot be used to collect the independent information requires for proving compliance to a third-party.

We consider four entities in our architecture: the *cloud provider*, *monitored organization*, *monitoring-as-a-service provider*, and (optionally) *external auditor*. We assume that the cloud and the MaaS providers are neutral entities and do not collude with the organization. Additionally, we assume that it is hard for an attacker to compromise both the organization and the cloud provider (or the MaaS), as they are managed by different entities. In a typical MaaS architecture, the organization runs its services on the cloud provider's infrastructure and sends event logs to the MaaS (Step 1, Figure 1a). The MaaS provides compliance reports and notifies the organization about any violations detected (Step 2, Figure 1a). In this architecture, the organization cannot perform any secondary checks on the correctness of the reports generated by the monitoring system. Since the logs are provided by the organization, the generated reports cannot be used to prove its compliance to external auditors.

Our evidence-based framework adds an additional interaction (Step 2-3, Figure 1b) between the MaaS provider and the cloud provider. After receiving the logs, the MaaS can verify the log integrity and accuracy (Step 2) through the use of the CSM API. The cloud provider sends (Step 3) the information about the state of the requested resource. After a small delay added from collecting all the event data about the resource, the MaaS confirms or disputes the trustworthiness of the logs sent by the organization. This information is delivered to the organization as a signed message. By collecting several pieces of evidence over a period of time, the organization can show partial evidence of compliance to the external auditors (Step 5, Figure 1b).

III. Collecting Compliance Evidence

A key component of our architecture is the CSM API, which provides access to advanced monitoring functionalities of the cloud provider.

Cloud providers generally provide an API for monitoring performance, load, and failures. We extend this API with methods to collect security-relevant information. However, IaaS cloud providers often have limited knowledge about the user systems' security state. For example, the semantic gap [10] introduced by virtualization or the overhead of a continuous monitoring of network communications make it difficult to monitor all events that are relevant to policy compliance.

To address these issues, we use the CSM API to collect a small amount of evidence data that can be used to verify the trustworthiness of reports generated by the monitoring system. The CSM API is based on a "query" approach: the organization or the MaaS requests specific information about a system resource; the cloud provider collects the requested information and responds to the message. Only a little persistent state is saved on the cloud provider's side. To provide the advanced monitoring services at a low cost, the API is designed to incur just small monitoring overhead.

A. *Case Study: PCI-DSS Compliance*

The rest of the paper is presented using a concrete case study that considers a subset of the Payment Card Industry Data Security Standard (PCI-DSS) policies. PCI-DSS compliance is a requirement for organizations handling credit card data for any of the large credit card companies (i.e., Visa and Mastercard). The policies cover security procedures, software development, access control, and network configurations. PCI-DSS is aimed at assessing compliance of an entire organization, but was not designed specifically for monitoring purposes. Generally, the organizations would define their own monitoring rules for checking compliance. However, for the policies that are directly related to network management, PCI-DSS provides representative examples of the type of security requirements that the organizations would enforce on their systems.

Because our focus is on network and system configuration policies, we only consider requirements identified in Sections 1, 2, 5, and 6 of the PCI-DSS compliance document [1]. Information about these sections and what data is required for validating the policies are shown in Table I.

TABLE I
Association between requirements, number of rules, and information to acquire. In parenthesis we put the number of policies we consider relevant to monitoring.

Description	Num	Evidence
1 - Install and maintain a firewall configuration to protect cardholder data	24(18)	procedures; documentation; firewall information; topology; network traffic; cardholder data.
2 - Do not use vendor-supplied defaults for system passwords and other security parameters	8(8)	procedures; documentation; custom-agents; running programs; topology;
5 - Use and regularly update anti-virus software or programs	3(3)	documentation; custom-agents; running program; program integrity; network.
6 - Develop and maintain secure systems and applications	22(2)	procedures; documentation; running program; OS identification; program integrity.

The cloud provider can collect a significant part of the required information using three types of monitoring systems: cloud configuration information, network monitoring, and VMI. All of these monitoring techniques have high practicality and have implementations available for use. First, configuration information is immediately available through the provider API. Second, network monitoring can be performed by implementing agents in a monitoring VM (e.g., Xen Dom0 or KVM's Host) with technologies such as OpenVSwitch [6] or libpcap. Third, VMI can observe the internal state of VMs and can be implemented with technologies such as libVMI [3]. We imagine that the cloud provider can recover the implementation costs and small performance overheads incurring from this advanced monitoring by charging a per-request fee in a way consistent with the pay-per-use cloud computing model.

TABLE II
List of event types that need to be acquired for validating PCI-DSS compliance

Event Type	Evidence	Description
topology	EC2 API	Firewall configurations and host IPs
network traffic	OpenVSwitch	Capturing packets for identifying flows
cardholder data	libVMI	Memory analysis for credit card patterns
running programs	libVMI	Memory analysis for extracting running programs
program integrity	libVMI	Memory analysis for checking program integrity

Table II lists the monitoring techniques that we consider in the development of the CSM API. Other monitoring techniques can also be added to allow verification of additional information.

B. Cloud Security Monitoring API

At the core of the CSM API is a set of methods that can be used for obtaining information about system resources. As information about resources is expressed in our model using logic statements, each method accepts one or more resources and returns a set of statements. The methods return a signed response containing the submitted query, response statements, and timestamp—this can be used to prove compliance of the infrastructure to an auditor. With the exception of methods returning general information about the system, the invocation of a CSM API method requires the IDs of resources (accessible to the cloud provider) to be specified. Queries that collect the state of a large set of resources at once are not practical as this would require a high workload from the cloud provider.

We summarize in Table III the monitoring techniques that we consider in our use case. Table IV provides a connection between the methods and the statement provided. The column "API call" contains the name of the method and the parameters to pass to the function. The domain of the parameters is provided in the second column. The third column reports the list of statements contained in the response. When a variable is qualified with $\forall$, the method returns the entire list of statements for the specified resource. For example, method ev_{run} returns the entire list of programs running on the given machine m at time t. Hence, we know that for every P not appearing in the result, the statement $runs(m, P, t)$ is false.

We consider monitoring techniques that are transparent for the guest VM (i.e., an organization is not aware of a sample being taken), and we consider the sample selection to be randomized: in this way, hosts cannot change their configurations or state just before the information is collected.

TABLE III
EXAMPLES OF VIRTUAL MACHINE INTROSPECTION CAPABILITIES.

Capability	**Name**	**Reference**
List host and network resources	ev_{list}	EC2 API [7]
Running processes	$ev_{run}(M)$	Payne et al. [3]
Network connections	$ev_{nt}(M)$	Payne et al. [3]
Cardholder data	$ev_{chd}(M)$	Hizver et al. [8]
OS identification	$ev_{id}(M)$	Christodorescu et al. [9]
Program integrity	$ev_{int}(M, P)$	Garfinkel et al. [10]

C. Validation Process

During the validation process, the MaaS provider selects and obtains evidence for supporting the compliance of its client infrastructure. As the CSM API exposes only a partial view of the events in the system, we select evidence using a *resource-based strategy* based on random sampling. This strategy uses the partial information available through the CSM API to prove compliance of a resource at a time t. Our strategy works as follows. First, we select a random policy p and, for each primary resource in the policy, a random resource m. Second, we convert the policy p into a set of expressions called *minimal evidence set*. Finally, we use the minimal evidence set to guide the acquisition of evidence from the CSM API.

TABLE IV
CSM API METHODS FOR OUR USE CASE

API call	**Domain**	**Response**
ev_{list}	-	$\forall M : computer(M)$.
$ev_{run}(M)$	M : hosts	$\forall P : runs(M, P)$.
$ev_{nt}(M)$	M : machine	$\forall N : connected(M, N)$.
$ev_{chd}(M)$	M : hosts	$cardholder(M)$.
$ev_{int}(M, D)$	M : hosts, D : program signature	$integrity(M, D)$.

1) Minimal Evidence Set: Minimal evidence sets define combinations of evidence that are sufficient for proving compliance of a resource. For example, we consider the requirement 1.3.7 of PCI-DSS that requires hosts storing cardholder data to be connected to an internal network that is segregated from the DMZ and the public Internet. We can represent this requirement with the following rule.

$$\begin{aligned} violation \leftarrow \quad & cardholderdata(M), \\ & connected(M, N), \neg internal(N). \end{aligned} \tag{2}$$

We consider the application of the policy to a specific resource by substituting the resource name in the policy (e.g., M/m). A direct verification of such an expression requires collecting evidence about each single statement in the rule body. In the given example, we need to (1) verify whether m contains $cardholderdata$, (2) find the list of networks that m is connected with, and (3) check which networks are internal. If this set of additional evidence can be collected, the MaaS verify compliance to the policy and determine the presence of violations. However, as the CSM API provides only a partial view of the system, collecting evidence about each single statement in a complex policy might not be possible or might impose a large overhead.

In many cases, it is possible for a MaaS provider to prove, by collecting very little evidence, that a resource is not violating the selected policy. In this example, compliance for a resource can be demonstrated by collecting evidence that proves that m does not hold cardholder data, **or** that m is only collected through the internal network. Intuitively, we show compliance by collecting evidence about the absence of a violation by considering the negation of a rule body. Generally, rules are composed of several conjunctions: we can use simple manipulations to show that negating the rule body creates a set of negated expressions connected by OR. If **any** of these expressions can be proven true, we know that the negation of the rule is true, and hence a violation cannot exist for the selected resource. As each OR-connected expression contains only a portion of the original statements, its validation only requires a partial view of the system state. Hence, it is more likely that the CSM API is used to collect evidence for showing compliance.

Given a rule, we define a set of expressions called *minimal evidence set*. Collecting evidence for any of the expressions in a minimal evidence set is sufficient for proving that the selected resource is compliant. We generate a minimal evidence set by considering the statements in the original rule body

$$\begin{aligned}
&\neg cardholderdata(m).\\
&\forall N: \neg connected(m,N).\\
&\forall N: internal(N).\\
&\forall N: \neg(connected(m,N) \wedge \neg internal(N)).\\
&\forall N: \neg(cardholderdata(m) \wedge connected(m,N)).\\
&\forall N: \neg(cardholderdata(m) \wedge connected(m,N) \wedge \neg internal(N)).
\end{aligned} \tag{3}$$

Fig. 2. Minimal evidence set for an example policy.

that can be validated using the CSM API. This algorithm is described in Algorithm 1, and how it is used in our example policy is shown in Figure 2.

The set includes both simple negated statements (e.g., $\neg cardholderdata(m)$) and the negation of statement groups (e.g., $\neg(connected(m,N) \wedge \neg internal(N))$. We include negations of statement groups to increase the possibility of collecting a complete set of evidence. Free variables in the expressions represent universally quantified variables. It is often impossible to collect evidence for such a general statement. For example, proving the expression $internal(N)$ would require showing evidence that all networks in the system are classified as "internal." On the other hand, by considering expressions obtained through negating statement groups, we can reduce the scope of the universal quantification. In our example, the expression $\forall N : \neg(connected(m,N) \wedge \neg internal(N))$ requires collecting evidence that shows, for all networks, there is no network connected to m, which is not internal. When looking at the conjunctions of statements, we only consider semantically meaningful groups by taking the subsets that are "safe" (i.e., variables appear in at least one non-negated statement) and semantically connected (i.e., statements share variables). Verification of statements which do not share variables (e.g., verifying the group of two statements $cardholderdata(m) \wedge internal(N)$) would not provide any additional information compared to when each statement is verified independently.

2) Verification: We use the minimal evidence set to select the evidence that needs to be collected. Each expression in the set is considered, and the evidence is gathered using the method listed in Table IV. In the given example, the MaaS randomly selects a resource m from the set returned from ev_{list}. We can use $ev_{chd}(m)$ to verify if m stores cardholderdata. If the API returns a values specifying that m does not contain such data, then we can immediately show that the rule cannot be true at the current time for resource m. This response can be stored, signed, and returned to the organization. Otherwise, the analysis for other expressions in the minimal evidence set continues.

For a given resource, the verification has three possible outcomes. First, the evidence supports a minimal evidence set expressions, indicating that a proof of compliance has been found. Second, the evidence contradicts information provided by the user. If the inconsistency remains after a timeout, during which all messages have been received, an evidence for a possible security problem has been found. Third, the evidence is consistent with the user information but insufficient to prove any minimal evidence set expression. Here, the users are notified to perform a detailed inspection of the resource.

Algorithm 1 Determine minimal evidence sets

```
$var(P)$ : returns the set of variables in $P$
$\mathcal{P}(A)$ : power set obtained by taking each predicate of $A$
$R_b =$ rule body
$MES = \phi$ minimal evidence set
$R_v = \phi$
for all $s \in R_b$ that are CSM-verifiable do
    $MES = MES \cup \neg s$
    $R_v = R_v \cup s$
end for
$ES = \mathcal{P}(R_v)$
for all $e_i \in ES$ do
    if $e_i$ is safe, connected then
        $MES = MES \cup \neg e_i$
    end if
end for
```

TABLE V
SUMMARY OF THE CAPABILITIES OF THE CSM API TO VALIDATE PCI-DSS POLICIES.

Section	Total	Monitorable	Complete	Partial	CSM/ Monitorable
1	24	18	11	7	100%
2	8	8	1	2	37.5%
5	3	3	0	2	33.3%
6	22	2	1	1	100%

The process of sampling the compliance of a resource is repeated multiple times by selecting different resources. The random selection ensures that the organization does not know which resources will be selected. If the evidence collected through this randomized process does not contradict the organization's own monitoring information, it can help validate the authenticity of the monitoring data and increase the confidence in the overall state of compliance.

IV. Initial Evaluation

Our initial evaluation investigates the potential effectiveness of the proposed technique to verify PCI-DSS compliance. Looking through the PCI-DSS documents, we identified the policies that could be validated using the CSM API.

Table V summarizes our analysis. We consider PCI-DSS policies that involve network management and identify subsets of rules that our evaluation focuses on. As PCI-DSS policies are defined for manual auditing, several policies are not expressed in the way suitable for direct monitoring. For example, policy 1.1.1 requires the existence of a formal process for testing and approving changes in firewalls, and such a policy cannot be verified by any IT monitoring system. We define policies as **monitorable** if an automated system is able to gather the required information.

As the next step, we analyzed the type of information required for validating such policies. We considered the current literature in virtual machine introspection and identified techniques that can gather such information. These techniques are listed in Section III-A. We identified (1) a set of **complete** policies for which the CSM API is capable of gathering complete information required to check compliance; and (2) a set of **partial** policies for which it will only collect partial information required to check compliance, indicating that external data is required for completing the evaluation. The other monitorable policies cannot be validated through CMS API as they require monitoring through specialized software (e.g., agents). For example, software agents can check the use of default passwords (policy 2.1.1) or test the state of anti-virus software (policy 5.5.1). The CSM API does not provide access to information obtained through such means. The last column shows the ratio between the policies we can validate using CSM and the monitorable policies: both complete and partial policies are considered as the resource-based sampling can collect compliance evidence for both policy types. Our analysis shows that a significant portion of the policies can be validated using our framework.

V. Related Work

Bleikertz et al. [11] introduce an approach for verifying the correctness of a cloud infrastructure deployment. The authors acquire information about the user's configurations from the cloud API and use model checking and theorem proving for verifying the correctness of the deployment. While such an approach permits the verification of static and slowly changing properties, its application to frequently changing states such as network connections or running programs is challenging. Our framework complements user-managed monitoring systems and uses sampling to reduce the load on cloud provider.

Previous research analyzed techniques for preserving log integrity through digital signatures and encryption [12], [13]. While such techniques provide an effective protection against attacks aimed at modifying the stored logs, they cannot prevent false information being injected into the monitoring system by compromised applications or machines. To provide stronger guarantees on the logs being generated, numerous researchers [14]–[16] have explored the use of host VMs (e.g. Dom0 in Xen) or secure, isolated logging VMs that log the I/O responses and requests independent to the guest VMs. A compromised VM, without also compromising the host VM or isolated logging VM, cannot bypass this type of logging mechanism. These approaches, however, often do not address the 'semantic gap' problem and generate a very large amount of log data, making it difficult to analyze them efficiently. In contrast, our approach only requires a small amount of data to be sampled and analyzed.

VI. Conclusion and Future Work

We introduce a framework for validating integrity/accuracy of a large volume of logs in a policy-based monitoring system using a small sample of evidence data. Our CSM API enables organizations to acquire additional evidence about the compliance of their systems and resources. The VMI techniques used by the CSM API only provide access to a limited view of the system's state. We introduce a resource-based sampling strategy that can select evidence to show compliance of a resource out of such a partial view.

Our future work will address the issues in implementing the CSM API in an OpenStack testbed, and will quantify the reduction in the amount of evidence required to prove compliance with PCI-DSS policies due to the minimal evidence set approach. Additionally, Platform-as-a-Service cloud providers have access to high-level semantic information about the users' systems. Future work should extend the CSM API to take advantage of such information.

Acknowledgment

This material is based on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was partially supported by a grant of The Boeing Company.

References

[1] Payment Card Industry Security Standards Council, "Payment Card Industry (PCI) Data Security Standard," Payment Card Industry Security Standards Council, Tech. Rep., 2010.

[2] United State Government, "Federal Information Security Management Act (FISMA)," 2002. [Online]. Available: http://csrc.nist.gov/groups/SMA/fisma/index.html

[3] B. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," *ACSAC 2007*, pp. 385–397, 2007.

[4] R. H. Campbell and M. Montanari, "Multi-Aspect Security Configuration Assessment," in *ACM Workshop on Assurable & Usable Security Configuration (SafeConfig)*, 2009.

[5] K. Walzer, T. Breddin, and M. Groch, "Relative temporal constraints in the rete algorithm for complex event detection," in *DEBS '08*, 2008.

[6] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," *Proc. HotNets (October 2009)*, 2009.

[7] Amazon Web Services, "Elastic Compute Cloud (EC2)," 2011. [Online]. Available: http://aws.amazon.com/ec2

[8] J. Hizver and T.-c. Chiueh, "Automated Discovery of Credit Card Data Flow for PCI DSS Compliance," *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pp. 51–58, Oct. 2011.

[9] M. Christodorescu, R. Sailer, D. Schales, D. Sgandurra, and D. Zamboni, "Cloud security is not (just) virtualization security: a short paper," in *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 2009, pp. 97–102.

[10] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *NDSS Symposium*, 2003.

[11] S. Bleikertz, T. Groß, and S. Mödersheim, "Automated Verification of Virtualized Infrastructures," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 47–58.

[12] Wensheng Xu and David Chadwick and Sassa Otenko, "A PKI Based Secure Audit Web Server," in *Communication, Network, and Information Security*. ACTA Press, 2005.

[13] V. Stathopoulos, P. Kotzanikolaou, and E. Magkos, "A framework for secure and verifiable logging in public communication networks," in *CRITIS*, ser. LNCS, vol. 4347. Springer, 2006, pp. 273–284.

[14] Sujata Garera and Aviel D. Rubin, "An independent audit framework for software dependent voting systems," in *CCS '07*. New York, NY, USA: ACM, 2007, pp. 256–265.

[15] J. H. Huh and A. Martin, "Trusted logging for grid computing," in *Trusted Infrastructure Technologies Conference, 2008. APTC '08. Third Asia-Pacific*. IEEE Computer Society Press, October 2008, pp. 30–42.

[16] N. A. Quynh and Y. Takefuji, "A central and secured logging data solution for xen virtual machine," in *24th IASTED International Multi-Conference Parallel and Distributed Computing Networks*, Innsbruck, Austria, February 2006.