

# Automatic Management of Logging Infrastructure

Christopher R. Johnson  
Department of Computer Science  
Knox College  
Galesburg, Illinois  
crjohnso@knox.edu

Mirko Montanari, Roy H. Campbell  
Department of Computer Science  
University of Illinois at Urbana-Champaign  
Urbana, Illinois  
{mmontan2,rhc}@illinois.edu

**Abstract**—A secure and reliable network logging infrastructure is necessary for anomaly detection systems to be effective. Without reliable logs, an anomaly detection system would be unable to detect malicious activity. An attacker could even target the logging infrastructure before carrying out an attack to make their actions undetectable. Unfortunately, logging infrastructure is very difficult to maintain manually as compliance to separation of duty policies and constant monitoring are a burden on administrators. In this paper we present an architecture for a system that is able to automatically react to problems with the logging infrastructure. This system uses a method we present to detect that a problem exists, determine the best way to fix that problem, and then notify automated agents on the network to take action. It does all of this with little or no administrator involvement to reduce the burden of maintaining a secure logging infrastructure.

## I. INTRODUCTION

A secure and reliable logging infrastructure is at the root of most systems designed for detecting malicious insider behavior. If the logging infrastructure fails or becomes compromised, these systems may be unable to detect malicious activity: important events might not be properly logged and personnel might be able to falsify logs to make their actions untraceable. Hence, protecting the logging infrastructure is a priority in the management of any anomaly detection system. To provide a better protection against attack, the logging infrastructure should be resilient to failures and to intentional compromises. Failure resilience is generally obtained using redundant systems, while protection against intentional compromises can be obtained by enforcing separation of duty constraints between the personnel managing the systems and the personnel managing the logging infrastructure: personnel in charge of administering systems should not have privileges for modifying the logs of their actions, while personnel in charge of the logging infrastructure should have no privileges that allow them to perform actions on parts of the system not related to logging.

As enterprise systems frequently experience failures and changes in configuration or policy, the logging infrastructure needs to be properly changed to ensure reliable monitoring

of the system. Enforcing reliability constraints and separation of duty in a large enterprise system is challenging. Personnel performing changes on systems may need to wait for a separate person before they act so that corresponding changes can be made in the logging systems.

In this paper, we present an automated system which is capable of automatically ensuring that these separation of duty constraints and reliability policies on the logging infrastructure are satisfied even after planned or unplanned changes in the system configuration. Our design allows for automatic reconfiguration of the logging infrastructure as a consequence of changes in the system resulting from failures or from administrators' actions. Using this automated management of the logging infrastructure, we can reduce the burden on administrators because reactions to failures are automated.

Our system monitors the current system configuration and reacts to changes. The reaction process is composed of two phases. During the first phase, information about the logging and network configurations is verified for compliance to reliability requirements and to separation of duty policies. This phase is implemented by creating a Datalog model of the network configuration and by specifying associated inference rules that represent network security policies. Datalog inference is used to verify whether violations of policies can be derived from the current network and logging configuration [1]. Once the violation is detected, the system generates a *logical attack graph* [2]. The attack graph is a directed graph that describes the logging or network configurations that cause the policy violation. For example, suppose that we have a policy that all workstations must log to a secure logging server and that we have a network configuration where the logging server is placed behind a firewall to protect a vulnerable software that cannot be removed for legacy purposes. In this setting, an administrator could modify the firewall configuration, expose the vulnerability, and make the logging server vulnerable to attacks. Using our system, we can automatically integrate information about the network configuration, derive that the logging server is not a secure server anymore, and hence detect that the system is

not configured properly because the logging policy has been violated.

During the second phase, our system analyzes the attack graph and identifies the least-cost set of actions that must be taken to resolve the violation. The resulting set of actions can then be used by the system to change the configuration to a state that satisfies the logging policies. In our previous example, the action could be either to fix the firewall configuration (if admissible by the business constraints) or to move the logging service to another server that is currently secure.

The organization of the rest of this paper is as follows: In Section II, we give an overview of related work. In Section III, we describe the architecture of our system and how it can be used to enforce auditing policies. In Section IV, we describe our method, its underlying algorithms, and how it compares to previous work. In Section V, we look at the performance of our method and present results and analysis of simulations that were run on randomly generated network models. Finally, in Section VI, we present a direction for future work that may increase the performance of our method and may increase its applicability to more areas of Insider Threat and network security in general.

## II. RELATED WORK

Insider threat is a growing problem for organizations. Disgruntled employees can use their authorized access to sabotage computer systems, while others may be paid to copy sensitive documents for outsiders. Even honest, but careless employees can contribute to this problem by failing to pick secure passwords, thereby giving others their access [3].

A major focus of insider threat research is on anomaly detection systems that compare the current activity of each user with precomputed normal usage profiles [4]. If a user is engaging in activity outside of their normal usage pattern, it draws attention to their abnormal activity on the assumption that it is more likely to be malicious.

Organizations that use anomaly detection systems often use centralized, redundant, or distributed logging systems. The anomaly detection systems that we have surveyed all depended on reliable logs to be effective [4] [5] [6]. However, while techniques for the secure monitoring of machines [7] [8] can be used for generating trustworthy logs on each machine, and cryptographic techniques can be used to prevent log tampering [9], without proper configuration of the logging infrastructure the information collected on each machine cannot be securely analyzed in a central location. If the logging infrastructure is disrupted before

an attack begins, then the logs will not be there to begin with. Our solution complements these systems by detecting and by reacting to misconfigurations and failures in the logging infrastructure that would have compromised the ability of anomaly detection systems to collect data reliably and securely.

The motivation for our architecture was to extend configuration model checking solutions to react to the problems that they detect. Configuration checking systems proposed by Montanari et al and Lobo et al can be used to infer how changes in configuration may yield complex attack vectors [1] [10]. However, this type of system cannot propose solutions to the detected configuration problems. Our system extends such systems by computing solutions to complex violations.

Noel et al. [11] proposed an algorithm for the computation of minimal reconfiguration actions in polynomial time. However, their system is unable to account for the complex separation of duty policies and the reliability requirements that are common in the configuration of the logging infrastructure of an enterprise. The method presented in this paper can represent these constraints and can quickly compute the reconfiguration actions to perform for managing complex logging configurations.

## III. ARCHITECTURE

Our architecture consists of a centralized system that obtains information about the network configuration from a network of *agents*. These agents are trusted resident programs and/or kernel modules that collect data about the configuration of networked machines and send those data to the centralized system. These trusted agents use secure monitoring techniques for reliably monitoring the configurations of hosts even in the case of compromises [7] [8]. Additionally, each agent is able to perform a specific set of actions that change the configuration of the system.

Our system aggregates data from agents and discovers violations of security or reliability policies, and it sends that data to be processed using our reaction method. The output of that algorithm is a minimal set of actions required to solve the violation, called a *minimal hardening set*. The minimal hardening set is sent to the relevant agents, which perform the actions they are instructed to carry out and report back to the centralized system with the resulting configuration changes. A diagram of this architecture is shown in Figure 1. The components surrounded by a dotted line are present in systems for the detection of policy violations (e.g., [1]), while the others are specific to our system.

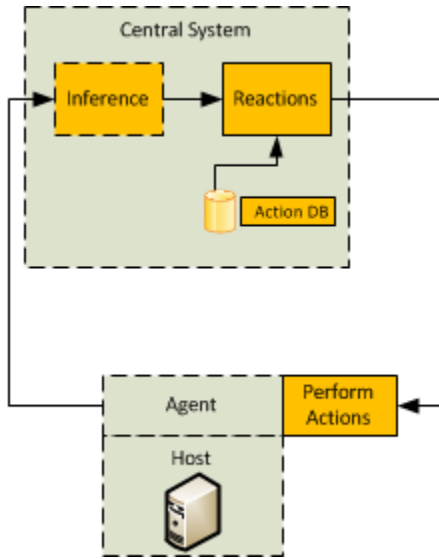


Figure 1. Reaction System Architecture

Another central aspect of our architecture is the *action database*, which is used by the centralized system to determine how to fix certain problems. Our action database is a list of entries composed of four parts: (1) the condition to be removed, (2) a set of preconditions to satisfy before it can be removed, (3) the commands necessary for the agents to remove the condition, and (4) a score for determining which actions are to be preferred, where lower is better. For example, we can negate the fact that a certain machine is providing a specific service by telling the agent to shut down the service. We can only do this, however, if the machine is running the agent and the agent knows how to disable that specific service. The specification of this action is shown in Figure 2. There can also be multiple ways of removing a condition, each with separate preconditions, scores, and commands. The action database is pre-populated depending on the capabilities of the agents and does not depend on the configuration of the system.

```
{?Machine srg:provideService ?Service ->
?Service srg:hasAction #Shutdown,
?Machine srg:hasAgent #TRUE :
shutdownService(?Machine, ?Service) [30, 80]}
```

Figure 2. Example entry in action database

### A. Logging Infrastructure Applications

In a simple example, our reconfiguration system can be used to ensure that at least one server is configured for remote logging at all times. Our policy is simply specified by saying that there must exist a machine to which all machines that must be audited are sending log information. A machine that

is not configured to log to the remote logging server will be found to be in violation of policy. The reaction system will determine the best way to resolve this violation, which will most likely be to change the configuration so that it logs to the correct server. Example rules and policies to enforce this are shown in Figure 3.

```
[rules]
/*
 * A server has a logging service if
 * it is running that service and that
 * service is a logging service.
 */
[(?Server srg:hasLogging ?Service) <-
(?Server srg:hasService ?Service),
(?Service srg:isType #LoggingService)]

/*
 * A machine is logging to a service if
 * the machine is configured to log to the
 * service and the machine can reach the service.
 */
[(?Machine srg:loggingTo ?Logging) <-
(?Machine srg:hasLoggingConfig ?Config),
(?Config srg:logsTo ?Logging),
(?Machine srg:reachService ?Logging)]

[policies]
/* Machines must log to the central server */
[(?Machine srg:loggingTo #CentralServer)]
```

Figure 3. Example of centralized logging configuration

A more complicated example that showcases our system’s ability to preserve auditability is in preventing an administrator from having control over all logging servers on a network. If someone had control over all of the logging servers, they could compromise the integrity of the logs or shut down all logging. Our policy would be that a single user cannot be an administrator on all of the existing logging servers. Possible reactions to a violation of this would be to restrict their access or to relocate logging. Example rules and policies to enforce this are shown in Figure 4.

Our goal is to reduce the burden to administrators by delegating maintenance of the logging infrastructure to an automated system. Administrators will be able to make configuration changes and rely on the automated system to make corresponding changes to the logging system.

## IV. REACTION METHOD

Our reaction method depends on the processing of logical attack graphs [2]. Each node in the graph represents some logical proposition, for instance, “User A has control of host H”. The incoming edges connect the node with its preconditions (i.e. the conditions that together imply the proposition), and outgoing edges connect the node with

```

[rules]

/*
 * A user controls a service if
 * they control the server on which
 * the service runs
 */
[(?Service srg:controlledBy ?User) <-
(?Server srg:hasService ?Service),
(?Server srg:controlledBy ?User)]

[policies]

/*
 * A user may not simultaneously control
 * a machine and the service to which
 * the machine sends logs
 */
![(?Machine srg:loggingTo ?Logging),
(?Service srg:controlledBy ?User),
(?Machine srg:controlledBy ?User)]

/*
 * A user may not simultaneously control
 * all logging services.
 */
![(#Logging_1 srg:controlledBy ?User),
...,
(#Logging_n srg:controlledBy ?User)]

```

Figure 4. Example of centralized logging configuration

others for which it is a precondition. The whole graph represents an attack by associating the goal of the attack (which represents some policy violation) with its preconditions, and it associates those conditions with their preconditions and so on to the fundamental reasons for the attack. An example of an attack graph describing how a user can use an exploit to control a system is shown in Figure 5. In that example, the preconditions for User A gaining control of System 2 are that User A controls System 1 and that System 2 is running a service with a vulnerability that allows User A to gain administrator access.

Logical attack graphs as described have been found to particularly suit the goal of finding and describing configuration errors [2].

Similar logical attack graphs are used by Noel et al to generate minimal hardening sets [11]. They provide an algorithm to simplify the graph under certain assumptions, and then they process this simplified graph to find a conjunction of disjunctions that logically represents the attack goal in terms of its preconditions. By partially ordering these preconditions and picking the “least” one, they find a minimal hardening set that can be used to make the attack goal unreachable.

However, by dealing directly with attack graphs their method does not allow more complicated logical restrictions that can

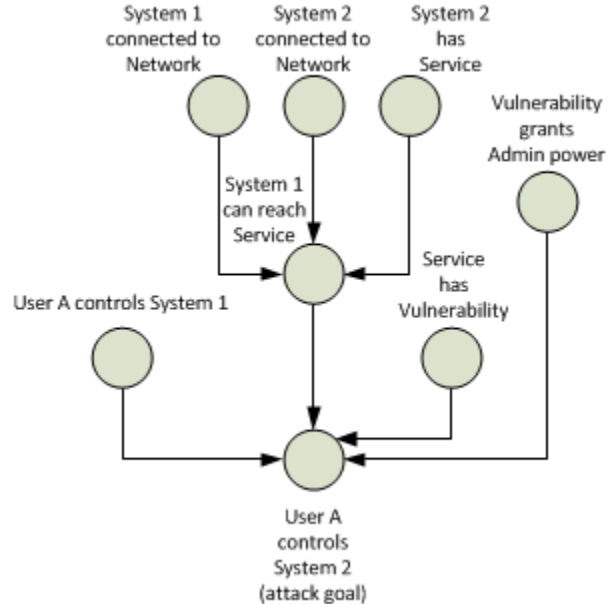


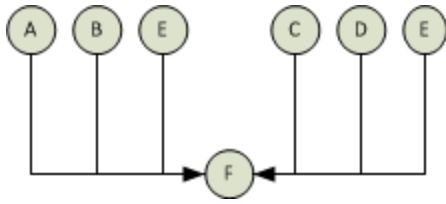
Figure 5. Example of an Attack Graph

change which hardening sets are considered. For example, in a configuration with several distributed logging servers, we may have the policy that at least one logging server must be running at all times. This would eliminate from consideration any hardening sets that call for disabling all remaining logging servers. Our reaction method supports such restrictions, which are necessary for practical use in application domains such as auditability assurance.

In the rest of this section we present our algorithm for calculating minimal hardening sets. Our algorithm consists of four phases: Phase 1 queries the inference system to construct a hypergraph representation for each violation. Phase 2 merges all of the resulting hypergraphs into a single hypergraph. Phase 3 converts this hypergraph into an instance of the Maximum Satisfiability Problem. Finally, Phase 4 feeds this problem to a SAT-solver and interprets the result.

#### A. Phase 1: Hypergraph Representation

Instead of using an attack graph, our solution generalizes the concept by representing the attack as a directed hypergraph, also known as an AND/OR graph due to its applicability to representing logical statements under conjunction and disjunction [12]. A hypergraph  $G$  is represented as  $G = (V, E)$ , where  $V = \{v_1, v_2, \dots, v_n\}$  is the set of vertices (in our case, vertices are conditions that have been found to be true) and  $E = \{(T_1, H_1), (T_2, H_2), \dots, (T_m, H_m)\}$  is the set of hyperedges. A hyperedge is a pair of disjoint



$$(A \wedge B \wedge E) \vee (C \wedge D \wedge E) \Rightarrow F$$

Figure 6. Example of a Hypergraph

subsets of  $V$ , where each  $T_i$  is called the tail of a hyperedge and each  $H_i$  is called the head; a hyperedge is a directed connection from all of the vertices in the tail to all of the vertices in the head (a regular directed graph is just a hypergraph in which  $|T_i| = |H_i| = 1$  for all hyperedges in  $E$ ). For our AND/OR graph representation of attacks, the conjunction of all vertices in the tail is a precondition of the vertex in the head. If a single vertex is in the head of multiple hyperedges then we interpret this as a disjunction of conjunctions; any one of the incoming conjunctions implies the common condition. An example is shown in Figure 6.

The inference system we use [13] allows us to find the sets of preconditions for any condition being true. We construct the initial hypergraph for each violation by starting at the goal of the violation, finding its preconditions, and then recursively repeating this process for each precondition. The result of this logical traversal is represented as a directed hypergraph that describes how the goal is derived from preconditions.

Representation as a directed hypergraph allows us to directly create a conjunction of propositional logic statements that entirely describe the violation. More immediately, however, this representation allows us to merge several attack hypergraphs into a single hypergraph that represents the reasoning behind all policy violations that exist on the network.

### B. Phase 2: Hypergraph Merging

Merging all attack hypergraphs into a single one that needs to be resolved has the benefit of detecting and preventing cycles and contradictions that would arise from solving each existing violation in turn. For instance, two violations may exist, but solving either violation may make the other impossible to solve given the current policies. Merging the attack hypergraphs allows the system to discover the best solution for the whole configuration that may not have been optimal for individual attack hypergraphs. We can detect the case that a complete solution is not possible and allow the system to prompt an administrator for a solution or take some other action. A discussion of possible ways to deal

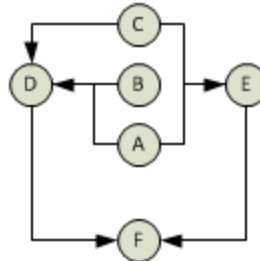
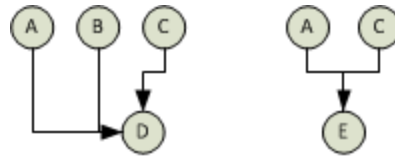


Figure 7. Example of Hypergraph Merging; F becomes the Master Goal

with this situation appears later in Section VI.

Our algorithm for merging a list of hypergraphs into a single one is as follows. We first create a new, empty hypergraph. Then we put all vertices from each hypergraph in the list into the new hypergraph, without creating duplicates. Next, we add each edge appearing in any hypergraph to the new one. Because we have no duplicates, this relinks the new hypergraph using all of the edges from each of the original hypergraphs. We construct a new vertex, which we will call the Master Goal, and insert this into the graph. We finally create a new edge from each of the original graphs' goals to the Master Goal. The result is an interconnected single hypergraph that has a single root goal. Because the precondition of the Master Goal is a disjunction of all of the original goals, finding a hardening set for that goal is the same as finding a single hardening set that solves all violations simultaneously. An example of hypergraph merging is shown in Figure 7. Pseudo-code for the algorithm is shown in Figure 8.

### C. Phase 3: Conversion to Weighted MAXSAT

The next part of our implementation is direct conversion of this resulting hypergraph into an instance of the Weighted Maximum Satisfiability Problem (Weighted MAXSAT) [14]. The Boolean Satisfiability Problem was the first known NP-Complete problem, and it is stated as follows: Given any propositional logic expression formed with variables, AND, OR, and NOT, is there an assignment of TRUE and FALSE to those variables that will satisfy the expression (is the expression satisfiable)? The Maximum Satisfiability Problem asks the question: Given a number of propositional clauses, what is the maximum number of those clauses that

```

L is a list of hypergraphs
G is the list of goals

// Final is an empty Hypergraph
Final := (Vfinal = {}, Efinal = {})

// Add each vertex from each graph,
// but do not include edges yet
foreach Hypergraph (V,E) in L:
  foreach Vertex vert in V:
    if Vfinal does not contain vert:
      Add vert to Vfinal
    endif
  end
end

// Now add all edges
foreach Hypergraph (V,E) in L:
  foreach Edge (T,H) in E:
    if Efinal does not contain (T,H):
      Add (T,H) to Efinal
    endif
  end
end

// Now create and link the Master Goal
Create and add MasterGoal to Vfinal
foreach Vertex goal in G:
  Add ({goal}, {MasterGoal}) to Efinal
endif

// MasterGoal is the new goal,
// Final is the new hypergraph

```

Figure 8. Mathematical Pseudo-code for Hypergraph Merging

can be simultaneously satisfied? Weighted MAXSAT is an extension of this in which each clause is given a weight, and the problem is then to maximize the sum of weights of satisfied clauses. Both MAXSAT and Weighted MAXSAT are NP-Hard [14]. Even though these problems have no known polynomial-time solution, fast SAT-solvers exist that can deal with the moderately sized Weighted MAXSAT problems we generate very quickly.

Our algorithm to convert our attack hypergraph into a Weighted MAXSAT problem takes advantage of our hypergraph representation. We associate a variable with each node in our hypergraph, and we convert each edge, representing logical implications, to their corresponding disjunction, i.e.,  $(A_1 \wedge A_2 \wedge \dots \wedge A_n \Rightarrow C) \rightarrow (\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n \vee C)$ . We add all of these as clauses (to be joined by conjunction) with a practically infinite weight. Because the solver tries to maximize the score of the satisfied clauses, these clauses must be satisfied unless it is impossible to solve.

Next, we integrate information about the possible actions that the system can perform by looking at the action database. For each vertex that has no incoming edges we check in our action database to see if there is an action we can take that will negate the corresponding condition. If no such action exists, we assert the corresponding variable alone with another practically infinite weight so that it cannot be negated. If an action does exist, we use the corresponding

```

1 : A
1 : B
1 : C
1 : D
1 : E
∞ : ¬A ∨ ¬B ∨ ¬E ∨ F
∞ : ¬C ∨ ¬D ∨ ¬E ∨ F
∞ : ¬F

```

Figure 9. Weighted MAXSAT representation of Figure 6

score of that action as the weight for the variable. While cycles can be present in the hypergraph, they do not affect the association to actions. The actions performed by the agents change conditions which have no incoming edges because they change explicit configuration settings on hosts. This type of configuration has no preconditions because they are directly read by the agent from the host, and not inferred by rules in the central system. Hence, if a policy violation is solvable, all actions that can be performed for solving it are going to be represented by a vertex with no incoming edges, even in the merged hypergraph, and they are going to be outside of any cycle.

The scoring of actions can be defined by each organization to represent an overall “cost” or “risk” of performing a particular modification to the configuration. The scoring we used for testing has two components (which are summed for the final score). First, we consider a rough estimation (based on experience) of how likely it would be for the application of this action set to fail. Second, we consider the potential negative consequences of the action that might require manual intervention (e.g., disabling a critical service has a high chance of administrator involvement because an administrator must restart the service as soon as possible). In the example in Figure 2, we had a moderate to low chance of failure (30 out of 100), and a high chance of administrator involvement (80 out of 100) if an unknown service is suddenly shut down. The final score is  $30 + 80 = 110$  out of 200.

After converting actions into the MAXSAT problem, the system adds to the MAXSAT representation additional constraints to the system that are not captured by the hypergraph. These are rules such as, “at least one logging server must be left running.” This would be represented by asserting that Logging1 is running OR Logging2 is running, and so on. Finally, we add the negation of the Master Goal’s variable with a practically infinite score. The result of this procedure being applied to the hypergraph from Figure 6 is shown in Figure 9.

#### D. Phase 4: Solving the Problem

Asserting the negation of the Master Goal introduces an immediate contradiction. Since all of the clauses together show that the Master Goal is true, the solver is forced to remove clauses to make the remaining ones satisfiable. Because the solver is trying to maximize the weight of the final system, it is equivalently trying to minimize the weight of what it removes. It will therefore find the set of clauses with the lowest combined score that need to be negated to remove the contradiction presented by the negation of the Master Goal. This is precisely the minimal hardening set for all of the violations.

Using the example in Figure 9, the minimal hardening set is  $(\neg E)$ , as this would resolve the contradiction (by removing the clause that asserts  $E$ ) while maximizing the score of the remaining clauses.

In cases where it is not possible to fix the problem, the system will attempt to discard an implication or negate a variable that it is unable to negate. We can detect when it tries to discard something it should not have, and a few ways of dealing with this situation are discussed in Section VI. The chances of such a situation occurring can be reduced by creating a well-populated rule database, as that will give the system more capabilities to fix a wide range of problems.

### V. PERFORMANCE AND RESULTS

We tested our method by solving policy violations in randomly generated network configurations with a certain number of hosts, subnets, users, and software. This was done by adding a number of hosts to different subnets in a model, then randomly assigning some amount of software to each host, where the software has probability  $P$  of being vulnerable. The primary violation that our performance-test simulations resolve are whether or not any systems can be compromised by a hypothetical “malicious attacker” on the network who exploits software vulnerabilities to gain control of systems, but the results are applicable to our target application in verifying the logging infrastructure. We used this test to examine the performance of our method on small-to large-scale networks.

We ran our simulations on an Intel Xeon 2.3 GHz system with 6MB cache and 4GB of RAM. Results from our simulations show that the time to compute hardening sets given attack graphs is very small, less than one second on networks of up to 3,000 hosts. Tests on larger networks of around 10,000 hosts have exhibited computation time of approximately 1.5 seconds. As seen in Figure 10, performance of our algorithms appear to be mostly independent

#### Average Hardening Time vs. Network Size

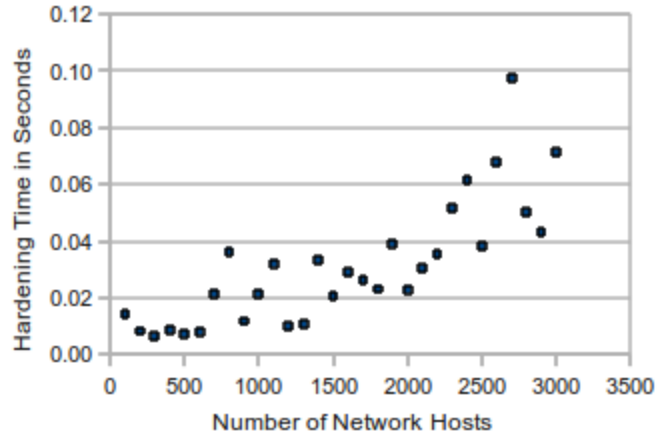


Figure 10. Relationship between hardening time and network size

#### Hardening Time vs. Hypergraph Nodes

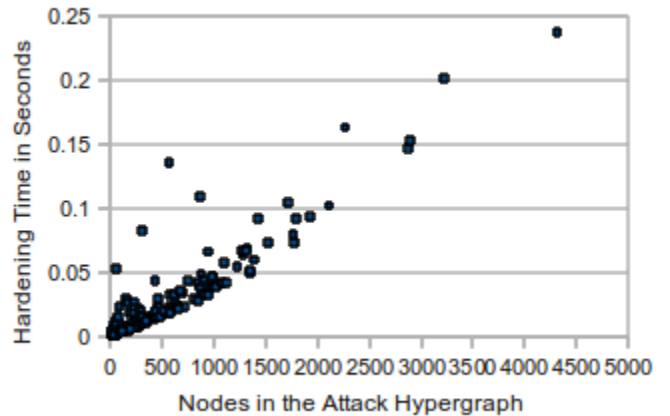


Figure 11. Relationship between hardening time and hypergraph size

of network size. Each data point in that graph is the average hardening time of four simulations. The hardening time generally trends upwards as the number of hosts increases, but it is not a very high correlation. Figure 11 shows that the size of the attack hypergraph in question plays a much larger role in influencing the performance of our algorithms. While larger networks have the potential for very large attack hypergraphs, large attack hypergraphs will not necessarily arise all the time, which is why most of the data points in Figure 11 are clustered toward the lower x-values. The results show that our algorithm is able to merge hypergraphs, convert them to Weighted MAXSAT, and solve the resulting problem all in far less than one second for even large attacks.

The limiting factor in our simulations is the time required to derive and recover the original attack graph from the

inference system. The time to derive the attack from the inference system can occasionally exceed 30 minutes for networks of approximately 2,500 hosts and can take days on networks of more than 10,000 hosts. These delays, however, are greatly exaggerated by the simulation method. New information and vulnerabilities will arise periodically in an actual deployment, which means that a lot of the inferences will already be precomputed when analysis starts. Our simulator attempts to derive possible attacks from a clean system, causing it to take a very long time. Incremental updates of the model in a production environment significantly reduce this delay [15].

## VI. FUTURE WORK

Several important research problems still need to be addressed to provide a complete solution to the problem of automatic reconfiguration of the logging infrastructure.

We have not yet addressed the problem of a merged hypergraph being impossible to solve. Even when given a very rich action database, it may simply be impossible to automatically resolve some issues that arise. We can detect when this is the case by detecting if the solver says it is necessary to remove a constraint or to negate a condition that it is unable to negate. Because these clauses have such high weights in the converted problem, requiring the removal of any of them implies that even taking all possible actions together is not sufficient to fix the violation.

If simply notifying an administrator of the problem and asking them to resolve the issue manually is not an option, an alternative is solving each unmerged hypergraph in turn and notifying administrators to manually fix the rest. An ordering can be used so that important violations are resolved first before other reconfigurations make them impossible. Another option would be to present the administrator with a few different compromises for how the problem can be partially fixed, and allow them to select the best one. This, however, has the obvious drawback of requiring that the administrator be available to make the change.

While the worst-case time of the Weighted MAXSAT is exponential in the size of the problem, our experiments show that computing optimal reactions for realistic systems is fast even for large networks. The real complexity of SAT depends on the structure of the problem that is represented by the interconnectivity between variables. Because real networks are designed to be managed manually by people, the complexity of the system and the relation between variables is limited to avoid overwhelming the capacity of experts in detecting security problems. We are planning to investigate the relationship between the complexity of the

network and the security of the system: a SAT model of the network that is very complex could be used to notify administrators of potential problems in the systems and suggest methods for simplifying the structure of the network by reducing interactions between variables to attain a system which is easier to manage.

The solution proposed in this paper assumes to have complete and correct information about the current configuration of the system and assume to have a complete understanding of the consequences of actions. While we believe that these assumptions are realistic when we restrict our attention to the well-known logging infrastructure subsystem, future work should analyze the problems of performing configuration changes when the information about the system is incomplete. Because of the partial knowledge about the system and about the interdependency between components, reconfiguration actions performed in this scenario can create unintended consequences.

Additionally, we are working on a full implementation of our system and logging infrastructure on a realistic network testbed that represents the production conditions of an enterprise network and incorporates realistic security policies. Using our testbed, we will be able to perform more experiments on the practical applicability of our design in an enterprise network.

## VII. CONCLUSION

In this paper we have presented an architecture and algorithms for a system that is capable of automatically reconfiguring the logging infrastructure of a network in response to problems. Our experiments demonstrate that the computation of automatic reconfigurations can be performed on the order of seconds, even for large networks.

This system eases the burden on administrators and complements anomaly detection systems that are primarily used to identify insider threats when they occur. If the logging infrastructure were compromised, it would severely inhibit threat detection as well as make attacks undetectable. By preserving that infrastructure, our system can ensure that no attacks are conducted silently.

## REFERENCES

- [1] M. Montanari and R. H. Campbell, "Multi-aspect security configuration assessment," in *SafeConfig '09: Proceedings of the 2nd ACM Workshop on Assurable and Usable Security Configuration*. New York, NY, USA: ACM, 2009, pp. 1–6.

- [2] X. Ou, W. Boyer, and M. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 2006, p. 345.
- [3] T. Noonan and E. Archuleta, "The insider threat to critical infrastructures," The National Infrastructure Advisory Council, Tech. Rep., 2008.
- [4] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, no. 12, pp. 3448–3470, 2007.
- [5] W. Lee and D. Xiang, "Information-theoretic measures for anomaly detection," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, pp. 130–143.
- [6] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2003, pp. 251–261.
- [7] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.
- [8] B. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *acsac*. IEEE Computer Society, 2007, pp. 385–397.
- [9] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 2, pp. 159–176, 1999.
- [10] J. Lobo and V. Pappas, "C2: The case for a network configuration checking language," *IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 29–36, 2008.
- [11] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs, "Efficient minimum-cost network hardening via exploit dependency graphs," in *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2003, p. 86.
- [12] G. Gallo, "Directed hypergraphs and applications," *Discrete Applied Mathematics*, vol. 42, no. 2-3, pp. 177–201, Apr. 1993.
- [13] J. J. Carroll, D. Reynolds, I. Dickinson, A. Seaborne, C. Dollin, and K. Wilkinson, "Jena: Implementing the semantic web recommendations," in *The 13th World Wide Web Conference*, 2004, pp. 74–83.
- [14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.
- [15] D. Saha, "Extending logical attack graphs for efficient vulnerability analysis," in *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2008, pp. 63–74.