Decentralized Enforcement of Security Policies for Distributed Computational Systems

Arie Orlovsky
Decentralized Enforcement of Security Policies for Distributed Computational Systems

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

ARIE ORLOVSKY

Submitted to the Senate of the Technion – Israel Institute of Technology
ALUL 5766 HAIFA August 2007
The Research Was Done Under the Supervision of Assoc. Prof. Danny Raz from the Department of Computer Science, Technion, in the Department of Computer Science, Technion

THE GENEROUS FINANCIAL HELP OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY IS GRATEFULLY ACKNOWLEDGED
I would like to express my appreciation to people without whom this work would not be completed.

My advisor Assoc. Prof. Danny Raz, for his valuable guidance and support throughout every step of the research and beyond.

I thank Prof. Shmuel Kats, Dr. Orr Dunkelman, Kfir Karmon, Denis Krivitski, Itay Ma- man and Lina Zarivach for many helpful discussions about the paper that led to this thesis. Many thanks to anonymous reviewers and other people that I may have forgotten to mention.

And finally to my family and my beloved fiancee, Luba, for their support and belief in me.
# Contents

Abstract 1

1 Introduction 2

2 Related Work 5

3 Computational Model, Local and Global Policies and Their Enforcement 8
   3.1 Computational Model 8
   3.2 EM-Enforceable Security policies 9
   3.3 From Local to Global Policies 10
      3.3.1 HB-Interpretation of Local Security Policies 12
      3.3.2 PHB-Interpretation of Local Security Policies 15

4 Enforcement Mechanisms 17
   4.1 A Centralized Enforcement Mechanism 18
   4.2 Distributed Enforcement Mechanisms 19
      4.2.1 Decentralized PHB Policy Enforcement Mechanisms 20
      4.2.2 HB Algorithm 28
   4.3 The Loosened Model 30
      4.3.1 Correctness Proof 35

5 Implementation 37

6 Performance Evaluation 42
## List of Figures

3.1 “No Send After Read” - the local automaton. ........................................ 11
3.2 “No Send After Read Unless Certify” - the local automaton. ................. 14
3.3 An execution example .................................................................................. 15

4.1 “No Send After Read” - A global automaton for 2 nodes. ...................... 19
4.2 An example of an PHB algorithm execution, enforcing PHB interpretation of “No Send After Read”, in three a node domain. ......................... 24
4.3 HB Algorithm, Enforcing HB Interpretation of “No Send After Read Unless Certify”................................................................. 30

5.1 The GUI of Distributed Security Enforcer ................................................ 39
5.2 Execution and data Flow ............................................................................ 40

6.1 “Chinese Wall” Local Automaton............................................................. 43
6.2 Overhead as a function of application behavior in a single node. .......... 44
6.3 Application Execution Times In Six Node Domain .................................. 45
6.4 Percentage Overhead Breakup. ................................................................. 46
6.5 Overhead Breakup in Six Node Domain.................................................... 47
List of Tables
Abstract

The shift from single server environments to globally distributed systems presents a great challenge in terms of defining and enforcing appropriate security policies. This is, among other things, due to the fact that the actual order of events in an asynchronous distributed environment is not always defined. In addition, security policies often depend on the actual information exchange among the distributed entities.

In this thesis we study the problem of adapting security policies to distributed environments such as grids and mobile code systems. We define what a global security policy is, and indicate some of the difficulties in translating local policies to the entire distributed environment. Then, we propose efficient and scalable security mechanisms for the enforcement of global security policies in distributed computational systems. These mechanisms are based on multiple instances of execution monitors (smart sandboxes) running on the distributed entities and on efficient security information sharing among them. We show that the subclasses of EM policies enforceable by these mechanisms, contain useful and real live security policies such as global information flow policies.

We provide prototype implementation of the security mechanism capable of defining and enforcing global security policies. This mechanism uses AspectJ to intercept security relevant events before they occur and terminates the execution if a target application is about to violate security policy.
Chapter 1

Introduction

Distributed computational systems such as grids, mobile agents systems and large scale clusters are gaining more and more popularity both in industry and academia. Providing security mechanisms for these systems is still a very big challenge. This is partially due to the large number of users, and the fact that code produced by one party is executed on resources owned by others. For instance, TeraGrid\(^1\) and EGEE\(^2\) grids may run millions of jobs on behalf of thousands users using resources located on several continents.

Naturally, resources as well as internal data should be protected against intentional or unintentional misuse. The system should be protected against unauthorized resource utilization and denial of service attacks; internal data should only be accessed by authorized parties and no leakages of classified data should be allowed. Such security requirements are commonly kept by defining and enforcing appropriate security policies.

A security policy is a very broad term and most generally can be described as specification of the security properties that a given system should possess. When dealing with mobile code systems we choose to use the following definition: security policy defines executions that, for one reason or another, has been deemed unacceptable. That is, the policy defines exactly all the prohibited state transitions.\(^3\)

Current security mechanisms in distributed systems are based on Authentication, Authorization and Access control (AAA). These mechanisms allow enforcement of general-

\(^1\)http://www.teragrid.org/
\(^2\)http://public.eu-egee.org/
\(^3\)Note that we do not mean that each such prohibited executions is stated explicitly, but rather that the short presentation of the policy defined them in an unambiguous way.
purpose security policies such as: User X is allowed to access resource Y between 8:00 and 16:00. However, in many cases AAA based policies may be too restrictive. For instance, security policy preventing the leakage of internally read data to the outside world cannot be enforced exactly, because the decision on whether to deny or allow the send operation depends on the current state of the application. I.e., “send” should be denied if the data was accessed and allowed otherwise. AAA based security policies can deny the target from reading internal data or from communicating with outside world, but the authorization decision is not based on the current security state of the running target. Such policies are referred to as \textit{stateless} policies.

We are primarily interested in \textit{stateful} security policies. These policies allow or prohibit security relevant events based on the security state of the target. The state is a function of the execution history of a the target.

Our current work is based on the theoretical work of Schneider \cite{schneider2007, schneider2013} in which characterizations of stateful security policies enforceable by Execution Monitoring is presented. An Execution Monitor, or EM, is an enforcement mechanism that works by monitoring execution steps of a target and stopping it if it is about to violate the security policy being enforced. An EM can be conceptually divided into two mechanisms: security relevant event recognizer and enforcement engine. The recognizer detects security relevant events and forwards them to the enforcement engine. The enforcement engine, in its turn, decides whether the event is allowed, based on the current security state of the target.

The work of Schneider and its followers uses a centralized enforcement decision mechanism which may receive events from different components of the system. These security mechanisms are planned to enforce local security policies where all the security relevant events are produced by the local node. The shift of the computational environment towards a more scalable distributed computing, poses a major challenges in terms of interpreting and efficiently enforcing security policies in such distributed environments. The interpretation complexity is, among other things, due to the fact that global order of events cannot be efficiently determined. Centralized enforcement in large scale computational systems is not feasible due to the heavy congestion resulted by the large number of events. Moreover, centralized enforcement reduces the concurrency, increases the latency of local executions and becomes a single point of failure.
In this thesis we study this interesting problem and propose efficient decentralized enforcement mechanisms for stateful security policies. The enforcing mechanisms are based on multiple instances of EMs performing local monitoring and efficient security information sharing between them. We also present our AspectJ [15] prototype system implementing our techniques.

The rest of the thesis is organized as follows: In Section 2 we present related work. In Section 3 we present the computational models, give a formal definition of security policies and discuss the challenges related to the shift from local to global policies. Section 4 describes a centralized enforcement mechanism for global policies and in Section 5 we describe our novel distributed enforcement mechanisms and prove their correctness. In Section 6 we present our prototype implementation of the security mechanisms, and in Section 7 we evaluate their performances. Finally, we summarize our work and discuss future directions in Section 8.
Chapter 2

Related Work

Several fields are related to our work. The theoretical part relies on the work that was done in specification of classes of security policies enforceable by various mechanisms. Schneider [26] provided the specification of policies enforceable by EM. Execution Monitor, or EM, is an enforcement mechanism that monitors the execution steps of a target and terminates it if it is about to violate the security policy. The EM is based on Buchi-like security automata that enforce safety properties. Mechanisms that are capable of performing other actions (and not only termination) upon policy violation, or capable of using knowledge about future execution steps of the targets are excluded from the class of EM mechanisms. We provide formal definition of a security automaton in Chapter 3.

Bauer et al. [4] extended the original class of EM enforceable security policies by including the policies enforceable by editing automata [18]. These automata are capable of modifying the execution sequences of the target rather than just terminating it. I.e., an editing automaton can suppress an event without terminating an execution or it can insert additional events to make the sequence legal under the policy being enforced. The set of policies enforceable by editing automata strictly contains all EM enforceable policies.

Fong on the other hand [9], tried to narrow the EM class by limiting the run-time information available to the Execution Monitor. He studied the power of shallow history automata or SHA. SHA remember the shallow history of previously allowed security events. I.e., the state of the automaton is the set over previously allowed security events. These automata are formally defined in Section 5.
Our prototype enforcement tool relates to language based security systems such as Java Virtual Machine (JVM) [19] and Common Language Runtime (CLR) [21]. These mechanisms use stack inspection rather than the entire history of the computation as a base for the security mechanism. Therefore, many useful security policies can not be enforced exactly by this mechanism. In addition, the Java Security Manager only allows monitoring of a limited number of the predefined security relevant events, therefore restricting the events space available for the enforcement mechanism.

The theoretical work which classified the policies enforceable by Buchi-like and editing automaton led to real systems implementing these models [7, 8, 5]. These systems work by rewriting target code and inserting the security automaton and monitor into the target application. These systems are designed to enforce local security policies. Even though our prototype implementation works in a similar way, its main goal is to enforce global security policies in a distributed environment.

We study the problem of enforcing security policies in distributed settings, therefore our work relates to the field of policy enforcement in distributed systems. Extensive work has been done in the field of global stateless policies and their centralized enforcement in distributed computational systems. Foster [11, 10] specified security requirements for the grid systems. Later on, various centralized enforcement systems such as those described in [25, 10] were developed to enforce stateless global policies in virtual organizations [10]. Our work differs from these systems in that it allows the distributed enforcement of general stateful security policies.

Various decentralized mechanism were also presented. Karmon and Schuster [14] developed highly scalable decentralized mechanism capable of enforcing budget based policies. However, it is limited only to budget consumption policies and not general enough to enforce stateful EM-enforceable policies.

Our work is closely related to the decentralized mechanism proposed by Minsky [24, 22, 23] and Sen [27]. Both present stateful mechanisms for distributed systems. The “Law-Governed Interaction” (LGI) [24] of Minsky is based on controllers that moderate the network traffic going to and coming from monitored nodes. Minsky et-al showed that several global stateful policies such as Distributed Chinese-Wall [22] can be enforced using LGI. There are several significant differences between LGI and our work. LGI only
deals with the exchange of messages between targets, and is not sensitive either to the behavior of targets, or to changes in their internal state. LGI is not automatically enforced on the targets and the enforcement depends on agents cooperation. Finally, unlike the LGI related work this thesis concentrates on the space of the enforceable policies and their relation to the EM enforceable policies.

Sen et al. [27] introduced the past time distributed temporal logic (pt-DTL), (a variant of the past time linear temporal logic), a decentralized algorithm for monitoring the requirements specified in pt-DTL, and a software system implementing the proposed techniques called DIANA. Sen’s techniques were developed to perform decentralized monitoring of safety in the context of software testing for distributed systems. DIANA allows the monitoring of the safety properties, specified in terms of local variables, in distributed applications developed in the DIANA framework. In contrast with Sen’s work, we concentrate on enforcing security and interpretation of local EM enforceable policies in distributed environments. The safety properties are specified by the Buchi-like security automata in terms of events performed by targets rather than local variables. In the context of security for distributed mobile code application, our framework can execute any external code and not only code developed internally using a specified framework.

In our work we present decentralized mechanisms capable of enforcing EM-enforceable stateful security policies. To the best of our knowledge, this is first study of interpretation of local EM-enforceable stateful policies and their decentralized enforcement in distributed systems.
Chapter 3

Computational Model, Local and Global Policies and Their Enforcement

In this chapter we start with a specification of the computational model, proceed with a formal definition of EM-enforceable security policies and security automata capable of enforcing them, specify the differences between local and global policies and finally discuss the difficulties in the interpretation of local policies in a distributed settings.

3.1 Computational Model

A distributed system is a collection of computational nodes located in various physical locations throughout the network. Each node possesses separate CPU and memory and is connected via reliable communication channels to the other nodes. FIFO is assumed on each of these communication channels.\(^1\)

**Definition 1.** A domain is a collection of computational nodes governed by the same global security policy.

Virtual organizations \([11,10]\), clusters and mobile code administrative domains \([2]\) as well as whole grids are examples of domains.

**Definition 2.** The targets of a policy are instances of (mobile) code that are executed within the boundaries of a domain.

\(^1\)In some of the algorithms we present, the reliability and FIFO requirements are not essential.
We assume that all the communication between targets during computation is done via unidirectional message passing. In addition, we assume that a message is not modified intentionally or unintentionally after it leaves the source node.\footnote{Note that our techniques are general enough and can be applied to multiple threads running within the same JVM. However, to simplify the presentation we assume throughout the thesis that only one target is executed in each node.}

We distinguish between two communication models:\footnote{We will show that communication models have direct impact on the enforcement mechanisms.}

- The restricted model assumes that the enforcement mechanism controls all possible communication channels used by the targets. The model also assumes the absence of side channels such as time or power consumption of target’s execution, which cannot be monitored by the enforcement mechanisms.

- The loosened model on the other hand assumes possible information exchange undetectable by the enforcement mechanisms.

### 3.2 EM-Enforceable Security policies

Let $I$ be a finite alphabet of all observable security events, and let $\Sigma$ be set of finite executions\footnote{Schneider~\cite{26} considered finite as well as infinite sequences of access events. Following the practice of Bauer et al.~\cite{4} and Fong~\cite{9}, we only consider finite sequences of access events in this thesis.}, that is, $\Sigma$ is a language over the alphabet $I$. We consider the class of EM-enforceable policies\footnote{This requirement can be omitted, by adding message authentication codes to the messages, but it is beyond the scope of our work.} \cite{26}. These are prefix-closed policies characterized by a detector $\hat{P}$ that detects all the forbidden words over $I$. The detector $\hat{P}$ must satisfy the following conditions:

- $\hat{P}$ detects whether an individual execution satisfies the policy, based on that execution alone. Not all policies satisfy this requirement. In Information flow policy, for example, detecting whether data moves from one variable to another cannot be necessarily determined based on single execution.
\[ \neg \hat{P}(\sigma) \Rightarrow \forall \omega \in \Sigma \neg \hat{P}(\sigma \omega) \]

Hence, the EM-enforceable security policies are a subset of the safety properties \([16]\) stipulating that “nothing bad” happens during execution. More informally it states that if a bad sequence has taken place it cannot be extended to satisfy the policy.

- \(\hat{P}\) is recursively decidable.

EM-enforceable security policies are specified by the detectors satisfying these conditions.

It is proved in \([26]\) that the class of EM policies is enforceable by security automata. A security automaton is variant of Buchi automaton \([3]\) capable of recognizing safety properties. It is defined as follows:

\[ A = (I; S; s_0; \delta)^S \]

- \(I\) is a finite or countably infinite set of observable security events,
- \(S\) is a finite or countably infinite set of automaton states,
- \(s_0 \in S\) is an initial state,
- \(\delta : I \times S \rightarrow S\) is a (possibly partial) transition function.

The final state is not specified and the input sequences is accepted by the automaton if a transition function is defined for every event in the sequence. Otherwise the sequence is rejected and the target is terminated. We assume that the local policies for a single node are defined by such an automaton.

### 3.3 From Local to Global Policies

When we move from a single node to a distributed environment we need to transfer the local policy into a global policy that is applied to the entire domain. Local policies are defined by security automata. For the purpose of this thesis we only consider automata with finite number of states. It is actually not a real limitation since Shneider \([26]\) himself acknowledged that security policies in real systems require finite number of states.
specified exclusively in terms of local events performed sequentially by a single target on specific node. Global policies, on the other hand, depend on the global state of the whole system and are influenced by events occurring at different nodes of the domain. In particular, different local security events, each occurring at a single computational node, may move the whole system from a legal to an illegal global state.

Consider the “No Send After Read” policy. This policy, discussed in the Introduction, prohibits the flow of internal data to the outside world. It is formally defined by the automaton in Figure 3.1. The question we are facing is: what is the appropriate way to transfer this local policy to a distributed setting?

One way to do it is by enforcing the local policy in every node on the domain. However, such interpretation will be too restrictive because even internal information exchanges within the borders of the domain will be prohibited. Consequently, this interpretation will not allow the execution of many distributed application performing collaborative computations based on the internal data located within the domain.

Another possible interpretation would allow internal sends, but prohibit external ones if the sending node previously read internal data. However, this interpretation may be vulnerable to the following attack. The first target reads the internal data and sends it to another target located within the boundaries of the domain. In turn, the second target sends the data to an external destination. The local policy is not violated in any of the

---

Figure 3.1: “No Send After Read” - the local automaton.

---

6Observe, that no transition is defined for Send operation in After Read state. This is the way security automaton rejects the input sequence.
nodes but internal data can leak.

Another possibility is to allow nodes in the domain to perform external *sends* only if no node in the domain had previously read internal data. We refer to this interpretation of a security policy as truly global interpretation. Note that unlike the first interpretation enforcing this policy is more complex and cannot be done by local mechanisms in each node. Once there was a local *read* in any node, all other nodes must change their local states. However, this policy may be too restrictive, especially in large systems. Information may leak only if the process attempting to perform external *send* had access (via internal domain communication with nodes that performed local *read*), otherwise we can allow the external *send* operation.

Additional difficulty that contributes to the complexity of transforming local to global policies is the ordering of the events. The order of the events in a local systems is straightforward. However, when moving to distributed systems, ordering becomes a much more complex issue. For instance, consider the interpretation of the policy given in the previous paragraph. In that interpretation, it is essential to determine whether the *Read* occurred before *Send*. In general, the ordering of events in distributed systems is not a trivial task and various alternatives such as total and causal ordering [17] have been proposed (see next section).

In order to overcome the drawbacks of the previous interpretation and make use of ordering techniques in distributed systems, we suggest another interpretation. This interpretation would allow internal *sends* and block external ones only if a node obtained internal data directly (by reading some local file) or indirectly (by communicating with other nodes that previously obtained internal data). I.e., internal *read* may have causally affected consecutive external *send*. This interpretation will be referred to as an *HB*-interpretation of a local security policy.

### 3.3.1 HB-Interpretation of Local Security Policies

In distributed loosely-coupled systems without shared memory or global clock, the ordering of global events is a very difficult task. To address this difficulty Lamport [17] defined the *happened-before* relation as follows:
**Definition 3.** The happened-before relation on a set of events, denoted by $\rightarrow$, satisfies the following three conditions: (1) If $a$ and $b$ are events in the same process, and $a$ occurred before $b$, then $a \rightarrow b$. (2) If $a$ is the sending of a message and $b$ is a corresponding receiving of the same message then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two events $a$, $b$ are said to be concurrent (denoted $a||b$) if neither $a \rightarrow b$ nor $b \rightarrow a$. An alternative way to understand the $a \rightarrow b$ relation is by observing that if $a$ happened before $b$ then event $a$ may potentially causally affect event $b$. The events $a$ and $b$ are concurrent if neither can potentially causally affect the other.

**Definition 4.** A node $i$ is aware of event $\sigma^k$, if $\sigma^k$ occurred locally in $i$ ($i=k$) or $i$ received a message from a node that was aware of $\sigma^k$.

**Definition 5.** A sequence of events $a_1a_2..a_n$ is an HB (happened before) sequence iff for each $1 < i \leq n$ $a_{i-1} \rightarrow a_i$.

Schneider [26] proved that EM-Enforceable security policies (as described in Chapter 2) are safety properties. A safety property can be characterized using a set of finite executions that are prefixes of all executions excluded from the property. Therefore, every EM-Enforceable security policy can also be characterized by a set of finite prefixes of the executions prohibited by the policy. These prefixes are sequences of events ordered by their occurrence time. When we move from a single node to a distributed setting it is natural to order these events using the happened-before relation.

The **HB-interpretation** of EM-enforceable policy states that prohibited prefixes and the execution sequences governed by the policy are HB sequences. In a local system with only one target and one node the definition is identical to the original definition of EM-enforceable policy. Such interpretation of a policy $P$ will be denoted by $\overrightarrow{P}$. Consider the “No Send After Read” global policy. The happened-before interpretation of this policy prohibits an execution in which the read operation happened-before external send. This is the best way to interpret this policy since the node attempting to perform communication with external target can only have access to the internal data if read$\rightarrow$send. On the other hand if read$\rightarrow$send does not hold it means that read cannot causally affect the send and no information can potentially be leaked.
Under the HB-interpretation of a policy, a node may be aware of concurrent events that cannot be ordered by the happened before relation. In the “No Send After Read” policy consider a node that performed a local read and after that received a message from a node that performed send. That node is now aware of read and send operations such that neither read happened before send nor read happened before send.

**HB-Interpretation Limitation**

The HB-interpretation considers all the sequences that are ordered by the happened — before relation. Consider a global HB sequence of events in a given system. Any subsequence of that sequence constructed by omitting a number of events would also be considered an HB sequence. The HB-interpretation of a security policy will prohibit an actual global HB sequence of events if any of its subsequences are prohibited by the policy. In order to demonstrate the limitation of the HB-interpretation of a security policy, consider the “No Send After Read Unless Certify” policy depicted by the automaton in Figure 3.2.

This policy resembles the “No Send After Read” policy, but contains an event certify after which the execution is allowed to perform send even if it is previously performed the read operation. The “No Send After Read” policy represents a class of policies, in which the space of allowed operations decreases over time while the “No Send After Read Unless Certify” is an example of a policy in which the previously prohibited operation send becomes legitimate after the execution of certify operation.
Consider an execution in a two node domain governed by the “No Send After Read Unless Reset” policy depicted in Figure 3.3. In the figure arrows between nodes represent internal messages. Observe that the sequence read, certify and send is an HB-sequence. However, read, send is also an HB-sequence prohibited by the policy and therefore the execution will be rejected. The execution will be rejected despite the fact that the full HB sequence read, certify, send is allowed by the policy.

**Definition 6.** A monotonic security policy is a policy in which the space of allowed actions can only shrink over time.

**Definition 7.** A non-monotonic security policy is a policy which is not a monotonic security policy.

The problem depicted in Figure 3.3 is characteristic to the non-monotonic policies and never occurs in monotonic policies.

### 3.3.2 PHB-Interpretation of Local Security Policies

In order to overcome the limitation of the HB interpretation we define properly happened-before sequence as follows:

**Definition 8.** A sequence of events $a_1a_2..a_n$ is a properly HB or (PHB) sequence iff it is HB sequence, and for each $1 < i ≤ n$ there is no event $b$ such that both $a_{i-1} → b$ and $b → a_i$. 
**Definition 9.** A sequence of events \( a_1 a_2 \ldots a_n \) is a longest PHB or LPHB sequence in node \( i \) iff all the following conditions hold. It is a PHB sequence, there is no event \( b \) such that \( b \rightarrow a_1 \), node \( i \) is aware of \( a_n \) and is not aware of \( c \) such that \( a_n \rightarrow c \).

The PHB interpretation of EM-enforceable policy considers only PHB, i.e., full HB sequence, treating monotonic and non monotonic policies in the same manner. The PHB interpretation disregards partial HB sequences that a certain node is aware of and only considers full HB sequences. For instance, the execution depicted in Figure 3.3 will be allowed under the PHB interpretation since the only PHB sequence \( \text{read, certify, send} \) is allowed by the policy and all the partial sequences including \( \text{read, send} \) are ignored.
Chapter 4

Enforcement Mechanisms

In this chapter we present various enforcement mechanisms capable of enforcing the interpretations discussed in the previous chapter.

**Definition 10.** A mechanism enforcing a security policy is conservative iff it rejects all the sequences prohibited by the policy.

**Definition 11.** A mechanism enforcing a security policy is exact iff it is conservative and allows all the sequences that are legal under the policy.

In order to provide the needed security every enforcement mechanism has to be conservative and since we would like to allow as much freedom as possible we also seek that enforcement mechanism will be as exact as possible.

The ability to enforce the policy interpretations depends, among other things on the communication model of the system. Under the loosened model, a more conservative global policy should be applied since internal data may be transferred to a remote node within the domain via covered communication channels without being detected by the enforcement mechanisms. Such a communication model would induce the enforcement of a global interpretation of the policy that prohibits any external sends performed after internal read regardless of the specific location of the events. The PHB and HB interpretation, on the hand, will be unenforceable since the exact flow of data cannot be determined. Under the controllable model, in which all the communication channels are monitored by

---

1. We assume that the covered channels between domain and the outside world are blocked by other security mechanism such as firewalls.
the enforcement mechanisms, we can enforce more permissive interpretations such HB and PHB.

4.1 A Centralized Enforcement Mechanism

We begin by describing a centralized enforcement mechanism that is based on a centralized enforcement engine and multiple local monitors that run on the execution nodes and forward local security relevant events to the enforcement engine. The local monitors intercept security events, forward them to the centralized enforcer and block the target until a positive answer is received. After performing an event, an acknowledgment is sent to the centralized enforcer. The decision to allow an action does not necessarily mean that the action will be successfully performed by the remote node. Therefore, the central enforcer does not accept new events from other nodes until the acknowledgment is received. This centralized construction provides a solution to the event ordering problem described before by creating a total ordering of the security relevant events.

In order to be as general as possible, the centralized enforcement mechanism is based on general security automaton with extended alphabet. The alphabet contains not only the security relevant events such as open socket or disc read, but also an indication to the exact node in which the particular event took place. The letters of this alphabet are of the following form: $A^i$ stating that event $A$ was observed at node $i$. In addition, for each message send from node $i$ to the node $j$ we have $S^{i\rightarrow j}$. For instance, the alphabet for the HB-interpretation of the “No Send After Read Policy” policy in a domain with two nodes is: $\{Read^1, Read^2, ExtSend^1, ExtSend^2, Send^{1\rightarrow 2}, Send^{2\rightarrow 1}\}$. The automaton enforcing this policy is depicted in Figure 4.1. This automaton assumes controllable communication model in which communication between nodes is always detected. Whenever a global violation is detected, the centralized enforcer will stop all the jobs executing in the domain and reset the global security state of the system.

Centralized enforcement in large scale systems suffers from the usual problems related to scalability. In large grids the centralized node needs to monitor a large number

\footnote{Recall that we assume that messages are reliable and thus it also indicates that the message was received by node $j$.}
of events. Moreover, the local executions will have to be blocked after every security relevant event for at least two periods of round trip time between the executing and enforcing nodes even if the event cannot violate the policy. In large scale systems spanning over several continents such an enforcement will cause unbearable performance overhead. Moreover, the concurrency of the distributed system will be severely reduced as a centralized enforcer acts as a global synchronizer of all the security events. Finally, the centralized enforcer is a single point of failure of the distributed system.

In addition, the centralized mechanism suffers from a state explosion problem. Observe the automaton in Figure 4.1. The simple two state local automaton in Figure 3.1 is extended to contain four states. In general, in an $N$ node domain, the centralized mechanism enforcing the HB-interpretation of the local policy will be based on automaton containing $2^N$ states. The reason for this phenomenon is that the centralized automaton needs to remember the sub-set of nodes in which the particular event has occurred.

### 4.2 Distributed Enforcement Mechanisms

In this section we present our distributed enforcement mechanisms that are based on multiple instances of local EMs without any centralized components. Each such EM performs
local enforcement based on the local security state. The local security state is updated by a security state transfer process. A security state transfer is a process in which local state of one node is passed to another node and is incorporated in the state of the remote node to create a new security state which combines the states of both node. In this section we show how certain policies can be enforced efficiently both in the loosened and in the controllable communication models.

4.2.1 Decentralized PHB Policy Enforcement Mechanisms

In this subsection we assume a restricted communication model in which the security mechanism is capable of detecting all the communication channels. State transfer is done exclusively by piggybacking on the messages sent during the normal computation and no additional messages are generated by the enforcement mechanism. Clearly, from communication perspective the performance overhead is very small; no new messages are generated and the only affect is that the size of the messages produced by the monitored computation is increased. The local EMs are based on the general security automaton presented in Chapter 2.

Recall that as stated before, we assume that every communication channel is monitored by the enforcement mechanism. We prove that the EM-Enforceable security policies interpreted by the happened-before relation can be enforced by this efficient distributed enforcement mechanism. This is done by describing the distributed automaton and proving that the distributed construction in fact enforces the policy.

Let $\overrightarrow{P}$ be a happened-before relation based interpretation of EM-enforceable security policy. $P$ is EM-enforceable security policy and therefore can be enforced by general security automaton which we call $A_{local}$. We use $A_{local}$ as input for the automatic construction of a powerset automaton $A_{global}$. This is done using a standard powerset construction. I.e., the state of the powerset automaton is a set over the extended states of $A_{local}$: $S_{global} = 2(S_{local})$. Let $s_i$ be a state in $S_{local}$ then in $S_{global}$ it is extended to include both the id and the timestamp fields. The id field represents the state in $A_{local}$, i.e., $i$ for $s_i \in S_{local}$. For simplicity, instead of using $s.id = i$ we use $s_i$. The timestamp field is
discussed later in this section. The powerset automaton $A_{\text{global}}$ is the base of every local
EM, and is located in every node in the system. The automaton in node $i$ is denoted by
$A^i$ and its state is denoted by $S^i$. Events in $i$ are denoted by $\sigma^i$. $A^i = (\Sigma; S^i; \{s_0\}; \delta_{A^i})$.
The (possibly partial) transition function $\delta_{A^i} : \Sigma \times 2^{S_{\text{local}}} \rightarrow 2^{S_{\text{local}}}$ is defined as follows:

- $\delta_{A^i}(S^i, \sigma) = \{\delta(s, \sigma) : s \in S\}$ if $\forall s \in S^i \delta(s, \sigma)$ is defined.
- $\delta_{A^i}(S^i, \sigma)$ is undefined if $\exists s \in S^i$ such that $\delta(s, \sigma)$ is undefined.

We use vector clocks [20] to determine the casual ordering of the security states. Each
node maintains its own vector clock which is denoted by $VC^i$. It is updated as described
in [20]. Only security relevant events cause vector clock updates. Local send and receive,
which are not a part of the alphabet, are not considered local events, yet, they generate

Every sub-state $s \in S^i$, which is a state of $S_{\text{local}}$, represents at least one global LPHB
sequence that the local EM is aware of. If a powerset state contains $n$ sub-states it indi-
cates that the EM is aware of at least $n$ concurrent LPHB sequences. A time stamp of an
event $\sigma^i$ is equal to the value of $VC^i$ immediately after its occurrence. Each sub-state $s$
contains a time stamp denoted by $s.t.s$. The value of the time stamp is always equal to the
time stamp of the last event in the LPHB sequence that is represented by this sub-state.
The time stamp of a sub-state is updated after every security relevant event. Next we
present the algorithm that enforces $\mathcal{P}^\prime$.

The PHB Algorithm

The PHB algorithm is invoked when local event or internal message send or receive is de-
tected. Upon invocation, the algorithm manipulates its state or terminates if the transition
is undefined. External sends and receives (i.e., communication with elements outside the
domain) are considered as regular local events and treated according to the policy being
enforced.

When a local event $\sigma$ occurs in $i$, the algorithm performs several operations. It updates
its vector clock by advancing the counter associated with the local process by 1. It updates
its state as described in lines 10-16 of Algorithm [1]. If the transition is undefined then the
### Algorithm 1: The PHB Algorithm

1. **Local variables:**
   - $VC^i$ - the local vector clock of node $i$
   - $S^i$ - the set of concurrent sub-states of node $i$
   - $s, s.ts$ - sub-state and its time stamp

2. **Initialization:**
   - $VC^i = 0$
   - $s_0.ts = VC^i, S^i = \{s_0\}$

3. **On local event $\{\sigma^i\}$:**
   - $VC^i[i] = VC^i[i] + 1$
   - foreach $s_j \in S^i$ do
     - remove $s_j$ from $S^i$
     - if $\delta(s_j, \sigma)$ is undefined then
       - terminated the execution
     - else
       - $s_k = \delta(s_j, \sigma)$
       - $s_k.ts = VC^i$, add $s_k$ to $S^i$

4. **On message $\{m\}$ sent:**
   - $m.state = S^i$
   - $m.VC = VC^i$

5. **On message $\{m\}$ receive:**
   - foreach $s_j \in S^i$ do
     - foreach $s_k \in m.state$ do
       - if $s_k.ts \leq s_j.ts$ then
         - remove $s_k$ from $m.state$
       - else if $s_k.ts > s_j.ts$ then
         - remove $s_j$ from $S^i$
     - foreach $s_k \in m.state$ do
       - add $s_k$ to $S^i$
     - foreach $k < n$ do
       - $VC^i[k] = \max(VC^i[k], m.VC[k])$

---

Last action violates the policy and the targets are terminated. Otherwise, the time stamp of every sub-state in the updated state receives the value of $VC^i$ immediately after $\sigma$. This value is equal to the time stamp of $\sigma$ since now it is the last event in the LPHB sequence represented by each of the sub-states. $S^i$ is a set of sub-states categorized both by a state of the local automaton and a time stamp. Occasionally, state may contain more than one
sub-state referring to the same state in $A_{local}$ but with different time stamps. However after a local event, all of them will have the same time stamp and therefore will be merged into one sub-state.

Upon an internal message send, the sender attaches its state and vector clocks to the message. The receiver of the message extracts the vector clock and the state of the sender. It updates its vector clock by selecting the maximal value for every entrance from both vector clocks as described in [20]. The updated local vector clock represents the combined knowledge of both nodes. After that, the algorithm performs state update. In the first stage the outdated sub-states are removed both from local and received states. The reason for this is that as a result of the message the local process becomes aware of all the events the sender process is aware of. Some of the sub-states of the sender state may represents LPHB sequence of events that happened-after the events represented by some of the local sub-states. We later show that only concurrent sub-states can be part of the state and therefore the sub-states that happened-before any of the other sub-states should be removed from the local or remote states. After the outdated sub-states are removed, the remaining sub-states from the remote state are added to the remaining local sub-states.

**Examples of algorithm execution.**

Figure 4.2 illustrates an execution of the algorithm in a three node domain. The local automaton enforcing this policy consists of two states: $s_0$ is the initial state indicating that no read had been performed and $s_1$ indicating that local read occurred at some point in the past. The automaton in each node starts with a state containing a single sub-state $s_0$ and a vector clock initialized to zero. Local reads in nodes 1 and 3 move $S^1$ and $S^3$ to state $\{s_1\}$ and increase the local entry of their vector clocks by one. After receiving a message sent by 3, the algorithm in node 2 removes the outdated sub-state $s_0$ from $S^2$, according to lines 25-26 of the algorithm and replaces it with $s_1$ containing a higher time stamp. It is also updates the local vector clock by choosing the maximal value for each entry (lines 29 – 30 of the algorithm). When a message from Node 1 arrives it contains a sub-state $s_1$ with time stamp which is concurrent to the local sub-state $s_1$ and therefore the new sub-state is added to $S^2$. $S^2$, which is a set of sub-states, contains now two concurrent sub-states distinguished only by their time stamps. When local read occurs in 2, it does
Figure 4.2: An example of an PHB algorithm execution, enforcing PHB interpretation of “No Send After Read”, in three a node domain.

not move any of the sub-states to a new state; however the time stamp of the sub-states is changed. Now both sub-states $s_1$ contain a time stamp equal to the time stamp of the local read and therefore become undistinguishable. Both sub-states are automatically merged since $S^2$ cannot contain several identical sub-states. Finally, send violates the policy since the transition from $s_1$ with send is undefined.

**Correctness Proof.**

**Lemma 4.1.** $s_i \in S^j$ iff there exists an LPHB sequence $w = w'\sigma^k$ in $j$ such that $\delta(s_0, w) = s_i$.

*Proof.* We start by proving that if there exists an LPHB sequence $w = w'\sigma^k$ in $j$, such that $\delta(s_0, w) = s_i$ then $s_i \in S^j$. We prove this by induction on the size of $w$.

Base case $|w| = 1$: Let $w = \sigma^k$; $w$ is an LPHB sequence and therefore no event happened-before $\sigma^k$, so $s_0$ was part of $S^k$ when $\sigma^k$ occurred. As a result, $s_i$ became part of $S^k$. $w$ is an LPHB sequence in $j$ and therefore $j$ is aware of $\sigma^k$ and is not aware of any event that happened-after $\sigma^k$. $s_i$ become part of $S^j$ as a result of a local transition if $k = j$ or as a result of a sequence of messages sent between $k$ and $j$.

We assume the claim holds for $|w| = n$ and prove it for $w\sigma^k$. Let $l$ be a node where the
last event of $w$ occurred. According to the assumption $\delta(s_0, w) = s_x \in S^j$ holds. Let $\beta^l$ be the last event in $w$, then $\beta^l$ and $\sigma^k$ are consecutive events in the LPHB sequence. I.e., just before the execution of $\sigma^k$, $k$ was aware of $\beta^l$ and it was not aware of any event that happened after $\beta^l$ and before $\sigma^k$. Therefore $s_x$ was a member of $S^k$ before $\sigma^k$ occurred. Since $w\sigma^k$ is an LPHB sequence in $j$ no event happened after $\sigma^k$ and therefore $\delta(s_x, \sigma) = s_i \in S^j$.

Next we prove that if $s_i \in S^j$ holds then there exists an LPHB sequence $w = w'\sigma^k$, in $j$ such that $\delta(s_0, w) = s_i$. We prove it by induction on the total number of events in the system (including local events and message sends in all the nodes). Let NOF (number of events) be the total number of events in the system in the current run.

Base case: $NOF = 0$ is trivial. Now we prove the claim for $NOF = 1$. If the only event in the system is a local event that occurred in node other than $j$ or is a send to node other than $j$, then $S^j = \{s_0\}$ and an empty sequence $\epsilon$ is the LPHB sequence that satisfies the claim. If the only event is a send to $j$ then $S^j$ will not be influenced since the only sub-state received as a result of a message will contain the initial time stamp equal to the time stamp of the local sub-state $s_0$ and therefore will be removed by the algorithm (lines 23-24). Therefore, the empty sequence $\epsilon$ satisfies the claim. If the only event is a local event $\sigma^j$ then $S^j = \{\delta(s_0, \sigma)\}$ and $\sigma$ is the LPHB sequence that satisfies the claim.

We assume that the claim holds for $NOF = n$ and prove it for $NOF = n + 1$. If the last event was a local event at some other node or message send to node other than $j$, then according to the assumption the claim held before the last event and since $j$ is not aware of the last event the claim also holds after it. If the last event was $\sigma^j$ then for every $s_i \in S^j$ there was $s_k \in S^j$ just before the last event such that $s_i = \delta(s_k, \sigma)$. According to the assumption for every such $s_k \in S^j$ there was a LPHB sequence $w$ such that $\delta(s_0, w) = s_k$. Therefore for every $s_i \in S^j$ after $\sigma$ there is LPHB sequence $w' = w\sigma$ such that $\delta(s_0, w\sigma) = \delta(s_k, \sigma) = s_i$ so the claim holds.

If last event is a send from $m$ to $j$, then after receiving the message $S^j$ contains sub-states that originated from $S^m$ and did not happened-before any sub-states of $j$, as well as, sub-states that were part of $S^j$ before the receiving of the message and did not happened-before any sub-state of $S^m$ (assured by lines 21-28 of Algorithm [1]). For every $s_i$ that originated from the $S^j$ LPHB sequence $w$ was assured by the assumption. The fact that
remained after the states were combined indicates that \( j \) did not become aware of any event that happened-after the last event of \( w \), because otherwise the sub-state would have been removed from the combined state (Algorithm 1, lines 25-26). Therefore, \( w \) remained an LPHB sequence even after the receiving of the message and thus for the sub-states originated from \( S^j \) the claim holds. For every sub-state originated from \( S^m \) the assumption assured the existence of an LPHB sequence \( w \) in \( m \) that satisfied the claim. As a result of the message, \( j \) became aware of the last event of \( w \) and therefore the same sequence became an LPHB sequence in \( j \). This is due to a fact that \( j \) is not aware of any event that happened after the last event of that sequence, because otherwise the sub-state would have been removed from the combined state (Algorithm 1, lines 23-24). Thus the claim also holds for the sub-states that originated from \( S^m \) and therefore the claim holds if the last event is message send to \( j \).

\begin{lemma} \label{lem:4.2}
Let \( \sigma^j \) be the last event in \( j \). Then \( A^j \) rejects iff there exists an LPHB sequence \( w = w'\alpha^k\sigma^j \) in \( j \) such that: \( \delta(s_0, w) \) is undefined.
\end{lemma}

\begin{proof}
\( \Leftarrow \) Let \( w = w'\alpha^k\sigma^j \) be an LPHB sequence in \( j \) such that: \( \delta(s_0, w) \) is undefined and let \( \delta(s_0, w'\alpha) = s_x \). \( k \) is aware of \( \alpha^k \) and \( \alpha^k \) and \( \sigma^j \) are consecutive events in a PHB sequence. We get that \( w\alpha^k \) was an LPHB sequence in \( k \) and therefore according to Lemma \[lem:4.1\] \( s_x \in S^k \) immediately after \( \alpha^k \). Since \( \alpha^k \) and \( \sigma^j \) are consecutive events in a PHB sequence \( s_x \) was also a member of \( S^j \) before \( \sigma^j \). \( \delta(s_x, \sigma) \) is undefined and therefore \( A^j \) rejects the input.

\( \Rightarrow \) We use Lemma \[lem:4.1\] on the previous state. If \( A^j \) rejected \( \sigma^j \) then just before \( \sigma^j \) there was \( s_i \in S^j \) such that \( \delta(s_i, \sigma) \) is undefined. According to Lemma \[lem:4.1\] at that moment there existed an LPHB sequence \( w \) in \( j \) such that \( \delta(s_0, w) = s_i \). Therefore \( w\sigma^j \) is also an LPHB sequence in \( j \) such that \( \delta(s_0, w\sigma^j) \) is undefined. Thus the claim holds.
\end{proof}

\begin{theorem} \label{thm:4.3}
The PHB interpretation of EM-enforceable security policy \( P \) is enforced by our distributed construction.
\end{theorem}

\begin{proof}
If \( w \) is forbidden by the policy then \( \delta(s_0, w) \) is undefined and according to Lemma \[lem:4.2\] for each sequence of events \( w = w'\sigma^j \) that is an LPHB sequence in \( j \) \( A^j \) will reject it as well.
\end{proof}
If $A^j$ detects a violation as a result of $\sigma^j$ then according to Lemma 4.2 there exists an LPHB sequence so that $\delta(s_0, w)$ is undefined and therefore this LPHB sequence is forbidden by the policy.

**Upper Bound**

**Lemma 4.4.** Any two sub-states of the same $S^j$ are either concurrent or represent the same last event (same time stamp).

**Proof.** We prove the Lemma by induction on the total number of events in the system (including local events and message sends in all the nodes). Let NOF (number of events) be the total number of events in the system in the current run.

Base case: $NOF = 0$ is trivial. Now we prove the claim for $NOF = 1$. If the only event in the system is a local event that occurred in node $j$ then the Lemma holds since the local state will only contain one sub-state after local event. Message sends do not change the local state of the sending node. Message receive cannot occur before corresponding message send. Therefore, for $NOF = 1$ the Lemma holds.

We assume that the claim holds for $NOF = n$ and prove it for $NOF = n + 1$. According to the induction assumption the Lemma held prior to the last event $\sigma$. I.e., all the $S^j$ contained sub-states with concurrent or equal time stamps. Let $\sigma$ (the last event) be a local event in node $j$. Local events only move all the sub-states to the same time stamp (Algorithm 1, lines 9-16). I.e., immediately after the local event all the sub-state have the same time stamp (the time stamp of the last event) and thus the Lemma holds. If $\sigma$ is a message sent from node $j$ then the Lemma holds since sending a message does not change $S^j$. Let $\sigma$ be a message received at node $j$. Let $y$ be the sending node and let $S^y$ be its state. Prior to Receiving message the Lemma held according to the induction assumption. I.e., both $S^j$ and $S^y$ contained sub-states with concurrent or equal time stamps. Let $S^j_{old}$ be the state of $j$ prior to receiving the message and let $S^j_{new}$ be its state immediately after it. According to the algorithm if there exists $s_k \in S^y$ and $s_l \in S^j_{old}$ such that $s_k.ts < s_l.ts$ then $s_k$ is removed from the combined $S^j_{new}$ (Algorithm 1, lines 23-24). If on the other hand there exist $s_k \in S^y$ and $s_l \in S^j_{old}$ such that $s_l.ts < s_k.ts$ then $s_k$ replaces $s_l$ in $S^j_{new}$ (Algorithm 1, lines 25-28). Therefore, the correctness of the Lemma claim is preserved.
during state transfer and the Lemma holds for $NOF = n + 1$.

Lemma 4.5. For every sub-state $s_i$ we have a:
$|\{s_i \in S^j\}| \leq N$, where $N$ is the size of the domain

Proof. Assume by contradiction that there exists a node $j$ and a sub-state $s_i$ such that $|\{s_i \in S^j\}| > N$. Then according to the pigeon-hole principle there exists a node $k$ such that at least two $s_i$ represent an LPHB sequences in $j$ with last events of both occurring locally in $k$. Let these events be $e$ and $e'$. Observe that $e$ and $e'$ are different events because otherwise both sub-states with same local state and time stamp would have been combined by the enforcement algorithm. Since both are sub-states of $S^j$ then according to Lemma 4.4 they are concurrent. But two events in the same processor cannot be concurrent (one always happens before the other) and therefore the contradiction assumption is erroneous and the lemma holds.

Theorem 4.6. $N \times |S_{global}|$ is an upper bound on the size of the local state $S^j$.

Proof. According to Lemma 4.5 for every $i$, $s_i$ can appear at most $N$ times in $S^j$. Therefore the upper bound on the size of the state $S^j$ is $N$ times the number of the possible sub-states. I.e., $N \times |S_{global}|$.

In this section we presented a fully distributed scheme that enforces a PHB interpretation of any EM-enforceable security policy. We believe that this is an important subclass of all relevant global policies. We proved that the scheme indeed works, and provided upper bounds on the sizes of the state and messages. In the Sections 5 we describe our prototype implementation and present its performance evaluation.

4.2.2 HB Algorithm

In sections 3.3.1 and 3.3.2 we described the differences between PHB and HB interpretation of the security policies. We also showed the limitation of the HB interpretation of the non-monotonic policies. The HB as opposed to the PHB interpretation prohibits an actual global HB sequence of events if any of its subsequences are prohibited by the
policy. However, for a class of monotonic policies the HB interpretation is as exact as the PHB interpretation. In this section we present an HB algorithm capable of enforcing HB interpretation of security policies. The HB algorithm is a simplified and more efficient version of the PHB algorithm and uses no vector clocks.

When a local event occurs, the algorithm performs local state update as described in lines 4-10 of Algorithm [2]. If a transition is undefined then the last action violates the policy and the targets are terminated. Otherwise, new sub-states are added to the local state. Observe that the old sub-states are never removed since they represent subsequences of the of the actual global HB sequence of events that the node is aware of.

Upon an internal message send, the sender attaches its state to the message. The receiver, in his turn, simply adds the remote sub-states to its local state without removing outdated states.

The algorithm uses no vector clocks and each sub-state can appear only once in the local state, therefore the message and the state size is significantly smaller than the size the PHB automaton.

Algorithm 2: The HB Algorithm

<table>
<thead>
<tr>
<th>Local variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^i$ - the set of concurrent sub-states of node $i$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initialization:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^i = {s_0}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>On local event ${\sigma^i}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreach $s_j \in S^i$ do</td>
</tr>
<tr>
<td>if $\delta(s_j, \sigma)$ is undefined then</td>
</tr>
<tr>
<td>terminated the execution</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$s_k = \delta(s_j, \sigma)$</td>
</tr>
<tr>
<td>$S^i \cup s_k$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>On message ${m}$ sent:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$.state = $S^i$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>On message ${m}$ receive:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^i \cup m$.state</td>
</tr>
</tbody>
</table>

The algorithm provides exact enforcement of monotonic policies. However, non-monotonic policies are enforced conservatively. I.e., the algorithm may reject a legal
execution under non-monotonic policies. Observe the HB algorithm executions depicted in Figure 4.3. The algorithm rejects the sequence in spite of the fact that the full PHB sequence is allowed by the policy. When applied to monotonic policies, the HB algorithm is exact and is much more efficient than the PHB algorithm. For instance, a simple “No Send After Read” can be enforced exactly by the HB algorithm.

One can use a proof similar to one we in the PHB case in order to prove the following theorem.

![Figure 4.3: HB Algorithm, Enforcing HB Interpretation of “No Send After Read Unless Certify”](image)

**Theorem 4.7.** The HB interpretation of EM-enforceable security policy P is enforced by HB algorithm.

The state of the local node can contain only one copy of each sub-state, therefore its size is bounded by $|S_{global}| = 2^{S_{local}}$.

### 4.3 The Loosened Model

Under the loosened communication model, the enforcement mechanism does not have complete control over the communication channels, making the enforcement mechanism
presented in previous section insecure. In this case, a different decentralized enforcement mechanism is needed. In this section we do not use the Happened Before relation to order global events, since we have no control over the communication between the distributed entities and therefore we cannot determine whether one event was casually influenced by another.

Our decentralized architecture uses no centralized components and assumes unsynchronized clocks making global ordering of the events unfeasible. Note, however, that in general, state transfer may depend on the exact order of events. Consider a policy in which event $a$ followed by event $b$ leads to a state that is substantially different from a state resulted by event $b$ followed by event $a$. If one of the nodes observed $a$ and another one observed $b$ the state incorporating common knowledge of these nodes can determined only if $a$ and $b$ could be ordered. Therefore, state transfer for general security automaton requires the EM to remember and transfer the entire history of the local execution in which each event in the sequence contains the global time of its execution, so the global order could be determined. This makes distributed enforcement of such policies in this model unfeasible. Therefore, we use Shallow History Automata [9] as a bases for the local EMs.

Shallow History Automata [9] or SHA, are a special subclass of a general security automata. An SHA remembers the shallow history of previously allowed security events and is formally defined as follows:

$$SHA = (I; F(I); H_0; \delta), \text{where}$$

- $I$ is a finite or countably infinite set of observable security events,
- $F(I)$ is a set of all finite subsets of $I$. The current state of the automaton is a set of all the security relevant events observed by it,
- $H_0$ is an initial empty state,
- $\delta : I \times F(I) \rightarrow F(I)$ is a (possibly partial) transition function. $\delta(q, i) = q \cup i$ if $\delta$ is defined.

A state of the SHA represents a set of security events that were allowed by the automaton.
The SHA therefore does not remember the number or sequence of the allowed security events and thus the policies enforceable by such an automaton are policies in which the enforcement decisions is based solely on whether a specific subset of security events was or was not performed by the target without the ability to distinguish between different sequences that led to the same state. Similarly to all security automata described in this thesis the SHA rejects the input whenever the transition for the current state and symbol is undefined. Fong [9] showed that SHA enforceable policies are a proper subset of all EM-enforceable policies.

We assume that the global centralized automaton, as described in Section 4, is a SHA, and we present a general decentralized SHA based mechanism capable of enforcing the policies enforced by the centralized SHA.

All the local EMs contain a copy of the SHA that would have been used in the centralized enforcement of the same policy. This copy of SHA contains the local knowledge and most of the time is not updated with global events performed elsewhere. Let \( i \) be a node with a local SHA in state \( S_j \) and a local event \( \sigma^i \). The SHA has to check whether the local action \( \sigma^i \) can potentially violate the global policy being enforced. I.e., the local EM checks whether there exists state \( S_k \in F(I) \) such that the transition from \( S_k \) with \( \sigma^i \) is undefined. If such a state does not exist then the transition is allowed and defined as follows: \( \delta(S_j, \sigma^i) \rightarrow S_j \cup \sigma^i \). Similarly, if for all the supersets of \( S_j \) including \( S_j \), the transition with \( \sigma^i \) is undefined then the sequence can be rejected locally without a need to perform any global operations. If neither holds, the global state should be determined before deciding on whether the action is allowed.

The global state is determined by a global query process in which a specific node gathers the global state of the system by querying the local states of all the nodes in the system. The state transfer for a SHA is simply defined as the union of the local and the remote states creating a state that incorporates all the action known to both nodes. Let \( S_i \) be the current state of the local node performing a global query, and let \( S_j \) be the state of the node being queried then the new state of the local node is \( S_i \cup S_j \). At each step before continuing to the next node, the mechanism checks whether the authorization decision can be reached based on the state gathered so far. At the end of the process, the local security
state of the node represents the current global state of the system\textsuperscript{3} and the transition from this state is checked. If the transition is undefined then the policy is violated and the targets are terminated.

A node performing the global query operation remains blocked during the operation. I.e., the next events of the target are not executed. Since the target remains blocked during global query process, the only events that could have influenced the current events are the events that occurred prior to the the beginning of global query process. These events must have occurred at a specific node and were recorded in that node local state. Therefore, all the events that could have potentially influence the local event are represented in the state upon completion of global query process.

**Definition 12.** An event $a$ can potentially influence another event $b$ under the Loosened model if $a$ was completed before the execution of $b$ was started.

If $a$ occurs before the execution of $b$ it can potentially causally affect it. And since we do not have full control over communication channels we have to assume that all the events that occurred prior to the event $b$ can causally affect it.

When a node $j$ receives a state transfer request from node $i$, it immediately responds even if it is in the middle of event handling process. Let $\sigma$ be the event node $j$ is currently handling and let $S_j$ be its state. Node $j$ will respond immediately and send $S_j$ since its local $\sigma$ has not been executed yet and therefore cannot influence the event under examination of remote node. I.e., it will send its local state without the event it is currently handling, since this event is not completed and therefore cannot influence the event under examination of remote node.

The algorithm works as follows: On local event it attempts to determine whether the event can be authorized or rejected based solely on local knowledge (lines 12-18). If authorization decision cannot be reached based on local knowledge the algorithm performs global query (lines 19-29). When a node receives response from one of the queried nodes it checks whether the definitive decision can be reached and whether it should continue querying process. Upon completing the global query process, it checks that a transition is defined for the local event and the global state that represents all the events that could

\textsuperscript{3}In fact, it contains the set of all the events that occurred in the domain and could have potentially influence the event $\sigma$ that is being checked.
Algorithm 3: The SHA Based Algorithm

1 Definitions:
2 \( \sigma^i \) event in node \( i \)
3 SHA = \( (I; F(I); \phi; \delta) \) - local copy of the automaton

4 Local variables:
5 \( S^i \) - the state of SHA in node \( i \).
6 isInGlobalQuery, currentEvent

7 Initialization:
8 \( S^i = \phi \), isInGlobalQuery = false

9 On local event \( \{\sigma^i\} \):
10 if \( \neg \) (CheckIfLocallyDetermined(\( \sigma^i \)) \) then
11 \( \) PerformGlobalQuery(\( \sigma^i \))

12 Boolean CheckIfLocallyDetermined(\( \sigma \)):
13 if \( \delta(S', \sigma) \) is defined for every \( S' \subseteq F(I) \) such that \( S^i \subseteq S' \) then
14 \( S^i = (S^i \cup \{\sigma\}) \)
15 return true
16 if \( \delta(S', \sigma) \) is undefined for every \( S' \subseteq F(I) \) such that \( S^i \subseteq S' \) then
17 terminated the execution
18 return false

19 PerformGlobalQuery (\( \sigma \)):
20 isInGlobalQuery = true, currentEvent = \( \sigma \)
21 foreach node \( j \) in domain do
22 \( \) get state from node \( j \)
23 \( S^i = (S^i \cup S^j) \)
24 if CheckIfLocallyDetermined(\( \sigma^i \)) then
25 \( \) isInGlobalQuery = false; exit
26 if \( \delta(S^i, \sigma) \) is undefined then
27 \( \) terminate execution
28 else
29 \( S^i = (S^i \cup \{\sigma\}) \)
30 isInGlobalQuery = false
31 On stage request from \( j \):
32 send \( S^i \) to \( j \)

have potentially influence the local event. If a node receives states request it responds immediately even if it is in the middle of a GlobalQuery process (lines 30-31).
4.3.1 Correctness Proof.

We first prove that the GlobalQuery process always terminates and then proceed to show that the algorithm conservatively enforces the local security policy.

GlobalQuery termination

Theorem 4.8. The GlobalQuery process always terminates.

Proof. The GlobalQuery process terminates when the security state of the last target is received. We assume the existence of reliable communication channels and an upper bound on the message delay in network. Therefore, all the messages reach their destinations and there is an upper limit $T_{max}$ on the period of time to receive a response from a single node. Recall that upon a state request, any node responds immediately even if it itself performing GlobalQuery process. I.e., if a node performing the global query did not receive a response during this period of time, it can assume that the remote node is down. The global query process time then is bounded by $N \times T_{max}$ where $N$ is a number of nodes in the domain. Since $N$ and $T_{max}$ are finite the process terminates after a finite period of time.

Correctness

We will prove that an event that would have been prohibited by the local SHA based security mechanism is rejected by the distributed construction. SHA does not remember the exact order of events in a sequence, therefore events are allowed and prohibited based on a set that includes all the events that occurred in the system. In the local mechanism this set is located in a single node, while in a distributed setting the same state can be spread over several nodes. When deciding whether to allow an event in a distributed environment, we consider all events that occurred before or can potentially occur before it. For the sake of the proof events are ordered by an actual time of their occurrences.

Theorem 4.9. Let $\sigma_n$ be an event occurring in node $i$ and let $S = \{\sigma_1, \sigma_2...\sigma_{n-1}\}$ be a set of all the events that could have potentially influence $\sigma_n$. If no corrective events occur between the start of the execution of $\sigma_n$ and the end of the GlobalQuery process, then $\sigma_n$ is rejected if $\delta(S, \sigma_n)$ is undefined. I.e., $\sigma_n$ violates the policy being enforced.
Proof. Let assume that \( \sigma_n \) is not rejected despite the fact that before the start of its execution the transition \( \delta \) with \( S \), the set of all events that occurred in the domain, and \( \sigma_n \) was undefined in the local automaton. During the GlobalQuery process node \( i \) receives states from all the nodes in the domain. Let an event \( \sigma_k \) that occurred in node \( l \) be an event that could have potentially influence \( \sigma_n \). It means that it was completed before the start of GlobalQuery process and therefore is a part of the local state of node \( l \). Consequently, node \( i \) will learn about this event during the GlobalQuery process. This is true for every \( 1 \leq k < n \). Therefore, by the end of the GlobalQuery process node \( i \) learns about all the events that could have potentially influence \( \sigma_n \). According to contradiction assumption \( \sigma_n \) is not rejected. Therefore, the state received by \( i \) by the end of GlobalQuery process contains some additional event \( \sigma_{n+1} \) such that transition \( \delta(S \cup \{\sigma_{n+1}\}) \) with \( \sigma_n \) is defined. However, the theorem assumes that no corrective events happen after the start of execution of \( \sigma_n \), and therefore the contradiction assumptions is erroneous and the theorem holds.

Recall that monotonic policies are policies in which there are no corrective events, thus we have the following corollary.

**Corollary 1.** Let \( \sigma_n \) be an event occurring in node \( i \) and let \( S = \{\sigma_1, \sigma_2, ..., \sigma_{n-1}\} \) be a set of all the events that could have potentially influence \( \sigma_n \). If the policy being enforced is a monotonic policy then \( \sigma_n \) is rejected if \( \delta(S, \sigma_n) \) is undefined. I.e., \( \sigma_n \) violates the policy being enforced.
Chapter 5

Implementation

We have implemented a prototype system capable of enforcing the policies discussed in Section 4.2. The system uses AspectJ [15] to intercept security relevant events. The AspectJ language is an implementation of the Aspect Oriented Programming paradigm for java. The motivation for AspectJ (and likewise for aspect-oriented programming) is the realization that there are issues or concerns that are not well captured by traditional programming methodologies. Security policies is an excellent example since by its nature, security cuts across many of the natural units of modularity of the application. In object-oriented programming languages, the natural unit of modularity is the class. However, implementing security requirements in this class structure is inherently difficult precisely because these requirements cut across classes, and they are not reusable, they cannot be refined or inherited, and they are spread through out the program in an undisciplined way. Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns.

We are using the AspectJ language to define the space of the events that can be detected by our mechanism, allowing us to monitor accesses to data members, constructor invocations or executions of the method of the java API classes as well as user developed classes. For instance, call(* *.Connect(..)) query will match every point in the code containing call to Connect method of any class. AspectJ allows executing arbitrary code before, after, or instead of the event specified by the query string.

AspectJ was selected over other alternatives such as Java Security Manager [12] be-
cause of its expressiveness power. For instance, Java Security Manager allows monitoring of only limited number of the predefined security relevant events, therefore limiting the events space available for enforcement mechanism. AspectJ on the other hand, enables to consider each function call as a security relevant event.

The system consists of a GUI, a client, a repository, and runtime components. The GUI is used to define and store security policies (Figure 5.1). It allows a user to create a security automaton using visual means. Security states are associated with an action that should be taken when a state is reached. Security states are connected by edges representing transitions. Every edge contains security event, specified in AspectJ, that causes the automaton to move from a source to a target state. The automaton can be compiled into Aspect file or stored as an XML file.

When compiling a policy, the system asks which policy interpretation will be created. We have implemented HB and PHB interpretations of security policies. The policy compilers are extensible components, making it very easy to write new compilers for new types of policy interpretations. The policy compilers scan the data structures created by the GUI and create an output AspectJ file. In addition to transitions defined by the GUI it injects interpretation related code.

The repository component is the only centralized component in the system. It is used to store the current version of the security policy. The clients can access this repository to download the latest version of the policy.

The client component resides on each of the nodes of the domain. It has two modes of operations. In the first mode, whenever a node is about to execute a target, a client weaves an AspectJ file containing the security policy together with the runtime component into the target application using an AspectJ compiler. This compulsive process does not depend on the target cooperation and is executed without the target’s knowledge. The client executes the application and provides the runtime with the necessary parameters. These parameters are the domain and the node id. In the second mode of operation the security policy provided by the AspectJ file is weaved directly into the core classes of the JRE that will be executing targets.

During the target execution, security events are forwarded to the runtime component. This generic component implements the relevant security algorithm. Events, transitions,
and actions are specified by the AspectJ file and vary from policy to policy. Runtime contains implementation of PHB and HB algorithms. It was designed in extensible manner that allows simple creation of additional algorithms. The algorithm type is specified by the policy compiler and is a part of the AspectJ file. The client component instantiates a target component with appropriate enforcement algorithm. For instance, runtime component implementing an PHB algorithm maintains security state and a vector clock of the target. It performs state transfers by piggybacking the local state on outgoing messages, incorporates a state received from remote node into the local state and terminate the application upon detecting a violation.

Figure 5.2 describes the flow of an execution of the Distributed Security Enforcer. In the first stage the administrator decides on the “Security Policy” he wants to enforce on the executed applications. This policy will need to be transformed into an automaton form using the “Distributed Security Enforcer GUI”. After drawing the automaton that describes the required policy, the administrator compiles the automaton into an AspectJ-
format “Aspect File”. The administrator may also export it into an XML file which can be imported back to the GUI. The DSE Repository should be created and configured to the directory where the aspect file was saved to.

Executing the DSE Client with the a URL to the Repository component and the application file will cause the application’s classes to be instrumented with the an aspect file by the “AspectJ Compiler”. The outputs of this stage are the java classes of the application woven with the policy enforcing mechanism. This happens in each machine indepen-
dently. From this point in time, every time the distributed application is spawned by a JRE in the predefined subnet, it will be sandboxed and monitored by the state-full global security mechanism.
Chapter 6

Performance Evaluation

In order to evaluate the performance of the implemented enforcement mechanism we tested it over a variation of the Chinese Wall security policy [6]. The policy is designed to avoid conflicts of interest that may arise due to the unchecked flow of information belonging to competing parties. Our interpretation prohibits from targets to pass information to a party if it had previously obtained information about its competitor. The automaton depicted in Figure 6.1 describes the local policy for a specific setup.

The goal of the policy is not to allow the target to write to a file belonging to a party if it previously had obtained information belonging to its competitor. Network sends are not allowed after accessing a local file. In addition, our interpretation allows a target to access local files only if the target was previously authorized. In our setup every node contains 3 files. Coca-Cola and Pepsi-Cola belong to same conflict of interests group while Citigroup is in its own banks group. Therefore, an attempt to write to Coca-Cola file after reading Pepsi-Cola file or vice-versa will result in a policy violation. The access to Citigroup file will not prevent any future writes but will not allow future network sends.

We checked various distributed application to make sure that the policy is enforced correctly, and in fact, in all of our experiments the application was terminated just before it violated the policy.

The overall overhead of the enforcement mechanism depends on various factors such as the application itself and the policy being enforced. An application performing a large amount of security relevant operations will be affected stronger by the security enforce-
Figure 6.1: “Chinese Wall” Local Automaton.

In order to test this, we first tested our system on a one node domain. The goal of this experiment is to understand how application behavior in terms of percentage of security relevant operations influences the overall running time. The application includes security relevant events such as disk and networks writes as well as mathematical calculations which are irrelevant from security policy point of view and are not handled by the security mechanism. Figure 6.2 depicts the overhead of the executions of an application running under the PHB security enforcement mechanism compared to the same application running without any security mechanism. In the experiment we started with an application containing only security relevant events and gradually increased the portion of security relevant events and the performance will be degraded much more than applications with a smaller amount of security relevant operations. The reason for this phenomenon is obvious: security relevant events are intercepted by the AspectJ mechanism, analyzed by the enforcement mechanism and only then target is allowed to proceed. The overhead is also a function of the number of states in the security policy, number of nodes in the domain, and the number of network operations.
irrelevant events reducing the total percentage of security relevant events from 100 to around 40. We can see that the overhead decreases as the percentage of security relevant events decreases. The security irrelevant events do not contribute to the overhead since these events are not intercepted by our mechanism. Therefore, the overall execution time increases but the total overhead remains constant and this decreases the overhead in terms of percentage. It can be observed that when an application contains only security relevant operations the overhead is about 70%, and as a number of security irrelevant operations increases the overhead is reduced to about 30%.

![Figure 6.2: Overhead as a function of application behavior in a single node.](image)

In order to measure the overhead inflicted by our mechanism we developed a distributed application running on several nodes within the same domain. One target reads data from its local Citigroup file, and sends it to another nodes using messages of constant size. The consecutive node receives the data, performs local calculation and writes the results to its local Coca-Cola file. This application does not violate the policy. We measured the performance of the application with and without our security mechanism. We mea-
sured the performances of the HB and the PHB mechanisms and compared them with the execution times of an application without any security mechanism. We also performed measurements that allowed us to study the different factors that contribute to the overall overhead.

Figure 6.3: Application Execution Times In Six Node Domain.

Figure 6.3 depicts the execution times of the application in a six node domain as a function of the size of the Citigroup file. The Figure contains the execution times of an application under the PHB, HB, Only AspectJ and no enforcement mechanisms. The “PHB” graph depicts the overall execution time of the application running under PHB security mechanism enforcing PHB interpretation of the policy as described above. The “HB” graph reflects an implementation that enforces HB interpretation and uses no vector clocks (see Section 4.2.2). Note that the application does not violates the policy in both interpretations. The “Only AspectJ” graph reflects a mechanism that performs no state manipulation operations; all security relevant events that are monitored by the PHB and HB mechanisms are intercepted by the Only AspectJ mechanism. Immediately after the
jump into the AspectJ code the function returns and the action is performed. Therefore, the total execution time under the Only AspectJ mechanism is higher than the execution time with no enforcement mechanisms, but significantly lower than the executions times in the HB and PHB interpretations. The results confirmed that HB is more effective (see Section 4.2.2). It can be clearly seen that the HB algorithm produces less overhead, however, the difference is not significant. This can be explained by the small size of the domain (6 nodes) influencing the size of the vector clocks, security state, and the messages exchanged between nodes in a domain (see Section 4.2.1).

![Figure 6.4: Percentage Overhead Breakup.](image)

Figure 6.4 depicts the percentage overhead of the security mechanisms we implemented. The depicted results were calculated using the measurements presented in Figure 6.3. The graphs represent overhead of each mechanism compared to an execution in a enforcement free environment in terms of percentage.

The overhead inflicted by AspectJ constitutes less then 10% of the execution time without any security mechanisms. The overhead of PHB and HB are 32% and 24%
on average; acceptable values for network and security relevant intensive application. The overhead percentage is higher in short executions and declines as execution time increases. The reasons for this phenomenon is that initialization time is amortized over longer period of execution and the fact that the longer is the execution the less is the portion of the security relevant events.

The HB and PHB overheads contain the AspectJ since both HB and PHB algorithm procedures are invoked by the AspectJ mechanism when security relevant events or message send and receives are performed by the application. On the algorithm level, the overhead is caused by state manipulation operations and modification of the network messages. I.e., attaching security state at the sending node and extracting it at the receiving end.

![Overhead Breakup in Six Node Domain](image)

Figure 6.5: Overhead Breakup in Six Node Domain.

Figure 6.5 depicts the various factors contributing to the overall overhead. The “Only AspectJ” graph reflects the overhead inflicted the AspectJ based event handling mechanism in terms of total overhead. I.e., it reflects the difference between the execution times
of the application without any security mechanism and with the *Only AspectJ* mechanism. This measurement allows us to understand the overhead caused by AspectJ since our algorithm can be implemented using different event monitoring technology such as Java Security Manager. The “Total HB” and “Total PHB” graphs were calculated by subtracting the execution time with no enforcement mechanism from the execution times under the PHB and HB mechanisms.

The “PHB algorithm” and “HB algorithm” graphs depict the overhead of these algorithms without taking into account the AspectJ overhead. This data was received by measuring the time spent in the algorithm. I.e., we summarized the times the application spent in the procedures of the algorithms. Each time a security relevant event occurred we measured the time from first to the last line of the algorithm procedures. Therefore, by the end of the execution we could calculate the total time spent in the algorithm procedures without taking into account other overhead factors.

The “Calculated PHB” graph was received by adding the AspectJ overhead to the “PHB Algorithm” graph. I.e., we took the overhead inflicted by the AspectJ and added it to the “PHB Algorithm” graph. It can be seen that calculated results are very close to the “Total PHB” graph reflecting measurements done by a different method as described above. An additional factor that influences the overhead is the number of node in the domain. Recall that state and message size is a function of the size of the domain. In fact, our experiments indicated that, as expected, the overhead is an increasing function of the size of domain.

In conclusion, we believe that this overhead is an acceptable price for the enhanced security. Furthermore, this overhead can possibly be reduced by using a more efficient implementation.
Chapter 7

Discussion and Future Work

The move from a single server to a distributed computation environment presents interesting challenges with respect to policy definition and enforcement. In this thesis we took a first step toward understanding the problems and developing solutions in this area.

Our mechanism performs conservative policy enforcement when dealing with concurrent LPHB sequences. The execution is terminated if a transition from one of concurrent LPHB sequences violates the policy. While being correct, this behavior limits the space of the policies that can be enforced by the mechanism. In some cases we would like to allow an action if at least one of the concurrent LPHB sequences is allowed by the policy. In order to allow exact enforcement of such policies we plan to add an operator applied to concurrent states. An analog of the \( \text{and} \) operator will require that all the concurrent LPHB sequences are allowed by the policy while an analog of the \( \text{or} \) operator will require that at least one LPHB sequence is allowed.

The SHA based distributed enforcement mechanism used in the loosened communication model may be enhanced by redefining the meaning of its security state. Instead of being a simple set of previously allowed security events, the definition can be extended to include multisets. The multiset based state allows enforcement of policies dependable on the number of events performed by the target.

Under the general SHA automaton the transition \( \delta(S_j, \sigma^i) \) may lead to a violation but a transition \( \delta(S_k, \sigma^i) \) such that \( S_j \subseteq S_k \) may be legal. Therefore, the global query should not be stopped even if the state received so far already leads to violation. Moreover, even
if according to the local state the action leads to violation, the system still has to perform a global query since in the updated global state the same local action may be allowed. This is the motivation to add the following additional limitation to the general SHA:

**Definition 13.** The No Regret Shallow History Automaton or NRSHA is a SHA with the following additional restriction: if $\delta(S_j, \sigma^i)$ is undefined then for every $S_k$ so that $S_j \subseteq S_k$ $\delta(S_k, \sigma^i)$ is also undefined.

In the policy enforced by NRSHA the space of the legal events shrinks over time, i.e., monotonic policies. Every allowed actions can only restrict the space of the action that will be allowed in the future. NRSHA is strictly less expressive then general SHA and cannot enforce policies in which target can gain additional rights by performing some authority gaining action or by “forgetting” classified information in information flow policy.

The benefit of NRSHA enforceable policies is that it is sufficient to detect the local violation without checking the global state. For the same reason, during a global query if violation state have been reached there is no need to continue. As a result, the implementation of the distributed mechanism based on NRSHA is much more efficient then the one using SHA.
Bibliography


52


We consider the problem of finding a policy that can effectively block malicious activity on a network. The approach is based on the AspectJ framework, which allows for the definition of policies in a declarative manner. The policies are then enforced by the AspectJ runtime system, which intercepts method calls and checks if they violate the policy conditions.

The policy enforcement is performed at runtime, and it is transparent to the application. This means that the application continues to work as usual, and the policy enforcement is handled automatically by the AspectJ framework. The policies are written in a language that is similar to Java, and they can be written in a modular way, allowing for easy modification and extension.

In this thesis, we propose a new approach for policy enforcement, which is based on the idea of defining policies in a high-level language and then translating them to the low-level language of the target system. The translation is performed by a compiler, which generates the necessary code to enforce the policy at runtime.

The main advantage of this approach is that it allows for more flexibility in the definition of policies, since the high-level language is closer to the natural way of thinking about security policies. It also allows for easier modification and extension of the policies, since the translation is performed by a compiler, which can be easily modified.

In the next chapter, we will present the details of our approach and show how it can be applied to real-world systems. We will also discuss some of the challenges that we encountered during the development of our system, and we will present some of the results that we obtained.
העבורה של שירותים מורכבים<GameObject> המגננת חמש מערכות מבוזרות מכונות המייצגות פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מרוכביםurbed ניוזי ערכות העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצגות פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצגות פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצגות פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצגות פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות המייצガイド פיזיים וממשיכיו של המרחב שייכים במערכת מערכות מבוזרות. 

העבורה של שירותים מורכבים מספקים את ביצועים הלוחמים של תקע במערכת במערכת מכונות=my_guid 값을 מגדיריםjavacore sollenhilisiert werden mit der Initialisierung.
The title page of a thesis is shown, with some text in Hebrew. The thesis is titled "תקציר" (Summary) and contains the following content:

"The GRID model is adopted in many organizations as a flexible framework for distributed computing, especially in research and educational institutions. The separation of concerns is achieved through various user interfaces, such as the EGEE and TeraGrid systems, which handle tens of thousands of tasks in parallel.

One of the factors behind this success is the use of computers by many users on various systems. Systems such as EGEE and TeraGrid, for example, use thousands of tasks in parallel, which are executed on different locations. These systems use different resources for access, which are protected against attacks such as DOS attacks, which may compromise information and theft.

The security policy, on the other hand, is defined in the system and is based on AAA (Authentication, Authorization, and Access Control) mechanisms. This policy determines whether a user can access a resource, such as a file, during a certain period (e.g., 08:00-16:00). The security policy also includes proper mechanisms to prevent unauthorized access, such as monitoring the system's behavior and checking the legitimacy of actions.

The Security Monitor (EM) tool is used to monitor the system and notify the system administrator of any unauthorized actions. The Security Monitor tool is divided into two main parts: the security control part, which blocks unauthorized actions, and the access part, which allows authorized actions.

In brief, the security policy of the system is defined in the system and is based on the AAA mechanisms. The Security Monitor tool is used to monitor the system and notify the system administrator of any unauthorized actions. The Security Monitor tool is divided into two main parts: the security control part, which blocks unauthorized actions, and the access part, which allows authorized actions.
רשימת תаблицות
רשימה הרשימות

<table>
<thead>
<tr>
<th>מספר</th>
<th>תＰורח</th>
<th>תורח</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>&quot;No Send After Read&quot; האוטומטים המוקמיים</td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>&quot;No Send After Read Unless Certify&quot; האוטומטים המוקמיים</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>דוגמה הרצה</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>&quot;No Send After Read&quot; האוטומטים הגלובליים לשני צמחי</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>&quot;No Send After Read&quot; הרותה של אלגוריתם PHB האוקף מדרני</td>
<td>4.2</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>&quot;No Send After Read Unless Certify&quot; הרצת אלגוריתם HB האוקף</td>
<td>4.3</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>Distributed Security Enforcer משמך מותחני של</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>ריצה וריהוט מ müd</td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>&quot;Chinese Wall&quot; האוטומטים המוקמיים</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>תוקידה מבצעה של הנחמה בפאשקיצים</td>
<td>6.2</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>פנסי הרצה בפרור 6 צמחי</td>
<td>6.3</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>פורק תוקידה במאגרים</td>
<td>6.4</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>פורק תוקידה במאגרים מפותח עובר 6 צמחי</td>
<td>6.5</td>
<td></td>
</tr>
</tbody>
</table>

リンゲージ: 0
ה rekl וצרים
הענינים
וכן
tתקציר

1 
2 
3 
3.1 
3.2 
3.3 
3.3.1 
3.3.2 
4 
4.1 
4.2 
4.2.1 
4.2.2 
4.3 
4.3.1 
5 
6 
7 
8 
9 
10 
11 
12 
13 
14 
15 
16 
17 
18 
19 
20 
21 
22 
23 
24 
25 
26 
27 
28 
29 
30 
31 
32 
33 
34 
35 
36 
37 
38 
39 
40 
41 
42 
43 
44 
45 
46 
47 
48 
49 
50 

1 מבוא
2 סקר ספורות
3 מודל החישוב, מדרנויות מוקומיות וגלובליות ואכיפתן
3.1 מודל החישוב
3.2 מדרינת האבטחה-מנגנונית EM-Enforceable
3.3 מעבר מדרינת מוקומית לאכיפת גלובלית
3.3.1 פירוש-לiado מדרינת האבטחה מוקומית HB
3.3.2 פירוש-לiado מדרינת האבטחה מוקומית PHB
4 מנגנוני אכיפה
4.1 מנגנון אכיפה מרכז
4.2 מנגנון אכיפה מבוזרים
4.2.1 מנגנון מבוזרים PHB
4.2.2 מנגנון מבוזרים HB
4.3 מודל משותף
4.3.1 מודל משותף
5 פירוט
6 תיעוד ייצוגים
7 דיויכי רחובים מחקר ותיעודים
8 מקורות
המחקר נערך בנהנניה פורפסור דנ ו. פקולת להנדסת מחשבים – טכניון
בפקולטה להנדסת מחשבים

אני מודד לוייננקל - מחנך מחכננוגרפיל שירר על התוכניות המכללות והנדסיות בהשחלה

אפיפת מבזירת של מדיות לשכחת

ה данные המבזירות

תאור על מחקר

לשם מילוי חלקי של הדרישות לקבלי התחאר
מגיסטר למדעי מדעי המחשב

אוריה אארלובסקי

הוגשה לסטט הטכניון - מרכז טכניולגי לישראל
אלול לשט"ז חופה אוגוסט 2007
אדריכים מבויתים של מדיניות אבטחה

במערבות מבויתות

אריה אאורלובסקי