Published in IET Information Security
Received on 1st October 2009
Revised on 21st July 2010
doi: 10.1049/iet-ifs.2009.0202

Special Issue on Multi-Agent & Distributed Information Security

ISSN 1751-8709

Covert channel resistant information leakage protection using a multi-agent architecture

S. Bishop[1] H. Okhravi[2] S. Rahimi[1] Y.-C. Lee[1]

[1]Department of Computer Science, Southern Illinois University, Carbondale, IL, USA
[2]MIT Lincoln Laboratory, Massachussets Institute of Technology, Cambridge, MA
E-mail: ylee@cs.siu.edu

Abstract: Covert channel attacks utilise shared resources to indirectly transmit sensitive information to unauthorised parties. Current operating systems (e.g. SELinux) rely on tagging the filesystem with security labels and enforcing security policies at the time of access to a file or resource. However, such mechanisms do not provide strong protection against information laundering via covert channels. Colored Linux, an extension to SELinux, utilises watermarking algorithms to 'colour' the contents of each file with their respective security classification, or context, to enhance resistance to information laundering attacks. In this study, the authors propose a mobile agent-based approach to automate the process of detecting and colouring receptive hosts' filesystems and monitoring the coloured filesystem for instances of potential information leakage. Implementation details and execution results are included to illustrate the merits of the proposed approach. The authors have also evaluated the performance of their agent-based system over a single host as well as a local network of machines. Finally, using formal method techniques, the authors have proved correctness properties about the agent-based approach and identified and corrected a flaw in their initial implementation.

1 Introduction

Information security has been researched to considerable depth in the ongoing quest to provide users and corporate entities a more secure computing environment. Although an extraordinary range of effective approaches have been developed to mitigate threats to information security, new threats appear daily. Within the realm of such threats, among the most difficult to detect and prevent involve covert channel or side channel attacks. A covert channel is a byproduct of shared resources like memory, network interfaces and execution time on computing devices and can be created and accessed dynamically [1, 2]. Examples of information leakage can be found in [1, 3]. Because covert channels are created from shared resources, it is very difficult to detect and prevent their occurrences. Covert channel attacks are often employed to bypass conventional security mechanisms by an authorised insider while leaking oftentimes sensitive information between processes. It is common for such attacks to involve the transfer of data from highly privileged processes to processes that would otherwise lack the necessary permissions to access such data.

According to a 2006 Global Security survey by Deloitte, insider fraud and information leakage contribute 28 and 18% of internal breaches, respectively, [4]. The most critical factor of insider data leakage involves users with varying permissions and privilege levels as designated by their respective positions [5]. Kowalski *et al.*, [5] indicated that more than half of the insider data breaches occurred within organisations and that information was accessed through organisations' computers. Because internal security breaches are caused by legitimate and authenticated users, most conventional security measures cannot effectively detect and prevent such activities.

Modern operating systems counter unauthorised accesses through the use of access control tags or labels, applied to subjects (e.g. processes or users) and objects (e.g. files). These labels are compared with the permissions assigned to the users attempting to access the labelled files. Access is then granted or denied depending on these permissions. Although this mechanism provides effective access control in most situations, it is vulnerable to covert channel attacks. Such attacks enable laundering of the access control tags

© The Institution of Engineering and Technology 2010

applied by the operating system, allowing for arbitrary tag removal or tag reassignment.

Colored Linux [2], introduced by a group of the authors of this paper, provides an extension to SELinux based on data watermarking or colouring. The approach in Colored Linux is to generate blind watermark signatures for all files on a filesystem based on each file's access control tag. These watermarks are then applied to all files. When file access is requested, Colored Linux examines the requested file's watermark and compares it to the file's tag. Discrepancies between the watermark and security tag indicate that an unauthorised modification to the tag has been made and appropriate measures can then be taken. If the watermark matches the tag, SELinux access control measures take over as usual. Furthermore, if adequately robust watermarking algorithms are used, attempts to remove a watermark will render the associated file's contents useless to the attacker.

Colored Linux was implemented primarily through modification of the SELinux kernel modules. The main advantage of this approach is that it does not need any knowledge of covert channels since the modification of the operating system kernel allows for direct interception of access attempts to the filesystem. On the other hand, while an operating system-based colouring scheme works effectively in a 'closed' system (i.e. a system in which every machine is running the modified operating system), it is not as effective in an 'open' system (i.e. a system which is connected to machines with non-coloured operating systems). Unless the borders of an open system are tightly controlled, an insider can distort watermarked files beyond recognition (e.g. using encryption) and leak them to the outside using cross-border covert channels.

To overcome this drawback, we propose an information leakage detection (ILD) agent system to automate the processes of converting a regular machine to a coloured one. Benefits of such an approach include the ability to modify and add detection capabilities in a modular fashion while simultaneously providing conditional deployment of such capabilities. With mobile agents, such dynamism can be realised with little or no administrative involvement. Furthermore, the distributed reporting potential of mobile agent networks can lend itself well to future analysis of information leakage, as well as the underlying covert channel techniques. The agent-based approach also makes the colouring scheme effective in an open system, which is a hybrid of machines running modified operating systems and commodity ones. We have chosen Perl as the foundation of our agent system, given its out-of-the-box integration with Linux, access to the necessary system constructs and relative efficiency.

We have also performed performance evaluation and formal evaluation on our agent-based system in two different operational modes: real-time monitoring and deferred monitoring modes. First, we evaluate the performance of the agent-based ILD by measuring the time that it takes to detect instances of leakage across a local network of machines in real-time under heavy load as well as the time taken to perform initial watermarking. Next, we evaluate the performance of the agent-based ILD by measuring CPU utilisation based on different scanning intervals and number of file system events. Here, the aim is to determine what scanning interval results in less resource usage under varying degrees of load. Then, using formal method techniques, we prove theorems about the agent-based approach for ILD. The goal is to verify that if the agents are secure and authentic, the system can successfully detect instances of information leakage. Using the results of formal evaluation, we were able to identify and correct a flaw in our initial implementation.

In Section 2, related works on information leakage prevention and detection are presented. The threat model used in this paper is described in Section 3. Section 4 provides an overview of Colored Linux whereas our proposed ILD agent community is discussed in Section 5. Section 6 details several detection methodologies and their respective limitations. Section 7 provides the proposed strategies that are implemented in our system. Section 8 lists the inter-agent communications present in our system. In Section 9, implementation details of our system are examined. Section 11 describes the performance evaluation and formal evaluation of the agent-based ILD system. Finally, Section 12 provides conclusions and future directions.

2 Related work

Since most of the studies in the security community deal with preventing outsiders access, there are only a few literature that have proposed methodologies to address the issue of information leakage through insiders. Alawneh and Abbadi introduced a mechanism to protect shared information among organisations via trusted platform module (TPM) [6, 7]. By creating master controller and domains for TPM-equipped devices, contents can only be accessed through the allowed devices. Takesue proposed a scheme to prevent information leakage through portable devices [8]. A modified *i*-node with 1-bit flag bit and 1-bit lock bit imposes authentications with integrated network location checking between storage devices and security server, a user can only access the files when she is inside the company and authentication succeeds. Chang and Kim designed a system to prevent information leakage in ubiquitous computing environments [9]. Their approach utilises cryptographic algorithms and authentication methods in agents to secure sensitive data during communications. Although these approaches present potential solutions to insider information threats, none of them examine the risk of covert channel attacks.

An overview of covert channel attacks are discussed in the following efforts [10–12]. A network-based storage covert channel based on IP time to live field is designed in [13].

© The Institution of Engineering and Technology 2010

doi: 10.1049/iet-ifs.2009.0202

A link-layer network-based covert channel in the MAC protocol based on the splitting algorithm is proposed in [14]. Cabuk *et al.* have designed and studied network-based timing channels and mechanisms to disrupt such channels in [15]. A work by Wang and Lee [16] studies hardware-based (processor-based) timing channels and identifies two such channels in typical simultaneous multi-threaded processors.

There are also countermeasures proposed for known covert channel attacks such as information flow analysis techniques [17], time-domain anomaly [18], entropy-based approach [19], data-dependency scheme [20] and store-forward approach [21, 22]. However, most of these countermeasures work best for known covert channels. The problem, however, is that it is impossible to enumerate all covert channels in a real system. Hence, it is the unknown channels that pose the greatest threat to the security of the system.

Robust watermarking (colouring) offers a strong binding between data and its security tag and can detect information leakage from both insiders as well as through covert channel attacks. Further, our proposed approach to combine mobile agents with Colored Linux methodologies is novel.

3 Threat model

The threat model used in this paper assumes that the applications running inside the operating system are not trusted. They have the potential to leak information and violate the security policy as a result of intentional malicious behaviour or unintentional bugs or programming errors. The trusted computing base, on the other hand, includes the hardware of the system (CPU, main memory and devices) and the operating system kernel.

The operating system in this paper is trusted to enforce access control on 'known' communication channels. However, it is assumed that the operating system has no control over 'unknown' channels inside the system, which include covert and side channels. These channels can be used to leak information across the applications which are otherwise not allowed to communicate according to the security policy. Note that the operating system performs security tag assignment and modification, and the applications are prevented by the operating system from tampering with the tags. However, if a piece of information is leaked through a covert channel, the operating system has no control over the channel and the security tag becomes meaningless at the receiver side. The goal in this work is to detect information leakage as a result of covert channel attacks using an agent-based detection scheme and watermarking.

Finally, we also assume that the agents are authentic and they perform their respective tasks. The authenticity of the agents can be verified using different techniques discussed in the literature for agent security [23, 24] or by using hardware support (e.g. using TPM) [25]. However, agent security is not the focus of this paper and we assume that it is already provided.

4 Colored Linux

Information laundering through covert channel attacks is possible because the binding between the data and its security tag is loose; that is, the security tag is usually appended to the end of the file as a bit stream. If such a file leaks through a covert channel, the tag becomes meaningless and easily removable. The insight behind Colored Linux is to make this binding strong by colouring (watermarking) data files. If the watermarking algorithm is robust, it is impossible for an attacker to remove the watermark without destroying the data itself.

Colored Linux has a colouring algorithm, 'brush', for each file type (e.g. one for images, another one for text files etc.). The set of all algorithms in the operating system is called the 'brush set'.

The entire filesystem is coloured during the initialisation. Upon each access to a file, the colour of the file is compared to its security tag. If there is a discrepancy between the two, it means that the file has been leaked through a covert channel and its security tag has been laundered. Note that Colored Linux knows nothing about the mechanism of covert communication. However, it can detect any leakage and prevent any further damage by taking appropriate measures.

The watermarking algorithms used in Colored Linux must be blind; that is, they should be able to detect the watermark without needing the original file. This ensures that the security of the system is not endangered by storing the original uncoloured file in the filesystem.

Colored Linux is implemented by modifying SELinux hooks. SELinux hooks are invoked whenever a resource is accessed in order to check the policy. By modifying these hooks, whenever a file is accessed, the colour detecting algorithm is called to check the colour of the file and compare it with its security tag. If it matches, the control is passed over to the SELinux engine. Otherwise, the process is terminated and appropriate logs are created. Colored Linux modules are called whenever an object is created, accessed, changed or its tag is modified.

The assumption in Colored Linux is that there is a boundary beyond which covert communication is very difficult or impractical (a closed system). Every system inside that boundary is running colour-aware operating system. Moreover, colour awareness is manually installed on all of the machines inside this boundary. ILD agent-based system addresses these drawbacks by moving through hosts that are not colour-aware and automating the colouring mechanism.

© The Institution of Engineering and Technology 2010

5 ILD agent system

Separation of powers and responsibilities in an agent community encourages flexibility and encapsulation. As such, our agent system is heterogeneous with members belonging to one of six principal archetypes, each adhering to unique roles and possessing distinct abilities. Fig. 1 depicts the classifications of our Information Leakage Detection Agent system and the respective agent ranks. All inter-agent communications adhere to Foundation for Intelligent Agents (FIPA) agent communication language (ACL) specifications [26] in order to maintain communication interoperability between different agent platforms. Properties and responsibilities of each type of agent are discussed in following subsections.

5.1 Controller agents (CAs)

CAs are responsible for dispatching subordinate agents and coordinating their respective activities in a designated network. Additionally, CAs will coordinate the remote installation of the necessary mobile agent environment and other required software packages on target hosts with environment agents (EAs). Multiple instances of CAs can be dispatched to ensure proper coverage of large networks as well as to accomplish load distribution for the purposes of performance optimisation.

5.2 Detection agents (DAs)

The main functionality of DAs is to identify new hosts in the network and to verify the host's states. In our initial design, a host's state will refer to the presence or absence of SELinux and the Colored Linux infrastructure. Once determined, a host's state will be reported to the CA to aid in the identification of subsequent actions.

5.3 Queue agents (QAs)

To avoid overwhelming CAs and to provide an orderly approach to dispatching agents to newly discovered hosts, QAs will be useful. As stated above, when a DA identifies a new remote host, the host's state is reported to a CA. Rather than dispatching agents to a new host immediately, it may be preferred to defer such processing for some time, especially in the case when many such hosts are reported at once. In such cases, hosts are reported by CAs to QAs which prioritise hosts for subsequent processing by, and at the request of, CAs.

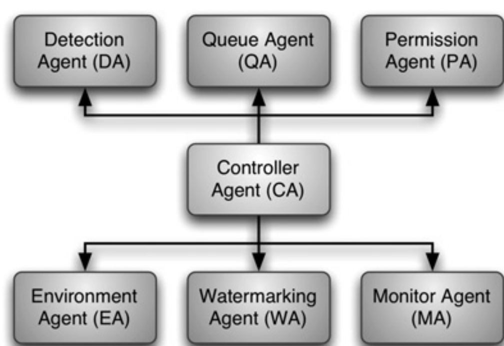

Figure 1 *Agent classifications and hierarchy*

5.4 Monitor agents (MAs)

MAs perform active monitoring on the host filesystem through the 'inotify' kernel subsystem to identify file write and creation operations. Details on the inotify kernel subsystem will be discussed in the next section. When a write operation or file creation operation takes place, MAs notify watermarking agents (WAs) which can then perform watermark analysis of the file in question. As comparable capabilities are already present in Colored Linux hosts, MAs only reside in non-coloured SELinux hosts.

5.5 Watermarking agents (WAs)

Similar to MAs, WAs shall only be present on non-coloured hosts, as determined by DAs. The responsibility of these agents is to watermark all files on a host's filesystem and to perform subsequent watermark analysis at the request of MAs.

5.6 Permission agents (PAs)

A central PA handles permissions issues involving MAs and WAs with their target hosts. Specifically, the PA should ensure that such agents are given only those permissions necessary to perform their respective tasks. In addition, the PA ensures that all permissions necessary for agent environment installation by the EA are in place.

5.7 Environment agents (EAs)

Minimally, WAs and MAs require the necessary agent environment installed on a target host in order to reside and function there. Also, depending on the type of watermarking employed, certain watermarking-specific software dependencies that may not reasonably be accommodated by the WAs themselves can exist. EAs are responsible for handling all such software dependencies without the intervention of the target host's administrator.

6 Detection methodologies

Detecting file 'write' or 'create' operations in a non-coloured host is the first step towards detecting potential information leakage. In this section, we examine four candidate methods and the feasibility and cost associated with performing such detection. One method is then selected and implemented in our proposed approach.

6.1 Memory scanning

Memory scanning involves systematic scans of the target machine's memory space (accessible via the /dev/mem virtual device). In cooperation with information obtained from /proc, it would be possible to locate any given process'

© The Institution of Engineering and Technology 2010

doi: 10.1049/iet-ifs.2009.0202

memory space in /dev/mem and scan that space for 'write' calls. However, the time cost of a single scan of a system memory snapshot depends on the number of processes in the system. As the number of the processes increases, so does the scanning time. Thus, there is no guarantee that the scanning time will always be less than a given processes' execution time, making this method prone to missed detections.

6.2 Process tracing

Process tracing works similarly to the previous method, with the exception that instead of scanning system memory, it will use the Process Trace, 'ptrace,' system call (as is used by the 'strace' command) to attach to a process and monitor all system calls, including 'write' (and 'open' calls with the create flag set, as is needed by the MA). Sharing the same concept as memory scanning, this method is susceptible to the same timing issue. Primarily, we must be aware of process creation and must attach to it with ptrace before the process issues any system calls (or terminates itself).

It is unknown how exactly this can be accomplished. A process polling method may be able to catch all process creations; however, this will dramatically decrease system performance. Therefore the performance overhead makes this method less appealing.

6.3 Kernel-level system call hooking

In order to maintain high system performance and mitigate the time cost associated with 'write' and 'create' operation detection, we explored the possibility and feasibility of detecting such operations via their respective system calls at the kernel level. Although this method would allow for interception of 'every' such system call easily and efficiently, several potential obstacles might prevent us from choosing this method. Machine architectures and kernel versions will surely differ throughout the network, and thus one pre-compiled kernel module carried as agent payload will not be injectable into every target host. It is certainly not reasonable to maintain pre-compiled modules for every possible architecture combined with every kernel version.

One potential solution would be to carry only the module source code as payload, build it on the target machine, and load it into the kernel. Although this may overcome differences in system architectures to some extent, modules for newer kernel versions are written quite differently from those intended for use in much older kernels.

6.4 Inotify Kernel subsystem

The 'inotify' kernel subsystem is a standard filesystem event notification service included in Linux kernels since release 2.6.13 three years ago [27]. This service enables a user to create applications from system libraries to monitor file operation events like read, write or delete on a set of specified files. By default, inotify imposes service limitations of 16 384 maximum events per queue, 128 maximum instances per user and 8192 maximum watches per instance to conserve kernel memory. We are confident that these limitations will not present overwhelming obstacles to the initial implementation of our approach; however, a more comprehensive study on trade-offs between kernel memory and inotify limitations will be conducted in the future.

Hence, our file operation detection in MA utilises the inotify kernel service because it provides not only stability and performance but is also accessible through uncomplicated system libraries. Although inotify is not available in kernel versions prior to 2.6.13, our initial targeted platform employs a fairly recent kernel version. We will later investigate the feasibility of dynamically building and installing inotify modules in older hosts.

7 Strategies

The following subsections illustrate the states and process flow of our system. Process flow is depicted in Fig. 2. Each subsection explains the objectives of each step and how they can be achieved.

7.1 Host discovery

In our proposed agent system, all operations begin with, and are coordinated by, the CA. Initially, it is assumed that all hosts in the network are clean, yet unknown. A DA is dispatched to scan the network for SELinux-based hosts. When the first such host is discovered, the DA determines whether or not the newly found host is 'coloured'. If the host is un-coloured, it is reported to the CA.

7.2 Non-coloured host queueing

When the first non-coloured, SELinux-based host is identified and reported by the DA, the CA shall create a QA and make it aware of the reported host. All subsequent host reports generated by the DA will also be forwarded to the QA. Hosts are enqueued, possibly with priorities, by the QA. At certain times, the CA will query the QA for a new host, which the QA will dequeue and forward to the CA.

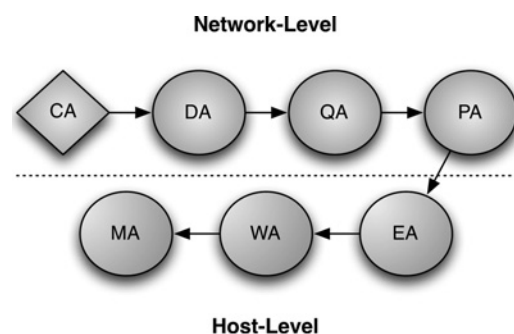

Figure 2 *Process flow of our proposed system*

© The Institution of Engineering and Technology 2010

7.3 Permission determination and management

Given a host report from the QA, the CA will create a PA and assign it to the new host. The PA (using standard Linux remote management facilities, as a mobile agent environment has not yet been installed on the target host) will attempt to determine if the proper permissions are in place for the successful remote installation of an agent environment on the target host, and for the proper operation of subsequently dispatched WAs and MAs. If proper permissions have not been assigned, the PA is responsible for coordinating with the target host to establish the lacking permissions. Once this process has completed, the CA remotely installs (with the aid of a helper EA) the appropriate agent environment on the target host.

7.4 Watermarking target hosts

Following the successful installation of the agent environment on the target host, the CA dispatches a WA to the host. Within the host, the WA 'colours' all files on the host's filesystem. Upon completion of initial colouring, the WA reports completion to the CA, and then awaits subsequent commands. Detection of a newly created file, or of write operations performed on an existing file, are reported to the WA by the MA, prompting the WA to analyse and possibly colour the new file. This process continues until the CA instructs the WA to terminate.

7.5 File creation and write monitoring

Once the WA has reported successful completion of initial colouring to the CA, a MA is sent to the newly coloured host. This agent will then use the methods described above in Section 4, and Section 5.3 to detect and handle potential instances of information leakage.

8 Communications among agents

In our agent architecture, communications among agents follow the FIPA communicative act specification which is based on the Speech Act Theory to facilitate communication interoperability between different agent platforms [26]. The specification defines 22 composite and macro communicative acts to provide conversational actions such as INFORM, REQUEST or PROPOSE. Tables 1–7 illustrates the communication details of the processes mentioned in the previous section.

9 Implementation

9.1 Agent environment

In choosing an appropriate foundation for our agent community, we considered primarily the ease of integration with our target environments, access to the necessary system-level constructs needed to perform the filesystem event detection and subsequent watermark analysis, and efficiency. Perl was chosen for this task, as it is installed with nearly every Linux distribution by default and can access the inotify system. Furthermore, adding functionality

Table 1 CA communications

From: CA
To: DA Ask the DA to notify CA when the first non-coloured host is found. (REQUEST-WHEN) After first host found, ask the DA to notify QA whenever non-coloured hosts are found. (REQUEST-WHENEVER)
To: QA Ask the QA to insert current non-coloured hosts to its queue. (REQUEST) Retrieve the hosts in the QA's queue. (REQUEST with INFORM)
To: PA Request PA to prepare target host for agent environment installation. (REQUEST)
To: WA Ask the WA to watermark the host's filesystem and report the completion. (REQUEST-WHEN)
To: MA Ask the MA to monitor the target host and notify the CA when information leakage occurred. (SUBSCRIBE)
To: EA Ask the EA to check for, and resolve, software dependencies on the target host which may inhibit the functionality of subsequently dispatched agents. (REQUEST)

Table 2 DA communications

From: DA
To: CA Confirm to CA that network scan to determine non-coloured host is proceeding. (AGREE) Notify CA when the first non-coloured host is found. (INFORM) Confirm to CA that notification to QA about non-coloured hosts can proceed. (AGREE)
To: QA Ask the QA to insert current non-coloured hosts in its queue. (REQUEST)

© The Institution of Engineering and Technology 2010
IET Inf. Secur., 2010, Vol. 4, Iss. 4, pp. 233–247
doi: 10.1049/iet-ifs.2009.0202

Table 3 QA communications

From: QA
To: CA Confirm to CA that queue insertion has been performed. (AGREE) Return the current hosts in queue to CA. (INFORM)
To: DA Confirm to DA that queue insertion has occurred. (AGREE)

Table 4 PA communications

From: PA
To: CA Confirm to CA to prepare the host for agent environment installation. (AGREE) Notify CA of the result of host preparation. (INFORM)

Table 5 WA communications

From: WA
To: CA Confirm with CA to perform watermarking operation. (AGREE) Return the result of watermarking operation to CA. (INFORM)

Table 6 MA communications

From: MA
To: CA Confirm with CA to perform queue insertion. (AGREE) Notify CA of the occurrence of information leakage. (INFORM)

Table 7 EA communications

From: EA
To: CA Confirm with CA to perform environment checking and dependency resolution. (AGREE) Notify CA of all resolved dependencies. (INFORM)

to the base Perl installation modularly is straight-forward and easily automated.

We developed a mobile agent system in Perl to implement the design discussed herein. Agents exist as Perl code and agent attributes encapsulated in XML files. Here, Perl's XML parsing modules are beneficial and provide a streamlined approach to extracting and evaluating agent code. Later in this section, implementation details of the WA and MA, the principal subjects of our performance analysis, will be discussed.

9.2 Watermarking algorithms

As different file types require different watermarking schemes, we focused on image files for our experiments. The watermarking algorithm utilised is the Dugad *et al.* [28] algorithm as implemented in Peter Meerwald's watermarking library [29]. This algorithm has many nice properties, especially that of blindness, which is required for our system. Meerwald's library, in turn, depends on the NetPBM library for reading, writing and converting images of a variety of formats.

The algorithm takes the discrete wavelet transform (DWT) of the image. Then instead of adding the watermark to a number of top coefficients, the Dugad algorithm adds the watermark sequence to all coefficients above a threshold T_1. Different images have different levels of smoothness; hence, this approach provides a natural way of adapting the amount of watermark added to the image. The algorithm excludes the lowpass component of the image. As a result, the watermark is effectively added to the edge related information of the image. The watermark is added to the image using (1) in which i is the index over all DWT coefficients greater than T_1, V_i is the DWT coefficient of the original image, and V_i' denotes the DWT coefficient of the watermarked image. x_i is the watermark value at position of V_i and α is taken as 0.2.

$$V_i' = V_i + \alpha|V_i|x_i \tag{1}$$

For watermark detection, Dugad chooses all the highpass coefficient above T_2 and correlate them with the original copy of the watermark. The correlation z between the DWT coefficients $\hat{\mathrm{V}}$ of the corrupted watermarked image and a possibly different watermark Y is computed using (2).

$$z = \frac{1}{M}\sum_i \hat{V}_i y_i \tag{2}$$

9.3 Handling dependencies

External dependencies, such as NetPBM, can be handled in several ways in mobile agent systems. Ideally, all necessary code can efficiently be carried with the agent itself. When this is not viable, the agent execution environment can be made to handle such dependencies. Functionality requiring

doi: 10.1049/iet-ifs.2009.0202
© The Institution of Engineering and Technology 2010

more code than can be reasonably accommodated by agents are installed as Perl modules at the request of the agency. Such modules may be from third-parties, or may be additions specifically suited for the ILD system.

9.4 Implementation of the WA

As described above, the primary role of a WA is to prepare a filesystem for ILD by watermarking all files with a particular permissions tag. Such tags essentially identify the sensitivity of a file and are used in conjunction with permissions assigned to individual users. A user's permissions regulate which files are accessible by the user. Here, accessibility can relate to the ability of a user to read, write or execute a file, or perform any combination of these actions. Information leakage via covert channels may result in the removal or modification of traditional permissions tags. The recipient of the leaked information may alter the tags in order to grant himself access to the information that he was not intended to possess. Watermarking embeds the permissions of a file within the file contents in such a way as to be (ideally) irremovable without rendering the file contents useless.

Algorithm 1

1: **while** *D has children* **do**
2: $d_i \leftarrow$ *child i of D*
3: **if** d_i *is a directory* **then**
4: *Watermark*(d_i)
5: **else**
6: boolean w = *DetectWatermark*(d_i)
7: **if** w = TRUE **then**
8: *Compare watermark of* d_i *with permissions tag*
9: **if** *Watermark does not match tag* **then**
10: *Quarantine or Securely Remove* d_i
11: **end if**
12: **else**
13: *Watermark* d_i *with signature = permissions tag*
14: **end if**
15: **end if**
16: **end while**
17: **return**

Figure 3 *Watermark(Directory D)*

Functionally, the WA developed for our experiments initiates a complete scan of the target filesystem upon entry into a target host. It does not, however, indiscriminately watermark all files encountered. It could be the case that the filesystem, or portions of it, is already watermarked but the agent, agency or supporting infrastructure was damaged or removed because of some unforeseen circumstance. Therefore the WA will attempt to detect the presence of a watermark in all scanned files prior to watermarking. If a watermark is not detected, the file is watermarked immediately with a signature corresponding to the files permissions tag. Conversely, if a watermark is detected, the WA will compare the watermark with the file's permissions tag. If an inconsistency is found, the file is assumed to have been previously leaked, and is either quarantined in a secure directory or securely deleted.

Once the initial watermarking phase is complete, the WA will become dormant. A WA will be awakened upon receipt of signal from the MA indicating that a new file has been created and will therefore need to be watermarked (for future work, the WA shall be made able to detect valid changes of permissions tags, and re-watermark files accordingly). Fig. 3 provides a broad representation of the operations performed by our WA.

Algorithm 2

1: $W \leftarrow$ *inotify event descriptor*
2: **for all** *Target directories* d_i **do**
3: *Add inotify watch descriptor for "write" and "create" operations within* d_i
4: **end for**
5: **loop**
6: $f \leftarrow$ *Read event from event descriptor W*
7: *Pass f to Watermarking Agent for Analysis*
8: **end loop**

Figure 4 *Monitor()*

© The Institution of Engineering and Technology 2010
doi: 10.1049/iet-ifs.2009.0202

9.5 Implementation of the MA

Although the WA effectively binds a files permissions tag to its content, it does not compare the watermark to the permissions of a user attempting to access the file. This task is the responsibility of the MA. The MA serves the primary role of monitoring the target filesystem for any file 'creation' or 'write' operations and notifying the WA of such events for subsequent processing. As stated above, the file operation monitoring is achieved via the inotify kernel subsystem. Fig. 4 represents the MA operations.

10 Covert channel attack scenario

To better understand the types of attacks thwarted by our system, we present a scenario involving a common covert channel.

Consider a Colored Linux system in which our ILD system is in place. A pair of nefarious users, U_A and U_B, wish to exploit a covert communication channel to transfer sensitive information across permissions boundaries and store the transferred data somewhere on the local filesystem. U_A has been assigned Top Secret level, while U_B has been given the more restrictive access level of Confidential. The aim of their subversive effort is to give U_B a copy of a Top Secret document that U_A has legitimate access to, but that U_B would normally not have legitimate access to. We assume here that the system has robust network security and physical security measures in place, so that U_B, upon receipt of the data, cannot merely offload the data to some remote site or store the material on some external physical medium.

The covert channel between U_A and U_B is constructed as follows. U_A has in his possession a small program or script that simply loops rapidly for some interval at pre-determined times (possibly agreed upon with U_B in advance). U_B likewise has a similar program or script, with a modification allowing for the counting of iterations over an interval of time.

When U_A's process is executing in a tight loop, the number of iterations executed by U_B's loop will be less than if U_A's process is not iterating. Therefore it can be agreed upon that if U_B's iteration count in some interval is below some threshold, a value of 1 will be recorded. Otherwise, a value of 0 will be recorded. For each bit in the Top Secret document to be transferred to U_B, U_A's process will then loop when the value of the bit is 1, and will pause when the value of the bit is 0. U_B can then interpret the correct values of the bits comprising the Top Secret document, and write those bits to a file for later review. Since U_B's process is executing with his permission level, the resulting file will be tagged by SELinux with a Confidential classification.

In a non-coloured system, the covert channel described here would successfully subvert the access control mechanisms of the system. Note that to do so requires no access to restricted resources. However, in a coloured host with our ILD system in place, the file creation event associated with U_B would be captured by the MA and the associated file would subsequently be submitted to the watermarking agent for analysis. Since the Top Secret document was transferred covertly bit-by-bit, the resulting file would contain the original watermark. On examination, the WA would identify the file's watermark as belonging to the Top Secret permissions classification, whereas the associated tag would now be Confidential. Owing to the mismatch between the tag and watermark, the offending file would be quarantined and no longer accessible to U_B. Regardless of the implementation details of the covert channel, any similar bitwise transmission of data across permissions boundaries would be detected upon realisation of the transferred data on the filesystem.

11 Evaluation and formal methods

We have evaluated the implementation of our agent-based system in two stages. First, we evaluate the performance of our leakage detection system and especially the time that it takes for the agents to detect instances of leakage under certain conditions. Then we perform a formal evaluation on the logic of the agent-based system. In fact, we could identify and correct a flaw in our initial implementation using the results from the formal analysis.

11.1 Performance evaluation

In evaluating the performance of our agent-based ILD system, we focused first on the time needed to perform initial watermarking of the filesystem and event detection time under heavy load across a network of four machines. Primarily, the time spent, and thus the resources consumed, in the course of scanning for instances of leakage is examined. Here, the aim was to determine what scanning interval resulted in the least resource usage under varying degrees of load. Load, in this case, refers to the number of filesystem events occurring in some period of time. For every such event, the watermark applied to the file associated with the event must be compared to the file's security classification. Additionally, time spent polling the event descriptor for such events is considered in our analysis.

Watermarking analysis was carried out by WA deployed to each of the four test machines. Each agent collected runtime statistics of the watermarking process over a subset of each machine's filesystem, consisting of image files. Each such set of images were identical across the test environment and the Dugad colouring 'brush' was used to perform the watermarking. The results of watermarking analysis, shown in Fig. 5, agree with our logical expectation that watermarking runtime increases linearly with the number of files being watermarked. Additionally, it was shown that the mean time required to watermark a single file is small

© The Institution of Engineering and Technology 2010

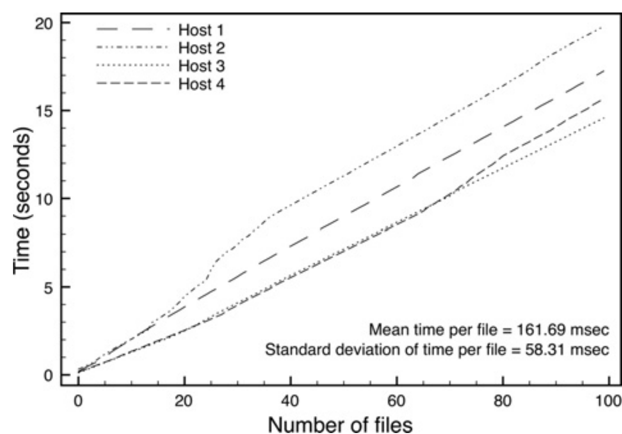

Figure 5 *Watermarking times per host*

enough to ensure that the initial watermarking of an entire filesystem of average size is not prohibitively expensive and that any such subsequent operations will not noticeably reduce system performance.

Aside from filesystem colouring, the detection system used to identify potential instances of information leakage entails the only other source of likely performance degradation. In conducting runtime analysis of the detection process, MA were dispatched to each test machine. At each machine, 1000 instances of information leakage were simulated in rapid succession. For each such instance, the average detection time for all test machines of the instance was determined. Additionally, the overall mean and standard deviation for all such detections was calculated. The results of our detection time analysis can be seen in Fig. 6. Note that the occasional spike in detection time can be attributed to the temporary execution of some CPU-intensive processes. The mean detection time demonstrates that even in heavy load situations, MAs will not produce any noticeable performance reduction.

Next, performance analysis was conducted under system loads ranging from five events per minute to one event every second. Events were simulated probabilistically. Denoting the number of events per minute as E, an event

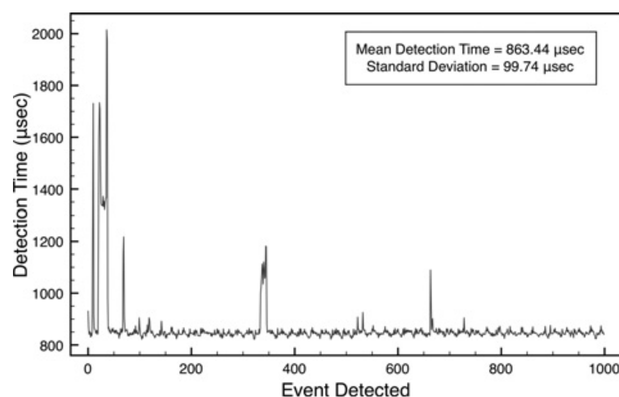

Figure 6 *Detection statistics from four hosts*

interval is then $60/E$. For each event interval, the probability of an event occurring was fixed at $P_E = 0.75$ and, among these events, the probability of leakage occurring was fixed at $P_L = 0.5$. The range of loads, $5 \leq E \leq 60$, were each tested against MA scanning intervals ranging from continuous scanning, to one scan cycle per minute. Fig. 7*a*–*f*, illustrates the results of this analysis for a sample of system loads.

Under the conditions of the performance analysis, continuous scanning was never the optimal solution, as even under the heaviest tested load the MA was wasting resources waiting for event notifications. Generally, as expected and demonstrated, optimal scan intervals decrease as E increases. In our test environment, natural system load averaged roughly 10 events per minute. From our analysis, scanning events roughly every 18.5 s results in the lowest resource consumption in this environment. In environments with widely varying loads, adjusting scan intervals according to E can occur automatically.

11.2 Formal verification

Formal method techniques are used to verify the logic of our agent-based ILD system. More specifically, assuming that the agents are authentic and the watermarking algorithm works correctly, we would like to verify that the logic of the agent-based system is sound; that is, it can correctly identify instances of leakage in the system.

We have used ACL2 to describe and formally prove the correctness of the agents, so we present a quick and brief background on ACL2.

ACL2 (a computational logic for applicative common Lisp) [30] is an automated reasoning system consisting of a language and a mechanical theorem prover. It is the 'industrial strength' successor to the Boyer–Moore theorem prover [31]. Both the ACL2 language and its implementation is built using the side-effect free version of Common Lisp [32].

In common Lisp everything including the code and the data is a list. The lists hold data such as integers, lists, fractions or characters. For instance, (120) is an integer, (1 2 3 6) is a list of integers, (a) is a character and (1/6) is a fraction, all represented as lists. The code is also written using lists, usually with the first element representing the operator or function name and the rest of the elements representing the arguments. For instance, (if x y z) is equivalent to 'if x then y else z', or (car (x y z)) represent the first element of (x y z) which is x.

When using ACL2, first the operation or the model is described using this syntax and a number of function definitions (defun.) Then, we describe a number of theorems (defthm) that ACL2 tool tries to prove about that operation or model. The ACL2 theorem prover then tries to prove the theorem using some basic axioms that it has in

© The Institution of Engineering and Technology 2010
doi: 10.1049/iet-ifs.2009.0202

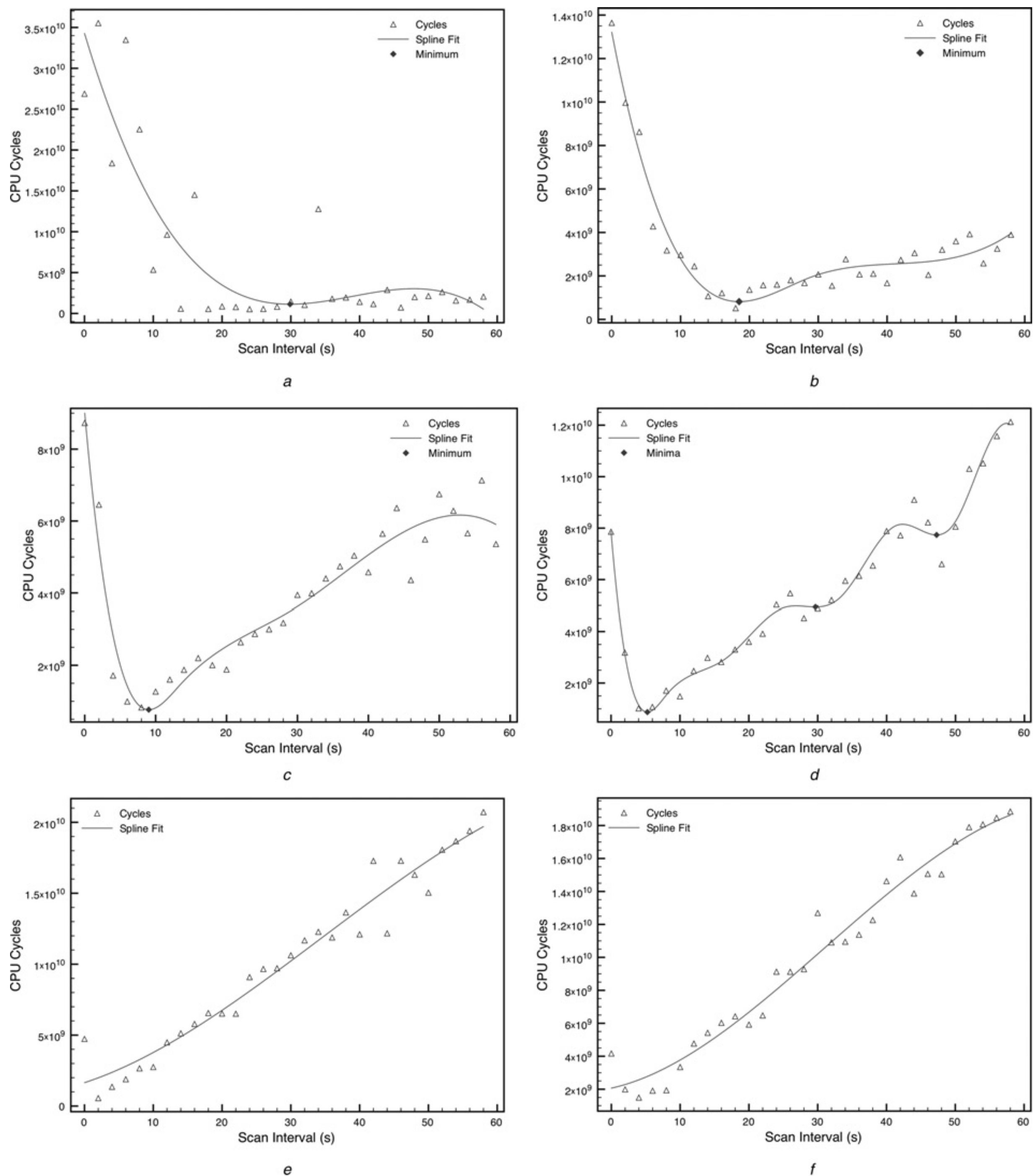

Figure 7 *CPU utilisation per scan interval under varying system load*

a Five filesystem events per minute
b Ten filesystem events per minute
c 20 filesystem events per minute
d 30 filesystem events per minute
e 40 filesystem events per minute
f 50 filesystem events per minute

its libraries and by breaking it into some smaller theorems (subgoals.) Upon successful proof of the theorem, ACL2 outputs the list of rules and axioms it used to prove that theorem. ACL2 theorem prover is sound, but incomplete. As a result, if it proves a theorem, the theorem is always true, but if it fails to prove it, the theorem might be true or false. Detailed description of how ACL2 works or how to check models with ACL2 is beyond the scope of this paper. The readers may refer to the ACL2 book [30] for more information.

© The Institution of Engineering and Technology 2010

11.2.1 Formal verification of agent-based ILD: To model the agent-based ILD system, we first build a model of the agents and a leakage using ACL2. The goal is to prove that if the agents are authentic and the watermarking algorithm works correctly, the agent-based system can successfully detect a leakage. Verifying the soundness of the model helps us to identify any flaw in the logic of the system.

The agents are modelled in Fig. 8. The model captures important features of the system while ignoring less important details. For instance, a file is modelled as a pair (`cons`) consisting of a security tag and a watermark. The content of a file is unimportant to our analysis, so it is not presented in the model. Note that formally verifying the entire low level implementation is prohibitively hard if not impossible because of the large size of the code. However, formally verifying the important features of the system gives us confidence that the system logic is sound and narrows the gap between what is being verified and the actual implementation.

In the model, first the DA finds a set of files to be watermarked on different machines. It then sends a message to the queuing agent to queue all these files. The queuing agent notifies the CA which in turn sends the queue to the WA. Upon the completion of watermarking, the CA calls the monitoring agent to verify that the security tags and the watermarks are equal for the files being accessed. The messages are modelled as ACL2 constants and are passed into the agents as arguments. An instance of leakage is modelled as a function which modifies the security tag of a file, but does not change the watermark. In the interest of space we have not included the less important parts of the model including the queue model.

The monitoring agent works in two different modes: real time monitoring and deferred monitoring. The monitoring agent model takes a delay argument. If the delay is zero, the agent models the real time monitoring behaviour and immediately monitors the file's watermark. If the delay is greater than zero, the agent models the deferred monitoring mode. In this mode, two scenarios are possible. The first scenario is that the delay is smaller than the acceptable threshold time during which the leakage is negligible and acceptable. In this mode, the file is monitored before it is communicated to another machine. Otherwise, if the delay is larger than a threshold, it is possible for the leaked file to be communicated to other machines before it is checked. Note that for high security applications, the agent must operate in the real time mode to prevent any possible leakage. The deferred monitoring mode is only for less critical applications that can tolerate a threshold delay.

By formally verifying the system, in the real time mode, we revealed a flaw in our implementation: namely, even if the

```
(defconst *QU_CONT_INFORM* 1)
(defconst *WM_CONT_INFORM* 2)
(defconst *MON_CONT_INFORM_PASS* 3)
(defconst *MON_CONT_INFORM_FAIL* 4)
(defconst *DET_QU_REQUEST* 5)
(defconst *ALARM* 6)
(defconst *SUCCESS* 7)
(defconst *MACHINE_ID* 10)
(defconst *NO_DETECTION* nil)
```

```
(defun detection_agent (flag)
   (if (car flag)
       (list *DET_QU_REQUEST* 1 2 3 4)
        nil))
```

```
(defun wm_agent (q)
   (cons *WM_CONT_INFORM*
         (wm_q q)))
(defun wm_q (q)
   (if
      (AND (cdr q)
           (car (dequeue q)))
      (cons (cons
           (car (dequeue q))
           (list (car (dequeue q))))
           (wm_q (cdr q)))
     (list (cons
           (car (dequeue q))
           (list (car (dequeue q)))))))
```

```
(defun monitor_agent_delay (delay machine q)
   (if (< delay 1)
      (if (equal machine (car q))
         (if (check_wm (car (cdr q)))
             (cons *MON_CONT_INFORM_PASS* nil)
             (cons *MON_CONT_INFORM_FAIL* nil))
         (cons nil nil))
      (if (equal machine (car (communicate_files q)))
         (if (check_wm (car (cdr q)))
             (cons *MON_CONT_INFORM_PASS* nil)
             (cons *MON_CONT_INFORM_FAIL* nil))
         (cons nil nil))))
(defun check_wm (q)
   (if (cdr q)
      (AND (check_wm (cdr q))
           (consp (dequeue q))
```

Figure 8 *Formal model of the agents*

© The Institution of Engineering and Technology 2010
IET Inf. Secur., 2010, Vol. 4, Iss. 4, pp. 233–247
doi: 10.1049/iet-ifs.2009.0202

```
(defun communicate_files (q)
   (cons (+ (car q) 1)
         (cdr q)))
(defun queue_agent (message q)
   (cons *QU_CONT_INFORM*
      (list
         (cadr message)
         (build_queue
            (cons
               (car message)
               (cddr message))
            q ))))
(defun build_queue (message q)
   (if
      (AND
         (= (car message) *DET_QU_REQUEST*)
         (consp (cddddr message) ))
      (build_queue
         (cons *DET_QU_REQUEST* (cdddr message))
         (enqueue
            (list
               (cadr message)
               (caddr message))
            q))
      (enqueue
         (list
            (cadr message)
            (caddr message))
         q)))
```

```
(defun controller_agent (delay message)
   (case (car message)
     (1 (wm_agent (cdr message)))
     (2 (monitor_agent_delay delay
           (cadr message)
           (cdr message)))
     (3 *SUCCESS*)
     (4 *ALARM*)
     (nil nil)
     (t nil)))
```

```
(defun leak (q)
   (cons *WM_CONT_INFORM*
         (fake_tag (cdr q))))
(defun fake_tag (q)
   (if (AND
```

Figure 8 *Continued*

```
(defthm working
   (equal
     (controller_agent
       (controller_agent
         (controller_agent
           (queue_agent
             (detection_agent '(t)) nil))))
     *SUCCESS*))
```

```
(defthm leak_detection_real_time
   (equal
     (controller_agent
       (controller_agent
         (leak
           (controller_agent
             (queue_agent
               (detection_agent '(t)) nil)))))
     *ALARM*))
```

```
(defthm leak_detection_channel
   (equal
     (controller_agent
       (channel
         (controller_agent
           (channel
             (leak
               (controller_agent
                 (channel
                   (queue_agent (channel
                     (detection_agent '(t)) nil)))))
                 ))))
     *NO_DETECTION*))
```

```
(defthm leak_detection_large_delay
   (equal
     (controller_agent 2
       (controller_agent 2
         (leak
           (controller_agent 2
             (queue_agent
               (detection_agent '(t)) nil)))))
     *NO_DETECTION*))
```

```
(defthm leak_detection_small_delay
   (equal
     (controller agent 1/2
```

Figure 9 *ILD theorems*

© The Institution of Engineering and Technology 2010

agents are secure and authentic, they communicate over a shared channel (network). An attacker can flood the network resulting in some messages being dropped randomly. For instance, if an 'INFORM' message from the monitoring agent notifying the CA of an instance of leakage is dropped, the CA never alarms the administrator. We have corrected this flaw by modifying the CA. If the CA does not receive any message back from the receiver after a short time, it automatically alarms the administrator. Also, a 'heartbeat' message is constantly sent back and forth between the controller and the monitoring agent to check the health of the communication channel. If the heartbeat stops, the CA again issues an alarm.

Then we model the system in the deferred monitoring mode. In fact, for an arbitrary delay, the detection is not sound. As the formal model shows, it is possible for a file to leak and then migrates to a new machine before the monitoring agent's delay expires. However, if the delay is sufficiently small, the monitoring agent detects the leakage before the leaked file is communicated to another machine. This value is normalised to '1' in the ACL2 model of the agents.

Fig. 9 shows the theorems to check the correctness of the model. The first theorem, 'working', checks the correct operation of the agents in a normal situation with no attack. The 'leak_detection_real_time' theorem verifies that in the presence of an attack, the agents can correctly detect the leakage. The 'leak_detection_channel' theorem checks the system when agents operate over an unreliable communication channel. The 'channel' model randomly drops some of the messages. Evaluating this theorem to SUCCESS fails because of the channel. After correcting the implementation, however, the last theorem can be evaluated to SUCCESS instead of NO_DETECTION. The fourth theorem, 'leak_detection_large_delay', models the deferred monitoring mode with a large delay which allows the leaked files to migrate from one machine into another one. In this case, the detection fails and the system is not sound. The last theorem, 'leak_detection_small_delay', on the other hand, models the deferred monitoring mode with a small amount of delay. In this mode, the detection is still successful. Note again that this does not mean that leakages cannot happen; it means that the leakages happened during the small delay are negligible and tolerable for that specific application.

ACL2 was able to model the agents and prove the theorems about them by breaking the theorems into 20 subgoals and by using 76 rules and axioms.

12 Conclusion

The primary benefit of an agent-based ILD system lies in the ability to modify and add detection capabilities, modularise those capabilities and then conditionally employ such capabilities at the discretion of a central control mechanism (in our system, the CA). The use of mobile agents as described in this paper, and in general, reduces the per-host administrative complexity. Once the initial agent environment is properly installed and configured, all further necessary actions are performed by the agents themselves. Additionally, mobile agents are able to provide unique reporting capabilities that, for the purposes of our research, may benefit the analysis of information leakage and the underlying covert channels through which information has been leaked.

Although the detailed ILD approach is based on the work of [2], future work in this area may lead to the inclusion of techniques aimed at detecting and blocking covert channels prior to the occurrence of information leakage. Given the highly varied nature of covert channel methods, detecting all such methods is likely a matter for which a solution can only be obtained through the liberal use of techniques rooted deeply in the field of artificial intelligence.

13 References

[1] National Computer Security Center: 'A guide to understanding covert channel analysis of trusted systems', NCSC-TG-30, November 1993, http://www.radium.ncsc.mil/tpep/library/rainbow, accessed May 2009

[2] OKHRAVI H., BAK S.: 'Colored linux: covert channel resistant OS information flow security'. University of Illinois at Urbana-Champaign, 2008

[3] TANAKA H.: 'Information leakage via electromagnetic emanation and effectiveness of averaging technique'. Proc. Int. Conf. on Information Security and Assurance, April 2008, pp. 98–101

[4] MELEK A., MACKINNON M.: '2006 Global Security Survey'. Research report, Deloitte, 2006

[5] KOWALSKI E., CAPPELLI D., MOORE A.: 'Insider threat study: illicit cyber activity in the information technology and telecommunications sector'. Technical report, National Threat Assessment Center and Carnegie Mellon University, CyLab, January 2008

[6] ALAWNEH M., ABBADI I.M.: 'Preventing information leakage between collaborating organisations'. Proc. Tenth Int. Conf. on Electronic Commerce, 2008, vol. 342, pp. 1–10

[7] ALAWNEH M., ABBADI I.M.: 'Preventing insider information leakage for enterprises'. Proc. Second Int. Conf. on Emerging Security Information, Systems and Technologies, 2008, pp. 99–106

[8] TAKESUE M.: 'A scheme for protecting the information leakage via portable devices'. Proc. Int. Conf. on Emerging

© The Institution of Engineering and Technology 2010
doi: 10.1049/iet-ifs.2009.0202

Security Information, Systems, and Technologies', 2007, pp. 54–59

[9] CHANG H., KIM K.: 'Design of inside information leakage prevention system in ubiquitous computing environment' (Springer Berlin, 2005), (*LNCS*, **3483**), pp. 128–137

[10] CABUK S.: 'Network covert channels: design, analysis, detection, and elimination'. PhD thesis, Purdue University, 2006

[11] MILLEN J.: '20 years of covert channel modeling and analysis'. Proc. 1999 IEEE Symp. on Security and Privacy, 1999, pp. 113–114

[12] ZANDER S., ARMITAGE G., BRANCH P.: 'Covert channels and countermeasures in computer network protocols', *IEEE Commun. Mag.*, 2007, **45**, (12), pp. 136–142

[13] QU H., SU P., FENG D.: 'A typical noisy covert channel in the IP protocol'. Proc. 38th Annual Int. Carnahan Conf. on Security Technology, 2004, pp. 189–192

[14] LI S., EPHREMIDES A.: 'A covert channel in MAC protocols based on splitting algorithms'. Proc. IEEE Wireless Communications and Networking Conf., 2005, vol. 2, pp. 1168–1173

[15] CABUK S., BRODLEY C., SHIELDS C.: 'IP covert timing channels: design and detection'. Proc. 2004 ACM Conf. on Computer and Communications Security, 2004, pp. 178–187

[16] WANG Z., LEE R.B.: 'Covert and side channels due to processor architecture'. Proc. 22nd Annual Computer Security Applications Conf., 2006, pp. 473–482

[17] TSAI C., GLIGOR V., CHANDERSEKARAN C.: 'On the identification of covert storage channels in secure systems', *IEEE Trans. Softw. Eng.*, 1990, **16**, (6), pp. 569–580

[18] WANG C., JU S.: 'Searching covert channels by identifying malicious subjects in the time domain'. Proc. Fifth Annual IEEE SMC Information Assurance Workshop, 2004, pp. 68–73

[19] GIANVECCHIO S., WANG H.: 'Detecting covert timing channels: an entropy-based approach'. Proc. 14th ACM Conf. on Computer and Communications Security, 2007, pp. 307–316

[20] MELLIAR-SMITH P.M., MOSER L.E.: 'Protection against covert storage and timing channels'. Proc. Computer Security Foundations Workshop IV, 1991, pp. 209–214

[21] KANG M.H., MOSKOWITZ I.S., LEE D.C.: 'A network pump', *IEEE Trans. Softw. Eng.*, 1996, **22**, pp. 329–338

[22] KANG M.H., MOSKOWITZ I.S., LEE D.C.: 'The Pump: a decade of covert fun'. Proc. 21st Annual Computer Security Applications Conf., 2005, pp. 352–360

[23] ZHONG S., YANG Y.R.: 'Verifiable distributed oblivious transfer and mobile agent security', *Mobile Netw. Appl.*, 2006, **11**, (2), pp. 201–210

[24] BURKLE A., HERTEL A., MLLER W., WIESER M.: 'Evaluating the security of mobile agent platforms', *Auton. Agents Multi-Agent Syst.*, 2008, **18**, (2), pp. 295–311

[25] MUNOZ A., MANA A., HARJANI R., MONTENEGRO M.: 'Agent protection based on the use of cryptographic hardware'. Proc. 2009 33rd Annual IEEE Int. Computer Software and Applications Conf., 2009, **2**, pp. 312–317

[26] FIPA communicative Act Library Specification: 'Foundation for intelligent physical agents', 2000, http://www.fipa.org/specs/fipa00037/, accessed May 2009

[27] LOVE R.: 'Kernel korner: intro to inotify', *Linux J.*, 2005, p. 8

[28] DUGAD R., RATAKONDA K., AHUJA N.: 'A New wavelet-based scheme for watermarking images'. Proc. Int. Conf. on Image Processing, October 1998, vol. 2, pp. 419–423

[29] MEERWALD, P.: 'Digital watermarking', http://www.cosy.sbg.ac.at/~pmeerw/Watermarking/, accessed May 2009

[30] KAUFMANN M., MOORE J.S.: 'ACL2: an industrial strength version of Nqthm' (Computational Logic, Inc., 1996)

[31] KAUFMANN M., BOYER R.S.: 'The Boyer–Moore theorem prover and its interactive enhancement', *Comput. Math. Appl.*, 1995, **29**, (2), pp. 27–62

[32] STEELE G.: 'Common LISP: the language (LISP series)' (Digital Press, 1984)

© The Institution of Engineering and Technology 2010