Preventing Intrusions through Non-Interference

Marco Giunti
Dipartimento di Informatica
Università Ca' Foscari di Venezia
giunti@dsi.unive.it

***Abstract*— The ability to prevent and to detect intrusions in computer systems is often heavily conditioned by having some knowledge of the security flaws of the system under analysis. Discover intrusions is particularly hard in concurrent systems, which contain several interactions among their components; suspicious interactions are usually studied manually by security experts which need to establish if they are dangerous. In this paper, we present an automated method to prevent intrusions in concurrent systems that does not require any previous knowledge of the flaws. We study the behaviour of an abstract model of the system that captures its security-related behaviors; the model contain the trusted components of the system such as the file system, privileged processes, etc. We then check all possible interactions with unprivileged processes to decide if the system contain security flaws. This is accomplished by introducing a non-interference security property which holds for models where unprivileged processes do not have direct or indirect write access to resources with an high security level. The property is based on traces and can be decided by using standard concurrency tools. Our method apply even to models containing information flows among their components; this turns out to be a necessary condition for analyzing interactions of actual computer systems, where privileged processes usually have both read and write access to low resources.**

I. Introduction

In the last years, several techniques have been developed to the purpose of guarantee the security of computer systems. Among these, two of the most used by security administrators are intrusion detection and vulnerability analysis. In the literature, intrusion detection refers to a range of techniques that discover run-time attacks to computer systems by means of analysis of system logs [1], [2]. A common approach consists in parse logs to find attack patterns picked from a database of signatures, e.g. [3], [4], [5]. Vulnerability analysis, instead, concerns with the problem of identify security flaws of computer systems, e.g. [6], [7]. We call this technique *intrusion prevention*, since it focuses on preventing attacks, rather than on detecting attacks at run-time.

Unfortunately, such techniques rely on previous knowledge of the security flaws arising in computer systems. For practical purposes, such flaws are often expressed as traces considered dangerous w.r.t.the system under analysis, and no general definition for undesired behaviour is provided. For instance, most signatures-based detectors and vulnerability tools contain patterns of some interaction among unprivileged users and privileged programs which causes the system's password file to change, but they not explain which general property is unsatisfied when the interaction occur. The need of foundations for these techniques is thus emerging as a central issue [8].

Our interpretation is that both intrusion prevention, and intrusion detection, are special cases of access control; given a security policy for the system, then an intrusion is the sequence of steps which directly or indirectly breaks the policy (e.g., similarly to [9], [10]). For instance, the malicious interaction depicted above is an intrusion for a system where is supposed that unprivileged processes cannot write the password file. Indeed, an unprivileged user may indirectly avoid this prohibition by means of a privileged program which acts like a *Trojan Horse*. Such internal attacks, usually hard to discover without any previous information, can be actually find out by using non-interference [11].

In this paper, we develop a method to prevent intrusions that does not require any previous knowledge of the flaws of the system under analysis. The technique we propose may discover unknown attacks, and apply to a wide range of concurrent systems; these are indeed essential requirements for foundations for discovering intrusions [8]. We adopt the model-based approach [12], [7] to vulnerability analysis, which consists in analyze the behaviour of an abstract model of the system that captures its security-related behaviors; the model contain the trusted components of the system such as the file system, privileged processes, etc. To prevent internal attacks of apparent honest privileged programs hiding malicious code, we check all possible interactions with unprivileged processes, and we decide if these interactions comport security flaws. We identify flaws as direct or indirect violations of the mandatory policy which denies to low level processes to have write access to resources with an high security level; roughly speaking, this policy may be called the "no write up" policy. We characterize secure systems by means of a non-interference [11] security property called *Non Write-Modifiability on Compositions*, which holds for those systems that, when composed with unprivileged processes, do not change their behaviour w.r.t. modification of sensible resources; that is, in these systems low processes may not directly and indirectly write high resources. The property is based on traces and can be decided automatically by using standard concurrency tools (e.g., [13], [14]); in the presence of a violation, a counter-example is given: this may be useful to generate traces of unknown attacks. Our approach apply even to models containing information flows among its components; this turns out to be a necessary condition for analyzing interactions of actual computer systems, where privileged processes usually have both read and write access to low level resources. We use the property we proposed to

analyze two abstract models of UNIX-like systems that contain known vulnerabilities [15], [16].

Non-Interference

In general, in a system, information is typically protected via some access control policy, limiting accesses of entities (such as users or processes) to data. There are different levels of flexibility of access control policies depending on the possibility for one entity to change the access rights of its own data. Here we will consider *mandatory* policies [17] in which entities have no control on the access rights. Unfortunately, even when direct access to data is forbidden by (strong) security policies, it might be the case that data are *indirectly* leaked or modified by trusted programs which hide inside some malicious code.

The necessity of controlling information flow as a whole (both direct and indirect) motivated Goguen and Meseguer in introducing the notion of *Non-interference* [11], which formalizes the absence of information flow within deterministic systems. Given a system in which *confidential* (i.e., high level) and *public* (i.e., low level) information may coexist, *non-interference* requires that confidential inputs never affect the outputs on the public interface of the system, i.e., never interfere with the low level users. If such a property holds, one can conclude that no information flow is ever possible from high to low level. A possibilistic security property can be regarded as an extension of non-interference to non-deterministic systems [18]. Various such extensions have been proposed, e.g. [19], [20], [21], depending on the notion of behaviour that may possibly be observed of a system, i.e., the semantics model that is chosen to describe the system; most of these properties are based on *traces* or execution sequences.

In [22], Focardi and Gorrieri express the concept of non-interference in the *Security Process Algebra* (SPA) language, which is an extension of CCS [23] where the visible actions are partitioned in two sets, high and low. In particular in [22] the authors introduce the notion of *Non Deducibility on Compositions* : a system E is "non deducible" if what a low level user sees of the system is not modified by composing any high level process with the system. The low vision of the system is expressed by means of trace-based (*NDC*) or bisimulation (*BNDC*) semantic properties, with important differences: the former property may be expressed equivalently avoiding universal quantification, while the latter need to consider all possible high processes. Indeed the decidability of *BNDC* is still an open problem, although proof techniques for a large class of processes have been proposed [24], [25]. The whole approach is similar to testing equivalence [26].

Preserving the integrity of data

In this paper we focus on integrity policies, i.e. policies which care about improper modification of data, rather than secrecy policies, i.e., policies which prevent unauthorized disclosure of data. In the literature, the standard mandatory integrity policy is known as the Biba model [27] and maybe roughly summarized through the rules "no read down", "no write up"; the Biba policy is the dual of the standard secrecy policy [17]. We refer to *Non Modifiability on Compositions* as to the property requiring that the high view of the system does not change in presence of low users, that is, this property is the dual of Non Deducibility on Compositions.

Unfortunately, Non Modifiability on Compositions is of little help in the analysis of models of actual operating systems, where it is practically infeasible that no information flows from low to high. As a matter of fact, in such systems privileged programs have often read access to low, potentially insecure resources; it turns out that most models of these systems do not satisfy Non Modifiability on Compositions. Since our aim is to discover potential intrusions in abstract models of actual systems, we need to distinguish the observable behaviour of systems where information flows from low to high, and thus we often cannot use this property.

For these reasons, we introduce a weaker security property which takes care of different types of access, i.e. distinguish read from write accesses. The property holds for models where unprivileged users do not have write access to sensible resources, both directly and indirectly, i.e. by exploiting malicious code of trusted programs. We verify this invariant by checking if the high behaviour w.r.t. write actions of the model does not change in the presence of any possible low process. The property is defined in a trace-based setting, and enjoys a characterization which permits to avoid universal quantification on low processes, making the property suitable for actual systems analysis. The main drawback of the choice of a trace-based semantics model is the inability to detect deadlocks, which seems to us less significant in the scenario under consideration.

To illustrate our technique, we draw an example of a core file system process. We use a CCS-like language where the set of visible actions is partitioned in the high (H) and the low (L) level as in SPA, but each of these sets is in turn partitioned in the read and the write subset; actions and co-actions belong to the same subset.

We abstract a file system as a process that change it's state if and only if executes write actions.

$$\begin{aligned} F &\triangleq write_L.F' + read1_H.F \\ F' &\triangleq read2_H.F' \end{aligned}$$

Executing the $write_L$ action causes F to move to state F', and F' exhibits a new action $read2_H$ that was not available in F. Since $read2_H$ has an high security level, this represent a flow from the low to the high level. However, since the high behaviour of F w.r.t. to write actions does not change by executing low actions, we conclude that F is safe.

The paper is structured as follows: in Section II we present some basic notions on the process algebra SPA^{rw}; Section III contain the definition of a SPA^{rw} model for a small subset of a UNIX-based system which contains a well-known vulnerability. In Section IV we present the *Non Write-Modifiability on Compositions* security property, and we verify that the SPA^{rw} model presented in Section III is flawed. In Section V we

study recent attacks based on symbolic link vulnerabilities. Comparison with related work and a few concluding remarks are reported in Section VI.

II. Read/Write Security Process Algebra

We present the syntax and the semantics of our model language, the *Read/Write Security Process Algebra* (SPA$^{\mathsf{rw}}$, for short). The SPA$^{\mathsf{rw}}$ language is a variation of CCS [23], where the set of visible actions is partitioned into two levels, high and low, which are in turn partitioned into two access type sets, read and write, in order to specify multi-level systems with read/write access control. SPA$^{\mathsf{rw}}$ syntax is based on the same elements as CCS that is: a set $\mathcal{L}$ of *visible* actions containing *input* actions $a, b, \ldots$ and *output* actions $\bar{a}, \bar{b}, \ldots$; a special action τ which models internal computations, i.e., not visible outside the system; a complementation function $\bar{\cdot} : \mathcal{L} \rightarrow \mathcal{L}$ such that $\bar{\bar{a}} = a$, for all $a \in \mathcal{L}$. $Act = \mathcal{L} \cup \{\tau\}$ is the set of all *actions*. The set of visible actions is partitioned into four sets Act_i^c, $i \in \{H, L\}$, $c \in \{\mathsf{r}, \mathsf{w}\}$ such that $\overline{Act_i^c} = Act_i^c$. Moreover, the sets Act_i^c are disjoint and they cover $\mathcal{L}$: $\bigcap_{c\in\{\mathsf{r},\mathsf{w}\},i\in\{H,L\}} Act_i^c = \emptyset$ and $\bigcup_{c\in\{\mathsf{r},\mathsf{w}\},i\in\{H,L\}} Act_i^c = \mathcal{L}$. We indicate with Act_L,Act_H, respectively the set $Act_L^{\mathsf{r}} \cup Act_L^{\mathsf{w}}$, and the set $Act_H^{\mathsf{r}} \cup Act_H^{\mathsf{w}}$.

The syntax of SPA$^{\mathsf{rw}}$ *terms* (or *processes*) is defined as follows:

$$E ::= \mathbf{0} \mid a.E \mid E + E \mid E|E \mid E \setminus v \mid E[f] \mid Z$$

where $a \in Act$, $v \subseteq \mathcal{L}$ and $a \in v$ iff $\bar{a} \in v$, $f : Act \rightarrow Act$ is such that $f(\bar{a}) = \overline{f(a)}$, $f(\tau) = \tau$, $f(Act_i^c) \subseteq Act_i^c \cup \{\tau\}$ for each $c \in \{\mathsf{r}, \mathsf{w}\}$ and $i \in \{H, L\}$, and Z is a constant that must be associated with a definition $Z \stackrel{\triangle}{=} E$.

The syntax of processes is standard for CCS and needs no comments; we will abbreviate often the process $E.0$ with E. For the definition of security properties it is also useful the *hiding* operator, /, of CSP [28], which can be defined as a relabelling as follows: for a given set $v \subseteq \mathcal{L}$, $E/v \stackrel{\text{def}}{=} E[f_v]$ where $f_v(x) = x$ if $x \notin v$ and $f_v(x) = \tau$ if $x \in v$. In practice, E/v turns all actions in v into internal τ' s. For the sake of brevity we will write often $E\backslash_L$ to indicate the process $E\backslash_{Act_L}$. We denote by $\mathcal{E}$ the set of all SPA$^{\mathsf{rw}}$ processes. We let $\mathcal{L}(E)$ denote the *sort* of E, i.e. the set of the actions occurring syntactically in E, $E \in \mathcal{E}$. We indicate with $\mathcal{E}_L$ the set of *low* processes: $\mathcal{E}_L \stackrel{\triangle}{=} \{E \in \mathcal{E} : \mathcal{L}(E) \subseteq Act_L \cup \{\tau\}\}$.

The operational semantics of SPA$^{\mathsf{rw}}$ processes is given in terms of *Labelled Transition Systems*; the inference rules are reported in Tab. I and are standard for CCS. The syntax of the *value-passing* version of SPA$^{\mathsf{rw}}$ is reported in Tab. II; value expressions $e_1, ..., e_n$ need to be consistent with arity of actions a and of constants A, respectively, whereas b is a boolean expression. A constant A is defined by $A(x_1, ..., x_m) \stackrel{def}{=} E$ where E is a value-passing SPA$^{\mathsf{rw}}$ agent that must not contain free variables except $x_1, ..., x_m$, which need to be distinct. As described in [23], value-passing calculus semantics is given by translating the calculus into the pure calculus.

Finally let $\Longrightarrow$ be a multi-step transition relation such that if $E(\stackrel{\tau}{\rightarrow})^* \stackrel{a}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* E'$ then $E \stackrel{a}{\Longrightarrow} E'$. If $\gamma = a_1, \ldots a_n$, we write $E \stackrel{\gamma}{\Longrightarrow}$ to mean that there exists E' s.t. $E \stackrel{a_1}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} E'$. The set of (weak) traces of E, $T(E)$, is defined as follows: $T(E) = \{\gamma \in \mathcal{L}^* : E \stackrel{\gamma}{\Longrightarrow}\}$. Two processes are *trace equivalent*, noted $E_1 \approx_T E_2$, if $T(E_1) = T(E_2)$.

III. Modeling Security-related Behaviour of Systems

We define a SPA$^{\mathsf{rw}}$ model of a small subset of a UNIX-based system which will exhibit a vulnerability [15], [12] due to /etc/utmp, a file containing the association between logged users and related terminals. We model the security behaviour of the trusted components of the system, which are a simplified file system process, the mail utility *comsat*, and an user sending mail. To ensure that our method is effective, we need to take care to assign actions to the appropriate subsets. To this aim, we follow these guidelines.

- Security levels
  - (i) system programs execute both low and high actions
  - (ii) privileged programs execute high actions
  - (iii) normal programs execute low actions
- Access mode
  - (iv) an action belongs to the write subset if its synchronization comport the file system state to change
  - (v) an action belongs to the read subset if its synchronization does not comport the file system state to change

The code of the file system is given in the value-passing SPA$^{\mathsf{rw}}$; the values we pass abstract contents and files names; we will use V to indicate the finite set of values, and we suppose that V contain a value $\varnothing$ representing empty contents. We let *pwd*, *utmp*, *tty* abbreviate respectively the values /etc/pwd, /etc/utmp, and /dev/tty, and we let $F = \{pwd, utmp, tty, mailbox\}$, $F \subseteq V$. We restrict to this set of files only for convenience; what follows extends to arbitrary finite set of files. The set of (value-passing) labels is $\{write_H, write_L, read_H, read_L\}$; the translation of each of this label in the pure calculus give raise to actions belong to the appropriated subsets. We model a simplified file system process $Fs(s)$ as reported in Table III; the process regulates read and write accesses to the set of files represented by F. The variable s indicates the actual values contained in files, i.e. the values which are available through the actions $\overline{read_l}(f, \mathsf{extract}(s, f))$; the state s may change by means of the function update, which writes a given content on a file $f \in F$. To illustrate, let s_0 be the state where /etc/utmp contains the value /dev/tty; we have that $Fs(s_0)$ contains the thread $\sum_{l\in L,H} \overline{read\_l\_utmp\_tty}$. If $Fs(s_0) \xrightarrow{write\_L\_utmp\_new} Fs(s_1)$ then the reduct contains the thread $\sum_{l\in L,H} \overline{read\_l\_utmp\_new}$.

In the file system, we implement the mandatory access control policy which we want to enforce; roughly speaking, the policy is expressible as "no write-up", that is, no controls on the read accesses are needed to enforce this policy, while low

Prefix	$\dfrac{}{a.E \xrightarrow{a} E}$	Sum	$\dfrac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \quad \dfrac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$
Parallel	$\dfrac{E_1 \xrightarrow{a} E_1'}{E_1 \mid E_2 \xrightarrow{a} E_1' \mid E_2} \quad \dfrac{E_2 \xrightarrow{a} E_2'}{E_1 \mid E_2 \xrightarrow{a} E_1 \mid E_2'}$ $\dfrac{E_1 \xrightarrow{a} E_1' \quad E_2 \xrightarrow{\bar{a}} E_2'}{E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'} \; a \in \mathcal{L}$	Restriction	$\dfrac{E \xrightarrow{a} E'}{E \setminus v \xrightarrow{a} E' \setminus v}$ if $a \notin v$
Relabelling	$\dfrac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$	Constant	$\dfrac{E \xrightarrow{a} E'}{Z \xrightarrow{a} E'}$ if $Z \triangleq E$

TABLE I
THE OPERATIONAL RULES FOR SPA$^{\mathsf{rw}}$

$a(x_1,...,x_n).E$, $\overline{a}(e_1,...,e_n).E$, $\tau.E$	$Prefixes$
if b **then** E **else** E	$Conditional$
$A(e_1,...,e_n)$	$Costant$
$\sum_{i\in I} E_i$, $E_1 \mid E_2$, $E\backslash v$, $E[f]$	$Sum,\ Parallel\ Composition,\ Restriction,\ Relabelling$

TABLE II
VALUE-PASSING SPA$^{\mathsf{rw}}$ SYNTAX

$$
\begin{array}{lcl}
Fs(s) & \triangleq & Write(s) + Read(s) \\
Write(s) & \triangleq & \sum_{l\in L,H} write_l(x_f, x_c)\,.\,\textbf{if } \mathsf{w} \in \mathrm{ACL}(l, x_f) \\
 & & \quad \textbf{then } Fs(\mathsf{update}(s, x_f, x_c)) \\
 & & \quad \textbf{else } Fs(s) \\
Read(s) & \triangleq & \sum_{l\in L,H,f\in F} \overline{read}_l(f, \mathsf{extract}(s,f)).Fs(s)
\end{array}
$$

TABLE III
FILE SYSTEM

	$/etc/utmp$	$mailbox$	$/etc/passwd$	$/dev/tty$
L	r, w	r, w	r	r
H	r, w	r, w	r, w	r, w

TABLE IV
THE ACCESS CONTROL LIST (ACL)

processes cannot write high resources. In the value-passing version of the system, we enforce the no write-up policy by means of the access control matrix defined in Tab. IV. Each entry of the matrix may contain r, w or both; that is, if $\mathrm{ACL}(l, f)$ contain r, w then the level l has the capability to both read and write the file f. We build the matrix by following a standard method [17] consisting in assigning security levels to files; according to the scenario we are modeling, /etc/passwd and /dev/tty have an high security level; the labels in the matrix express which permission is granted on files to security levels. As introduced, this access control method block direct attempts of low users to write high resources; indirect accesses exploited by using back-doors of trusted programs, or bad configurations of the system, need to be analyzed through non-interference. We will see how to do that in the next section.

The comsat server checks the mailbox for incoming mail and prints the new message on the terminal in which the user is logged on; this information is retrieved from /etc/utmp. We let $Comsat$ read and write labels belong to H since in the scenario under consideration comsat is a privileged program.

$$
\begin{array}{rcl}
Comsat & \triangleq & \sum_{m\in V\setminus\varnothing}\ read\_H\_mailbox\_m\,. \\
 & & \sum_{t\in V} read\_H\_utmp\_t\,.\,\overline{write\_H\_t\_m}.Comsat
\end{array}
$$

The model of the system is the process

$$S \triangleq Fs(s_0) \mid Comsat \mid U$$

where $U = \sum_{v\in V} \overline{write\_H\_mailbox\_v}$ represents an high user process writing some value v in the mailbox. Remember that in the state s_0 the terminal file $/etc/utmp$ contains the value $/dev/tty$; the value of the other files is irrelevant.

IV. NON-INTERFERENCE PROPERTIES

The *NDC* [22] security property aims at guaranteeing that no information flow from the high to the low level is possible, even in the presence of malicious processes. The main motivation is to protect a system also from internal attacks, which could be performed by the so called *Trojan Horse* programs, i.e., programs that are apparently honest but hide inside some malicious code. We refer to the converse security property of *NDC*, which denies information flows from low to high, as to the *Non Modifiability on Compositions (NMC* for short). Property *NMC* is based on the idea of checking the system against all low level potential interactions, representing every

possible low level malicious program. In other words, a system E is *NMC* if what a high level user sees of the system is not modified by composing any low level process Π to E. *NMC* may be used to verify if a system respects the Biba policy [27] w.r.t. both direct and indirect information flows, that is, in *NMC* systems both high processes do not read low resources and low processes do not write high resources.

We introduce some notations. We indicate with $\mathcal{E}_L$ the set of *low* processes. More formally, let $\mathcal{L}(E)$ denote the *sort* of E, i.e. the set of the (possibly executable) actions occurring syntactically in E, $E \in \mathcal{E}$; then the set of low level agents is defined as $\mathcal{E}_L \stackrel{\triangle}{=} \{E \in \mathcal{E} : \mathcal{L}(E) \subseteq Act_L \cup \{\tau\}\}$. We have the following definition.

Definition 4.1 (Non Modifiability on Compositions): Let $E \in \mathcal{E}$.

$$E \in NMC \quad \text{iff} \quad \forall\, \Pi \in \mathcal{E}_L \,.\, E/_L \approx_T (E \mid \Pi)\backslash_L$$

The main obstacle to a practical use of safety properties that consider all possible attackers is the universal quantification on processes; indeed the decidability of the bisimulation version of *NDC* (and in turn of the bisimulation *NMC*) is unknown [22]. The use of less discriminating properties based on traces permit to avoid this problem by supplying a characterization which is decidable by standard concurrency tools [22], [14]. To decide if a systems is *NMC*, we avoid universal quantification on low level processes by representing all possible low level interactions by relabelling low actions to τ. The intuition is that all possible backdoors or bugs of the system exploitable by low level attackers are now disclosed; indeed the interaction with low processes is no more needed as synchronization of actions and co-actions has been replaced by internal reduction.

Theorem 4.2: $E \in NMC \Leftrightarrow E/_L \approx_T E\backslash_L$.

Proof: The proof follows the same rationale of the proof of Theorem 4.5, but is simpler.

Based on this (strict) notion of security, we may check if the file system model defined in Section III is indeed safe.

Example 4.3: $S \notin NMC$. This result can be established by checking if holds Theorem 4.2 by means of a concurrency tool which accepts the CCS syntax, e.g. [13], or more directly by using the CoSEC tool [14]. We illustrate the intuition of why S is not *NMC*. In process S the state s_0 of the file system may change by executing write requests on low resources; this provokes the high behaviour of S and particularly of $Fs(s_0)$ to change. Similarly to the example of the introduction, writing /etc/utmp with some value v causes the filesystem to change the state s_0 as low users have write rights on the terminal file (i.e. w $\in ACL(L, /etc/utmp)$). Consequently, in the state reached after the reduction, is visible an action $\overline{read\_H\_utmp\_v}$ originated by the file system in order to communicate the new content v of /etc/utmp to high processes. More formally, we have that there is $v \in V \setminus tty$ s.t. $\overline{read\_H\_utmp\_v} \in T(S/_L)$) and $\overline{read\_H\_utmp\_v} \notin T(S\backslash_L)$. Intuitively this holds since no processes overwrite /etc/utmp in S alone, and thus the reducts of $S\backslash_L$ have /etc/utmp containing the init value tty.

This example shows the inadequacy, in such cases, of the *NMC* security property. For instance, if comsat specification was consisting in reading on which terminal is logged the recipient of a new message, this system would be still considered unsafe. But it is clear that such system would be normally acceptable by most security administrators; indeed no violations of the integrity of the system could happen if comsat does not use the information available in the terminal file to execute write actions. As a matter of fact, we observe that *NMC* is too strong for the security analysis of models of actual systems, and thus is of little help.

To capture this situations that are very likely in modern systems, we introduce a security property weaker than *NMC* called *Non Write-Modifiability on Compositions* (*NWMC* for short). The idea is to check systems by analyzing the high level behaviour w.r.t. write actions, rather than on all high level actions (in fact write and read actions). A system E is *NWMC*, if for every low level process Π, a high level user cannot distinguish the *write* behaviour of E from the one of $(E|\Pi)$, i.e., if Π cannot interfere with the high level write execution of the system E. In other words, a system E is *NWMC* if what a high level user sees of the system with respect to write actions is not modified by composing any low level process Π to E.

Definition 4.4 (Non Write-Modifiability on Compositions): Let $E \in \mathcal{E}$.

$$E \in NWMC \;\; \text{iff} \;\; \forall\, \Pi \in \mathcal{E}_L, \;\; E/_L\backslash_{H^r} \approx_T (E \mid \Pi)\backslash_{L \cup H^r}$$

As in the case of *NMC*, it is fundamental to supply a proof technique for *Non Write-Modifiability On Compositions*.

Theorem 4.5: $E \in NWMC \;\Leftrightarrow\; E/_L\backslash_{H^r} \approx_T E\backslash_{L\cup H^r}$.

Proof: Let $E \in NWMC$. Thus $E/_L\backslash_{H^r} \approx_T (E \mid \mathbf{0})\backslash_{L\cup H^r}$ for the specific $\Pi = \mathbf{0}$. From $T(E \mid \mathbf{0})\backslash_{L\cup H^r}) = T(E\backslash_{L\cup H^r})$ we infer $E/_L\backslash_{H^r} \approx_T E\backslash_{L\cup H^r}$. For the if direction, let $E/_L\backslash_{H^r} \approx_T E\backslash_{L\cup H^r}$. We have that $T(E\backslash_{L\cup H^r}) \subseteq T(E \mid \Pi)\backslash_{L\cup H^r}$ and in turn $T(E/_L\backslash_{H^r}) \subseteq T(E \mid \Pi)\backslash_{L\cup H^r}$. The opposite direction also holds; indeed if Π and E synchronize on some action, this is a low action as only such labels syntactically occur in Π. $E/_L\backslash_{H^r}$ can do this internal reduction since the low co-action participating in the synchronization is relabelled to τ in $E/_L$.

The following proposition formalizes the intuition that *NWMC* is less discriminating than *NMC*.

Proposition 4.6: $NMC \subset NWMC$

Proof: Let $E \in NMC$. By definition, for all $\Pi \in \mathcal{E}_L$, we have $E/_L \approx_T (E \mid \Pi)\backslash_L$. Since trace equivalence is closed under restriction [23], we infer $E/_L\backslash_{H^r} \approx_T (E \mid \Pi)\backslash_L\backslash_{H^r}$, that is $E \in NWMC$. To see that the inclusion is strict, consider the example of the introduction:

$$F \;\stackrel{\triangle}{=}\; write_L.F' + read1_H.F \quad F' \stackrel{\triangle}{=} read2_H.F'$$

We have that $read2_H \in T(F/_L)$ while $read2_H \notin T(F\backslash_L)$; by Theorem 4.2 it follows that $F \notin NMC$. From $T(F/_L\backslash_{H^r}) = \varnothing = T(F\backslash_{L\cup H^r})$ and Theorem 4.5 we infer $F \in NWMC$.

Once we have weakened our notion of security concentrating on the violations of the integrity of the system, we check if the system of Section III contains potential security flaws. We find out that actually that system mounts attacks a-lá [15]; indeed any privileged file ([15] mentions /etc/passwd) may be overwritten by comsat if a malicious user subverts the terminal file /etc/utmp.

Example 4.7: $S \notin NWMC$. A counter-example is that $S/_L\backslash_{H^r}$ has trace $\overline{write\_H\_pwd\_v}$, while $S\backslash_{L\cup H^r}$ has not. To illustrate, notice that when low actions are blocked we have the invariant that in all states /etc/utmp contains /dev/tty; this holds since the specification of the system does not include any process overwriting such file. Thus only write co-actions $\overline{write\_H\_mbx\_v}$ or $\overline{write\_H\_tty\_v}$ are observable; the former represent mail sended by U, the latter originate from Comsat printing mail on the terminal. Conversely, when low actions are hidden, we can reach a state where the file system makes available a port $\overline{read\_H\_utmp\_pwd}$ which corresponds to communicating that /etc/ utmp contains the value /etc/passwd. To see that, remember that low level users can overwrite /etc/utmp by synchronizing with the file system; in this case such synchronization is turned into an internal reduction by the hiding operator $/_L$. Comsat synchronizes with the file system though the port $\overline{read\_H\_utmp\_pwd}$ and subsequently exhibits an action $\overline{write\_H\_pwd\_v}$ that is not visible when low actions are blocked. Particularly, when v is the value root::0:0, Comsat set root's password to blank [15]. To conclude, we verify that the solution proposed in [15] to this flaw actually recover the security of the system. Indeed once patched the system by protecting file /etc/utmp against malicious users, i.e. by setting $\mathrm{ACL}(L, /etc/utmp) = \mathsf{r}$, we actually obtain that $S/_L\backslash_{H^r} \approx_T S\backslash_{L\cup H^r}$; this can be automatically verified through the Concurrency workbench [13]. By Theorem 4.5 we obtain that the patched system is NWMC.

Applying our method, we have thus found that the model introduced is flawed, since a low user of the system may indirectly break the "no write up" mandatory system's security policy obtaining that the high level password file (in fact any privileged file) is overwritten. Using a security property less discriminating than *NMC* gave us the possibility to individuate the core of the intrusion. That is, by checking if the system satisfies the *NWMC* property we individuated the action which exactly cause the security policy to be broken. Moreover, we verified that the solution suggested in [15] actually patches the vulnerability.

V. Symbolic Links Vulnerabilities

In this Section we draw another example of the use of the *NWMC* property to prevent intrusions arising in the specification of concurrent systems.

Many recent attacks (e.g, [16], [29], [30], [31], [32]) exploit the use of symbolic links to overwrite sensible files. In many UNIX-like systems often privileged programs create files in a temporary directory like /tmp using predictable filenames without checking whether a file with that name already exists. Malicious local users can therefore create a symbolic link to a file that will be used by such programs. The malicious user can then overwrite or append to the file as it is being used by the privileged program. We show how such attacks can be discovered by using the NWMC property.

To this aim, we extend the file system of Table V to permit the use of symbolic links. We abstract a simplified file system where paths are associated to file descriptors. To read (write) a file, one first requires the file descriptor associated to the path, then directly reads (writes) the file descriptor; the read / write operations succeed only if the process may access the file descriptor. The set FD is the finite set of file descriptors available. The access control list now defines accesses from security levels to file descriptors, rather than to paths. We consider two files, $/etc/passwd$ and $/tmp/config$ and we let s_0 be the initial state where /tmp/config and /etc/passwd have respectively file descriptor $strc$ and $strp$, which are different. We let high processes to read and write both $strc$ and $strp$ while low processes can read both file descriptors.
The file system defined in Table V formalize the intuitions above. We use a value passing channel $slink$ to set symbolic links among paths. i.e. to assign the file descriptor of the target path to the file descriptor of the source path. Pure actions obtained from the translation of $slink$ belong to the write subsets, as this operation comport a change of the state of the system. To obtain the file descriptor of a path, a value passing channel $open$ is used; we let pure actions obtained from $open$ to belong to the read subsets, as the execution of such actions do not comport a system change.

$$
\begin{array}{lcl}
Fs(s) & \triangleq & Write(s) + Read(s) + Link(s) \\
Link(s) & \triangleq & SetLink(s) + GiveLink(s) \\
SetLink(s) & \triangleq & \sum_{l\in L,H} slink_l(x_s, x_d).Fs(\mathsf{updateFD}(s, x_s, x_d)) \\
GiveLink(s) & \triangleq & \sum_{l\in L,H,f\in F} \overline{open}_l(f, \mathsf{extractFD}(s, f)).Fs(s) \\
Write(s) & \triangleq & \sum_{l\in L,H} write_l(x_a, x_c)\,. \\
 & & \quad \textbf{if } \mathsf{w} \in \mathrm{ACL}(l, x_a) \textbf{ then } Fs(\mathsf{update}(s, a, x_c)) \\
 & & \quad \textbf{else } Fs(s) \\
Read(s) & \triangleq & \sum_{l\in L,H,a\in A} \overline{read}_l(a, \mathsf{extract}(s, a)).Fs(s)
\end{array}
$$

TABLE V
File System with Symbolic Links

We consider a program $P \in \mathcal{E}_H$ writing some content c on the file /tmp/config

$$P \triangleq P' \mid \sum_{a\in FD} open\_H\_/tmp/config/\_a.\overline{write\_H\_a\_c}$$

and such that program P both does not write the file descriptor $strp$ and does not setup a symbolic link from /tmp/config to any other file. More formally we assume

$(\bigcup_{v\in V} write\_H\_strp\_v \quad \cup_{v\in V} \quad slink\_H\_/tmp/config\_v) \cap \mathcal{L}(P) = \varnothing$.

The model of the system is the process $S \triangleq Fs(s_0) \mid P$.

We follow the approach of Section IV and we use *NWMC* property to check if the system defined above contains violations of its integrity.

Example 5.1: We use Theorem 4.5 to show that $S \notin NWMC$. As in the previous flaw, the presence of a privileged process acting as a Trojan Horse is fundamental. Indeed, $S/_L\backslash_{H'}$ may evolve to a state where a symbolic link from /tmp/config to /etc/passwd is setted up. Intuitively this holds since the definition of $FS(s)$ let low users setup symbolic links between any possible files (this represents the actual situation in UNIX-like systems); such low level actions are turned into internal synchronizations by the hiding operator $/_L$. Now program P synchronizes with the updated system and receives as a file descriptor associated to /tmp/config the value $strp$ which points to /etc/passwd. In the next step P overwrites the file descriptor associated to /etc/passwd and flaws the system. This indeed cannot happen in $S\backslash_{L\cup H'}$ since both low actions are blocked and by the assumptions we did on P (actually on $\mathcal{L}(P)$) saying that P does not overwrite /etc/passwd and does not setup a link from /tmp/config to other files. More formally, we have that there is $v \in V$ s.t. $\overline{write\_H\_strp\_v} \in T(S/_L\backslash_{H'})$ and $\overline{write\_H\_strp\_v} \notin T(S\backslash_{L\cup H'})$.

Unfortunately in this case no reasonable workaround are available to patch the system [16] (block linking in /tmp is probably not feasible). This suggests that using files with predictable names and without subsequent checks in the specification of programs that will run as privileged is highly not recommended.

Summing up, we showed that the property $NWMC$ can be effectively used to verify the safety of of systems where trusted parts as the file system interact with privileged processes possibly containing back-doors or vulnerabilities.

VI. Discussion

In this paper, we studied how to prevent intrusions in CCS-like models representing the security-related behaviors of concurrent systems. We interpret an intrusion as the sequence of steps which breaks the mandatory security policy which denies to unprivileged users to have both direct and indirect write access to sensible resources. We introduced a non-interference property based on traces which is valid for models satisfying this policy. The property may be decided by using standard concurrency tools, and does not need previous knowledge of the vulnerabilities of the system. We drew two examples of how to use the property to verify the security-related behaviour of abstract systems where it is supposed that unprivileged processes cannot directly or indirectly have write access to privileged resources.

We took inspiration from [12] in modeling the behaviour of abstract systems to verify security properties . In [12], [7], the authors define security properties trough an intentions model which captures the intended outcome of executing every program. Their implementation find vulnerabilities as violations of invariant properties, e.g. in all states the password file is not overwritten. The authors focus on the automated analysis of interactions among the system components, which in usual vulnerability analysis techniques is done manually. We adopt the same model-based approach to vulnerabilities analysis, while we express security properties as mandatory access control policies. The use of non-interference let us discover both direct violations, and attacks exploited by means of trusted malicious programs containing back-doors. Moreover, we do not require any previous knowledge of the vulnerabilities of the system modeled.

Focardi and Gorrieri [22] introduced non-interference properties of CCS-like models where actions have different security levels. The authors studied the behaviour of models representing small components of a system, e.g. a buffer cell, by means of trace-based and bisimulation semantic properties. The approach based on bisimulation properties is more discriminating, and can discover attacks which exploit covert channels; however, the study of bisimulation properties is more expensive, and the decidability of the bisimulation non interference is unknown. In both cases the aim is to establish if no information flows among the security levels of the model analyzed. We argue that this requirement is too strict for models of actual systems where it is practically infeasible that no information flows from the low to the high level. We weakened this condition by permitting flows that do not comport any modification of the high resources.

The use of types to enforce access control in abstract mobile systems have been recently an active field of research, e.g. [33], [34], [35], [36], [37]. Particularly, in [36], [37] it is shown how non-interference can be enforced by using type systems. Extending our approach to a typed analysis of configuration vulnerabilities appears a promising field of research. We reserve to future work an integration among the two approaches.

References

[1] J. P. Anderson, "Computer security threat monitoring and surveillance," Fort Washington, Pennsylvania, Tech. Rep. 79F296400, April 1980.

[2] D. E. Denning, "An Intrusion-Detection model," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 2, pp. 222–232, 1987.

[3] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State transition analysis: A rule-based Intrusion Detection approach," *IEEE Transactions on Software Engineering*, vol. 21, pp. 1–22, March 1995.

[4] M. Roger and J. Goubault-Larrecq, "Log auditing through model checking," in *Proc. of 14th IEEE Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada, June 2001*. IEEE Comp. Soc. Press, 2001, pp. 220–236. [Online]. Available: citeseer.nj.nec.com/roger01log.html

[5] J.-P. Pouzol and M. Ducassè, "Formal specification of intrusion signatures and detection rules," in *Proc. of 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, 2002.

[6] D. Farmer and E. H. Spafford, "The COPS Security Checker system." in *USENIX Summer*, 1990, pp. 165–170.

[7] C. Ramakrishnan and R. Sekar, "Model-based analysis of configuration vulnerabilities," *Journal of Computer Security (JCS)*, vol. 1/2, no. 10, pp. 189–209, 2002.

[8] T. F. Lunt, "Foundations for intrusion detection?." in *Proc. of Computer Security Foundations Workshop (CSFW' 00)*, 2000, pp. 104–106.

[9] J. Zimmermann, L. Mé, and C. Bidan, "An improved reference flow control model for policy-based intrusion detection." in *ESORICS*, 2003, pp. 291–308.

[10] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *Proc. of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002, pp. 177–188.

[11] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proc. of the IEEE Symposium on Security and Privacy (SSP'82)*. IEEE Computer Society Press, 1982, pp. 11–20.

[12] C. Ramakrishnan and R. Sekar, "Model-based vulnerability analysis of computer systems," in *Proc. of the 2nd Int. Workshop on Verification, Model Checking and Abstract Interpretation (VMCAI'98)*, 1998.

[13] R. Cleaveland, J. Parrow, and B. Steffen, "The concurrency workbench: A semantics-based tool for the verification of concurrent systems." *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 1, pp. 36–72, 1993.

[14] R. Focardi and R. Gorrieri, "The compositional security checker: A tool for the verification of information flow security properties." *IEEE Trans. Software Eng.*, vol. 23, no. 9, pp. 550–571, 1997.

[15] C. Coordination Center, "Advisory CA-1994-06 Writable /etc/utmp Vulnerability," 1994, http://www.cert.org/advisories/CA-1994-06.html.

[16] S. Focus, "Gnu automake symbolic link vulnerability," 2004, Gentoo Linux Security Advisory 200404. [Online]. Available: http://www.securityfocus.com/advisories/6542

[17] D. E. Bell and L. J. L. Padula, "Secure computer systems: Unified exposition and multics interpretation," MITRE MTR-2997, Technical Report ESD-TR-75-306, 1975.

[18] D. Sutherland, "A Model of Information," in *Proc. of the 9th National Computer Security Conference*, 1986, pp. 175–183.

[19] S. N. Foley, "A Universal Theory of Information Flow," in *Proc. of the IEEE Symposium on Security and Privacy (SSP'87)*. IEEE Computer Society Press, 1987, pp. 116–122.

[20] D. McCullough, "Specifications for Multi-Level Security and a Hook-Up Property," in *Proc. of the IEEE Symposium on Security and Privacy (SSP'87)*. IEEE Computer Society Press, 1987, pp. 161–166.

[21] J. McLean, "Security Models," *Encyclopedia of Software Engineering*, 1994.

[22] R. Focardi and R. Gorrieri, "Classification of Security Properties (Part I: Information Flow)," in *Proc. of Foundations of Security Analysis and Design (FOSAD'01)*, ser. LNCS, R. Focardi and R. Gorrieri, Eds., vol. 2171. Springer-Verlag, 2001, pp. 331–396.

[23] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[24] F. Martinelli, "Partial Model Checking and Theorem Proving for Ensuring Security Properties," in *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'98)*. IEEE Computer Society Press, 1998, pp. 44–52.

[25] A. Bossi, R. Focardi, C. Piazza, and S. Rossi, "Verifying Persistent Security Properties," *Computer Languages, Systems and Structures*, vol. 30, no. 3-4, pp. 231–258, 2004.

[26] R. D. Nicola and M. C. B. Hennessy, "Testing equivalences for processes," *Theoretical Computer Science*, vol. 34, no. 1–2, pp. 83–133, Nov. 1984.

[27] K. Biba, "Integrity considerations for secure computer systems," MITRE Corporation, Tech. Rep. ESD-TR-76-372, 1977.

[28] C. Hoare, *Communicating Sequential Processes*, ser. International Series in Computer Science. Prentice Hall, 1985.

[29] SecuriTeam, "Maelstrom symbolic link vulnerability," 2002. [Online]. Available: http://www.securiteam.com/unixfocus/5FP0S0U60I.html

[30] US-CERT, "Redhat linux diskcheck.pl creates predictable temporary file and fails to check for existing symbolic link of same name," 2000, Vulnerability Note VU#570952. [Online]. Available: http://www.kb.cert.org/vuls/id/570952

[31] ——, "gpm creates temporary files insecurely," 2001, Vulnerability Note VU#426456. [Online]. Available: http://www.kb.cert.org/vuls/id/426456

[32] ——, "Netscape vulnerable to arbitrary file overwriting via symlink redirection of temporary file," 2001, Vulnerability Note VU#356323. [Online]. Available: http://www.kb.cert.org/vuls/id/356323

[33] B. Pierce and D. Sangiorgi, "Typing and subtyping for mobile processes," *Mathematical Structures in Computer Science*, vol. 6, no. 5, 1996.

[34] M. Hennessy and J. Rathke, "Typed behavioural equivalences for processes in the presence of subtyping," *Mathematical Structures in Computer Science*, vol. 14, no. 5, pp. 651–684, 2003.

[35] M. Bugliesi and M. Giunti, "Typed processes in untyped contexts," in *Proc. of TGC 2005, Symposium on Trustworthy Global Computing*, ser. Lecture Notes on Computer Science, R. Nicola and D. Sangiorgi, Eds., vol. 3705. Springer-Verlag, 2005, pp. 19–32.

[36] M. Hennessy, "The security picalculus and non-interference," *Journal of Logic and Algebraic Programming*, vol. 63, no. 1, pp. 3–34, 2004.

[37] S. Crafa and S. Rossi, "A theory of noninterference for the pi-calculus." in *TGC*, 2005, pp. 2–18.