On Run-Time Enforcement of Policies

Harshit Shah[1] and R.K. Shyamasundar[2]

[1] Dep. Informatica & TLC, Univ. of Trento, Italy
shah@dit.unitn.it
[2] School of Tech. & Comp. Science, TIFR, Mumbai, India
shyam@tcs.tifr.res.in

Abstract. Monitoring untrusted code for harmful behaviour is an important security issue. Many approaches have been proposed for restricting activities and the range of untrusted code. Among these, run-time monitoring is a promising approach for constricting run-time behaviour of programs. In this paper we describe a method of containing the effects of untrusted code with respect to a specified policy. We use a guarded command like language for specifying policies that could monitor system calls, APIs or library routines of the underlying system. We also discuss a system call monitoring architecture for an operating system like Linux. We provide semantics of the language in terms of Security Automata and also discuss how pure past temporal properties can be automatically compiled into policies in guarded command language. This allows users to specify policies in terms of logical formulae and automatically generate monitoring algorithm for the same in terms of guarded commands. We show how simple modifications allow us to specify constraints on the overall behaviour of a group of processes.

1 Introduction

Software is a very important and very complex component of computer systems. A user does not have a thorough understanding of most of the code running on his machine. In such a scenario, code containing malicious payload becomes a serious threat [14]. Such a piece of code tricks the unsuspecting user into believing that it adds functionality to the device. In [18], Ken Thomson showed how a seemingly harmless compiler could be loaded with harmful content that would open a backdoor to the system.

The amount of damage that can be incurred by malicious code can be huge. It can perform a range of nasty activities from corrupting a system to obtaining complete control over it. The situation becomes worse when one considers mobile platforms that have limited capabilities (which also come at a price). It therefore becomes important to make sure that these capabilities are not abused and to also provide mechanisms that ensure that untrusted code does not breach user's expectations. Achieving this objective is not easy. It is difficult to check code for malicious content. Code writers use a high level of sophistication to make sure that the unwanted behaviour goes undetected. It is impractical to expect the end

I. Cervesato (Ed.): ASIAN 2007, LNCS 4846, pp. 268–281, 2007.
© Springer-Verlag Berlin Heidelberg 2007

user to carefully examine and execute untrusted code so as not to compromise system security.

Run-time monitoring of code comes across as a promising approach to constrain activities of a running program. In such scenarios, a user typically writes a policy which is then used to instrument code (to insert checks at appropriate places) or which is compiled into a separate monitor. Many specification languages and monitoring mechanisms have been proposed. Some of the specification languages are platform specific (e.g., for Linux) while some are specific to a programming language (e.g., Java). In order to achieve platform independence, it has been proposed to abstract the underlying system. The policy specification can then be provided in terms of the abstraction [19]. Often, policy specification becomes difficult and user requires good amount of knowledge about the specification language to ensure that he has written the correct policy.

In order to make specification easier, we use a simple Guarded Command Policy Specification Language (GCPSL) to specify security policies that have to be enforced on untrusted code. GCPSL has the advantage that it describes the actions that monitor has to perform and the state information it has to maintain to enforce the policy. When untrusted code is executed, the actions performed by the code are monitored. At any point of time, if the code is about to violate a security policy, then the execution is terminated. We show the connection between pure past temporal specification and GCPSL. This helps us to automatically generate monitoring algorithm for a policy specified as a temporal logic formula (with past temporal operators). We discuss a sample monitoring architecture for system calls and extend the specification language to allow formulation of policies constraining the behaviour of a set of processes. The rest of the paper is organized as follows: section 2 provides the syntax and semantics of the (GCPSL), section 3 describes a system call monitoring architecture, in section 4 we outline the extension to GCPSL to handle a set of processes, in section 5 we show how temporal constraints can be compiled into monitor specification in GCPSL, in section 6 we discuss the state of the implementation, section 7 describes related work and section 8 presents the conclusion.

2 Guarded Command Policy Specification Language

Guarded command language [4] is a well founded and structured formulation widely used for a variety of specification and applications. A guarded command is of the form $G \rightarrow S$ where G is a proposition and S is a statement. When G evaluates to true, S is executed. Guarded command language also contains conditional and repetitive structures as shown below:

if
  $G_0 \rightarrow S_0$
  $G_1 \rightarrow S_1$
  ...
  $G_n \rightarrow S_n$
fi

do
  $G_0 \rightarrow S_0$
  $G_1 \rightarrow S_1$
  ...
  $G_n \rightarrow S_n$
od

In each case, more than one guard could evaluate to true at a time. In this case, one of the statements is chosen non-deterministically. Thus, non-determinism is an intrinsic property of guarded command language. When none of the guards evaluate to true, execution is aborted.

GCPSL, like guarded command language, is a list of guarded commands. To ensure determinacy, the guards must be mutually exclusive (i.e., at most one guard can evaluate to true at a given time). However, to allow for easier specification of policies, we do not require that the guards be mutually exclusive. Whenever more than one guard evaluates to true, we execute the instructions following the first true guard. Here, we assume that the policies are written at the level of system calls, library routines or APIs i.e., a policy imposes constraints on how these calls can be used by untrusted code. State variables keep track of current state of execution of code and are updated upon witnessing relevant actions (in this case, method calls with actual arguments).

2.1 Syntax of GCPSL

A policy in the language consists of two sections; first part (which is optional) is used to declare state variables and the second part is a list of guarded commands. The syntax is shown below.

state:
    $var_type_1\ state_var_1 = initial_value_1;$
    $\vdots$
command:
    $(method_call_1(x,y,z)) \wedge (condition_1 \vee condition_2) \rightarrow statement_1; \ldots ;$
    $(method_call_2(w)) \wedge (condition_3) \rightarrow$ **terminate**;
    $\vdots$
default: skip | terminate;

State variable declaration consists of the keyword “state” followed by a series of variable declarations. The state variables have to be initialized to some value so that a unique start state can be determined. Guarded command list consists of the keyword “command” followed by at least one guarded command and the “default” action is either skip or terminate. Each guarded command consists of a guard followed by one or more statements. A guard can be a boolean combination of conditions: it could be an event denoting method call/s (with the name of the call and a list of arguments) or a comparison of expressions (involving only constants, state variables and formal arguments to the current method call). Statements can either update the state variables and allow the action or terminate the program under supervision or skip updation and allow the aciton. State variables are updated after the actual method call returns. Statements can use the return value of the call through the keyword “result”. The “default” declaration tells what should be done when none of the guards evaluate to true.

Example 1. Consider a policy which states that processes cannot write more than 80KB (total) into files. This policy could be specified as:
state:
  int *written*=0;
command:
  (`write`(*fd*, *buff*, *num_bytes*)) $\wedge$ (*written* < 80000) $\wedge$
    (*written* + *num_bytes* ≤ 80000) $\rightarrow$ *written*= *written* + **result**;
  !(`write`(*fd*, *buff*, *num_bytes*)) $\rightarrow$ **skip**;
default: **terminate**;

2.2 Semantics of GCPSL

We provide semantics using *security automata* as defined in [16]. The security automaton is a 4-tuple $\langle Q, \Sigma, \delta, Q \rangle$. The set of states Q is the set of values that the state variables can take. Suppose that the state variable declaration is as shown below:
state:
  var_type_1 $s_1 = c_1$;
  ⋮
  var_type_n $s_n = c_n$;

Let **val** be a function that takes as input a variable and returns the set of values (of appropriate type) that it can take. Then the set of states Q can be defined as:

$$Q = \mathbf{val}(s_1) \times \cdots \times \mathbf{val}(s_n).$$

The initial state is the set of initial values that are assigned to state variables (i.e., $\langle c_1, \cdots, c_n \rangle$). The alphabet Σ is the set of monitored method call events. It is defined as:

$$\Sigma = \{\text{method_call}(v_1, \cdots, v_n) \mid \text{method_call}(x_1, \cdots, x_n) \text{ is a method call with formal arguments } x_1, \cdots, x_n \text{ and } \forall 1 \leq i \leq n.\ v_i \in \mathbf{val}(x_i)\}$$

We now define the transition function. Consider the command section of a policy as defined below:
command:
  $guard_1 \rightarrow statement_1$
  ⋮
  $guard_m \rightarrow statement_m$

Let G be a function that evaluates each guard on current state and current input symbol and returns the index of the first guard that evaluates to true (if none of the guards evaluates to true, it returns 0). G is defined as:

$$\mathrm{G}\colon Q \times \Sigma \rightarrow \{0, \cdots, m\}$$

Let F_i be the partial function associated with the statement part of guarded command i. This function captures the way in which statement i updates the state. We use function G to select appropriate function F_i on a given $\langle$input state, symbol$\rangle$ pair. We then define the transition function in terms of F_i. Formally,

$$\forall 1 \leq i \leq m.\ \mathrm{F}_i : Q \times \Sigma \rightarrow Q \text{ and}$$

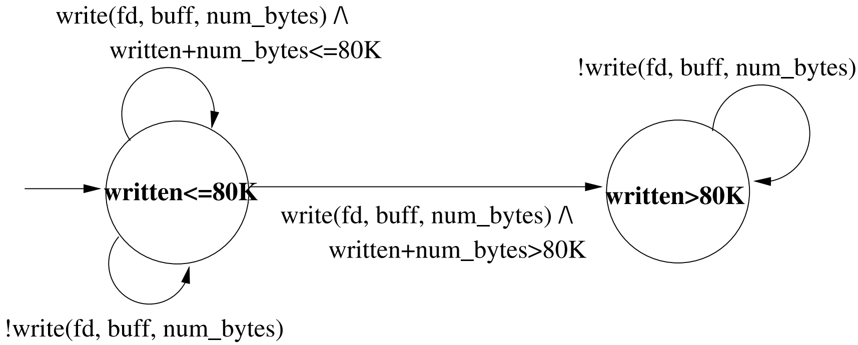

Fig. 1. Automaton for example 1

$\forall q \in Q$ and $\forall a \in \Sigma,\ \mathrm{F}_0(q, a) = q$
(if specification has "**default**: skip") and
$\forall q \in Q$ and $\forall a \in \Sigma,\ \mathrm{F}_0(q, a)$ is not defined
(if specification has "**default**: terminate")

Thus, $\forall q \in Q$ and $\forall a \in \Sigma$, $\delta(q, a) = \mathrm{F}_{G(q,a)}(q, a)$.

Figure 1 shows the security automaton (reduced to 2 states) for policy shown in example 1.

3 A Sample Monitoring Architecture: Monitoring System Calls Without Program Instrumentation

System calls constitute an important programming aid. They abstract away low level hardware details from the programmer. All access to underlying hardware is made through these system calls (which are provided by the operating system through libraries). Thus, system calls form an important boundary at which we can monitor program actions.

Monitoring system calls at run-time can be done without modifying untrusted code. System calls are made by generating an interrupt (e.g., `int 80x` on Linux/i86). This transfers the control to kernel mode. The service for this interrupt uses a system call table (set by the operating system) that is looked up (using the value in register eax as index) and control is transferred to code for the appropriate system call in kernel space. The process is shown in fig. 2.

Since all the necessary information is already maintained by the operating system, we only have to intercept system calls inside the kernel and check whether they should be allowed. To make this decision, the kernel module consults a user space monitoring module. The user space module allows user to specify policies, compiles them and then takes a decision based on information provided by the kernel module (figure 3).

For example, consider a policy that no file in "/bin" can be written into. To enforce this policy, we write the following specification:

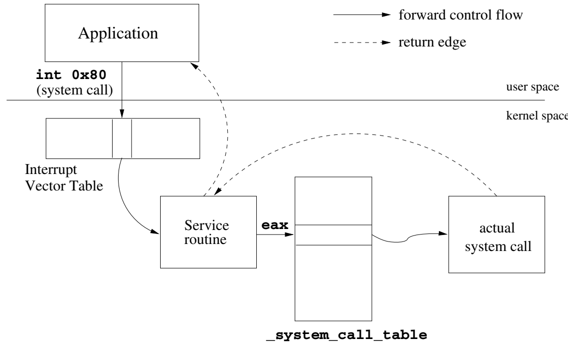

Fig. 2. System call architecture in Linux

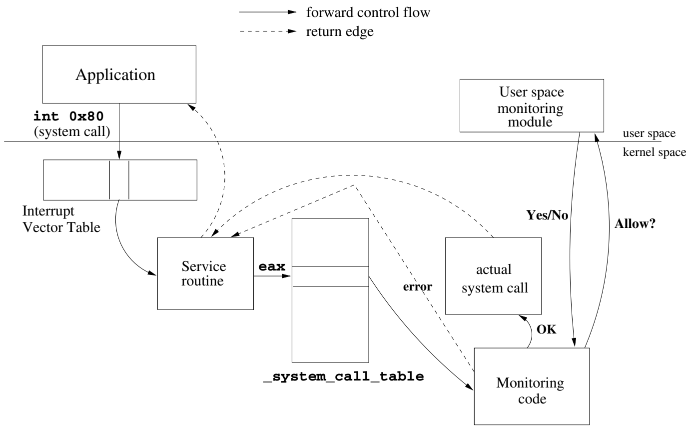

Fig. 3. System call monitoring architecture in Linux

commands:
(`open`(*file_name*, *access_mode*)) ∧ (`file_path`(*file_name*)==“/bin”) ∧
((*access_mode*==O_WRONLY) ∨ (*access_mode*==O_RDWR))
→**terminate**;
default: skip;

The “`file_path`()” function takes a file name and returns the absolute path of the file. This function is not a system call or a library routine but such functions can be used to make specification easier. Based on the target platform, the run-time enforcement mechanism implements the necessary functionality. The policy can be easily compiled into user space module of the monitor. The kernel module signals the user space module when relevant system calls are made (along with state information and arguments). Shown below is a part of C code (for user space component) compiled from the GCPSL policy shown above:

```
action = get_event();
if ( (strcmp(action.call_id,"open") &&
(file_path(aciton.arg_1,"/bin")) && ((action.arg_2==O_WRONLY) ||
(action.arg_2==O_RDONLY)) )
    signal(deny);
 else
    signal(allow);
```

The "action" structure contains information about the system call (name, arguments, etc.) passed by the kernel module (shown by edge labeled "Allow?" in fig. 3). Whether the system call should be allowed or not is decided by the code fragment shown above and the reply is sent back to the kernel module via signal() function (shown by edge labeled "Yes/No" in fig. 3).

In the monitoring architecture, kernel mode provides good insulation to the in-kernel module that intercepts the system calls. The state variables could also be maintained inside the kernel (and hence be safe from modification by user space code). Since GCPSL captures the working of the monitor, it is easily compiled into the user space module that takes decisions regarding actions. The monitoring framework does not require major changes in the kernel code or a reboot when the user policy changes. The drawback of the architecture is that the decision is taken after execution enters kernel mode and the switch is very time consuming. Care should be taken to ensure safety of the user space module from interference by malicious code.

4 Enforcing Constraints on a Set of Processes

The monitoring architecture described in the previous section enforces run-time constraints (specified in GCPSL) on individual processes. However, there are certain cases where restrictions have to be imposed collectively on a set of processes. Here, we provide extension to GCPSL so that specification of such collective constraints becomes easy.

Consider a mobile phone game that can have multiple instances running on the platform. The user interacts with other online players through messages. However, since messages cannot be sent free of cost, user wants to limit the number to MAX_MESSAGES (some pre-defined constant). Once the limit is reached, user should be asked each time before sending a message. To handle such policies, we equip the GCPSL with extra constructs that allow us to handle a set of processes. We partition the state variables into two sets namely, "global" and "local". For the example mentioned above, we could write a policy as follows:

state:
  global:
    int *num_msg*=0;
  local:
    int *asked*=0;

command:
(`send_message`(*remote_host*)) ∧ (*num_msg*<MAX_MESSAGES) → *num_msg*++;
(`ask_user`(*msg*)) ∧ (*num_msg*==MAX_MESSAGES) → *asked*=1;
(`send_message`(*remote_host*)) ∧ (*num_msg*==MAX_MESSAGES)
∧(asked==1) → *asked*=0;
!(send_message(remote_host))→ **skip**;
default: terminate;

Thus, GCPSL can be easily adapted to enable a user to specify policies over multiple processes. It is important to note that global state variables should be locked before access to ensure that concurrent execution does not corrupt them. Also, in some cases, like the example mentioned above, the global state of the monitor may have to be persistent. In the example above, if 2 sessions of the game use up the MAX_MESSAGES limit set by the monitor, then any session that may be invoked later should always have to ask before sending a message. Thus, for example, a monthly limit on the number of messages that can be sent without permission can be set. When the month is over, the global state of the monitor has to be reset again. We do not include this in specification. The local state variables pertaining to individual game sessions can be created when the process starts and can be removed when the process terminates. Thus, the extension to the GCPSL can be easily handled by the system call monitoring architecture described in the previous section with minor changes to the kernel module.

4.1 Semantics

Let us assume that we are given a policy to enforce on a set of processes $P_1, P_2, \cdots, P_p$. Also, each process has its own copy of local state variables (we tag local variables with the process name).

Suppose that the policy has state variable declaration as:
state:
global:
var_type_1 $g_1 = c_1$;
⋮
var_type_n $g_n = c_n$;
local:
var_type_1 $l_1 = c'_1$;
⋮
var_type_n $l_{n'} = c'_{n'}$;

Then, the set of states is given by
$$Q = \mathbf{val}(g_1) \times \cdots \times \mathbf{val}(g_n) \times \{\mathbf{val}(l_1) \times \cdots \times \mathbf{val}(l_{n'})\}^p.$$
Initial state is given by $\langle c_1, \cdots, c_n, \{c'_1, \cdots, c'_{n'}\}^p \rangle$.
The alphabet is given by
$$\Sigma = \{\langle P_i, \text{method_call}(v_1, \cdots, v_k)\rangle | \text{ method_call}(\mathrm{x}_1, \cdots, \mathrm{x}_\mathrm{k}) \text{ is a call with formal parameters } x_1, \cdots, x_k \text{ and for } 1 \le i \le k.\ v_i \in \mathbf{val}(x_i) \text{ and } \mathrm{P}_\mathrm{i} \text{ is the process which made the call}\}$$

Consider the command section of a policy as defined below:
command:

$guard_1 \rightarrow statement_1$

$\vdots$

$guard_m \rightarrow statement_m$

Let G be a function that evaluates each guard on current state and current input symbol and returns the index of the first guard that evaluates to true (or returns 0 when none of the guards is true). G is defined as:

$$\mathrm{G}\colon Q \times \Sigma \rightarrow \{0, \ldots, m\}$$

Let F_i be the partial function associated with the statement part of guarded command i. This function captures the way in which statement i updates the state. We use function G to select appropriate function F_i on a given ⟨input state, symbol⟩ pair. We then define the transition function in terms of F_i. Formally, $\forall 1 \leq i \leq m$. $\mathrm{F}_i : Q \times \Sigma \rightarrow Q$ (and F_0 is defined the same way as before).

The information about which process performed corresponding action is needed to update the local state variables pertaining to that process. For updation of global state variables, this information is discarded. The transition function can therefore be written as:

$$\forall q \in Q \text{ and } \forall a \in \Sigma,\ \delta(q, a) = \mathrm{F}_{G(q,a)}(q, a)$$

5 From Pure Past Temporal Logic to GCPSL

GCPSL policies describe the operation of the run-time monitor in terms of states and allowed actions. Many policies enforce temporal restrictions on the actions (e.g., library routines, system calls, etc.) of an untrusted program. These restrictions can sometimes be difficult to capture through such a detailed specification of monitoring mechanism as GCPSL. Temporal logic is better suited for specification of such policies. Temporal logic formulae are more convenient when complex policies are composed by conjunction/disjunciton of several simple policies. A composition of simple policies could lead to a huge number of guarded commands in GCPSL. A separation of concerns can therefore be achieved if users can specify such complex policies in temporal logic which could then be compiled into GCPSL (which, as we have seen in section 3, can be easily compiled into user space component).

Since the monitoring architecture presented in section 3 relies on GCPSL policies to make run-time decisions (which are compiled into the user space component of the architecture), we will see how conversion of pure past temporal logic formulae into GCPSL policies can be done so that they can be incorporated without any change in the architecture.

As we are concerned with security policies that can be monitored during execution, we focus our attention on *safety properties* which stipulate that the execution never enters a forbidden state. Every past formula is interpreted on a finite sequence of states $\sigma = s_0, \cdots, s_n$ (where S is the set of states and

each $s_i \in S$). The formula is built from a set of atomic propositions AP and logical and temporal connectives. A labelling function $l: S \rightarrow 2^{AP}$ tells which propositions are true in a particular state. The syntax for pure past temporal logic is as follows:

$$\varphi\!: \;= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, \mathcal{S}\, \varphi \mid \mathcal{Y}\varphi$$

Here, $p \in AP$ is an atomic propostion, $\mathcal{S}$ is the "since" operator and $\mathcal{Y}$ is the "yesterday" or the "previous step" operator. Other operators can be expressed in terms of these as follows:

$\mathcal{O}\varphi \equiv \mathbf{T}\, \mathcal{S}\, \varphi$ ($\mathcal{O}$ - once in the past)

$\mathcal{H}\varphi \equiv \neg\mathcal{O}\neg\varphi$ ($\mathcal{H}$ - always in the past)

Let $p \in AP$ be an atomic proposition and $\sigma = s_0, s_1, \ldots, s_n$ (each $s_i \in S$) be a finite sequence of states. Let σ_x denote the prefix sequence $s_0, s_1, \ldots, s_x$ of σ. The satisfaction relation is defined by:

$\sigma \models p$ iff $p \in l(s_n)$

$\sigma \models \neg\varphi$ iff $\neg(\sigma \models \varphi)$

$\sigma \models \varphi \vee \psi$ iff $(\sigma \models \varphi) \vee (\sigma \models \psi)$

$\sigma \models \varphi\, \mathcal{S}\, \psi$ iff $\exists\, j,\, 0 \leq j \leq n \,.\, \sigma_j \models \psi$ and $\forall\, i,\, j < i \leq n \,.\, \sigma_i \models \varphi$

$\sigma \models \mathcal{Y}\, \varphi$ iff $n > 0$ and $\sigma_{n-1} \models \varphi$

A sequence σ satisfying a formula is called the model of the formula. All safety properties expressible in temporal logic can be represented as $\mathcal{H}\varphi$ where φ is a past LTL formula [12]. The models of such a safety formula are infinite sequences of states $\sigma = s_0, s_1, \ldots$ such that

$$\forall\, i \geq 0.\, \sigma_i \models \varphi$$

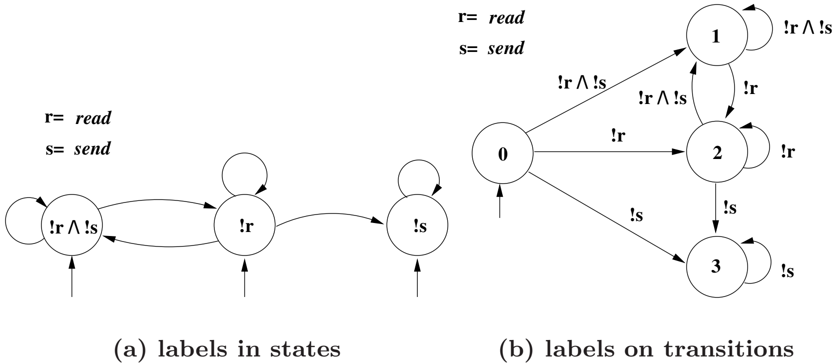

Fig. 4. Automaton for $\mathcal{H}(send \rightarrow \neg\mathcal{O}(read))$

In [17], authors present an efficient approach for converting a formula in pure past temporal logic into a finite state automaton. This paves the way for converting temporal restrictions on program actions into guarded command statements. The simple idea is to have an enumerated state variable that can range from 1 to i if there are i states $(q_1, \cdots, q_i)$ in the automaton. For each state and a transition

going out of it, we take conjunction of index of the state and the label on outgoing transition as the guard and update the state variable to the index of the state reached via the transition. We set the "**default**: terminate" to indicate a violation.

For example, consider the policy that a message cannot be sent after a file has been read. Let *send* and *read* be the propositions that correspond to `send()` and `read()` method invocations. Then, the policy can be stated in pure past LTL as $\mathcal{H}(send \rightarrow \neg\mathcal{O}(read))$. The automaton produced for this formula is shown in fig. 4(a). The same automaton with labels moved to transitions is shown in fig. 4(b). Thus, the state variable would be declared as:

state:

int q=0;

Consider state 0 in the automaton shown in figure 4(b). Since it has 3 outgoing edges, we add a guarded command for each edge as follows:

(q==0)∧(!`send()`)∧(!`read()`)→ q=1;
(q==0)∧(!`read()`)→ q=2;
(q==0)∧(!`send()`)→ q=3;

Continuing this way, we add guarded commands for each outgoing transition from each state and use the default condition to signal violation. Note that the GCPSL policy obtained this way could be longer than what one would write directly using GCPSL. But, complex policies are easer to specify in pure past temporal logic and the translation can be made automatic. The most important aspect of the conversion is mapping of the propositions in the formula to the actions in the guards. To enforce restrictions on a set of processes, we would require a larger set of atomic propositions to incorporate information about the process that performs an action. But guarded command specification can handle it better (as can be seen from the example stated in the previous section).

6 Implementation

We are working on implementation of the architecture described in section 3 with the extensions to GCPSL. We have built a parser for GCPSL policies and implemented a prototype version of the in-kernel module that intercepts the system calls. We are working on implementation of the user space module and communication between user and kernel space components. We also intend to enhance flexibility of the tool by incorporating pure past temporal logic formulae into the tool (by using the translation procedure of the previous section).

7 Related Work

In [19], Uppuluri and Sekar propose a specification language called Behavioural Monitoring Specification Language (BMSL) to specify constraints on program behaviour. They also describe how abstraction can be used to specify policies by grouping system calls into categories. This makes the task of policy specification easier and the policies portable across platforms. The authors also show how more involved policies can be obtained by refining generic policies. Policies in

BMSL can be easily translated into guarded commands. Our approach allows specification of constraints for a set of processes and can even take advantage of the abstraction mechanisms mentioned in [19]. A similar approach for specifying software wrappers for commercial off-the-shelf components was presented in [8]. The specification language proposed in [8] also uses abstraction but is much more complex and often leads to lengthier policies. In addition, the approach uses databases to store and share information among wrappers.

Substantial amount of research has been done on run-time enforcement of security policies. In [9], authors provide a theoretical classification of security policies that can be enforced by different mechanisms (e.g., static analysis, execution monitoring and program re-writing). The authors show that the class of properties that can be monitored at run-time is co-RE (i.e., a policy violation can be detected in finite amount of time but execution that conforms to policy goes on forever). In [16], Schneider presents a theoretical study of execution monitors (mechanisms that monitor program execution and terminate the program before it can violate a security policy) and shows that they can only enforce safety policies. Schneider also presents a *security automata* model of an execution monitor (and also shows that the same can be coded in guarded command language).

Many tools provide system call interception (e.g., [1] and [3]). In [15], author proposes an interactive policy generation tool for monitoring system calls. The specification language is very simple and not as expressive as the guarded command language. A policy in their framework just states whether a particular system call (with specific arguments) should be allowed. The system allows for interactive policy generation through training runs. If a particular system call is not covered by the policy, then the tool asks the user to make a decision (which can be added to the policy). The tool does not allow specification of policies that restrict the order in which system calls are made. Also, the tool cannot constrain the amount of resources used by a process as shown in example 1 in section 2. Another tool called "syscalltrack"[1] allows monitoring of system calls through a simple specification language. On observing a monitored system call (with appropriate arguments), it either allows the call (with logging) or returns an error. Both the tools mentioned above cannot specify policies for a set of processes.

In [13] different models of monitors were proposed which were more powerful than execution monitor in that they could suppress or insert program actions or could truncate execution. It was also shown that edit automata (which can truncate, suppress and insert actions) could enforce any property on execution (even if it were not a safety property). A specification language and a run-time monitoring system called Polymer [2] was also provided for monitoring of Java programs. In this framework, every policy extends an abstract policy class. One has to define security relevant actions (method invocations) and then the policy provides various suggestions (e.g., skip, insert actions, replace action, etc.) when these actions are performed. Different mechanisms for composing policies are also

[1] available at `http://syscalltrack.sourceforge.net/index.html`

provided in [2]. Instead of writing complex policies directly in Java, one could use GCPSL to specify an edit automaton and then compile it into a Polymer policy. By including actions in the instruction part of the guarded commands, one can easily specify an edit automata policy. For example, the specification can be changed as follows:

(Guard)→ terminate;
OR
(Guard)→ {action}*;
update state | skip;

Thus, whenever a guard is true, either the execution is terminated or sequence of actions is performed and is followed by a possible change of state.

In [7], authors propose a model for run-time monitoring (called *Shallow History Automata*) that tracks only the execution history and does not remember the order of events. Although less powerful than security automata, SHA can enforce some important security policies. In [5], security automata implementation of Software based Fault Isolation (SFI) was discussed and prototypes for x86 and JVML were provided in SAL (Security Automata Language). The language is just a text based representation of security automata with macro definitions in the beginning. The author also presents PSLang (Policy Specification Language) and PoET (Policy Enforcement Toolkit) for in-line monitoring of Java code. In this approach, the untrusted program is instrumented with proper checks so that the modified program does not violate security policy. In [6], author presents Naccio framework for policy enforcement through in-line monitoring. Naccio allows user to specify policies in a platform independent way by providing abstract system interface (the interface, however, is not very easy to use).

All specification languages mentioned so far are platform specific (except Naccio). GCPSL can be seen as a simple way for specifying monitoring algorithm in a platform independent way. Other approaches for run-time monitoring of Java code have been proposed in [11,10] (these approaches were mainly aimed at checking program correctness). In [20], a tool for sand-boxing of untrusted applications is presented.

8 Conclusion

We have described how GCPSL can be used for easy specification of policies. Through a simple extension, GCPSL can enforce a policy collectively on a set of processes. Validation of GCPSL policies is achieved through security automata semantics. The most important objective was to provide a simple specification language that allows a rich set of policies to be formulated and validated. A *separation of concerns* can be achieved by specifying policy in terms of pure past temporal logic and then automatically producing monitoring instructions (guarded commands). A robust system call monitoring architecture for enforcing GCPSL policies was described. GCPSL can be easily modified to incorporate other models of enforcement like suppression, insertion or edit automata (by allowing corresponding actions in the statement part of the guarded command).

References

1. Acharya, A., Raje, M.: MAPbox: using parameterized behavior classes to confine untrusted applications. In: SSYM 2000. Proceedings of the 9th conference on USENIX Security Symposium, Denver, Colorado, p. 1. USENIX Association, Berkeley, CA, USA (2000)
2. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
3. Chari, S.N., Cheng, P.-C.: Bluebox: A policy-driven, host-based intrusion detection system. ACM Trans. Inf. Syst. Secur. 6(2), 173–200 (2003)
4. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
5. Erlingsson, U.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Department of Computer Science, Cornell University (2003)
6. Evans, D.: Policy-Directed Code Safety. PhD thesis, Dept. of Electrical Engg. amd Computer Science, Massachusetts Institute of Technology (February 2000)
7. Fong, P.W.L.: Access control by tracking shallow execution history. In: Proceedings, IEEE Symposium on Security and Privacy, 2004, pp. 43–55 (May 2004)
8. Fraser, T., Badger, L., Feldman, M.: Hardening COTS software with generic software wrappers. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999, pp. 2–16 (1999)
9. Hamlen, K., Morrisett, G., Schneider, F.: Computability classes for enforcement mechanisms. Technical Report 2003-1908, Department of Computer Science, Cornell University (2003)
10. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. International Journal on Software Tools for Technology Transfer 6(2), 158–173 (2004)
11. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: a run-time assurance tool for java. In: 1st International Workshop on Run-time Verification, vol. 55 (2001)
12. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Parikh, R. (ed.) Logics of Programs. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
13. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4(5), 2–16 (2005)
14. McGraw, G., Morrisett, G.: Attacking malicous code: a report to the infosec research council. Software, IEEE 17(5), 33–41 (2000)
15. Provos, N.: Improving host security with system call policies. In: SSYM 2003. Proceedings of the 12th conference on USENIX Security Symposium, Washington, DC, p. 18. USENIX Association, Berkeley, CA, USA (2003)
16. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
17. Shah, H., Shyamasundar, R.K.: Efficient automata generation for pure past LTL. Technical report, School of Technology and Computer Science, TIFR (2007)
18. Thomson, K.: Reflections on trusting trust. Communication of the ACM 27(8), 761–763 (1984)
19. Uppuluri, P., Sekar, R.: Experiences with specification-based intrusion detection. In: RAID 2000. Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection, pp. 172–189. Springer, Heidelberg (2001)
20. Wagner, D.: Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, University of California, Berkeley (1999)