UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CAROLINA PARREIRA LORINI

A Framework for Usage Control Policy Enforcement

Trabalho de Graduação.

Prof. Dr. Alexander Pretschner, T. U. Kaiserslautern
Orientador

Ricardo Neisse, Fraunhofer Institute IESE
Co-orientador

Prof. Dra. Taisy Silva Webber
Supervisora

Porto Alegre, julho de 2010

TABLE OF CONTENTS

LIST OF FIGURES

RESUMO

Este trabalho de conclusão foi iniciado sobre a orientação do professor Alexander Pretschner da Universidade Técnica de Kaiserslautern, em 2009 e também com orientação do Ricardo Neisse, do Instituto Fraunhofer, e mais tarde com a supervisão da professora Taisy Weber. Este trabalho trata da aplicação de mecanismos de controle de uso de dados, com ênfase na aplicação no nível de sistemas operacionais.

No mundo digital atual, a segurança de dados tornou-se uma grande preocupação. Para lidar com esta preocupação foram desenvolvidos vários métodos, entre eles o controle de uso dos dados. Controle de uso é uma extensão do controle de acesso que determina não apenas quem pode acessar os dados, mas também o que pode ou deve ser feito com esses dados e pode ser aplicado nas mais diversas áreas, incluindo, mas não limitado a, gerenciamento de direitos digitais (DRM).

Controle de uso funciona através da aplicação de políticas a dados, as quais determinam obrigações do usuário em relação a esses dados. Por exemplo, é possível especificar que um usuário que baixa um determinado filme só poderá tocá-lo um número máximo de vezes e não é permitido a copiar ou re-distribuir o filme através de meios de armazenamento portáteis.

Para que as políticas de controle de uso sejam aplicadas corretamente, um tipo de mecanismo de execução deve ser implementado. Estes mecanismos podem ser divididos em duas categorias: mecanismos de observação, que relatam violações de políticas para o proprietário dos dados e que gera ações penalizantes, e mecanismos de controle, que impedem que as políticas sejam violadas através de restrições as ações do usuário.

Os mecanismos de controle de uso são compostos por três componentes principais: o monitor de obrigações, responsável por detectar violações de políticas, um componente de sinalização, que gera eventos com base em ações do usuário que envolvem dados controlados e que alimentam o monitor de obrigações, e uma componente de execução, que executa ações que penalizam o usuário ou que impede violações, de acordo com a categoria do mecanismo.

Controle de uso pode ser aplicado em diferentes níveis de abstração, como no kernel do sistema operacional, no nível de máquinas virtuais (VMWare, por exemplo), sistemas runtime (JavaVM), e na camada de aplicações. Em níveis de abstração mais altos as ações observadas são mais específicas e em níveis mais baixos são mais genéricos e com poucas distinções. Para ser mais eficiente, uma política deve ser implementada no nível que mais se adapte às características dos dados e da prórpia política. Uma política que especifica restrições com relação à reprodução de um filme vai ser melhor aplicada na camada de aplicação do que ao nível do sistema operacional, por outro lado, uma política que afirma que um arquivo não deve ser aberto, exceto em circunstâncias especiais, será mais fácil aplicar no nível do sistema operacional através do controle de chamadas do sistema do tipo "open". Controle do uso também pode ser aplicada em mais de um nível ao mesmo tempo.

O foco desta tese é sobre a aplicação de políticas de controle de uso no nível do sistema operacional utilizando a Obligation Specification Language (OSL). Políticas de controle de uso em OSL representam ações do usuário por meio de eventos, e podem ser aplicadas usando tanto mecanismos de observação quanto mecanismos de controle. Um mecanismo de observação verifica se uma seqüência de eventos está em conformidade com uma fórmula OSL e gera notificações quando não for compatível, enquanto um mecanismo de controle não apenas observa um evento, mas pode impedir, modificar, substituir ou atrasar o eventos que não é compatível com a política.

Nesta tese, será apresentado um parser OSL e monitor de obrigações genéricos, além de um monitor que permita a utilização de mecanismos de controle. Adicionalmente, será apresentado um arquitetura genérica para implementação de controle de uso, e a instanciação desta arquitetura para dois sistemas operacionais reais: OpenBSD e Windows. Finalmente, será avaliado o impacto da aplicação de controle de uso sobre o desempenho do sistema operacional.

Palavras-Chave: Controle de Uso, Systrace, Detours, OSL.

ABSTRACT

Usage control is an extension of access control that determines not only who may access data but also what can or must be done with this data. Usage control works by applying OSL policies to data, which state the user's obligations in relation to that data. In this work, we implemented an OSL parser and obligations monitor then extended this monitor to support enforcement by control using usage control mechanisms. We also present the architecture of a generic usage control enforcement framework with remote policy deployment. This framework was then instantiated in two real-world systems, OpenBSD and Windows, using Systrace and Microsoft Detours respectively. The security and performance of the implementations were evaluated and analysed.

Keywords: Usage control, Systrace, Detours, OSL.

1. Introduction

In an ever growing digital world, data security has become a major concern. Many methods have been developed to deal with this concern and to ensure control over data; usage control is one of these methods. Usage control is an extension of access control that determines not only who may access the data but also what can or must be done with this data and has many applications including, but not limited to, digital rights management.

Usage Control works by applying policies to data, which state the user's obligations in relation to that data. For example, it is possible to specify that a user that downloads a certain movie may only play it for a maximum number of times and is not allowed to copy or re-distribute this movie through portable storage media.

For usage control policies to work as intended, a type of enforcement mechanism must be applied. These mechanisms can be divided into two categories: observation mechanisms, which report policy violations to the data owner and might take penalizing actions; and control mechanisms, which prevent policies from being violated by restricting certain user actions.

Enforcement mechanisms are composed of three main components: and obligations monitor, responsible for detecting policy violations; a signalling component, that generates events based on user actions that involve controlled data and which are fed into the obligations monitor; and an enforcement component, which executes actions that penalize the user or prevents violations, according to the mechanism category.

Usage control can be enforced at different levels of abstraction such as the operating system kernel, system virtual machines (e.g. VMWare), runtime systems (e.g. JavaVM), and the application layer. At higher levels of abstraction the actions observed are more specific, and at lower levels they are more generic and with fewer distinctions. In order to be most effective, a policy should be enforced at the level that most suits the characteristics of the data and of the actual policy. A policy that specifies restrictions with respect to playing a movie will be better enforced at the application layer than at the operating system level; on the other hand, a policy that states that a file must not be opened except under special circumstances will be easier to enforce at the operating system level by controlling 'open' system calls. Usage control may also be enforced at more than one level simultaneously. The focus of this thesis is on the enforcement of usage control policies at the operating system level using the Obligation Specification Language (OSL).

Usage control policies in OSL represent user actions by means of events, and can be enforced using both observation and control mechanisms. An observation mechanism verifies if a sequence of events is in conformance with an OSL formula and generates notifications when it is not compliant, whereas a control mechanism not only observes an event, but might block, modify, replace, or delay the events that are not compliant with a policy.

In this thesis we present, in Section 2, a generic OSL parser and obligation monitor (which can be used for enforcement by observation), and subsequently we extend this monitor in Section 3 to support enforcement by control using mechanisms as shown in (Pretschner, 2008). Section 4 presents the design of a generic usage control policy enforcement architecture that allows remote deployment and management of OSL policies and data resources, and that focuses on the operating system level. Sections 5 and 6 deal with the implementation of usage control enforcement frameworks in two real-world operating systems: OpenBSD and Windows respectively. Finally, Section 7 evaluates the impact of usage control enforcement on the operating system's performance, as well as comparing the two implementations.

2. OBLIGATION MONITOR

The observation monitor implements a method of usage control enforcement where a monitor receives events that have already happened and checks if these events violate any usage control policies, in which case the violation is reported, allowing the data owner to take compensating measures. Figure 2.1 illustrates this method.

The obligation monitor receives events from an event signaller, which is responsible for intercepting user actions and generating events once these actions have occurred, and also receives a set of usage control policies to be monitored. At every event received, the monitor goes through the list of policies and verifies if they were violated. The observation monitor, which will be referred to from here on as the policy monitor, is an instantiation of an obligation monitor where the reporting of a violation means, for instance, logging this in a file. The monitor does not stop the action that generates the violation from happening.

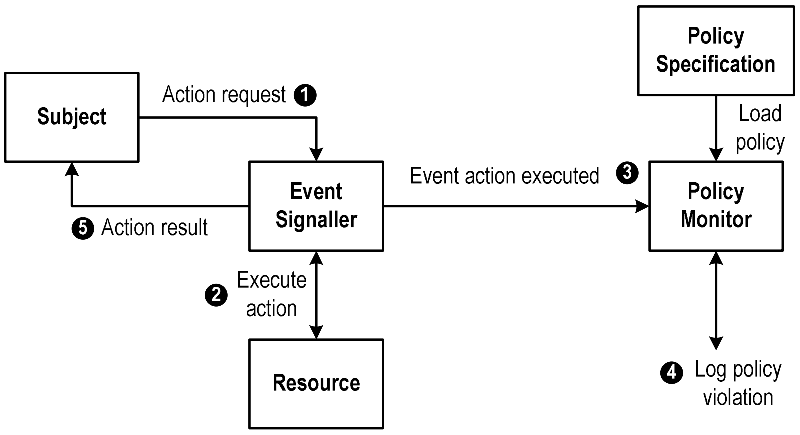


Figure 2.1: Enforcement by observation.

An important characteristic of a policy monitor implementation is related to when a policy violation is detected. If a violation is detected as soon as possible or immediately after a new event is received, the monitor is said to be synchronous. Conversely, if the violation is detected at a later time, the monitor is said to be asynchronous. For enforcement by observation, either option is valid. However, if the policy monitor will be used by control mechanisms, it must be synchronous, as these mechanisms require an immediate response from the monitor.

2.1 Policy Specification

The usage control policies that are inputted to the monitor are specified using the obligation specification language (OSL). More specifically, they are expressed using past OSL, as the policy monitor will be used by control mechanisms, which can only make decisions based on what is already known (and therefore is in the past).

OSL is an LTL based language, defined so as to be able to easily express usage control policies such as “delete file A after 30 days” and “file B must not be printed”. The compact OSL syntax is shown below. For more information on OSL see (Hilty, 2007)

$$\varphi^{-} ::= \underline{\mathit{true}}^{-} \mid \underline{\mathit{false}}^{-} \mid \mathit{Efst(Event)} \mid \mathit{Eall(Event)} \mid \underline{\mathit{not}}^{-} (\varphi^{-}) \mid \underline{\mathit{and}}^{-} (\varphi^{-} \times \varphi^{-}) \mid \underline{\mathit{or}}^{-} (\varphi^{-} \times \varphi^{-}) \mid \underline{\mathit{implies}}^{-} (\varphi^{-} \times \varphi^{-}) \mid \underline{\mathit{always}}^{-} (\varphi^{-}) \mid \underline{\mathit{before}}^{-} (N \times \varphi^{-}) \mid \underline{\mathit{within}}^{-} (N \times \varphi^{-}) \mid \underline{\mathit{during}}^{-} (N \times \varphi^{-}) \mid \underline{\mathit{repmax}}^{-} (N \times \varphi^{-}) \mid \underline{\mathit{replim}}^{-} (N \times L \times U \times \varphi^{-}) \mid \underline{\mathit{repuntil}}^{-} (N \times \varphi^{-} \times \varphi^{-})$$

For example, the policy “file /etc/passdw should never be opened” would be expressed in OSL as: *always(not Eall(open{(file, “/etc/passwd”)}))*

The formal semantics of past OSL are defined in (Pretschner, 2009). The operators can be sorted into three categories: propositional operators, which are the same from propositional logic; cardinality operators, which state how many times something should happen; and temporal operators, which determine when something should happen. Informally, the meanings of the OSL operators are:

- always (X): X must be true at every time step;
- before (N,X): X must have been true N time steps previously;
- within (N,X): X must have been true at least once in the previous N time steps;
- during (N,X): X must have always been true during the last N time steps;
- repmax (N,X): X must be true at most N times;
- repuntil (N,X,Y): X must be true at most N times, until Y is true;
- replim (N,L,U,X): X must have been true at least L times and at most U times during the last N time steps.

The policy monitor implements the stated OSL syntax with only one restriction: only propositional formulas are allowed inside the cardinality operators (repmax, replim, repuntil). Internally, the monitor uses an implicit mixed OSL formula (past and future); semantically this means that every policy is automatically encapsulated inside a future always statement, meaning that every formula will always be evaluated at every time step. Although this limits somewhat the expression of policies, this was an implementation choice as it greatly simplifies the monitor algorithm.

Additionally, each policy has an activation time, which corresponds to the moment the policy started being checked for violations in the monitor.

2.2 Obligation Monitor Algorithm

An aspect that must be considered when implementing an OSL monitor is how to adapt OSL’s discrete time steps to continuous system time. This can be done in two ways: by defining time windows in which all events that happen in that period are considered as being in the same time step (i.e., the events are said to have happened simultaneously), as illustrated by figure 2.2; or by assuming that every new event received is in a new time step (i.e., no two events occur in the same time step), as illustrated by figure 2.3.

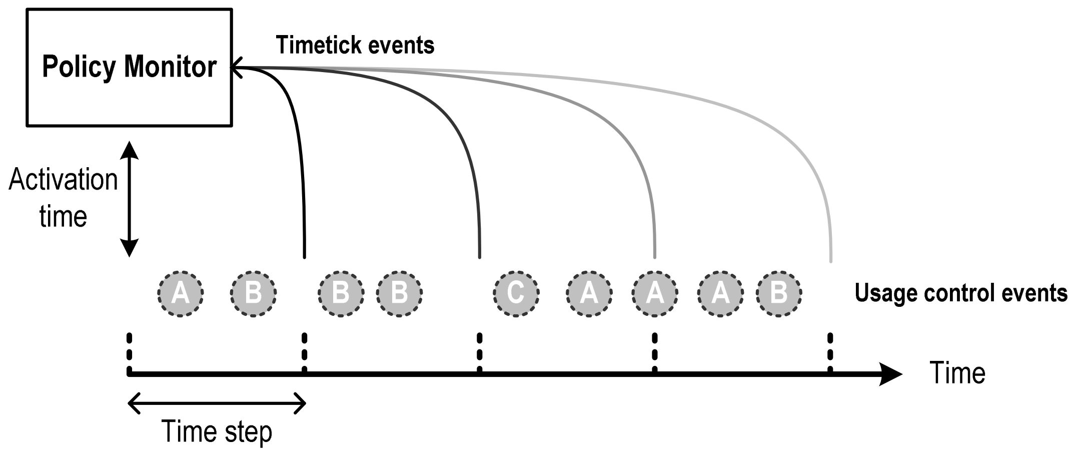


Figure 2.2: Time windows approach.

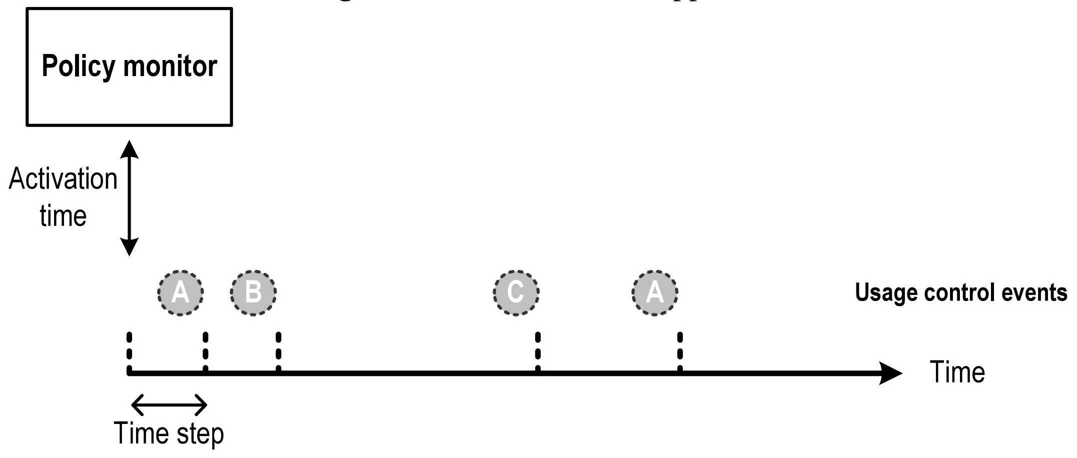


Figure 2.3: Individual time steps approach.

2.2.1 Asynchronous OSL Monitor

The implementation of time windows to emulate OSL's time steps results in an asynchronous monitor, as the detection of a violation does not occur immediately after receiving an event, but must wait until the end of the ongoing time window since only then the policies are evaluated; therefore a violation may not be detected as soon as possible. This method was implemented by creating an event queue which stores all the received events until a timer signals the end of a time window; all policies are then updated with this set of events and the queue is emptied. If an event happens more than once in the same time window, all occurrences of that event in the window are counted so as to accurately update the counters of the cardinality operators. If no events happen in a time window, the policies are still updated with the empty set, as some policies (more specifically, the one with temporal operators) may be violated if no events occur (the lack of an event may violate a policy). For example, the policy "delete after 30 days" will be violated if a delete event does *not* happen.

The advantages of this implementation are the possibility of having simultaneous events, meaning that we can have policies that state that event A and event B must happen at the same time. Also, the policies are updated less frequently and at regular intervals, improving the monitor performance. However, this monitor cannot be used together with control mechanisms, for reasons previously stated.

2.2.2 Synchronous OSL Monitor

Since the control monitor needs to incorporate an OSL monitor and the asynchronous OSL monitor described above is not adequate for this purpose, a second, synchronous, OSL monitor was implemented. This monitor treats every event as having happened in a new time step, as previously described. When an event is received, this monitor goes through the list of loaded policies and updates each one with the event, immediately reporting any policy violations. It is possible that no events are received for a relatively long period of time and it is necessary to regularly check the policies because of the temporal operators; to solve this problem the monitor has a thread that periodically generates a *null* event with which to update the policies.

There are two problems with this approach: the first is that it no longer makes sense for temporal operators to use time *steps* as a time unit, for the time steps now have variable length. To solve this problem, we adapted the operators to use standard time units such as seconds, minutes, hours or days, therefore the expression 'before(3,A)' that read "A must have been true 3 time steps before" now reads "A must have been true 3 seconds before". The second problem is that a policy that states '$A \wedge B$', where A and B are events, will never be true as there are no simultaneous events in this implementation (if either A or B is not an event but another formulae, the expression can be true), however this is acceptable for our monitors purpose.

2.2.3 OSL Formula Evaluation

The evaluation of the OSL policies is the same for both the synchronous and the asynchronous monitors and is based on the algorithm described in (Havelund, 2004). The monitor receives as an input a list of usage control policies expressed in OSL, each associated with a unique identifier defined by the policy administrator. In the initialization phase of the algorithm, each policy is parsed so as to verify that it is in the accepted syntax and, simultaneously, to initialize and fill the data structures that will be later used to evaluate the policy. The data structure is a table containing all the subformulae of the original OSL formula in post order. Additionally, each entry of this table is associated with an auxiliary data structure that stores the state information of that particular subformula. The monitor also stores the activation time (time of deployment) of each policy.

The actual evaluation of the OSL formula occurs every time a new event is received, in the case of the synchronous monitor, or when a time window ends, in the case of the asynchronous monitor, and consists of every subformula being evaluated according to its operator. The last subformula to be evaluated is the original OSL formula, so its state (true or false) is the policy state, meaning that if this state is false the formula was violated and if it is true the policy is correct. Each operator is evaluated as follows:

- True/False: obviously evaluated to true and false, respectively;
- Eall (event)/Efst (event): every received event has a type which determines whether it is an Eall or Efst event and the rest is a simple matching of the name and parameters of the received event with the event specified in the formula;
- Not/And/Or/Implies: these are the standard propositional logic operators and are evaluated as such;
- Always(X): a flag in the state structure indicates whether X was ever false or if X is currently false; if yes the formula is false, otherwise it is true;

- Before(N,X): a buffer in the state structure contains all the times X was true since the last N time units, and if at N time units previously X was true, the formula is true, otherwise it is false. The buffer is necessary so as to be able to evaluate the before operator at every moment. In the synchronous monitor, as it is very hard for an event to happen exactly 30 seconds previously (unless the time unit used has the same precision as a system clock tick), we define a window of time in which this event should be true, for example, between 29 and 31 seconds previously;
- Within(N,X): an auxiliary variable present in the state structure holds the last time X was true, if X was true sometime in the last N time units, the formula is true, otherwise it is false;
- During(N,X): an auxiliary variable holds the last time X was false, if X was false sometime in the last N time units, the formula is false, otherwise it is true;
- Repmax(N,X): the state structure includes a counter which is used for cardinality operators; in this case, the counter is increased every time X is true and the state of the formula is true if the counter is less than or equal to N, otherwise it is false;
- Repuntil(N,X,Y): the counter is increased every time X is true until Y is true, the state of the formula is true if the counter is less than or equal to N, otherwise it is false;
- Replim(N,L,U,X): also keeps a buffer with all the times X happened in the last N time units and counts the number of instances of X in the buffer, if the counter has a value in between or equal to L and U the formula is true, otherwise it is false.

When the synchronous monitor is used together with control mechanisms, it is necessary to be able to tell if an event A should come to happen, whether it would violate a policy or not. In this case, as the event has not actually happened, the internal state of the policy (such as the counters) should not be updated. Therefore, the monitor maintains two copies of the state structures, so that this verification can be done without affecting subsequent events.

2.3 Monitor API

All interaction with the policy monitor is done through the monitor API. The API for the synchronous and the asynchronous monitors are almost identical, therefore only the synchronous monitor API is listed below.

The monitor represents events using an event_t structure, which contains the event name, type, number and list of parameters; the parameters are represented using a param_t structure, which is a pair that consists of a parameter name and the actual value of the parameter. This representation was chosen so as to be easy to use and understand, and to be as faithful as possible to the OSL event format.

```
/* Structure used to describe an event */
typedef struct event_s{
  char *event_name;
  int type;
  int num_params;
  param_t *params;
} event_t;

/* Structure used to describe an event parameter */
typedef struct param_s {
  char *param_name;
  char *param_value;
} param_t;
```

Listing 2.1: Event structures.

The most important functions in the API are update_all_past_monitors() and update_past_monitor(). The first function is common to both implementations of the policy monitor and is responsible for updating all active policies loaded in the monitor with the event that it receives as a parameter. The second function is only available in the synchronous monitor as it involves an immediate return value; this function updates only one monitor at a time (whose identification is passed as a parameter) with the specified event (also passed as a parameter) and returns the updated state of the policy. Additionally, this function can receive as a parameter a No Update Flag, which means that the new policy state will not save the new internal policy state; this can be used to test if an event would violate a policy should it occur.

Besides the functions for policy creation and updates, the monitor API also implements functions for policy deployment management. These functions can set a policy as active or inactive, where inactive means the policy is loaded into the monitor and its state is maintained but this state will not change until the policy is activated. Another function resets the internal policy state to its initial configuration. Functions such as these are especially useful for the policy administrator as a policy can be reset after a violation without having to be reloaded (resetting a policy is, computationally, less cost-worthy) and are also useful to control mechanisms.

```
/* Initializes the monitor manager
 * @RETURN  0               sucess
 * @RETURN  -1              error: unable to initialize library
 */
int init_past_monitor();

/* Creates a new past monitor
 * @PARAM   id              unique monitor identifier
 * @PARAM   policy          osl policy to be monitored
 * @RETURN  0               sucess
 * @RETURN  -1              error: unable to create monitor
 * @RETURN  1               error: identifier already in use
 */
int add_past_monitor(char *id, char *policy);

/* Deletes the specified monitor, if it exists
 * @PARAM   id              unique monitor identifier
 */
void delete_past_monitor(char *id);

/* Resets the state of the monitor policy and the policy activation time
 * @PARAM   id              unique monitor identifier
 */
void reset_past_monitor(char *id);

/* Changes monitor state to active.
 *  @PARAM  id              unique monitor identifier
 */
void activate_past_monitor(char *id);

/* Changes monitor state to inactive
 *  @PARAM  id              unique monitor identifier
 */
void disactivate_past_monitor(char *id);

/* Checks if a monitor exists for the data container
 * @PARAM   id              unique monitor identifier
 * @RETURN  ACTIVE          monitor exists and monitor's state is active
 * @RETURN  INACTIVE        monitor exists and monitor's state is inactive
 * @RETURN  -1              monitor does not exist
*/
int lookup_past_monitor(char *id);

/* Updates the chosen monitor with a new event. The update flag determines if
 * the policy state will be updated or not; if the flag is DO_NOT_UPDATE, the
 * function checks if the new event turns the policy true or false but does
 * not change policy state; if the flag is UPDATE, the policy state will change.
 * @PARAM   id              unique monitor identifier
 * @PARAM   new_event       structure that contains the event information
 * @PARAM   update_flag     UPDATE or DO_NOT_UPDATE
 * @RETURN  1               if the updated policy is true
 * @RETURN  0               if the updated policy is false
 * @RETURN -1               error: monitor does not exist
 */
int update_past_monitor(char *id, event_t* new_event, int update_flag);

/* Updates all monitors with the new event
 * @PARAM   new_event       structure that contains the event information
 */
void update_all_past_monitors(event_t* new_event);
```

Listing 2.2: Policy Monitor API.

3. CONTROL MONITOR

The control monitor is an implementation of preventive enforcement, as it prevents actions that would violate usage control policies from actually happening. This is done by sending an action execution request to the monitor before every action occurs in the system; the monitor then evaluates the impact of this action on all the relevant policies and determines if the action should be allowed or not and if any additional actions should be executed before generating an action execution response.

There are four categories of control mechanism: inhibit, which simply cancels a requested action; modifier, which changes the parameters of the requested action so that it no longer causes a violation; executer, which executes a number of actions before the requested action takes place; and delayer, which delays the occurrence of an action. A single mechanism can implement more than one of these characteristics. For more information on control mechanisms see [REF NUM].

The behaviour of the control monitor is determined by a set of loaded control mechanisms which consider actions as events and which belong to at least one of the categories mentioned above. Each mechanism consists of a condition and trigger event, which defines when the mechanism should be applied, as well as a list of actions to be executed should the mechanism be triggered and the effect that the mechanism has on the requested event, whether it should be inhibited, allowed, modified or delayed. A mechanism is triggered every time the requested event is the same as the trigger event and the mechanism condition (updated with the requested event) evaluates to false. One or more mechanisms should be sufficient to enforce a usage control policy.

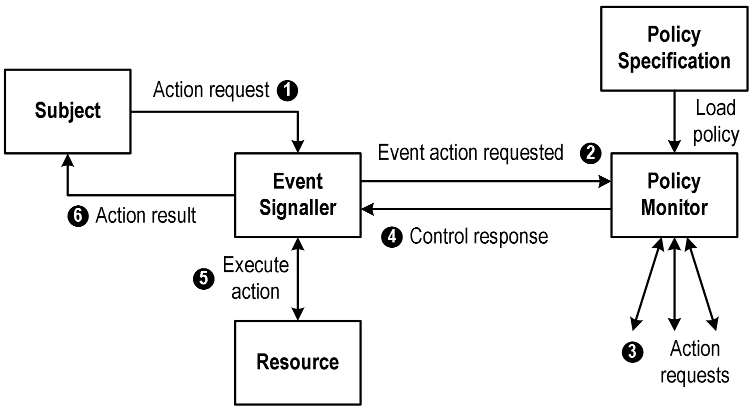


Figure 3.1: Enforcement by control.

3.1 Mechanism Specification

Every mechanism must have a unique name, which will be used to identify the mechanism in the monitor, a condition expressed in past OSL, which is evaluated using the synchronous past OSL monitor (described in section 2.2.2) and determines when the mechanism is triggered, and must choose to allow or inhibit the requested event. Additionally, a mechanism can have a trigger event (e.g., the event ‘play’ is the trigger for the condition “repmax(3,play)”), which informs the monitor when to check the mechanism condition (in other words, every time the requested event is the same as the trigger event), however a trigger is not always necessary (such as in the policy “delete after 30 days”) and therefore can be omitted or specified as *null*. The mechanism may also contain a list of actions to be executed, which are shown in more detail below. All these characteristics can be specified using the XML schema shown in figure 3.2 and detailed below. The use of the XML parser is recommended for interfacing with the monitor, but direct interfacing is also possible using the control monitor API.

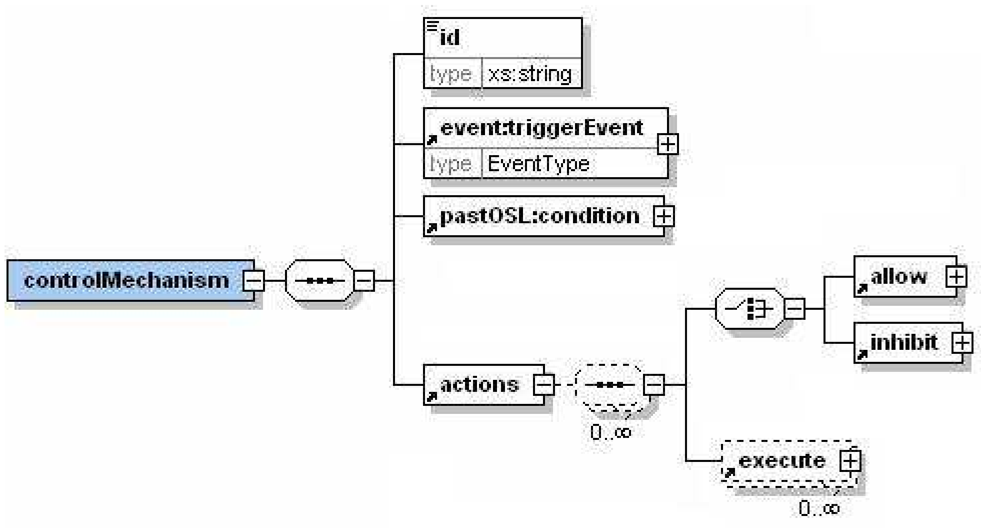


Figure 3.2: General XML mechanism schema.

3.1.1 Mechanism Responses

A control mechanism must always emit an answer to the event signaller, this answer will determine if the requested event will become an actual event. The mechanism response must be one of the two possible values: allow or inhibit. If the chosen response is inhibit, the event will not occur, but the mechanism may still execute a list of actions. If, however, the answer is allow, a few other parameters must be specified. In most cases, it does not make sense to allow an event without also delaying or modifying it; therefore, in the parser, it is mandatory to either delay or modify or do both to an allowed event. To delay an event, the duration and time unit of the required delay must be specified. Delaying an event is useful for policies such as “if the user did not pay a fee, the user must wait 1 minute for movie to play after loading”. It is important to point out that the enforcement of the delay does not occur inside the monitor, as it is a single-threaded application and the executing of a delay would temporarily block all other evaluations.

To modify the requested event, the name of the parameter to be modified and its new value must be specified. It is usually advisable to modify an event so that it becomes harmless instead of inhibiting it, this is because many programmers do not check the return value of functions or system calls and blocking an action may result in a program crash. For example, instead of inhibiting the system call “open file A” we can modify it to “open file /dev/null”. The response schema is detailed below.

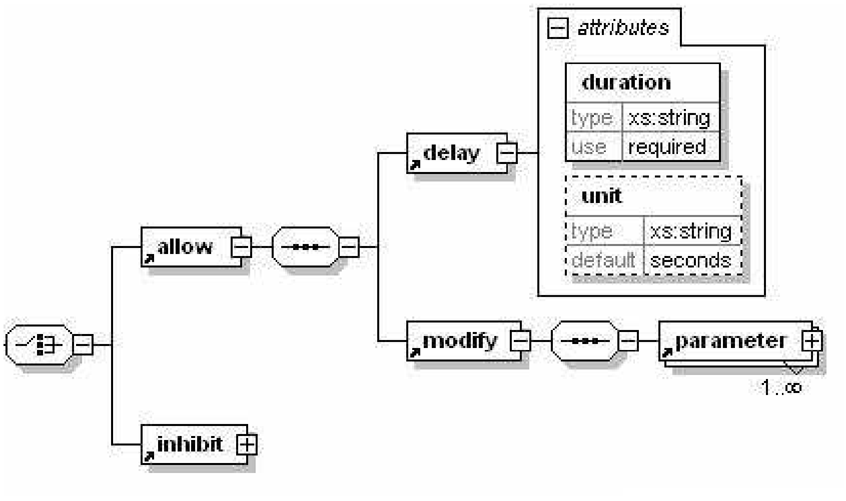


Figure 3.3: XML mechanism response schema.

3.1.2 Mechanism Actions

The mechanism actions are user-defined actions to be executed and in this scenario these are function calls, which are defined by a function name, a type, the component where they should be executed (the action does not have to take place inside the monitor, therefore a number is used to identify the target component; the number '0' is reserved for the monitor) and optional function parameters for the execution. The decision to allow actions to be executed outside the monitor was taken so as to add flexibility to the monitor and to decrease the time spent executing each monitor. Figure 3.4 represents the XML schema for specifying actions.

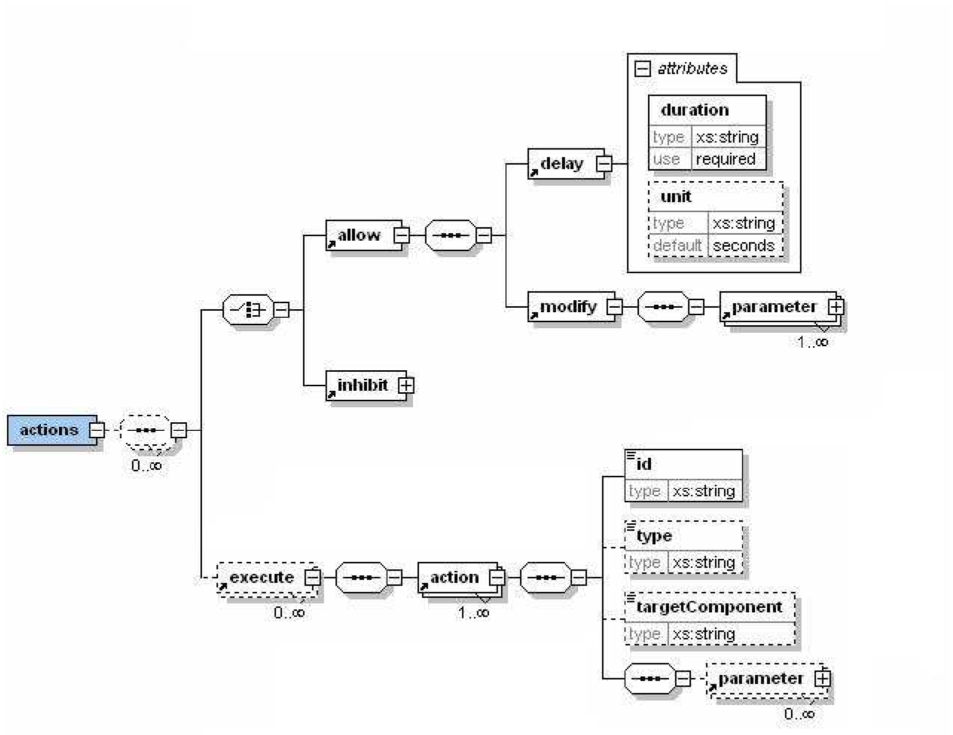


Figure 3.4: XML mechanism actions schema.

3.1.3 Restrictions

There are a few restrictions that are necessary for the control monitor to work correctly. Firstly, an event request can only trigger one mechanism, therefore if more than one mechanism would be triggered by an event only the first mechanism will actually be triggered; this restriction is necessary because otherwise we might trigger mechanisms that return conflicting responses (for example, one says modify and the other says delay, which response should be enforced?). One option would be to implement precedence rules for this kind of situation, but for this implementation it is sufficient to assume that these situations will not happen.

Secondly, the actions to be executed that must take place inside the mechanism must be known beforehand, as they must be hard-coded in the monitor. This is only a technical restriction as mapping between function names and addresses at runtime does not exist in our choice of implementation language (C), it is possible to create a dynamic mapping, however function names still have to be known beforehand (hard coded). This problem is not a real limitation as functions can also be executed outside the monitor. The functions that can currently be executed inside a monitor are to print a string, delete a file and reset a monitor; other functions can be easily added to the monitor, these were chosen for demonstration purposes.

Thirdly, the effects of one mechanism should not invalidate the condition of another mechanism, as this can cause inconsistencies within the system. However, this last restriction is not verified by the monitor, but should be avoided by the user.

3.2 Control Monitor API

The control monitor API is very similar to the policy monitor API, the same set of functions for creating, inactivating and resetting monitors are available for the mechanisms and they use the same structures to represent events and parameters and to receive input.

The most important function in the API is new_event_request(), where the received event is compared to all the mechanism triggers and if a match is found, the monitor checks if the mechanism condition would still be true if it was updated with the new event (this is done by calling the function update_past_monitor() with the DO_NOT_UPDATE flag); if yes the event request is transformed into an actual usage and all mechanism conditions are updated, otherwise the mechanism actions are executed.

A control monitor must be able to differentiate between an event request and an actual event; in this implementation, both kinds of events are represented using the exact same structure and the differentiation is done implicitly by the context of the monitor, i.e. some code sections deal only with requests while others deal only with actual events, so it is not necessary to distinguish them.

Additionally, the API uses a mechanism_actions_t structure to represent the body of the mechanism and it follows the same pattern as the XML schema, it is recommended to use the parser to fill these structures, but it is possible to do so manually as well.

```
/* Initializes the mechanism manager
* @RETURN  0                success
* @RETURN  -1               error: could not initialize library
*/
int init_control_monitor();

/* Creates a new mechanism
* @PARAM   id               unique mechanism identifier
* @PARAM   trigger          mechanism trigger event
* @PARAM   condition        mechanism condition in osl
* @PARAM   actions          array of mechanism actions
* @PARAM   num_actions      number of mechanism actions in actions array
* @RETURN  0                success
* @RETURN  -1               error: monitor not be created for mechanism condition
* @RETURN  -2               error: id is already being used
*/
int add_mechanism(char *id, event_t* trigger, char *condition,
mechanism_actions_t actions);

/* Deletes the mechanism, if it exists
* @PARAM   id               unique mechanism identifier
*/
void delete_mechanism(char *id);

/* Resets the state of the mechanism condition and the mechanism activation time
* @PARAM   id               unique mechanism identifier
*/
void reset_mechanism(char *id);

/* Changes mechanism state to active.
*  @PARAM  id               unique mechanism identifier
*/
void activate_mechanism(char *id);

/* Changes mechanism state to inactive
*  @PARAM  id               unique mechanism identifier
*/
void disactivate_mechaninsm(char *id);

/* Checks if a mechanism exists
* @PARAM   id               unique mechanism identifier
* @RETURN  ACTIVE           if mechanism exists and monitor's state is active
* @RETURN  INACTIVE         if mechanism exists and monitor's state is inactive
* @RETURN  -1               if mechanism does not exist
*/
int lookup_mechanism(char *id);

/* Checks if a desired event can become an actual event and if it triggers any
control mechanisms
* @PARAM   desired_event    requested event
* @RETURN  NULL             desired_event is allowed
* @RETURN  !NULL            event triggered
*/
mechanism_actions_t* new_event_request(event_t* desired_event);
```

Listing 3.1: Control monitor API.

4. GENERIC FRAMEWORK ARCHITECTURE

To successfully enforce usage control policies in any system, three main components must be correctly implemented in a framework: the obligations monitor, the event signaller, and the enforcement mechanisms. Aside from these major components, it is also necessary to have a means of deploying policies remotely, offering feedback to the policy administrator and storing the policy states as well as storing a log of events. In this section, each of these elements will be explained in detail, together with their interactions.

To begin with, one of the most important parts of the framework, the obligations monitor, is responsible for constantly checking if a series of events causes one or more policies from a loaded set to be violated. This component should be implemented similarly to the policy monitor from section 2.

The event signaller encapsulates a layer of the software stack with the aim of intercepting all user actions in that layer. These actions are translated into events, using the format defined in OSL, and are then sent to the obligations monitor to be evaluated. A system may have more than one event signaller working simultaneously and at different abstraction levels. At the operating system level, examples of frameworks that can be used for system call interception include Systrace in OpenBSD, Ptrace in Linux and Detours in Windows.

The enforcement component is responsible for taking actions after it has been detected that an event would violate a policy. These actions range between only logging the violating event to actually preventing the event from happening. This component interacts directly with the obligations monitor, and an example of an implementation of this component is the control monitor from section 3.

A few other, auxiliary, components are important in a framework, such as the event log. The purpose of the event log is to register every single event sent to the monitor, allowing policy violations to be analysed and justified should the need arise. Another auxiliary component is the policy and state repository which should store all the policies that have been loaded into the obligations monitor, as well as their internal state; this enables the usage control framework to recover after a crash (otherwise a user could force a crash of the system to avoid the detection of violations).

It is also advisable to implement a mechanism enabling remote deployment of policies and automatically informing the creator of a policy of necessary information such as a policy violation or the current state of the policy. This will allow the policy administrator more control over the data being protected and will provide an easy way to load policies and control mechanisms into the usage control framework.

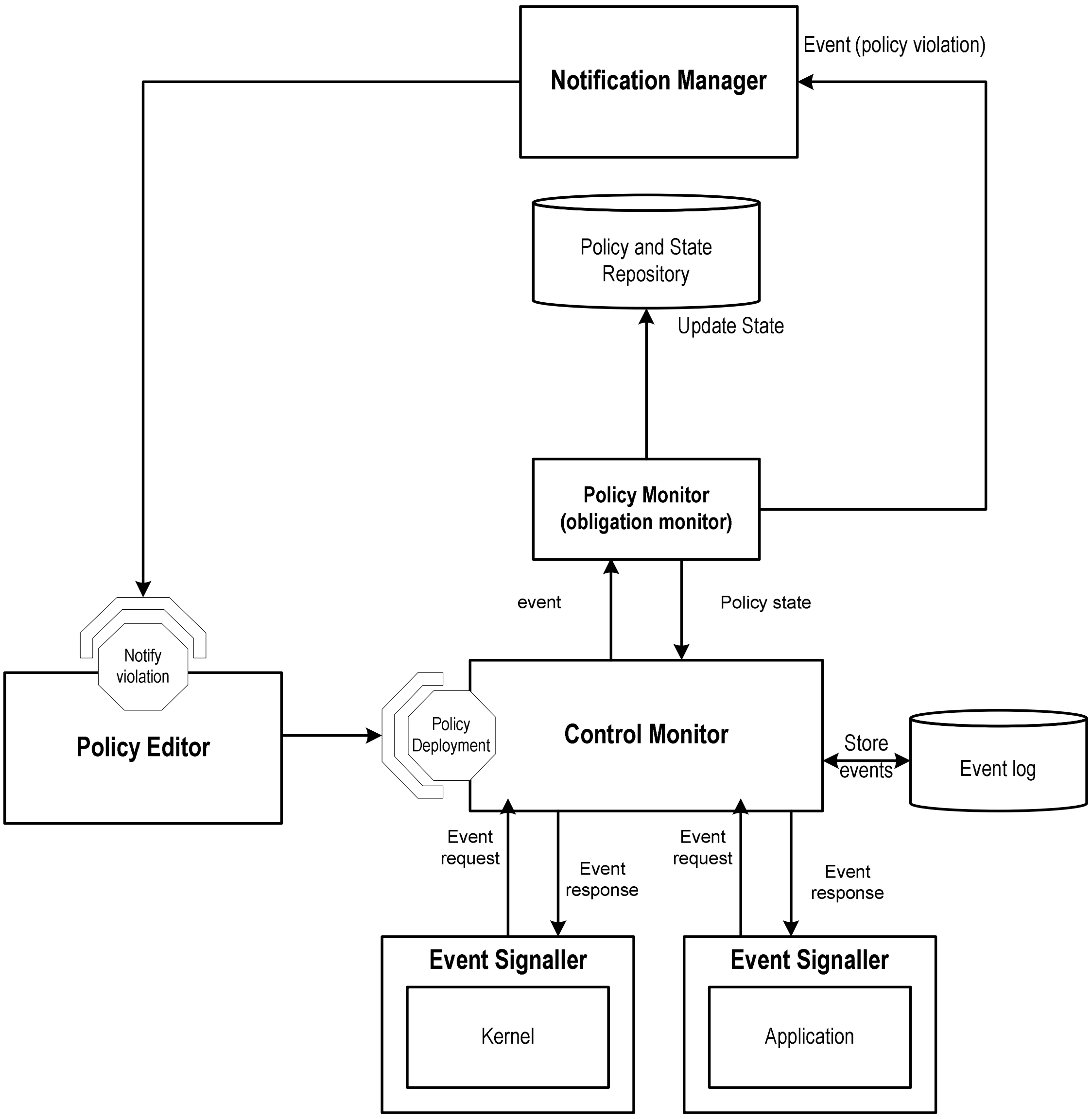


Figure 4.1: Generic usage control framework architecture.

Figure 4.1 shows a generic architecture for a usage control enforcement framework, and, although some minor components may be changed, merged or, sometimes even omitted to accommodate specific system needs, the basic structure must remain the same. Here, the control monitor represents the enforcement component and is also responsible for forwarding the events from the all the event signaller components and for receiving new policies and control mechanisms from a policy administrator. The obligation monitor maintains a backup of the internal states of all policies and is responsible for alerting the creator of the policy of violations via the notification manager.

5. OPENBSD SYSTRACE FRAMEWORK

This section describes the implementation of a secure framework on OpenBSD to enforce usage control policies in the operating system level. This is done by intercepting system calls in the kernel using the Systrace framework (described in detail below), transforming these calls into events, inputting these events into the control monitor described in section 3, and then applying the control mechanisms. The reason OpenBSD was chosen for our real system implementation was the availability and ease-of-use of the Systrace framework present in this platform.

5.1 Systrace

Systrace is a security utility tool that limits an application’s access to the system by enforcing system call polices. This is done by intercepting system calls and their parameters before and after execution, a process known as system call interposition. With Systrace it is possible to sandbox an application and observe, or even limit, its behaviour.

It is possible to specify many constraints in a Systrace policy; however the monitoring framework is only interested in one aspect of these policies: the policy determines whether a program may always execute a system call, never execute it or if additional information is needed for this decision to be taken. If a system call is to always be allowed or always denied, the Systrace processing and decision is made directly in kernel space, so as to introduce as little overhead as possible. However, if the decision requires other information, the system call together with its arguments is redirected to a user space process which is responsible for the decision. This process may alter the system calls arguments before replying to Systrace.

The user space process used by the OpenBSD framework is called the ‘call interceptor’ and is responsible for attaching processes to Systrace (Systrace will then intercept those processes calls), instantiating Systrace policies for every process (note that these policies have nothing to do with our usage control policies), and making decisions on the execution of any calls forwarded from the kernel space. In this implementation, the Systrace policies are the same for every process; all system calls are allowed except the calls: open, close, read, write and unlink; the execution of these calls must be decided by the call interceptor, as these are the calls that relate to data that is in the system and that will be used to enforce the usage control policies at the system call level.

The sequence of events, from the request of a system call execution until the reception of the execution response by a user process, is shown in detail in figure 7. First the user process requests a system call, which is then intercepted by the Systrace module and, if it is not allowed or denied by default, forwards it to the call interceptor. The call interceptor sends an answer to Systrace, denying or allowing the call, and the call is executed. Systrace then intercepts the call parameters (including the system call return value) and once more forwards it to the call interceptor, as it might desire to change the parameter values before returning to the user process.

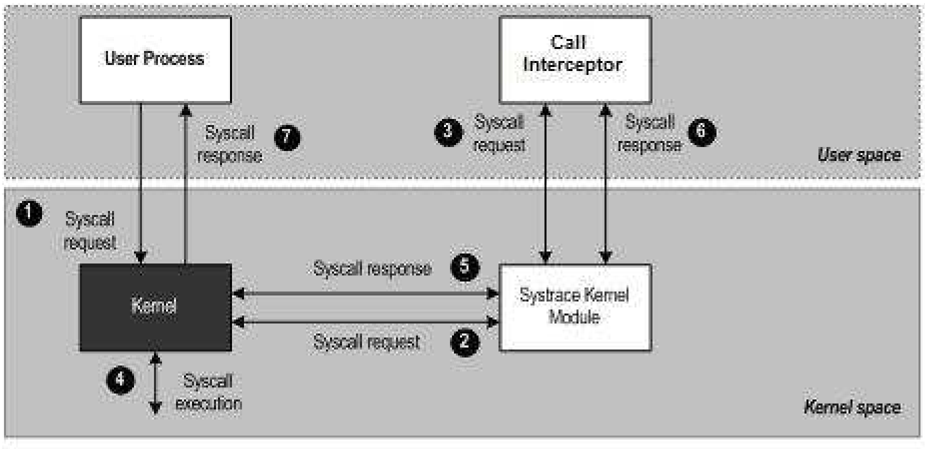


Figure 5.1: System call interposition.

5.2 OpenBSD Implementation Architecture

The overall framework architecture consists of four components: the Systrace kernel module, the call interceptor, the control monitor and the shell wrapper, which all work together to enforce usage control policies. The components and their interactions are shown in figure 8. The Systrace module forwards the relevant system calls to the call interceptor, as previously described. The Call Interceptor, as well as communicating with Systrace, instantiates and interfaces with the control monitor. The call interceptor assumes the role of event signaller, since every time a system call request is received, it generates an event request to the control monitor. The control monitor, as described in section 3, has a list of loaded mechanisms and at every event request, checks if any mechanism is triggered, and then generates an event response to the call interceptor. Upon receiving the event response, the call interceptor must execute the actions determined by this response, which can be to delay the execution of the system call, to modify the parameters of the system call, or to execute some arbitrary action, and then finally answering Systrace, allowing or denying the call. This framework is not interested in the results of the system call, therefore they are ignored.

The shell wrapper is the component responsible for informing the PID of the process that should be attached to Systrace, so as to be monitored. Since we want to monitor all user processes, this is done by creating a wrapper so that when a user logs in the shell wrapper is called instead of the initial user shell. This shell wrapper calls a soap service (attach process), that sends the PID of the shell wrapper to the call interceptor (which is loaded at boot time) and only then starts a user shell. This ensures that all processes created by that user will be monitored, as all processes are child processes of the shell wrapper and Systrace automatically monitors all child processes. If the shell wrapper is not able to communicate with the call interceptor, the user’s login request is denied.

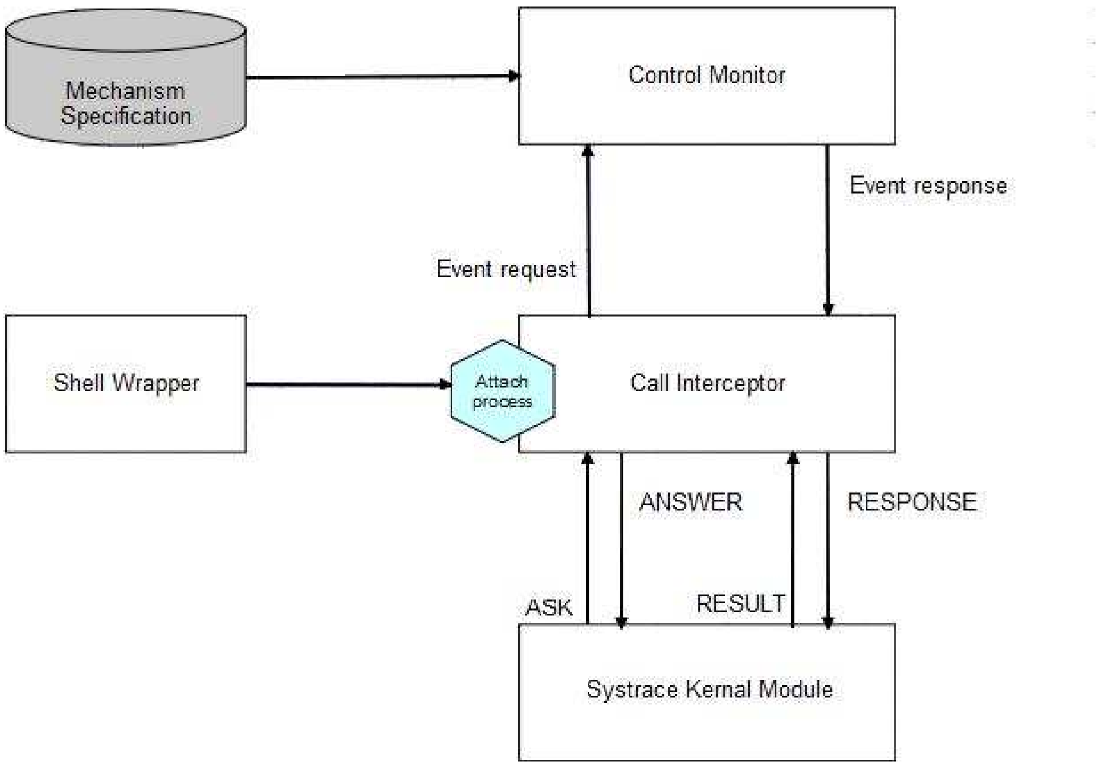


Figure 5.2: OpenBSD framework architecture.

5.3 Implementation Security and Limitations

The aim of the OpenBSD framework implementation was to provide a usage control enforcement mechanism that could not be tampered with or fooled by a user without super user rights. The framework manages to successfully monitor all user processes from the moment the user logs on, as there is no way a normal user can create a process without it being monitored by Systrace. Additionally, the monitoring process is started during the system boot and cannot be killed by a normal user. If by any chance the monitor crashes, all user processes that were being monitored are automatically killed and if the monitor is not running when a user tries to login, the user shell will not be created and the login will fail. These measures ensure that all user actions are being monitored at all times. There is also some related work being done in trusted computing that aims to guarantee that all the framework requirements are satisfied and that all the necessary components are loaded correctly by using a TPM chip to perform hardware and software configuration integrity measurements and to ensure that the appropriate mechanisms are loaded.

There is one important limitation to the framework that relates to the implementation of the delay mechanism. If a mechanism determines that a system call should be delayed, this delay ends up impacting all monitored processes and not just the process that invoked the system call. This happens because the Systrace module must receive an answer from the call interceptor relating to a system call request, before a new request can be generated; as the delay has to be enforced in the call interceptor (once the Systrace module receives an 'ALLOW' for a system call, the call is executed immediately), all processes end up suffering the delay period. The only way this can be avoided is if there is an instantiation of the call interceptor process for every user process to be monitored and an interprocess communication method is used to send events to the control monitor that also runs in its own process (there must only be one control monitor as the usage control policies are universal for all processes). Of course, as the monitor does not support parallel evaluations, the control monitor would become a bottleneck for the system.

6. WINDOWS DETOURS FRAMEWORK

While the OpenBSD framework provides a valid implementation of usage control enforcement on the operating system level, it is important to be able to enforce usage control on a more widely used operating system, such as Windows.

In the Windows framework, instead of intercepting the system calls from the kernel, we use the Microsoft Detours library to intercept relevant Windows API function calls, which are present in kernel32.dll. The actual monitoring part of the framework remains the same as in the OpenBSD implementation, only the generation and signalling of events is changed.

6.1 Detours

Detours is a library for detouring function calls; this is done by replacing the call to the target function by a call to a user defined detour function, which can still call the original function through a trampoline function, if required, as detours preserved the target function code so it can be called as a subroutine. The detouring of functions occurs at execution time by altering the code in memory, which creates the possibility of having an instance of a program that uses the detoured functions and another instance of the same program without the detours, executing at the same time.

The actual detouring is done by creating a process hook, which is a DLL that will be injected into the detoured process' code and that will redirect the chosen functions. For every function that will be detoured, the process hook must declare a trampoline and a detoured function with the exact same signature as the target function. The detoured function contains whatever we want to execute and can use the trampoline function to execute the original target function, which then returns to the detoured function. This allows function pre-processing and post-processing.

The process hook must also contain a DLLMain function where, for every process that attaches the DLL, the detoured functions are attached to their target functions. When a process detaches the DLL, the detoured functions must also be detached from their target functions.

The injection of the process hook DLL into the process that is to be monitored can be done by using the withdll utility that comes with the detours package.

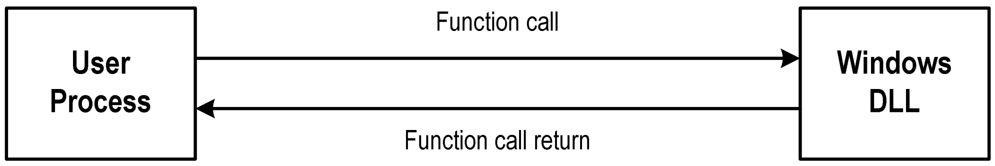


Figure 6.1: Normal windows API function call.

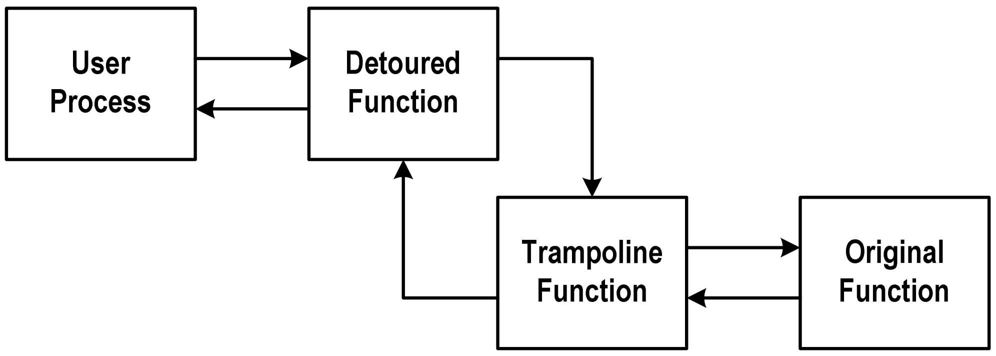


Figure 6.2: Windows API function call with Detours.

6.2 Windows implementation architecture

The Windows framework detours the functions CreateFile(), ReadFile(), WriteFile() and DeleteFile() from the windows API. Each one of the detoured functions created generates an event request that is sent to the control monitor and receives a response. Like in the OpenBSD framework, the process hook must execute the actions stated by the response, besides delaying or modifying the function call, and then executes the target function by calling the trampoline function (except if the response was to inhibit the function call). In other words, the process hook assumes the event signaller role.

The overall architecture of the Windows framework consists of the control monitor, the user processes that are being monitored and the process hook, that intercepts the calls to the windows API functions and also interfaces with the control monitor.

In the OpenBSD implementation, the event signaller (call interceptor) and the control monitor run in the same process. However, in the Windows implementation this is not possible because the event signaller (process hook) is not a process but a DLL and the control monitor cannot be initialized in a DLL. This is because DLLs share code but not data segments between all the processes that are using the DLL, which means that if the monitor was initialized inside the DLL, each process would have its own independent copy of the monitor structures, including the policy states; in other words, each process would have an individual and independent monitor. This is clearly not acceptable as usage control policies are universal in the system, therefore the control monitor must run separately from the process hook and the interaction between these two components should be done using an inter-process communication protocol such as SOAP or RPC.

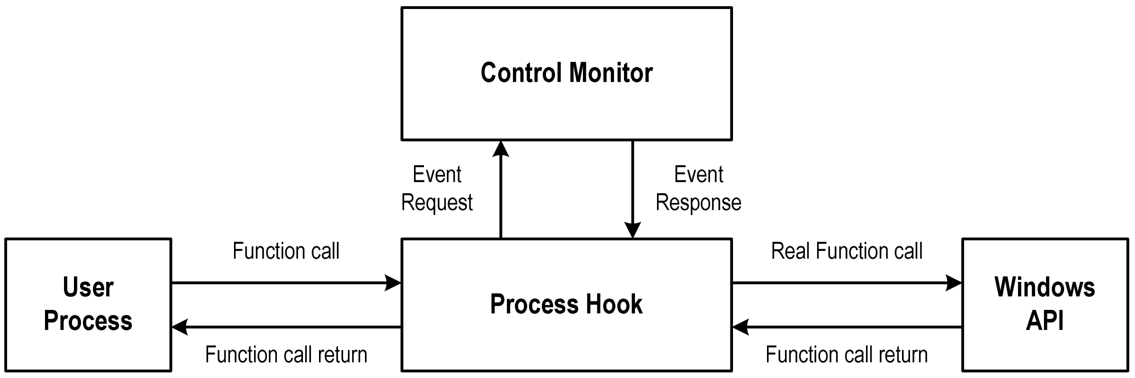


Figure 6.3: Windows framework architecture.

6.3 Process Hooking Techniques and Detours Limitations

There are two ways to hook a process with Detours: an individual hook and a system wide hook. The individual hook is the simplest approach and can be done using the withdll utility from the Detours package. In this approach, each process has to be manually hooked to the detoured API when the process starts and is a fairly efficient process. However, this approach is not suitable for this implementation, as the framework's aim is to monitor every user process since the user's logon and it is extremely difficult to monitor the creation of user processes, and even more so to inject the process hook as soon as the process is starting.

The second alternative covers much better our framework's requirements but is also much more costly. To implement a system wide hook, one must create a kernel driver with a process creation notify mechanism (using the PsSetCreateProcessNotifyRoutine function) which notifies userspace about every new process being created and which can then be used to automatically inject a detoured dll into the newly created process; but, as previously mentioned, this process significantly slows down the operating system and even a small bug can cause a crash, from which a user might possibly have to restart the whole operating system to recover from.

Lastly, it is important to point out that some functions used in creating the framework are not compatible with all versions of Windows that are currently in use and even that the behaviour of Detours might be slightly changed in different versions of Windows. The current Windows framework implementation is only compatible with Windows Vista and later releases.

7. PERFORMANCE MEASUREMENTS AND ANALYSIS

7.1 Monitoring overhead

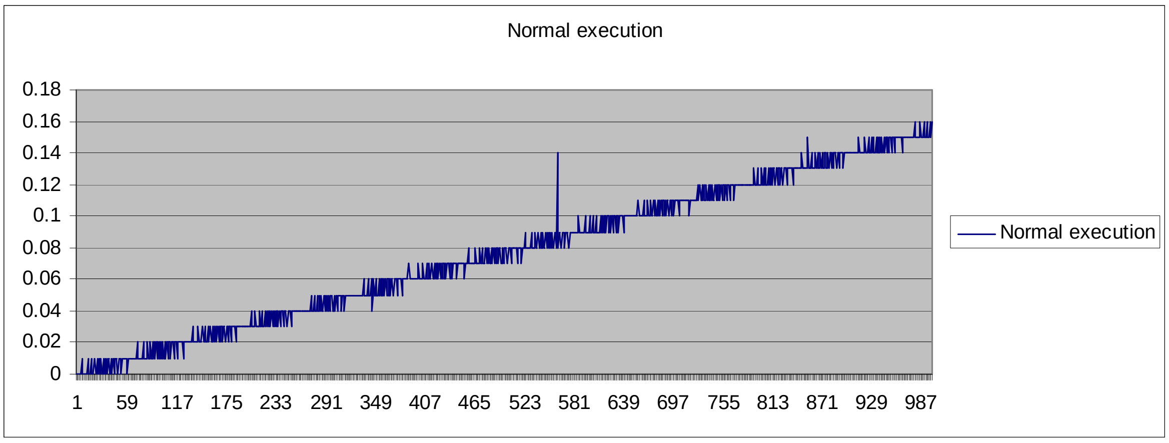
So as to evaluate the impact of our monitoring on the overall system performance, a series of tests were performed to measure the CPU time the system call interception framework adds to a monitored process.

With this in mind, a test program was created that executes a certain number of system calls to open, read, write and close files. Which system calls are executed does not matter, as there is no difference in the overhead generated by, for example, an open and a write system call.

The measurements were taken using the OpenBSD framework because it better represents a real-world system, as the Windows framework was implemented without a system wide hook. The scenarios that were measured were:

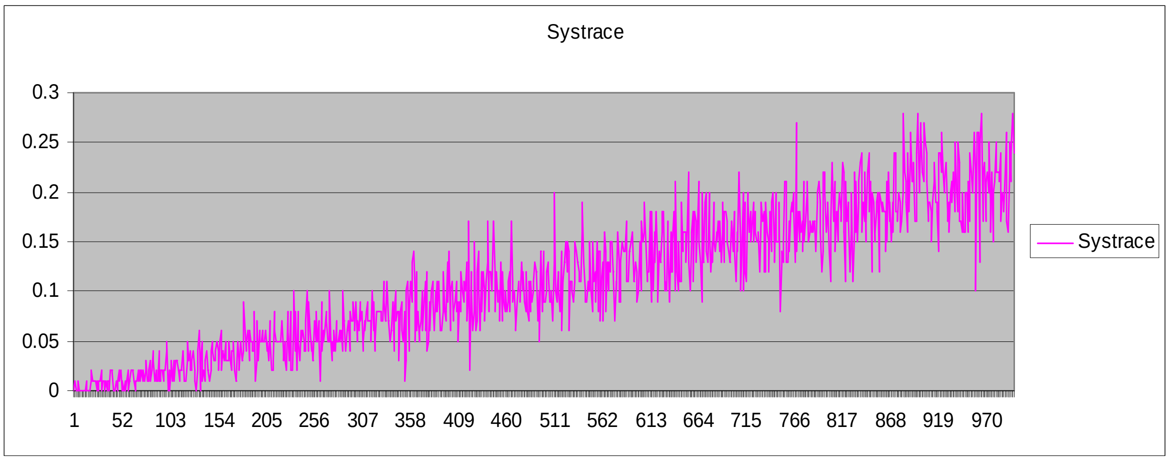
- firstly, we ran the test program without the framework to generate values with which to compare all other measurements;
- the second measurement was to run the test program attached to the framework but without any mechanisms being loaded, so as to measure the overhead caused by the signalling component (in this case, Systrace);
- lastly, the second measurement was repeated but this time a set of mechanisms were loaded, so as to calculate the total framework overhead.

Also the behaviour of the system with a variable number of mechanisms (between 10 and 100) and a variable number of executed system calls (between 1 and 1000) was assessed.



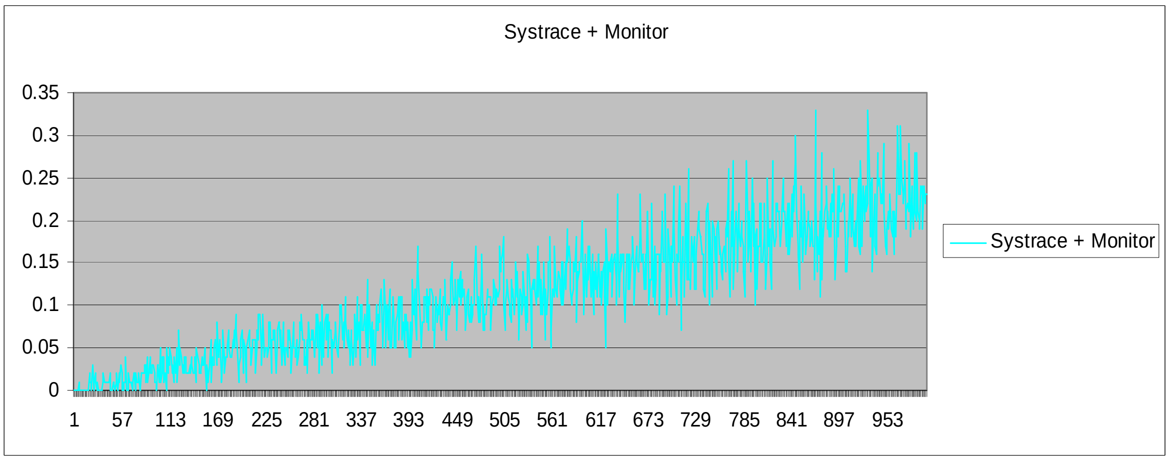
Graph 7.1: Normal system call execution times in seconds.

Graph 7.1 shows the execution times of the test program without the framework, and the execution time grows linearly with the number of system calls called.



Graph 7.2: Test program execution times with Systrace.

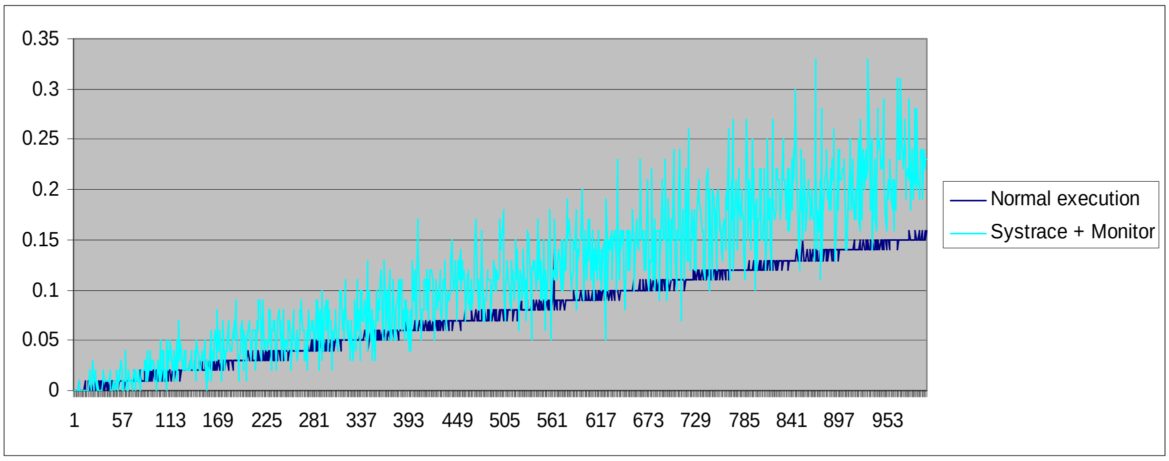
The second graph shows the results of the test program execution times when the program is being monitored by Systrace. Systrace intercepts the system call from the kernel, forwards it to the monitoring process in user space and immediately allows and returns the call to the test process, therefore, this is the overhead of using only Systrace, as no policy evaluation or modification of the call occurs. Comparing graphs 7.1 and 7.2, we can see that Systrace adds an overhead of approximately 40%.



Graph 7.3: Monitoring framework execution times.

Analysing graph 7.3, which shows the test program's execution times when attached to the monitoring framework loaded with mechanism, we can state that the total monitoring framework overhead is not much more than the overhead already caused by Systrace. Additionally, this last measurement scenario was run many times, with an increasing number of mechanisms and this did not generate perceptible differences in execution time.

Finally, graph 7.4 shows a comparison between the execution time of the test program with and without the monitoring framework. Although there is a significant overhead from monitoring the program, it does not make the monitoring unviable.



Graph 7.4: Comparison between execution times.

7.2 Evaluation of the OpenBSD and Windows implementations

It is important to point out that there is a slight difference in the level of abstraction in which the OpenBSD and the Windows frameworks are implemented. While Systrace allows the OpenBSD framework to intercept system calls directly in the kernel, this is not possible in Windows because kernel access is not available; therefore we have no choice but to intercept the calls at a slightly higher level: the Windows API. The practical consequences of this difference is that the Windows framework has access to more data pertaining to the system call and this access is much easier than in the OpenBSD implementation. For example, when executing a read or write call, the calling process passes a file pointer as a parameter to indicate which file should be accessed, however, usage control policies deal with file names; in the Detours framework this is not a problem, as the Windows API maintains a mapping of file pointers to file names; no such mapping is available in the OpenBSD kernel, forcing the framework to create a table with these values (which is less efficient and is a potential security threat). This is further complicated by the fact that a same file may have many file pointers, and a file pointer may represent more than one file name (because of soft and hard links).

Aside from the difference in the signalling component mentioned above, both implementations are quite similar, mostly because the policy evaluation component is the same. Also, both frameworks aim to monitor all user processes; monitoring system processes is unfeasible, partly because Systrace and Detours do not have this capability (Systrace can’t attach to system processes, and system processes in Windows use a different dll for system calls than the kernel32.dll whose documentation is not publicly available) , but mostly because enforcing mechanism actions might turn the operating system unstable; for example, an inhibited system call may cause the whole operating system to crash.

Finally, although the frameworks presented in this thesis are capable of enforcing many aspects of usage control, to make the framework really complete, an information flow component must be added. This component would be responsible for tracking sensitive data throughout the system, making it impossible for a policy violation to not be detected. Currently, simply renaming a file allows a user to sidestep a usage control policy, as it remains attached only to the previous file name. Another method of ensuring the correct running of the framework is to add trusted computing modules, which can be used to guarantee the correct monitor and mechanism configurations.

8. CONCLUSION

In conclusion, we successfully implemented an obligations monitor that can be used in usage control enforcement by observation and a control monitor that can be used in enforcement by control. A generic framework to enforce usage control in a system was also shown, along with two applications of this framework in real-world systems: OpenBSD and Windows.

Our test results show that, although process monitoring adds a significant overhead to the CPU time of said process, most of this overhead is not caused by the control and obligations monitors but by the signalling mechanism (more specifically, by the Systrace framework used to capture events). And, more importantly, the generated overhead is not too large as to make process monitoring unviable.

Finally, the proposed framework is able to enforce most aspects of usage control but in order for it to be really complete, an information flow component must be added. It is also necessary to guarantee that the monitors are running and with the correct configurations, trusted computing being the most adequate method for this.

REFERENCES

K. Havelund , Grigore Roşu, Efficient monitoring of safety properties, International Journal on Software Tools for Technology Transfer (STTT), v.6 n.2, p.158-173, August 2004.

M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A Policy Language for Distributed Usage Control. In Proc. ESORICS, pages 531-546, 2007.

M. Hilty, A. Pretschner, D. Basin, C. Schaefer, T. Walter, Monitors for Usage Control, Proc. Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM), Moncton, August 2007

G. Hunt, D. Brubacher, Detours: binary interception of Win32 functions, Proceedings of the 3rd conference on USENIX Windows NT Symposium, p.14-14, Seattle, Washington, July 1999.

A. Pretschner, M. Hilty, C. Schaefer, T. Walter, and D. Basin. Mechanisms for Usage Control. In Proc. ASIACCS, pages 240–245, 2008.

A. Pretschner, J. Rüesch, C. Schaefer, and T. Walter. Formal Analysis of Usage Control Policies. Proc. 4th Intl. Conf. on Availability, Reliability, and Security (AReS), Fukuoka, March 2009.

N. Provos. Improving host security with system call policies. In SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, pages 18–18, Berkeley, CA, USA, 2003.