International Journal of Security and Its Applications
Vol. 10, No. 8 (2016), pp.417-434
http://dx.doi.org/10.14257/ijsia.2016.10.8.37

Efficient and Fast Detection of Covert Channels using Mutation Analysis

Mohammed Ennahbaoui[1] and Said El Hajji[2]

[1,2] Mohammed V University in Rabat, Faculty of Sciences, Laboratory of Mathematics, Computing and Applications, BP1014 RP, Rabat, Morocco
[1]ennahbaoui.mohamed@gmail.com, [2]elhajji@fsr.ac.ma

Abstract

Covert channels are among the serious and most complicated security flaws that attempt to broke the access control policies, since they allow an unplanned communication medium within an information system. The important impact of this vulnerability gives it a wide interest by the researchers. In this paper, we propose a new approach based on an enhanced version of the shared resources matrix method, where the principle of mutation analysis is integrated to allow an efficient and fast detection of covert channels. We have conducted an evaluation within a real system and the obtained results show very promising performances.

Keywords*: Covert Channel, Security Policy, Mutation Analysis, Shared Resources Matrix*

1. Introduction

Access control is the core of any security policy within an information system, in such a way that every vulnerability in this system is necessarily due to a deficiency of this aspect. A security policy relies initially on the choice of an access control model, that satisfies the security needs of the relevant information system, in which this policy is implemented. Thus, a security flaw in an access control model influences directly the security policy of this model. Among the most tough and complicated flaws in the access control models are **Covert Channels**.

The problem of covert channels is not new, it was detected in 1973 by Lampson in [1], but still up to now without any efficient solution, which makes it one of the most complex problems in the area of access control. A covert channel is defined as "an un-intended communication medium by designers, within an information system". Actually, there are several types of covert channels, but the well-known are the Covert Storage Channels (CSC) and the Covert Timing Channels (CTC). In order to face this problem, most solutions adopt detection mechanisms to discover the covert channels, then, they restrict the entities that are responsible for producing them.

In this paper, we give an overview of the covert channel flaws, including their definitions, types and semantics. Thereafter, we describe in details our proposed solution which is mainly based on the technique of mutation analysis to detect cover channels. Thus, we make use of an enhanced version of the shared resources matrix method, in such a way we incorporate in our system a set of mutants corresponding to the various levels in the matrix, and we proceed to the verification progressively. A throughout evaluation of our solution is conducted, through implementing a test on a real information system, and which demonstrates the feasibility and the efficiency of our method.

This paper is organized as follows. Section 2 defines the covert channels and their types. Section 3 presents some related works investigated to deal with this vulnerability. A description of the proposed approach is provided in Section 4, while a thorough

ISSN: 1738-9976 IJSIA

evaluation is conducted in Section 5. At the end, a Conclusion is mooted with some perspectives.

2. Covert Channels

Covert channels represent one of the most complicated security flaws within an information system, even with the establishment of a security policy [4]. This is mainly because this vulnerability affects directly the concept of access control as one of the primary security requirements.

Since the discovery of the covert channels, several definitions were proposed in the literature. Among these definitions we can cite three famous ones:

- For Lampson [1]: Covert channels, *i.e.*, those not intended for information transfer at all, such as the service program's effect on the system load.
- For Kemmerer [3]: Covert channel, use entities not normally viewed as data objects to transfer information from one subject to another.
- For DoD (DEPARTMENT OF DEFENSE) [4]: A covert channel is any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy.

From the above definitions, we adopt the following definition as a generalization to be adopted along our work: *"A covert channel is a medium of communication not planned by the designers of an information system (either not originally intended or not allowed to transfer information). Therefore, the overall security policy of the system is violated, because this medium allows to transfer data between two different subjects not allowed to communicate with each other, such that one gives an information (in write mode) and the other receives it (in read mode)."*

Actually, there are several types of covert channels, including two major ones: the Covert Storage Channels and the Covert Timing Channels:

1. Covert Storage Channel, according to [4], includes all traffic that would allow the direct or indirect writing in a storage location by one process, and the direct or indirect reading in the same location by another process. In other words, it is a direct or indirect link between two processes (two subjects) using a data storage resource of information system (*e.g.*, an object, an object attribute, hard disc. ..), such that a process has a write access to this support and the other process has a read access.

2. *Covert Timing Channel*, according to [4], includes all traffic that would allow a process to report an information to another process, by modulating its own use of system resources, in such a way that the observation of the change in response time would provide information to the second process. In other words, it is a direct or indirect link between two processes (two subjects), using a temporal resource of the information system (*e.g.*, CPU computing time). More simply, the sending process calculates the time needed by the receiving process to detect a modification in an attribute or perform a task, while the receiving process interprets this delay or the changing in delay as an information. For example, assuming that the delay is one second, if the sending process occupies the CPU of the information system for more than one second before leaving the place to the receiving process, then the receiving process will translate this by 1. Contrariwise, if it occupies the CPU for one second or less, then the receiving process will translated this by 0.

In general, there are two approaches to solve this problem. The first one is to conceive a new access control model with multiple functionalities, able to face the covert channels. The second one considers how to protect an information system against this attack, while and after implementing an access control policy. In this

case, we focus on all components of the information system in order to detect the covert channels and lessen their risks [4].

3. Related Works

In practice, it is not possible to remove all the covert channels. But we can reduce their bandwidth, reduce the number of possible shared resources, fix the time of resource sharing, *etc*. However, the problem with the techniques of bandwidth reduction is that they have a direct influence on the intended productivity and efficiency for the system' legitimate users. Then, the appropriate solution would be to reduce the bandwidth of the covert channels, in such a way to not exceed the threshold of 100 bits per second. According to [4], an information system with a covert channel, whose bandwidth capacity is greater than this threshold, is considered as an insecure system. Unfortunately, up to now there are no explicit methods to eliminate the covert channels, but there are only methods to identify and detect covert them or cancel their effects.

There are several methods to counter covert channels, including two well-known ones:

- The Non-Interference Model (just for the covert storage channels).
- Shared Resource Matrix (SRM).

The **Non-Interference Model**, as defined by Goguen and Meseguer in [5], is the application of the non-interference property in one of the multi-level security (MLS) models. In a simple way, a non-interference model is a MAC model provided with the non-interference property to eliminate the existence of covert storage channels. According to [6] this property is defined as follows: Let s_1 and s_2 be two subjects of a system. We say that s_1 does not interfere with s_2, if no action taken by s_1 cannot have an effect on the view of s_2 within the system, and we write: $s_1 \not\rightarrow s_2$.

This explains how the non-interference model tries to ensure that, a non-privileged user can not interfere any information on private data to which he has no access, from the observation of public data to which he has access. Thus, the effect of covert channels is eliminated. The well-known non-interference model is Bell-LaPadula model integrated with non-interference principle, in order to allow information flow only between comparable applications, even with different levels.

Despite the power of the non-interference property, it is difficult to verify in practice, because it requires to be tested on every subject while running each of its applications sequences. Moreover, the non-interference models can eliminate only the covert storage channels, but we still need to implement other methods to counter precisely the covert timing channels.

The **Shared Resource Matrix** (SRM), invented by Kemmerer in 1983 [3], is a technique that aims to detect both types of covert channels, whether of the storage or the timing channels. It is performed in two steps. The first step is to determine all shared resources that can be modified or referenced by a subject, it is the creation phase of the shared resource matrix. The second step examines each resource, in order to determine if it can be used to transfer information from one subject to another in a hidden manner, it is the analysis phase of the shared resource matrix. This technique considers that the subjects are the basic processes, called "*primitives"*, while the shared resources are objects or collections of objects able to be modified by more than one process. It is necessary to refine each shared resource through indicating its attributes, because we may have two processes that share the same file, but the first has access to its length, while the second is able to read its content.

The creation phase includes three main manipulations to build the shared resources matrix:

1. Determine the objects and their attributes, in order to identify the headers of the matrix lines.
2. Determine the primitives, in order to identify the headers of the matrix columns.
3. Fill the cells of the matrix, in such a way that each cell indicates if the primitive of its column modifies or references the attribute of its line. There are three values possible in each cell: ***M*** if modifies, ***R*** if references, and **none** if there is no relationship between the primitive and the attribute.

In the analysis phase, we search the existence of covert channels, which must satisfy at least the following three criteria:

1. The sending and receiving processes must have access to the same attribute of the shared resources.
2. The sending process must hold the means to force the modification of the shared attribute, while the receiving process must hold the means to detect the change on the modified attribute.
3. A mechanism is needed to initialize the communication between the sending and receiving processes and to correctly sequence events, in such a way to have a logical chaining of events.

The shared resource matrix is one of the most efficient methods for the detection of covert channels. But its major problem, as described in [7], lies in the fact that it is "lazy", because it leaves the analysis and detection of the possible channels as a last step, which requires a heavy effort at the end. This approach appears easy when we use small systems, but when we deal with more complicated and voluminous systems, we will face an explosive growth of data flow.

4. Proposed Approach

All In this section, we describe our proposed approach that consists of an enhanced version of the SRM technique. It aims at analyzing the existence of covert channels, after inserting each data flow in the generated matrix of shared resources. In other words, the insertion of information flows, the creation of the matrix and the analysis are progressively performed, until generating the final matrix which requires only the analysis of the last flow. This should absolutely simplify and alleviate the analysis and detection of covert channels through updating the matrix of shared resources.

Our proposed method consists in applying the technique of mutation analysis on the information system, in which we search covert channels. Thus, we create a set of mutants for the system, then, we establish the matrix of shared resources for each mutant, in order to separately detect the covert channels.

4.1. Mutation Analysis

The principle of mutation analysis is related to the phase of software testing in the development of the applications, where it is necessary to eliminate the conception faults before validating a program. Mutation Analysis, introduced by DeMillo [8], consists in creating a set of erroneous versions of the program to be tested, called **mutants**. Thus, the basic idea is to invent a series of tests that allow to distinguish between the initial program and all its mutants, in such a way that many faults are injected to evaluate the ability of these tests to detect these faults.

The Mutation Analysis technique, as illustrated in Figure 1, includes two main steps:

- *The Creation of Mutants*: it begins by the research of the mutation operations, the application of these operations, and finally the elimination of the equivalent mutants. These later are mutants that do not differ from the initial program, which

requires their elimination since no test can distinguish them from the initial program.

- *The Execution of the Test on the Mutants*: in this step, if the test is able to detect mutants, then it is a confidence test for the software.

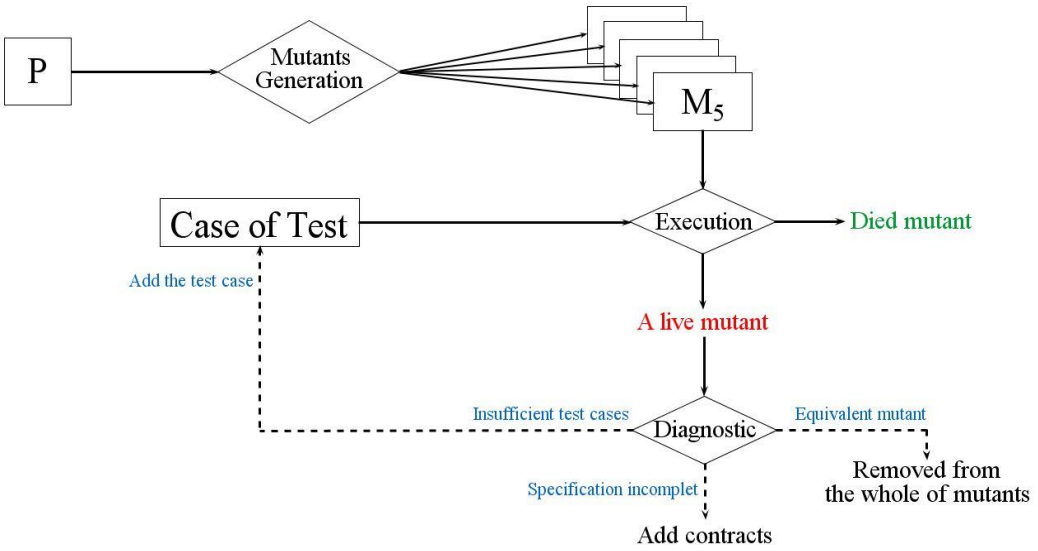

Figure 1. Operating Mode of Mutation Analysis

A mutant *M(i)* of the program *P* is created through modifying the original program *P*. This modification (error or fault) is called **Mutation**, and it should be introduced in the source code of *P* in an elementary manner and syntactically correct. A necessary condition is that *M(i)* must be different from *P* by one and only one modification, that could be a replacement of an operator, a symbol, variable name or numerical value. After the creation of all mutants, it is primordial to eliminate the equivalent ones before pursuing the analysis and go to the next step, because it is difficult for the test to detect them.

After the elimination of equivalent mutants, we proceed to the execution of the tests on the remaining mutants. If an executed test on a mutant produces a result different from this produced by the initial program, then, the test has detected or has killed a mutant. Else, we say that the mutant is alive. Once a mutant is killed, it can be removed from the process of testing, because the faults it represents have been detected which allow to spot a pertinent test. In case some mutants still alive at the end of the test, we verify that they are not equivalent. If so, we can either add more tests (also called "T*est Cases"*) in order to kill these mutants and obtain the satisfying mutation score for our system, or add the capacity of detecting an erroneous intern state or a fault in the initial software, which is called "*Contract"*.

4.2. Mutation Analysis for Covert Channels Detection

Our method is divided into several levels as shown in Figure 2, in such a way that each level contains a mutant and the corresponding shared resources matrix. Therefore, a level includes two steps: the creation of the mutant and the application of the SRM method on the created mutant.

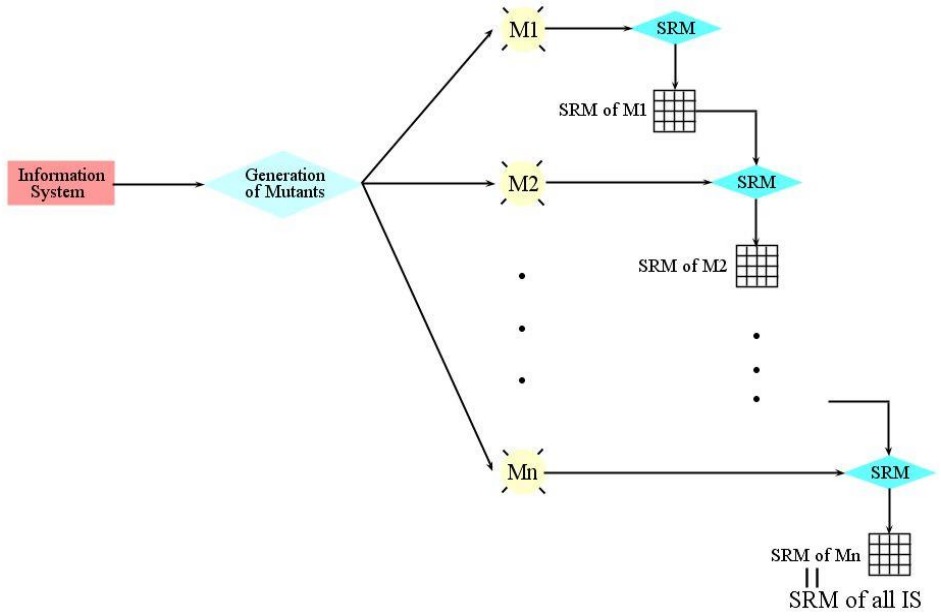

Figure 2. The Detection Approach of Covert Channels (SRM+MA)

In order to simplify the description of our approach to detect covert channels, we consider an information system with *m* primitives and shared objects, which have in total *n* attributes. Many mutants are created using a mutation operator called "**Add Attribute (AA)**", which allows to construct a mutant from another one by adding an attribute. Thus, the operator AA has M_{k-1} mutants as input and M_k as output, as described by the Equation 1,

$$where\ 0 \le k \le n: \quad M_k = \begin{cases} M_0 & if\ k = 0 \\ AA(M_{k-1}) & if\ 1 \le k \le n \end{cases} \tag{1}$$

In total, we have *n+1* mutants. The first mutant M_0 corresponds to the analysis of the information system without attribute, while the last mutant M_n represents to the overall system.

In order to create the matrix of the shared resources for each mutant, we begin by constructing the matrix SRM_0 for the mutant M_0. Knowing that this later has no data flows and contains only the primitives of the analyzed system, then, it does not include covert channels. Thereafter, the matrix SRM_1 for the mutant M_1 is created through adding a prior first line to the matrix SRM_0, containing the attribute added to M_0 in order to obtain M_1. The matrix SRM_2 for the mutant M_2 is created through adding a prior first line to the matrix SRM_1, containing the attribute added to M_1 in order to obtain M_2, and so on. This phase of SRM_k construction is represented by the Equation 2,

$$where\ 0 \le k \le n: SRM_k = \begin{cases} SRM_0 & if\ k = 0 \\ \begin{bmatrix} SRM_{k-1} \\ l_k \end{bmatrix} & if\ 1 \le k \le n \end{cases} \tag{2}$$

Such that:

- $\boldsymbol{SRM_0}$ is the matrix of shared resources of corresponding to the mutant M_0, with dimension $0 \times m$.
- $\boldsymbol{l_k}$ represents the new line corresponding to the attribute added to the mutant M_k.
- $\boldsymbol{SRM_k}$ is the matrix of shared resources with dimension $k \times m$.

During the creation of the shared resources matrices, we enclose the analysis to detect the possible covert channels in each mutant. Thus, SRM_k represents the matrix of the shared resources, with the possible covert channels, corresponding to the mutant M_k. In total, there are $n+1$ matrices for shared resources, such that the last matrix SRM_n corresponds to the overall analyzed system.

To sum up, our approach is divided into levels such that each level includes a mutant and its related shared resources matrix. Equation 3 describes the composition of each level L_k,

$$where\ 0 \leq k \leq n: \quad L_k = (M_k, SRM_k) \tag{3}$$

5. Evaluation

5.1. Description of the Evaluated System

Our approach is applied on the same information system used by [3, 9], and we subdivide the presentation of this system into two parts: the first one concerns the attributes of its objects, while the second concerns the primitives.

5.1.1. The Attributes of the System Objects: The analyzed information system contains three types of objects: the processes, the files and the current process. Table 1 shows the attributes of each object within the analyzed system.

Table 1. The Attributes of the Objects within the Analyzed Information System

Object	Attribute	Definition
Process	ID_Process	The unique identifier of the process.
	Access_Rights	A set of elements, such that each element is represented by a couple composed of: 1. A field "*Read/Write*": this field has three possibilities: Read, Write or Read-and-Write. 2. *Security_Class*: is the list of the objects (files) able to be accessed by the process, with one of the access rights in the first field.
	Buffer	The temporary file of the process.
File	ID_File	The unique identifier of the file.
	Security_Classes	The set of processes having any access to the file.
	Locked_By	Contains the ID_Process that currently open this file in write mode.
	Locked	It has two values: *True* if the file is locked and *False* otherwise.
	In_Use_Set	The list of processes that open this file in read mode.
	Value	The content of the file.
Current Process	Current Process	Contains the active current process in use within the system.

We say that a process has the right to read a file, if:

- this process is among the elements of the attributes "Security_Classes" of the specified file.

- this file belongs to one of the elements of the attribute "*Access_Rights* ={*Security_Classes, Read/Write*}" of the specified process, in such a way that it figures among the objects of the component "Security_Classes", and that the value of the second field corresponds to "Read" or "Read-and-Write".

The access right to write on a file for a process is defined in the same way.

Considering the process *P1* with the set of *Access_Rights* shown in Table 2, such that the File5, File6 and File13 have the process *P1* as one of the elements of "*Security_Classes*" attribute. In this case, we can say that the process *P1* has access to:

- read the files: File5 and File13;
- write in the files: File6.

Table 2. The Set of Access_Rights of the Process P1

	Security_Class	Read/Write
Access_Right1	File1, File4, File5	Read
Access_Right2	File3, File6	Write
Access_Right3	File2, File9, File10, File13	Read-and-Write

5.1.2. The Primitives of the Information System: The analyzed information system contains eight primitives, described in Table 3.

Table 3. The Primitives of the Analyzed System

Primitives	Role
Write_File	It is a primitive used by a process to modify the content of a file. If the file is "Locked_By" the current process, then the file "value" is modified by the content of the contemporary file "Buffer" relative to the current process.
Read_File	It is a primitive used by a process to read the content of a file. If the current process belongs to the attribute "In_Use_Set" of the specified file, then, the "value" of this file is copied in the buffer of the current process.
Lock_File	It is a primitive used by a process to modify the content of a particular file. A process must lock the file before modifying it, and unlock the file once the modification is performed. If the current process has the access to read in a specific file, if the specified file is unlocked and its attribute "In_Use_Set" is empty, then, the file is locked and his attribute "Locked_By" has the "ID_Process" of the current process.
Unlock_File	It a primitive that makes a file accessible when a process has finished the modification of its content. If the attribute "Locked_By" of the specified file contains the "ID_Process" of the current process, then, the file is unlocked.
Open_File	It a primitive used by a process to know the content of a file. It ensures that no process can modify the relevant file. If the current process has access to read in the specified file which is not locked, then, the "ID_Process" of the current process is added to the attribute "In_Use_Set" of this file.
Close_File	It a primitive used when a process has finished the reading of a file, and wants to release it in order to be modified. If the "ID_Process" of the current process is an element of the attribute "In_Use_Set" of the specified file, then, it is removed from this attribute.

File_Locked	It a primitive used by a process to determine whether a file is locked. If the current process has the access to write in a specific file, and this file is locked, then, the value "True" is returned. If the file is unlocked, then the value "False" is returned. If the current process has no access to write in the file, then, the result is undefined.
File_Opened	It a primitive used by a process to determine whether a file is opened for reading. If the current process has the access to read in a specific file, and the attribute "In_Use_Set" of the specified file is not empty, then, the value "True" is returned. If the attribute "In_Use_Set" of the file is empty, then, the value "False" is returned. f the current process has no access to read in the file, then, the result is undefined.

5.2. Testing

In this section, we consider the notations in Table 4.

Table 4. Notations used in the Testing

Notation	Definition	Notation	Definition	Notation	Definition
IP	ID_Process	*AR*	Access_Rights	*B*	Buffer
IF	ID_File	*SC*	Security_Classes	*LB*	Locked_By
L	Locked	*IUS*	In_Use_Set	*V*	Value
WF	Write_File	*RF*	Read_File	*LF*	Locke_File
UF	Unlock_File	*OF*	Open_File	*CF*	Close_File
FL	File_Locked	*FO*	File_Opened	*1*	Legitimate Channel
2	Useless data channel	*3*	Channel with same sending and receiving process	*4*	Potential storage covert channel

The first mutant is M_0, which represents our system without attributes. The matrix of shared resources SRM_0 of this mutant is represented in Table 5.

Table 5. The Matrix of Shared Resources SRM_0 of the Mutant M_0

Attributes	Primitives							
	WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**

While applying the operator AA on the mutant M_0 to get the mutant M_1, we add the attribute "*ID_Process*". This later is not used by any primitive, which means it cannot be referenced or modified. Table 6 represents the matrix of shared resources SRM_1.

Table 6. The Matrix of Shared Resources SRM_1 of the Mutant M_1

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								

While applying the operator AA on the mutant M_1 to get the mutant M_2, the attribute added is "*Access_Rights*" which is referenced by the primitives: "*Lock_File*", "*Open_File*", "*File_Locked*" and "*File_Opened*". Then, we insert "R" (Referenced) in the cells having these primitives as columns and the attribute "*Access_Rights*" as line. Table 7 represents the matrix of shared resources SRM_2.

Table 7. The Matrix of Shared Resources SRM_2 of the Mutant M_2

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR			R		R		R	R

While applying the operator AA on the mutant M_2 to get the mutant M_3, the attribute added is "*Buffer*" which is reference by the primitive "*Write_File*" and modified by the primitive "*Read_File*". Thus, we insert "R" and "M" respectively in the cells having the primitives "*Write_File*" and "*Read_File*" as columns and the attribute "*Buffer*" as line.

Table 8. The Matrix of Shared Resources SRM_3 of the Mutant M_3

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR			R		R		R	R
	B	R	M						

Table 8 represents the matrix of shared resources SRM3.

At this stage, we notice in SRM_3 the existence of two values "R" and "M" in the line of the attribute "*Buffer*". Therefore, it is necessary to discover the existing covert channels in the mutant M_3. During the evaluation, the detection of covert storage channels is satisfying, but concerning the covert timing channels the same procedure is used taking into consideration the time factor.

In SRM_3, the attribute "*Buffer*" is modified by the primitive "*Read_File*". Every current process making use of this primitive must be a member of the set "*In_Use_Set*" of the file, in order to modify the attribute "*Buffer*". Accordingly, the process needs the access to read in the specified file. Before continuing on the example, we think it necessary to explain the different and possible situations between the sending and receiving processes within a communication channel:

1. The sending and receiving processes have respectively the right to write and read in a file. Consequently, there is an authorized communication channel by the access control mechanism, since both processes use the file as a communication medium in the identical way as the detected channel, as shown in Figure 3-A.

2. The sending and receiving processes hold the right to write in a file. Consequently, there is a communication channel that can be a potential covert storage channel, since both processes get access to the file in a different way than the detected channel, as shown in Figure 3-B.

3. The sending and receiving processes hold the right to read in a file. Consequently, there is a communication channel that can be a potential covert storage channel, since

both processes get access to the file in a different way than the detected channel, as shown in Figure 3-C.

4. The sending and receiving processes have respectively the right to read and write in a file. Consequently, there is a communication channel susceptible to be a potential covert storage channel, since both processes get access to the file in a different way than the detected channel, as shown in Figure 3-D.

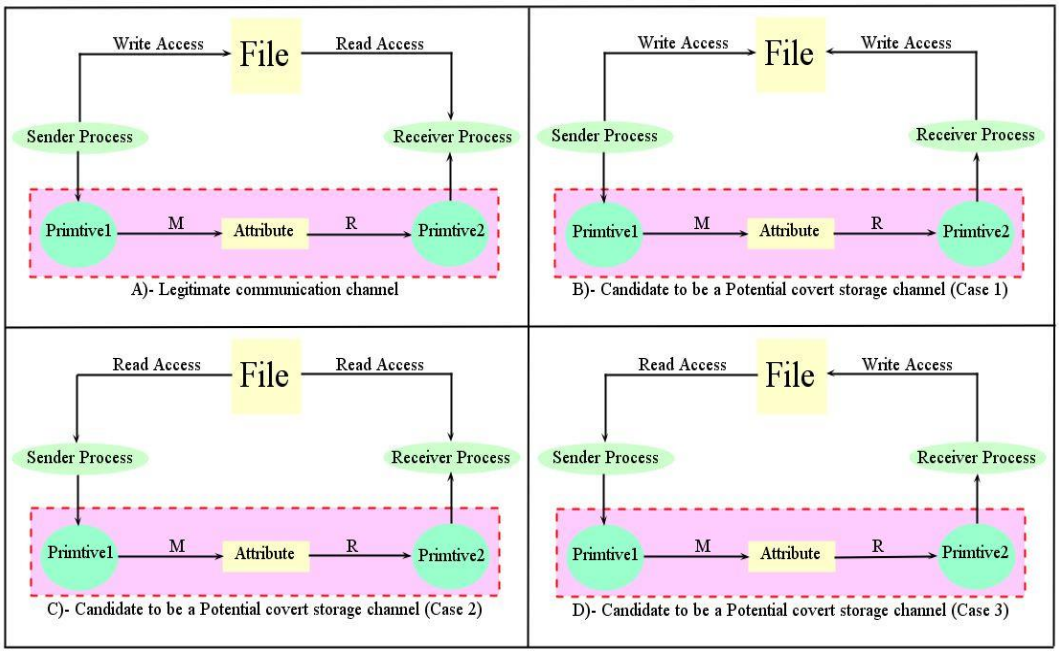

Figure 3. The Four Possible Cases of Sending and Receiving Processes within a Communication Channel

In case the attribute "*Buffer*" is modified by the primitive "*Read_File*", therefore, the sending process, which uses this attribute to send data, needs to get the right to read in this file. Thus, all primitives must be considered in the research of potential covert storage channels. However, we consider only the two last cases of possible covert channels, shown in Figure 3, because at this stage the sending process has only the right of reading.

Concerning the only primitive "*Write_File*" that references the attribute "*Buffer*", every current process using this primitive must first lock the file (*i.e.*, its ID_process must be in the attribute "*Locked_By*"). Consequently, the process needs the right to write in the specified file, because the locking of file by a process could be possible only if this process has the right to write in this file (case 4, Figure 3-D).

Within information systems, the temporary entities (files or attributes) are used by the processes during their executions, in such a way that each process has the right to read and write in its own temporary elements, which are automatically removed once the execution finishes. In our case, we have two processes: the first is the sending process that uses the primitive "*Read_File*" to modify the attribute "*Buffer*", while the second is the receiving process that uses the primitive " *Write_File*" in order to reference the attribute "*Buffer*". However, since both processes are different and are used by two different users, it is impossible for them to share the same temporary attribute "*Buffer*". In other words, if an information is shared, this can be only possible for a one process which is at the same time a sending and a receiving process. Table 9 summarizes the analysis phase and the detection of covet channels in the mutant M_3.

Table 9. The Matrix of Analysis and Detection of Covert Channels in M_3

Shared Attributes	Primitives observing the Attribute Modification							
	WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
B	(3,RF)							

We return to the creation of mutants. In order to create the mutant M_4 we apply the operator AA on the mutant M_3. Thus, the attribute "*ID_File*" is added and it is not used by any primitive, which means it cannot be referenced or modified. Table 10 represents the matrix of shared resources SRM_4.

Table 10. The Matrix of Shared Resources SRM_4 of the Mutant M_4

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR			R		R		R	R
	B	R	M						
File	IF								

The analysis and detection of covert channels in the mutant M_4 follow the same procedure performed on the mutant M_3.

In order to create the mutant M_5 we apply the operator AA on the mutant M_4. Thus, the attribute "*Security_Classes*" which is referenced by the primitives "*Lock_File*", "*Open_File*", "*File_Locked*" and "*File_Opened*". Then, we insert "R" in the cells having these primitives as columns and the attribute "*Security_Classes*" as line. Table 11 represents the matrix of shared resources SRM_5.

Table 11. The Matrix of Shared Resources SRM_5 of the Mutant M_5

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR			R		R		R	R
	B	R	M						
File	IF								
	SC			R		R		R	R

The analysis and the detection of covert channels in M_5 the mutant follow the same procedure performed in the mutant M_4, because the attribute "*Security_Classes*" is only referenced by primitives.

While applying the operator AA on the mutant M_5, we construct the mutant M_6 through adding the attribute "*Locked_By*", which is referenced by the primitives "*Write_File*" and "*Unlock_File*". Thus, we insert "R" in the cells having these primitives as columns and the attribute "*Locked_By*" as line. This attribute is modified by the primitive "L*ock_File*", then we insert "M" in the cell having the primitive "L*ock_File*" as column and the attribute "*Locked_By*" as line. Table 12 represents the matrix of shared resources SRM_6.

Table 12. The Matrix of Shared Resources SRM_6 of the Mutant M_6

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR			R		R		R	R
	B	R	M						
File	IF								
	SC			R		R		R	R
	LB	R		M	R				

In case the attributes are referenced by a primitive and can be modified by another one referencing traditional attributes, thus, these attributes must be referenced by the original primitive. In order to avoid such situations, we makes use of **The Transitive Closure** [9].

The transitive closure of the matrix is generated through the research of each input with "R". If there is "M" in the line where this input appears, then it is necessary ti verify the column containing this "M" to see if it references an attribute, which is not referenced by the original primitive (original column). In other words, if the column containing "M" has "R" in a line, where normally this "R" does not appear in the line corresponding to the original column, then, another "R" must be added in the line of the original column. This procedure is repeated until no input can be added to the matrix of shared resources.

During the analysis and the detection of covert channels in a mutant whose matrix of shared resources is issued using the transitive closure, we consider only the primitives which are modified at least by one primitive. Following the application of transitive closure on SRM_6, we get a new matrix illustrated in Table 13.

Table 13. The Matrix of Shared Resources SRM_6 of the Mutant M_6 Issued by the Transitive Closure

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR	**R**		R	**R**	R		R	R
	B	R	M						
File	IF								
	SC	**R**		R	**R**	R		R	R
	LB	R		M	R				

Since four references were added, then, the attribute "*Locked_By*" is only modified by the primitive "*Lock_File*" which requires that the process executing this primitive must have the access right to write in the file. Thus, the sending process must belong to the set of processes that hold the right to write in the specified file. Our attribute is referenced by two primitives "*Write_File*" and "*Unlock_File*", consequently, it is necessary to verify for each reference if it can be produced when the current process belongs to the set of processes which have not the access right to read (Figure 3-B). The primitive

"*Write_File*" can be executed by the current process on a file, if this later is locked by this process (appear in the attribute "*Locked_By*" of the file) which must have the right to write in this file. Thus, the current process, which uses the primitive "*Write_File*", does not require the access right to read, hence the possibility of existence of potential covert storage channels (Figure 3-B). However, the current process is the same process that has modified and referenced the attribute, since the process which writes in the file must be the same that locks it , we conclude that we have the case where the sending and receiving processes of this channel are the same. Using the same argumentation for the primitive "*Unlock_File*", we get the same situation. Table 14 summarizes the analysis and the detection of covert channels for the mutant M_6.

While applying the operator AA on the mutant M_6, we construct the mutant M_7 through adding the attribute "*Locked*", which is referenced by the primitives "*Write_File*", "*Lock_File*", "*Unlock_File*", "*Open_File*" and "*File_Locked*". Thus, we insert "R" in the cells having these primitives as columns and the attribute "*Locked*" as line. This attribute is modified by the primitives "L*ock_File*" and "*Unlock_File*", then we insert "M" in the cell having these primitives as column and the attribute "*Locked*" as line. Table 15 represents the matrix of shared resources SRM_7 with the application of transitive closure

Table 15. The Matrix of Analysis and Detection of Covert Channels in M_6

Shared Attributes	Primitives observing the Attribute Modification							
	WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
B	(3,RF)							
LB	(3,LF)			(3,LF)				

Table 14. The Matrix of Shared Resources SRM_7 of the Mutant M_7 Issued by the Transitive Closure

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR	**R**		R	**R**	R		R	R
	B	R	M						
File	IF								
	SC	**R**		R	**R**	R		R	R
	LB	R		**R,M**	R	**R**		**R**	
	L	R		R,M	R,M	R		R	

Before dealing the attribute "*Locked*", we return to the attribute "*Locked_By*" because three transitive references were added to this attribute. Since the attribute "*Locked_By*" is modified only by the primitive "*Lock_File*", then, another process that exploits this process must have the right to write to modify this attribute. Consequently, the sending process has the access right to write in the file.

The transitive reference of the primitive "*Lock_File*" is generated due to a reference and modification of the attribute "*Locked*", respectively by the primitives "*Lock_File*" and "*Unlock_File*". Moreover, the presence of a reference for the attribute "*Locked_By*" in the column of the primitive "*Unlock_File*", allows in turn to reference the attribute "*Locked_By*" is the column of the primitive "*Lock_File*". Thus, the transitive reference of

the attribute "*Locked_By*" by the primitive "*Lock_File*" allows to know the process that lock the specified file. This corresponds to the situation of a channel where the sending and receiving processes use the same primitive "*Lock_File*" to share the attribute "*Locked_By*". Thus, both processes must have the access right to write, but the only information transferred by this channel is that "*the last process which has unlocked the file is the same that has locked it*". Consequently, it is the case where no useful information can be transmitted through this channel.

Similarly, the channel where the sending process uses the primitive "*Lock_File*" and the receiving process uses the primitive "*File_Locked*", in order to share the attribute "*Locked_By*", is in the case where no useful information can be transmitted through this channel.

The channel where the sending process uses the primitive "*Lock_File*" and the receiving process uses the primitive "*Open_File*", in order to share the attribute "*Locked_By*", is in the case where a legitimate channel exists between the sending and receiving processes. This result can be explained due to the fact that the sending process needs the access right to write in the file, and that the receiving process uses the primitive "*Open_File*", thus, this process must have the access to read in the specified file in order to reference the attribute "*Locked_By*" (Figure 3-A). However, the sending and receiving processes of this channel can directly communicate through this file. That is to say that between these two processes there is a legal communication channel authorized by the access control mechanism.

Actually, we proceed to the detection of covert channels that share the attribute "*Locked*", which is modified by the two primitives "*Lock_File*" and "Unl*ock_File*". A process that executes one of these two primitives necessitates to hold the right to write in the file, then, the sending process must exist in the set of processes with the access right to write in the specified file. Our attribute is referenced by five primitives: "*Write_File*", "*Lock_File*", "*Unlock_File*", "*Open_File*", "*File_Locked*". Therefore, it is necessary to verify, for each reference, if it can be produced when the current process (receiving process) belongs to the set of processes which do not have the right to read (Fig. 3-B).Thus, with this attribute we find the same cases of channels that share the attribute "*Locked_By*", except for the primitives "*Lock_File*" and "*File_Locked*". Then, we explain these two cases.

The primitive "*Lock_File*" (or "*File_Locked*") references the attribute "*Locked*" of a file to know if it is locked or not. The only condition needed for a current process to use this primitive is to hold the access right to write in the file (Figure 3-B). Hence, each process having the right to write in the file can detect information, even locked or not. Consequently, we can meet the case of a potential storage covet channel that exploits the attribute "*Locked*".

The following scenario shows that a reference by the primitive "*Lock_File*" (or "*File_Locked*") on the attribute "*Locked*" may be used by a storage channel. If this attribute contains the value "*False*" (the file is unlocked), then, a process with the right to write in a file, can send **1** when executing the primitive "*Lock_File*" and **0** when executing "*Unlock_File*", while the attribute "*Locked_By*" contains the ID_Process or without executing any primitive. Thus, another process with only the right to write in the specified file, can determine the values sent when executing the primitive "*Lock_File*" (**0** if succeeded and **1** if failed). It is worth to notice the possibility that the file is locked, because in this case the primitive "*Lock_File*" can not return explicitly the success of failure, then the process must utilize "*File_Locked*" to verify the result. Applying this scenario for numerous files authorized for writing to the sender and receiver, then, a covert channel with large bandwidth is detected.

At this stage, we have detected two covert channels that share the attribute "*Locked*", whose sending and receiving processes have the right to write in the specified file, in a such way that: the first channel has a sending process using the primitives "*Lock_File*" and "*Unlock_File*" and a receiving process using the primitive "*Lock_File*", while the second channel has a receiving process using the primitive "*File_Locked*" and a sending process using the primitives "*Lock_File*" and "*Unlock_File*". Table 16 illustrates the analysis and the detection for the mutant M_7.

Table 16. The Matrix of Analysis and Detection of Covert Channels in M_7

Shared Attributes	Primitives observing the Attribute Modification							
	WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
B	(3,RF)							
LB	(3,LF)		(2,LF)	(3,LF)	(1,LF)		(2,LF)	
L	(3,LF) (3,UF)		(4,LF-UF)	(3,LF) (3,UF)	(1,LF) (1,UF)		(4,LF-UF)	

We continue the same procedure for the remaining mutants M_8, M_9 and M_{10}. At the end, we obtain the shared resources matrix of the overall system and which corresponds to the SRM_{10} of the last mutant M_{10} issued after the application of the transitive closure as showed in Table 17.

Table 17. The Matrix of Shared Resources SRM_{10} of the Mutant M_{10} Issued by the Transitive Closure

Attributes		Primitives							
		WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
Process	IP								
	AR	**R**	**R**	R	**R**	R	R	R	R
	B	R	R,M						
File	IF								
	SC	**R**	R	R	**R**	R	R	R	R
	LB	R	R	**R,M**	R	**R**	R	**R**	R
	L	R	R	R,M	R,M	R	R	R	R
	IUS	**R**	R	R	**R**	R,M	R,M	**R**	R
	V	R,M	R						
Current_Process		R	R	R	R	R	R	R	R

Table 18. The Matrix of Analysis and Detection of the Overall System

Shared Attributes	Primitives observing the Attribute Modification							
	WF	**RF**	**LF**	**UF**	**OF**	**CF**	**FL**	**FO**
B	(3,RF)							
LB	(3,LF)	(1,LF)	(2,LF)	(3,LF)	(1,LF)	(2,LF)	(2,LF)	(2,LF)
L	(3,LF) (3,UF)	(1,LF) (1,UF)	(4,LF-UF)	(3,LF) (3,UF)	(1,LF) (1,UF)	(1,LF) (1,UF)	(4,LF-UF)	(2,LF) (2,UF)
IUS	(2,OF) (2,CF)	(3,OF) (3,CF)	(4,OF-CF)	(2,OF) (2,CF)	(2,OF) (2,CF)	(2,OF) (2,CF)	(2,OF) (2,CF)	(4,OF-CF)
V	(3,WF)	(3,WF)						

Table 18 illustrates the matrix of analysis and detection of covert channels corresponding to the overall system.

6. Conclusion

Covert channels are viewed as a pertinent problem, that expresses serious obstacles for the majority of access control models, and which necessitates efficient solutions to deal with it. Our proposed solution is an evolution and improvement of shared resources matrix method, where it becomes possible to detect two types of covert channels: storage and temporal.

The obtained results are satisfactory and show a highly competitive performances, especially that they refer to real evaluation on a real information system.

As perspective, we would like to extend our approach in such a way it can proceed automatically to the execution of the mentioned steps one by one, which ensures a highly dynamic method able to evolve according to the development and modification of the information system.

References

[1] B. W. Lampson, "A Note on the Confinement Problem", Commun. ACM, vol. 16, no. 10, **(1973)**, pp. 613-615.
[2] AA. El Kalam, "Security policies and models for Health Care Computing and Communication Systems", Theses, Institut National Polytechnique de Toulouse - INPT, **(2003)**, Décembre.
[3] R. A. Kemmerer, "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels", ACM Trans. Comput. Syst., vol. 1, no. 3, **(1983)**, pp. 256-277.
[4] DoD 5200.28-STD, "Trusted Computer System Evaluation Criteria", Dod Computer Security Center, **(1985)**, December.
[5] J. A. Goguen and José Meseguer, "Security Policies and Security Models", In Security and Privacy, 1982 IEEE Symposium on, **(1982)** April, pp. 11-20.
[6] J. Rushby, "Mathematical foundations of the MLS Tool for Revised Special. Rapport technique", SRI International, **(1984)**.
[7] J. Shen, S. Qing, Q. Shen et and L. Li, "Optimization of covert channel identification", In Security in Storage Workshop, 2005. SISW'05. Third IEEE International, **(2005)** Dec., pp. 13-95.
[8] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer", Computer, vo. 11, no. 4, **(1978)** April, pp. 34-41.
[9] R. A. Kemmerer, "A practical approach to identifying storage and timing channels: Twenty years later", In Proceedings of the 18th Annual Computer Security Applications Conference, ACSAC '02, pp 109 , Washington, DC, USA, IEEE Computer Society, **(2002)**.

Authors

Mohammed Ennahbaoui, He received her Master's degree in Codes, Cryptography and Information Security from Mohammed V University in Rabat, Faculty of Sciences, Morocco. He is actually a PhD student in the laboratory of Mathematics Computing and Applications (LabMIA) in the University of Mohammed V in Rabat. His research interests include: information security, cryptography, Access Control, distributed computing.

Said El Hajji, He is a full professor at the Mohammed V University in Rabat,, Morocco, and the head of the laboratory of Mathematics, Computing and Applications (LabMIA). His research interests include: information security, cryptography, error correcting codes, Mathematics.

