A tool supported methodology to passively test asynchronous systems with multiple users

Mercedes G. Merayo[a], Robert M. Hierons[b], Manuel Núñez[a,*]

[a] *Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain*
[b] *Department of Computer Science, Brunel University London, Uxbridge, Middlesex, UB8 3PH United Kingdom*

Abstract

Context: Testing usually involves the interaction of the tester with the system under test. However, there are many situations in which this interaction is not feasible and so one requires a *passive* approach in which the system is analysed to look for failures or unexpected behaviours. The entities of a complex system usually communicate in an asynchronous manner and this complicates the testing tasks since the observed order of events need not be the same as the order in which the events were produced. In previous work, we presented a formal passive testing theory for a single user and system communicating through an asynchronous channel. We were able to check that a trace generated by the system satisfies a given property.
Objective: This papers extends our work, for detecting potential intrusions and unexpected behaviours, to the case where many users simultaneously communicate with a central server. We evaluate the practical value of the theoretical framework with a non-trivial system.
Method: We developed a novel complete theoretical framework to analyse asynchronous systems with multiple users. All the algorithms are fully implemented. Experiments were performed over the Nextcloud platform to show the applicability of our framework. The experiments include an attack so that actual vulnerabilities could be revealed.
Results: The application of our methodology, supported by a tool fully implementing it, was able to reveal a vulnerability in the WebDAV protocol running on Nextcloud.
Conclusion: The extension of our previous work is not only useful from a theoretical point of view but also increases the applicability of the original work. We are now able to analyse systems where the interaction with different users can lead to unexpected behaviours. We have been able to find a vulnerability in a real system, showing the usefulness of our work.

1. Introduction

1.1. Motivation

The development of reliable software depends on the use of appropriate verification and validation methods, with testing being one of the most widely used approaches [33]. The application of testing techniques has traditionally been manual and so expensive and error prone. This has led to the development of approaches that automate parts of the testing process, with some of the most promising approaches being based on methods that *formalise* aspects of software testing [13].

The usual understanding of the testing process is that testers apply inputs to the system under test (SUT), receive outputs and establish whether the obtained answers are the expected ones. However, there are situations where this schema cannot be applied. For example, we might have restricted access due to security issues or because the system is running 24/7 and our interaction might produce undesirable changes in the associated data. In this case we have to use a *passive* approach where we observe the interactions between the system and its users to try and detect unexpected or unwanted situations. In formal active testing we usually assume that the tester has a complete specification of the SUT and derives tests from it. However, in passive testing this is not always the case and it is more often assumed that the (passive) tester only has a certain set of properties that the observation of the SUT cannot violate. In other words, we have a property (or set of properties) and we check that the trace being observed satisfies that property. Ideally, we want the process of checking whether the property is satisfied to be quick and to take very little storage since this can allow passive testing to occur in real-time. The application of passive testing in real-time has an important benefit: a detected error can be notified to the operators of the system almost immediately and they can then take appropriate measures.

The initial work on passive testing assumed that the monitor that observes and checks the behaviour of the SUT observes the trace actually produced. However, this

*Corresponding author.
Email addresses: mgmerayo@fdi.ucm.es (Mercedes G. Merayo), rob.hierons@brunel.ac.uk (Robert M. Hierons), manuelnu@ucm.es (Manuel Núñez)

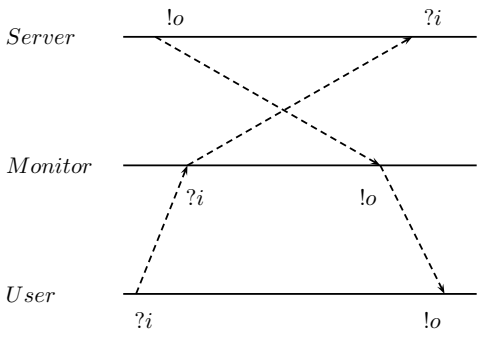

Figure 1: Different perception: monitor vs. server

assumption is unrealistic in situations in which there is asynchronous communications between the parties. Let us suppose, for example, that a user communicates with a server and we place a monitor between them (see Figure 1). If the user sends a message (that is, an input) $?i$ then the monitor will observe $?i$ before $?i$ is received by the server. Similarly, if the server sends a message (that is, an output) $!o$ then the monitor will observe $!o$ after $!o$ was actually produced. Therefore, it may happen that the server produced the trace $!o?i$ but the observer has the perception that the actual trace was $?i!o$. This situation can lead to false positives (the monitor thinks that the server is faulty but it is not) and false negatives (the monitor does not realise that there has been a faulty behaviour).

1.2. Contributions of the paper

In previous work we developed a passive testing theory for systems with asynchronous communications [19, 20], with the theoretical framework being completely supported by the PTTAC tool [5]. However, in experimenting with the framework, in particular, analysing different systems and protocols, we detected a weakness: we cannot appropriately handle systems where many users interact with a central server. The main problem is that users use different instances of the same message to communicate with the server and we need to distinguish between these when checking a property. Essentially, instead of simple input/output actions, we will also consider information about the party producing/receiving the action. This is similar to the role played by ids in datagram protocols. Therefore, our actions will be pairs (a, α) where a is either an input or an output and α is the user sending, respectively receiving, the action. In addition, our former framework processed the observed trace in a sequential way while now we need to skip actions of users not involved in the current matching, due to different instantiations, of a property. For example, consider the property $(?a, x)(!b, y), \{(!c, y)\}$ whose intended meaning is that if a certain user sends an input $?a$ and a different user (if we have different value names then we assume that they are instantiated to different users) receives an output $!b$ then the next output received by the second user must be $!c$. If we observe the sequence $(?a, u_1)(!b, u_2)(!d, u_3)(!c, u_2)$, then we cannot say that we observed an erroneous behaviour. Similar to our previous work, we assume that the SUT and monitor interact through a FIFO channel[1] but there are no such restrictions on how the monitor and users interact. In this paper we present a complete framework to deal with this type of systems and properties. In Figure 2 we present a schematics of part of a longer interaction between three users and a server. This schema shows our assumption about how the communications between the different entities take place. The users communicate with the server (SUT) via an asynchronous channel. The monitor is placed between users and server. Therefore, the monitor can observe messages in a different order with respect to the order in which they were actually produced. For example, in Figure 2 the event $(!o', u_1)$ is produced before the event $(?i, u_3)$ but they are observed in reverse order.

We have extended the PTTAC tool accordingly to the new framework. In order to assess the usefulness of our extension we used a real case study where multiple users communicate through a server: Nextcloud. We studied the protocols underlying this platform and we decided to focus on the WebDAV protocol, identifying some usual properties that the exchanges of messages should fulfil, expressing them in our framework and analyzing captured traces to check whether the properties are violated.

The rest of this paper is structured as follows. In Section 2 we review related work. In Section 3 we present our formal framework. In Section 4 we introduce an algorithm to detect suspicious behaviours in observed traces. In Section 5 we describe how the proposed technique and tool were used to analyse some properties of the WebDAV protocol in a real environment. Finally, in Section 6 we present our conclusions and some potential lines of future work.

2. Related work

In this section we briefly review related work on passive testing and related areas, in particular, we will mention the relation between passive testing and runtime monitoring and we will review some tools to perform passive testing.

There has been significant interest in approaches that combine formal methods and testing, associated tools that automate testing activities are widely available [42], there are several surveys [7, 17, 18, 49], and there is significant evidence of industrial uptake [3, 4, 14, 22, 38].

Formal passive testing is already a well established line of research and extensions of the original frameworks [2, 6, 24] have dealt with issues such as security and trust [26, 31, 44, 48].

[1] This might, for example, result from the SUT and monitor being on the same local network or the monitor being placed at the gateway between the local network that contains the SUT and another network.

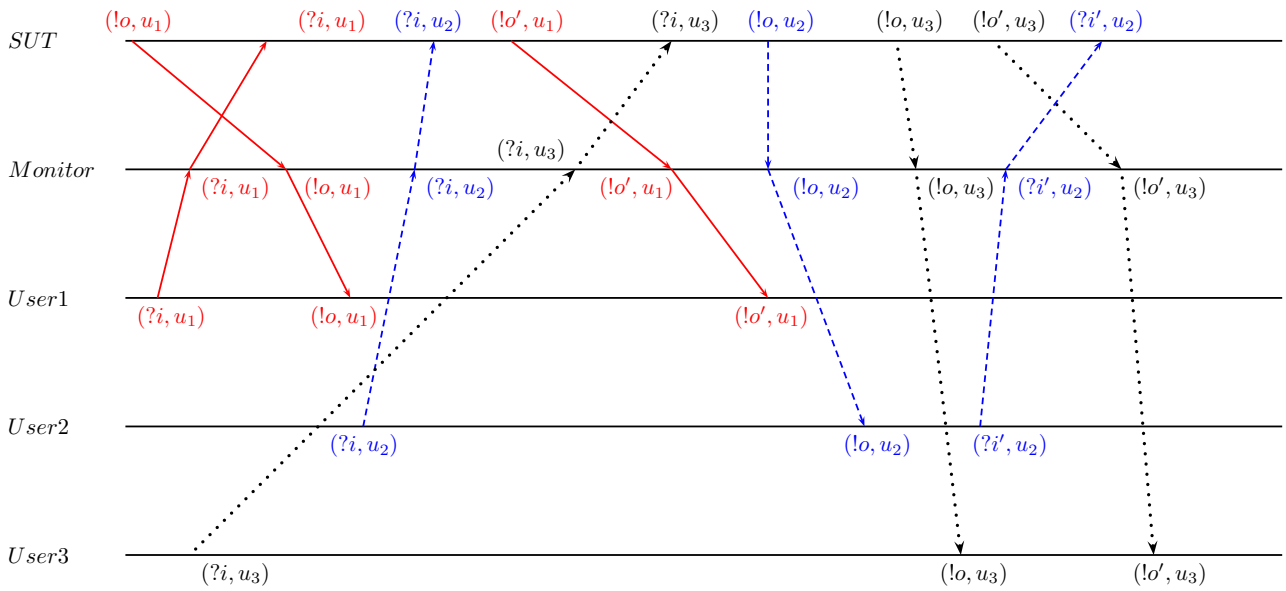

Figure 2: Multiple channels between the monitor and users

We are not aware of previous work on formal passive testing where there is an asynchronous communication channel between the system and the monitor, other than ours [19, 20, 29]. In contrast, there has been some work on active testing and several approaches have appeared in the literature for models where there is a distinction between inputs and outputs [15, 16, 21, 37, 45, 53].

Homing algorithms are used in many approaches to passive testing, in particular in the seminar contributions of the 1990s [24, 46], there exist surveys on homing algorithms [40] and this is still an active research area [50]. A homing algorithm applies a sequence of actions to a system so that it brings it to its initial state (or to a certain *special* state where we know that we can start testing the SUT). Homing is not used in our framework. In order to apply homing we need to have a complete specification of the system so that, by comparing the observed behaviour of the SUT with the specification, we can *deduct* that the SUT is in the desired state. However, we do not assume the existence of a complete specification: we consider that we are given a set of properties and we check that the SUT does not violate them.

There are two main lines to express properties in passive testing. One of them uses LTL safety formulas and creates checking automata by translating formulas to Buchi automata [39]. The use of LTL and other temporal logics is also the usual way to define properties in runtime verification. The second one considers properties expressing relations like "if we observe such a trace then the following action must belong to a certain set" [2, 6, 31]. In this paper we consider this second approach because it is closer to *usual* testing terminology and it is widely used in passive testing.

The main problem, from the theoretical point of view, to define a passive testing framework of asynchronous systems is that events might be observed by the monitor in a different order with respect to the one in which they were sent/received by the SUT. If we are working in a system where messages are timestamped then we can use this information to decide the right order of events. A recent work [29] uses this approach and considers that a potential swapping is admissible only if the difference between the timestamps is below a certain threshold error (due to a wrong synchronization of the local clocks). In this paper we do not assume time information and, therefore, we cannot discard potential swaps.

Passive testing is a monitoring technique and as such it is related to runtime verification. They share the same goal, checking the correctness of a system without interacting with it, but use different formalisms and methodologies. In runtime verification it is not usual to distinguish between inputs and outputs, since their observation makes them events of the same nature, and therefore it is difficult to compare the work from that area with ours. While some work has investigated asynchronous runtime monitoring [10, 41], the problems considered in this context are different: this line of work does not distinguish between input and output and does not explore potential reorderings of traces. Instead, it looks at the situation in which the monitor and system do not synchronise on actions: actions engaged in by the system might instead be recorded and analysed later. Once said this, it is worth noting that the *compensation* mechanism [10] can be used to partially simulate reorderings. The idea is that if we observe an action that does not match our property (because it was observed before it was expected by the property) then we can later *cancel* this occurrence with the complement of the observed action. Similarly, the interleavings considered in

the *last known position* mechanism [41] could also be used to deduct the reorderings allowed by a certain property.

We have to mention work on using passing testing for the validation of protocols [23, 32]. In our approach we focus on detecting unexpected behaviours, or behaviours violating some established rules, in the interaction of the users with the system and we do not look for errors in the actual protocol.

Finally, we would like to review some tools implementing passive testing approaches. Other than our previous version of PTTAC [5], passive testing tools do not deal with asynchronous frameworks. Since there are many tools, we will concentrate on those using a formalism similar to ours and/or dealing with security and vulnerabilities. The notion of property that we use in this paper originates from previous work on EFSMs [2, 6]. These papers introduced tools to implement their approaches and applied them to analyse the Simple Connection Protocol and the Wireless Application Protocol. Continuations of this work present tools to deal with timed systems [1, 8]. One of these papers introduces the tool PASTE [1] where a novel combination of passive and mutation testing is implemented. The TestInv-Code tool [26, 43] implements a passive testing approach based on *Vulnerability Detection Conditions* to automatically detect vulnerabilities in C code. In order to show its usefulness, Test-Inv-Code has been applied to several C applications and successfully revealed previously known vulnerabilities. MMT is a monitoring tool that has been used to perform security analysis, by inspecting network traffic, and applied to an industrial QoS-aware ad-hoc radio communication protocol [52]. MMT has also being extended to deal with temporal properties and applied to a collaborative programming project [48]. A recent contribution [25], although not properly a tool, introduces the concept of *Software Defined Network* and highlights how it can be used to solve the current limitations in legacy monitoring systems.

3. The methodology

In this section we introduce the main aspects of our formal passive testing methodology that deals with systems that have many concurrent users. In the original proposal [19, 20] we only considered the situation in which a single user communicates with the server. Consequently, the properties represented restrictions over one user. In the new framework we will be able to define new types of properties in which we can include variables that allow us to represent different users.

In this paper, when we refer to a system we will assume that there is a labelled transition system that represents the behaviour of the system. In such a model, transitions are labelled by either an input or an output. Next we introduce some notation that will be used throughout the paper.

Definition 1. *We fix the sets of inputs and outputs as I and O, respectively, and let $\mathcal{A}ct$ be equal to $I \cup O$. In order to distinguish between inputs and outputs we usually precede the name of an input by ? and precede the name of an output by !. We let $\mathcal{U}sers$ be the set of user identifiers. Given $(a, x) \in \mathcal{A}ct \times \mathcal{U}sers$ we have that* $\mathrm{act}(a, u) = a$ *and* $\mathrm{user}(a, u) = u$. *A* user trace *is a sequence σ belonging to the set $(\mathcal{A}ct \times \mathcal{U}sers)^*$.*

In this paper we classify actions from the point of view of the SUT. That is, an input will be an action produced by a user (and received by the SUT) while an output will be an action received by a user (and produced by the SUT). User traces are sequences of inputs and outputs annotated with the identifier of the user from/to which the actions are sent/received. We need to define the set of user traces that might be observed by the monitor if the SUT produces a trace. Since the monitor and server do not synchronise on actions (there is an asynchronous channel between the monitor and the SUT), we need to consider the traces that can be performed by a system and also the traces that can be observed as a result of this. Typically, the monitor will be part of the same network as the SUT and so, as explained in the introduction, we make the realistic assumption that communications between the monitor and SUT are FIFO. In contrast, there is no such restriction on communications between the users and the monitor. Finally, we assume that no packages are lost.

If a system performs a certain trace then we can observe a variation of this trace where the outputs appear later than they were actually performed and the inputs are observed before they arrive at the SUT. This idea is formally defined next.

Definition 2. *Let $\sigma, \sigma' \in (\mathcal{A}ct \times \mathcal{U}sers)^*$ be user traces. We say that σ' is an observation of σ, denoted by $\sigma \rightsquigarrow \sigma'$, if there exist sequences $\sigma_1, \sigma_2 \in (\mathcal{A}ct \times \mathcal{U}sers)^*$, $!o \in O$ and $?i \in I$, $u_1, u_2 \in \mathcal{U}sers$ such that $\sigma = \sigma_1(!o, u_1)(?i, u_2)\sigma_2$ and $\sigma' = \sigma_1(?i, u_2)(!o, u_1)\sigma_2$. We let $\mathcal{L}_u(\sigma)$ denote the set of traces that can be formed from σ through sequences of transformations of the form $\rightsquigarrow$, that is, we have that $\mathcal{L}_u(\sigma) = \{\sigma' | \sigma \rightsquigarrow^* \sigma'\}$, where $\rightsquigarrow^*$ represents the repeated (zero or more times) application of $\rightsquigarrow$. Given a set of traces Φ, we overload the previous notation to define $\mathcal{L}_u(\Phi) = \cup_{\sigma \in \Phi} \mathcal{L}_u(\sigma)$.*

The previous definition introduces an equivalence relation on traces, where certain observations can be exchanged in their order while preserving equivalence. It should be mentioned that this concept comes from the very roots of *trace theory* [27, 28]. In this section we use FTP as a running example [12]. This is a well known and studied protocol and, therefore, it will be easy to follow the intended meaning of the used traces and properties. The list of commands and return codes used in the paper is given in Table 1.

Example 1. *Let us assume that two users u_1 and u_2 try to connect to an FTP server using their valid username*

Service commands	RNFR	It specifies the old pathname of the file which is to be renamed
	RNTO	It specifies the new pathname of the file to be renamed
	DELE	It deletes the file specified in the pathname
	APPE	It causes the server to accept data and to store it in a file at the server site
	LIST	It causes a list to be sent from the server
	ABOR	It aborts the previous FTP service command and any associated transfer of data
	RETR	It transfers a copy of the file specified in the pathname
	USER	User identification required by the server for access to its file system
	PASS	Password associated to the user identification
Reply codes	150	File status okay; about to open data connection
	226	Closing data connection. Requested file action successful
	230	User logged in, proceed
	250	Requested file action okay, completed
	331	User name okay, need password
	350	Requested file action pending further information
	450	Requested file action not taken. File unavailable
	501	Syntax error in parameters or arguments
	503	Bad sequence of commands
	530	Not logged in
	550	Requested action not taken. File unavailable
	553	Requested action not taken. File name not allowed

Table 1: FTP service commands and reply codes used in the paper

and password. Then, the sequence of messages σ that appears in Figure 3 might be produced. Due to the asynchronous nature of the system, the monitor might observe any of the traces in the set $\mathcal{L}_u(\sigma)$.[2]

Next, we formally define our notion of a property to validate traces that include multiple users interacting with a server. Variables will be used to allow the names of users in different pairs in $\mathcal{A}ct \times \mathcal{U}sers$ to be matched and constants will be used in order to require that a particular user is involved in a particular pair. Properties will be of the form (ρ, O_ρ), with this meaning that if the SUT produces the sequence ρ then the next output belongs to O_ρ.

Definition 3. *Let $\mathcal{X}$ be a set of variables. We say that $P = (\rho, O_\rho)$ is a property if $O_\rho \subseteq O \times (\mathcal{X} \cup \mathcal{U}sers)$ and ρ is defined according to the following EBNF:*

$$\rho ::= (a, x)\rho'$$
$$\rho' ::= \epsilon | (a, x)\rho'$$

where $a \in \mathcal{A}ct$ and $x \in \mathcal{X} \cup \mathcal{U}sers$.

Let $P = (\rho, O_\rho)$ be a property. We let $act(\rho)$ and $act(O_\rho)$ (resp. $var(\rho)/cons(\rho)$ and $var(O_\rho)/cons(O_\rho)$) denote the set of actions (resp. variables/constants) appearing in ρ and O_ρ, respectively. We also let $act(P) = act(\rho) \cup act(O_\rho)$, $var(P) = var(\rho) \cup var(O_\rho)$ and $cons(P) = cons(\rho) \cup cons(O_\rho)$.

We say that a property $P = (\rho, O_\rho)$ is well-formed if $|var(P) \cup cons(P)| \leq |\mathcal{U}sers|$, $var(O_\rho) \subseteq var(\rho)$ and $cons(O_\rho) \subseteq cons(\rho)$.

Let $P = (\rho, O_\rho)$ be a property and $f : var(P) \longrightarrow \mathcal{U}sers \setminus cons(\rho)$ be a total injective function. The instantiation of ρ with respect to f is the user trace obtained from the replacement of each variable x in ρ by $f(x)$. The set of all instantiations of ρ is denoted by $\mathrm{Ins}(\rho)$.

We consider properties of the form (ρ, O_ρ). Such a property says that if the SUT produces the sequence ρ then the next output belongs to O_ρ. The previous EBNF expresses that ρ is a non-empty sequence of input and output actions parameterised by variables or associated with specific users. The variables are related to users to/from which the actions are sent/received. Each variable will be associated with the identifier of a different user, that is, if a property has different variables, then they always represent different users. In addition, a variable cannot take a value included in the property as a constant. An instantiation is the trace obtained when the variables in a property are replaced by actual users.

Let us note that in the current framework we do not consider data associated with actions. Therefore, in our properties we have to implicitly assume that the actions are associated with the same objects. For example, if a property indicates that a *delete* should not be followed by an *append*, then we assume that both actions are referring to the same file.

Example 2. *Consider again the communication of users with a server using the FTP. The next property represents*

[2] Note that we do not consider time in this paper. If we did then we should consider that the users should be NTP synchronised and, therefore, most of these traces should be discarded.

$$\sigma = (?USER, u_2)(!331, u_2)(?USER, u_1)(!331, u_1)(?PASS, u_2)(!230, u_2)(?PASS, u_1)(!230, u_1)$$

$$\mathcal{L}_u(\sigma) = \left\{ \begin{array}{l} (?USER, u_2)(!331, u_2)(?USER, u_1)(!331, u_1)(?PASS, u_2)(!230, u_2)(?PASS, u_1)(!230, u_1) \\ (?USER, u_2)(?USER, u_1)(!331, u_2)(!331, u_1)(?PASS, u_2)(!230, u_2)(?PASS, u_1)(!230, u_1) \\ (?USER, u_2)(!331, u_2)(?USER, u_1)(?PASS, u_2)(!331, u_1)(!230, u_2)(?PASS, u_1)(!230, u_1) \\ (?USER, u_2)(!331, u_2)(?USER, u_1)(!331, u_1)(?PASS, u_2)(?PASS, u_1)(!230, u_2)(!230, u_1) \\ (?USER, u_2)(?USER, u_1)(!331, u_2)(?PASS, u_2)(!331, u_1)(!230, u_2)(?PASS, u_1)(!230, u_1) \\ \ldots \end{array} \right\}$$

Figure 3: Possible observations of trace σ

a behaviour that might occur when a user requests that a file is renamed.

$$P_1 = \left(\begin{array}{l} (?RNFR, uid)(!350, uid)(?RNTO, uid), \\ \{(!250, uid), (!553, uid), (!503, uid), (!501, uid)\} \end{array} \right)$$

Once the command has been accepted and the information has been received by the server, the return codes that can be produced are: !250 *(action completed),* !553 *(invalid name / cannot rename file),* !503 *(cannot find the file which has to be renamed) or* !501 *(syntax error).*

The next property involves two different users represented by two variables uid_1 and uid_2. The property represents the sequence of actions that might be produced when a user requests that a file is deleted and another user tries to append information to the same file. In this property we assume that the user associated with uid_1 has admin permissions and uid_2 has read permission. Therefore, if we detect an error in an observed trace then we have to check that the specific users have the required permissions.

$$P_2 = \left(\begin{array}{l} (?DELE, uid_1)(?APPE, uid_2), \\ \left\{ \begin{array}{l} (!250, uid_1), (!450, uid_1), \\ (!550, uid_2), (!501, uid_2) \end{array} \right\} \end{array} \right)$$

In this case, the property indicates that the possible return codes that can be observed are: !250 *(delete completed) or* !450 *(cannot delete the file) to uid_1 and* !501 *(syntax error) or* !550 *(permission denied) to uid_2.*

Next we present a property that includes a specific user. In this case, for the sake of clarity, we use the identifier `adm` *to denote a specific user with administrator permissions. This property represents the behaviour of the system if* `adm` *tries to download a remote file and another user, connected to the same ftp server, tries to abort this transfer. The only possible action in this situation is* !530 *(connection rejected).*

$$P_3 = \left(\begin{array}{l} (?LIST, \texttt{adm})(!150, \texttt{adm})(!226, \texttt{adm}) \\ (?RETR, \texttt{adm})(!150, \texttt{adm})(?ABOR, uid_1), \\ \{(!530, uid_1)\} \end{array} \right)$$

Our methodology is based on the construction of automata representing the properties to be checked by the monitors. These automata are built in two phases. Given a property $P = (\rho, O_\rho)$, in the first phase we built an automaton accepting all the user traces that might be observed if the communications between the server and the users match ρ, assuming FIFO asynchronous communication. Another consequence of communications being asynchronous is that an output from before ρ can be observed after inputs from ρ and outputs from ρ can be observed after inputs that follow ρ. This observation leads to the second step, where we complete the automaton with transitions corresponding to it being possible to observe outputs from before ρ and inputs from after ρ.

Next, we explain how the initial automaton can be built. First, we transform traces into sets of events. Given a trace ρ, we derive a set of events that allows us to distinguish between repeated actions in ρ. The elements are constructed from actions by labelling each pair (action,variable/constant) in ρ with the occurrence of this pair in the trace. Next, we define a partial order $\ll_U$ on the labelled pairs to represent which actions must be observed before other ones if the system produces an instantiation of ρ.

Definition 4. *Let $\rho = (a_1, x_1) \ldots (a_n, x_n) \in (\mathcal{A}ct \times (\mathcal{X} \cup \mathcal{U}sers))^*$ be a sequence of pairs (action, variable/constant). We let $E_U(\rho)$ denote the set of events of ρ, where $e = ((a_i, x_i), k)$ belongs to $E_U(\rho)$ if and only if there are exactly $k-1$ occurrences of a_i in $a_1 \ldots a_{i-1}$. This says that the ith element of ρ is the kth instance of a_i in ρ.*

Let $e_i = ((a_i, x_i), k_i)$ and $e_j = ((a_j, x_j), k_j)$ be two events belonging to $E_U(\rho)$. We write $e_i \ll_U e_j$ if either $i = j$ or $i < j$ and one of the following conditions hold: a_i and a_j are inputs, or a_i and a_j are outputs, or a_i is an input and a_j is an output.

The first two cases in the definition of $\ll_U$ result from channels being FIFO. The last case results from the observation of outputs being delayed, while an input is observed before it is received by the SUT. Essentially, we have that $((a_i, x_i), k_i) \ll_U ((a_j, x_j), k_j)$ does not hold for $i < j$ if a_i is an output and a_j is an input since in this case it is possible that the observation of output a_i is delayed until after input a_j has been observed. In order to simplify the notation, we will remove labels from events (and just write the pair (action, variable)) if they are irrelevant, for example, if a property does not have repeated occurrences of an action.

$$
\begin{array}{lcl}
\mathcal{I}_0 & = & \{\} \\
\mathcal{I}_1 & = & \{(?APPE, uid_1)\} \\
\mathcal{I}_2 & = & \{(?APPE, uid_1), (!150, uid_1)\} \\
\mathcal{I}_3 & = & \{(?APPE, uid_1), (?DELE, uid_2)\} \\
\mathcal{I}_4 & = & \{(?APPE, uid_1), (!150, uid_1), (!226, uid_1)\} \\
\mathcal{I}_5 & = & \{(?APPE, uid_1), (?DELE, uid_2)(?LIST, uid_1)\} \\
\mathcal{I}_6 & = & \{(?APPE, uid_1), (!150, uid_1), (?DELE, uid_2)\} \\
\mathcal{I}_7 & = & \{(?APPE, uid_1), (!150, uid_1), (!226, uid_1), (?DELE, uid_2)\} \\
\mathcal{I}_8 & = & \{(?APPE, uid_1), (!150, uid_1), (?DELE, uid_2), (?LIST, uid_1)\} \\
\mathcal{I}_9 & = & \{(?APPE, uid_1), (!150, uid_1), (!226, uid_1), (?DELE, uid_2), (?LIST, uid_1)\} \\
\mathcal{I}_{10} & = & \{(?APPE, uid_1), (!150, uid_1), (!226, uid_1), (?DELE, uid_2), (!250, uid_2)\} \\
\mathcal{I}_{11} & = & \{(?APPE, uid_1), (!150, uid_1), (!226, uid_1), (?DELE, uid_2), (!250, uid_2), (?LIST, uid_1)\}
\end{array}
$$

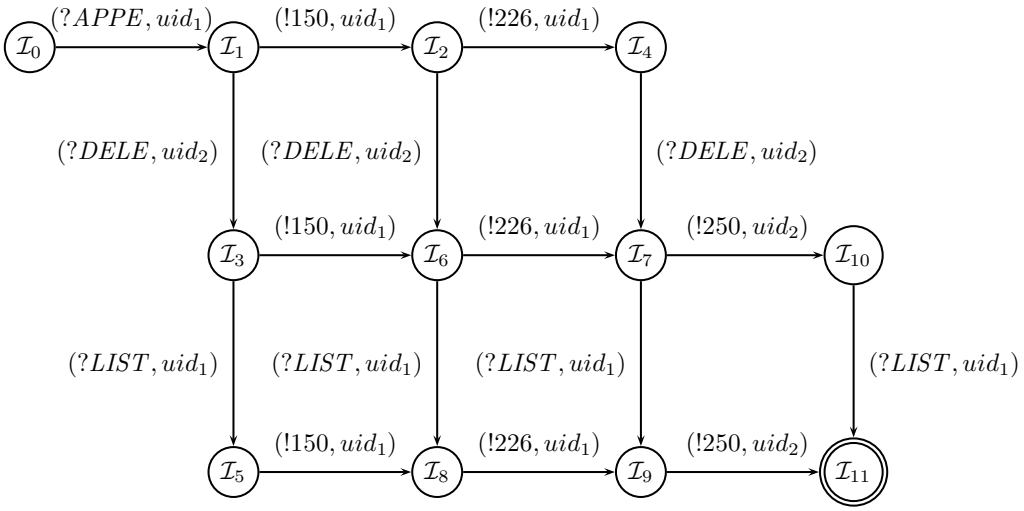

Figure 4: Automaton $\mathcal{A}_{\mathcal{U}}(\rho)$

Example 3. *Consider the trace*

$\rho = (?DELE, uid_1)(!250, uid_1)(?LIST, uid_2)(?DELE, uid_2)$

The corresponding set of events is

$$E(\rho) = \left\{ \begin{array}{l} ((?DELE, uid_1), 1), ((!250, uid_1), 1), \\ ((?LIST, uid_2), 1), ((?DELE, uid_2), 2) \end{array} \right\}$$

For instance, $((?DELE, uid_1), 1) \ll_U ((!250, uid_1), 1)$ *while* $((!250, uid_1), 1) \ll_U ((?LIST, uid_2), 1)$ *does not hold.*

Next we introduce the notion of *ideal*. We will use ideals to construct the states of an automaton that accepts the set containing all the user traces that can be observed if the system produces any instantiation of the considered property.

Definition 5. *Let* $\rho \in (\mathcal{A}ct \times (\mathcal{X} \cup \mathcal{U}sers))^*$ *be a sequence of pairs (action, variable/constant) and* $E_U(\rho)$ *be the set of its events. A set* $\mathcal{I} \subseteq E_U(\rho)$ *is an* ideal *of the ordered set* $(E_U(\sigma), \ll_U)$ *if for all* $e_i, e_j \in E_U(\rho)$*, if* $e_i \ll_U e_j$ *and* $e_j \in \mathcal{I}$ *then* $e_i \in \mathcal{I}$*.*

Intuitively, if the SUT produces ρ and e_i is an element of ideal $\mathcal{I}$, then $\mathcal{I}$ includes all events that *must* be observed before e_i is observed by the monitor.

Example 4. *Consider the sequence*

$\rho = (?DELE, uid_1)(!250, uid_1)(?LIST, uid_2)(?DELE, uid_2)$

of (action, variable/constant) pairs. The following sets of events are ideals of $(E_U(\sigma), \ll_U)$

$$
\begin{array}{l}
\mathcal{I}_1 = \{(?DELE, uid_1)\} \\
\mathcal{I}_2 = \{(?DELE, uid_1), (!250, uid_1)\} \\
\mathcal{I}_3 = \{(?DELE, uid_1), (?LIST, uid_2)\} \\
\mathcal{I}_4 = \{(?DELE, uid_1), (?LIST, uid_2), (?DELE, uid_2)\} \\
\mathcal{I}_5 = \{(?DELE, uid_1), (!250, uid_1), (?LIST, uid_2)\} \\
\mathcal{I}_6 = \left\{ \begin{array}{l} (?DELE, uid_1), (!250, uid_1), (?LIST, uid_2), \\ (?DELE, uid_2) \end{array} \right\}
\end{array}
$$

Next we present an alternative characterisation of the notion of ideal [19].

Proposition 1. *Let* $\rho \in (\mathcal{A}ct \times (\mathcal{X} \cup \mathcal{U}sers))^*$ *be a sequence of pairs (action, variable/constant). We have that* $\mathcal{I} \subseteq E_U(\rho)$ *is an ideal if and only if one of the following conditions holds:*

- *$\mathcal{I}$ contains a pair (a_i, x_i), where a_i is an input, and all the earlier pairs in ρ that contain an input;*

- *$\mathcal{I}$ contains a pair (a_j, x_j), where a_j is an output, and all earlier pairs in ρ; or*
- *$\mathcal{I}$ contains a pair (a_i, x_i) where a_i is an input, a pair (a_j, x_j) where a_j is an output, all the earlier pairs than (a_i, x_i) in ρ that contain an input, and all earlier pairs than (a_j, x_j) in ρ.*

This alternative characterisation indicates that an ideal $\mathcal{I}$ is a set of elements from $E_U(\rho)$ such that all *earlier* elements, under $\ll_U$, are contained in $\mathcal{I}$. In particular, the third item indicates that if an ideal contains a pair with an input $(?i, x)$ and another one with an output $(!o, y)$ then it must also contain, on the one hand, all the previous pairs that present inputs associated to the variable x in ρ and, on the other hand, all the pairs with actions corresponding to variable y. These additions are due to the FIFO nature of the channels.

Next, we show how the ideals associated with a sequence ρ are used to construct the automaton. We use the ideals of $(E_U(\rho), \ll_U)$ to represent states and based on this we define the transitions of a finite automaton $\mathcal{A}_{\mathcal{U}}(\rho)$. Our automata will include a set of variables that will be used to capture the actual users of the traces. Let us emphasize that in this paper, automata are associated to properties[3], not to specifications. Actually, we do not assume the existence of a complete specification; we only assume that the passive tester is provided with a set of properties. First, we introduce the notion of an extended finite automaton and auxiliary notation.

Definition 6. *Let S be a finite set of states, $\mathcal{A}ct$ be a set of actions, X be a finite set of variables, $C \subseteq \mathcal{U}sers$ be a finite set of constants, $s_I, s_F \in S$ be the initial and final states, $cu \notin X$ be the current user variable, $nu \notin X$ be the null user variable and $Tr \subseteq S \times \mathcal{A}ct \times (X \cup C \cup \{cu, nu\}) \times S$ be a set of transitions. We say that the tuple $A = (S, \mathcal{A}ct, Tr, X, C, cu, nu, s_I, s_F)$ is an extended finite automaton.*

A valuation *over X is a total function from X to $(\mathcal{U}sers \setminus C) \cup \{null\}$. We denote by $\mathcal{V}_X$ the set of all valuations of X. Given $v \in \mathcal{V}_X$, for all $x, y \in X$ such that $v(x) \neq null$ and $v(y) \neq null$ we have that $v(x) \neq v(y)$.*

A configuration *of A is a pair (s, v) where $s \in S$ is the current state and $v \in \mathcal{V}_X$ is the valuation corresponding to the current value of the variables belonging to X.*

Given a configuration (s, v), if an action a associated with a user u is produced then a transition (s, a, y, s') can be fired if one of the following conditions holds:

- $y = cu$.
- $y = nu \wedge \forall c \in C : c \neq u \wedge \forall x \in X : v(x) \neq u$.
- $y \in C \wedge y = u$
- $y \in X \wedge (v(y) = null \wedge \forall x \in X : v(x) \neq u) \vee v(y) = u)$.

In this case, the configuration will change to (s', v') where v' is the valuation such that

$$v'(x) = \begin{cases} u & \text{if } x = y \wedge v(y) = null \\ v(x) & \text{otherwise} \end{cases}$$

The initial valuation *of A, denoted by v_0, assigns null to every variable belonging to X.*

Our automata will recognise sequences of (action,user) pairs. The transitions of the automata will thus be labelled with pairs of the form (a, y), in which y might be a variable. If an (action,user) pair (a, u) is received by an automaton then the only transitions that could be potentially fired are those with the same action a. If a transition is labelled with (a, u) then the transition can be fired. However, if the label of a transition is (a, x), with $x \in X$, then the variable x plays several roles that we explain later.

In the definition of our automata we use two additional terms, cu and nu, that can label transitions. If we are processing a pair (a, u) and a transition is labeled by a and cu (standing for current user), then the transition can be triggered. Essentially, cu does not impose any constraints on the user and does not lead to the change in the value of any variable. Thus, cu can be seen as a 'do not care' term. The term nu (standing for null user) is similar except that it requires that none of the *regular* variables in X have already been instantiated to u. If a transition is labeled by a constant, then the transition can only be triggered if the constant is equal to u. A valuation assigns a user identifier to each variable in X. If y is a variable then a transition (s, a, y, s') will be fired if the current state is s, the automaton receives the action a associated with a user u and either y is not instantiated and no other variable is instantiated to u or $v(y) = u$ in the current valuation of the variables. If the transition is triggered then the variable y is instantiated to u if its previous value was *null*. Finally, the current state becomes s'. Next, we define the first type of automata that we use in our approach. Essentially, given a property (ρ, O_ρ), the extended finite automaton $\mathcal{A}_{\mathcal{U}}(\rho)$ will accept those traces that are a possible variation of an instantiation of ρ.

Definition 7. *Let (ρ, O_ρ) be a property. The extended finite automaton for ρ, denoted by $\mathcal{A}_{\mathcal{U}}(\rho)$, is defined as $(S, \mathcal{A}ct, Tr, X, C, cu, nu, \mathcal{I}_s, \mathcal{I}_f)$ where*

- *S, the set of states, is equal to the set of ideals of $(E_U(\sigma), \ll_U)$.*
- *$\mathcal{A}ct$ is the alphabet.*
- *$X = var(\rho)$ is a finite set of variables.*
- *$C = cons(\rho)$ is a finite set of identifiers.*
- *$\mathcal{I}_s = \{\}$ is the initial state.*
- *$\mathcal{I}_f = E_U(\rho)$ is the final state.*

[3] More properly, automata are a useful *view* of properties because we can reduce the problem of checking whether a sequence reveals a problem to a problem of an automaton accepting the sequence.

1. **Input** (ρ, O_ρ).
2. Let $\mathcal{A}_\mathcal{U}(\rho) = (S, \mathcal{A}ct, Tr, X, cu, nu, \mathcal{I}_s, \mathcal{I}_f)$
3. Let $\mathcal{A}_\mathcal{U}(\rho, O_\rho) = (S \cup \{s_f\}, \mathcal{A}ct, Tr, X, cu, nu, s_0, s_f)$ where $s_0 = \mathcal{I}_s$, and $s_f \notin S$ is a fresh state.
4. For all $a \in I \cup O$ add the transition $(s_0, (a, cu), s_0)$.
5. For every state s of $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$ that represents an ideal that does not contain any pair with an output action and for all $!o \in O$, add the transition $(s, (!o, cu), s)$.
6. For every state s of $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$ that represents an ideal that contains all the pairs from ρ that present an input action, add the transition $(s, (?i, cu), s)$ for all $?i \in I$.
7. For all $l \in var(O_\rho) \cup cons(O_\rho)$ add the transitions $(\mathcal{I}_f, (O \setminus \{!o | (!o, l) \in O_\rho\}, l), s_f)$. Additionally, for all $l \in (X \setminus var(O_\rho)) \cup (C \setminus cons(O_\rho))$ add the transitions $(\mathcal{I}_f, (O, l), \mathcal{I}_f)$. Finally, add a transition $(\mathcal{I}_f, (O, nu), \mathcal{I}_f)$.
8. Make s_f the only final state of $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$.
9. *Complete A.* Given a state s of $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$, let $ini(s)$ denote the set of variables of X and constants appearing in the pairs belonging to the ideal that represents s. For every state s of $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$ with $s \neq s_f$ and for all $a \in \mathcal{A}ct$ and $l \in ini(s)$ such that there is no transition from s with label (a, l), add the transition $(s, (a, l), s_0)$. Additionally, for all $s \in S$ and all $a \in \mathcal{A}ct$ such that there is no transition with label (a, l), with $l \in (X \cup C) \backslash ini(s)$, add a transition $(s, (a, nu), s)$.
10. **Output** $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$.

Algorithm 1: Producing $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$

A tuple $(\mathcal{I}, a, x, \mathcal{I}')$ belongs to the set of transitions Tr if and only if there exists an event $((a, x), k) \in E_U(\sigma)$ such that $\mathcal{I}' = \mathcal{I} \cup \{((a, x), k)\}$.

Example 5. *Consider the trace*

$$\rho = \begin{array}{l}(?APPE, uid_1)(!150, uid_1)(!226, uid_1) \\ (?DELE, uid_2)(!250, uid_2)(?LIST, uid_1)\end{array}$$

Figure 4 depicts the automaton $\mathcal{A}_\mathcal{U}(\rho)$ that accepts the set of sequences in $\mathcal{L}_u(\rho)$.

Since our methodology applies passive testing and, as explained before, we cannot use a homing algorithm, a trace of interest might not be the start of the overall observed trace. This fact has to be reflected in the construction of the automaton that will be used. The automaton $\mathcal{A}_\mathcal{U}(\rho)$ must be adapted to take into account different points. We must take into account the fact that an instantiation of ρ might be preceded by other actions and the observation of earlier outputs might be delayed. Besides, an instantiation of ρ might be followed by later actions and the outputs of the instantiation might not be observed until after later inputs. In addition, the observation will include actions associated with users that are not of interest. Algorithm 1 achieves this. We denote by $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$ the extended automaton.

Example 6. *Consider the automaton $\mathcal{A}_\mathcal{U}(\rho)$ depicted in Figure 4 corresponding to the trace ρ introduced in Example 5. Given $O_\rho = \{(!551, uid_1), (!501, uid_1)\}$, Figure 5 shows the automaton $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$ constructed by using Algorithm 1. Let us note that the transitions required to complete the automaton given in Step 9 of the algorithm are not drawn in the figure to not overcomplicate the graph.*

Next we explain how our algorithm works and justify its correctness. Initially we take the automaton $\mathcal{A}_\mathcal{U}(\rho)$ and add a new final state. The idea is that if the final state is reached then we can claim that the observed trace violates the property. The fourth step adds transitions to the initial state to ensure that we are considering all possible starting points in the observed trace. In the next step, we add transitions to deal with the possibility of earlier output being observed after input from ρ. Similarly, in the sixth step we add transitions to consider the possibility of later input being observed before some of the output from ρ is observed. In the seventh step we consider that we have observed an admissible reordering of ρ and check whether the next action shows an error. If the next action associated with any of the users in O_ρ does not belong to O_ρ, then go to the final (error) state. We also include loop transitions associated with users not in O_ρ to skip this action. After making the new state a final state (step eight) then we almost have our desired automata. Finally, in the ninth step we complete the automata with two types of transitions. The first one considers actions performed by users that have not previously matched the property. These actions are discarded by remaining in the same state. The second type considers the situation where a user that was involved in the matching of the property performs an unexpect action. In this case, the transition takes the automata to the initial state.

Concerning the complexity of the automata $\mathcal{A}_\mathcal{U}(\rho, O_\rho)$ in terms of states and transitions, let n be equal to the number of ideals generated from the property. The number of states of the automaton is equal to $n + 1$. Next, we provide the complexity, in the worst case, of the number of transitions. First, note that $var(O_\rho) \subseteq var(\rho)$, $cons(O_\rho) \subseteq cons(\rho)$ and that $|var(\rho)| + |cons(\rho)| \leq |\rho|$.

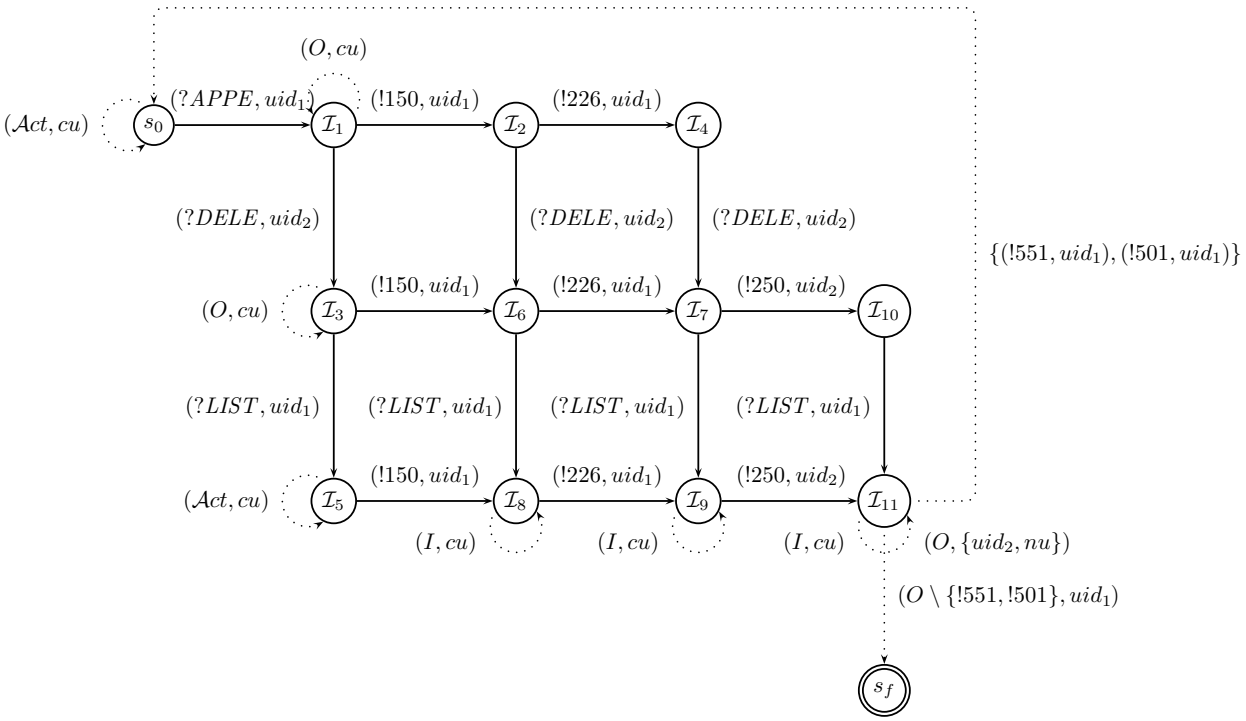

Figure 5: Automaton $\mathcal{A}_{\mathcal{U}}(\rho, O_\rho)$

Taking into account that $|\rho| \leq n$ we have that in the worst case the number of transitions is in $\mathcal{O}(n^2 \cdot |\mathcal{A}ct|)$.

4. Checking Traces

In this section we present the overall method that analyses an observed trace with respect to a property. In the case where there is only one user, it is sufficient to decide whether the application of the trace to the associated automaton leads to the final state; if this is the case then the trace does not fulfil the requirements expressed in the property. However, since there are multiple users we must take into account the fact that the observed trace might contain actions related to different users interacting with the server. In this section we provide an algorithm that shows how one can analyse a trace ρ taking into account the fact that ρ could allow different instantiations of the considered property and the corresponding actions can be interleaved.

Example 7. *Consider the observed trace*

$$(?i, u_2)(?i, u_3)(?i, u_1)(!o', u_3)(!o, u_1)$$

and the property $((?i, x)(?i, y)), \{(!o', y)\}$. *There are two ways in which one can match the property and the observed trace, with these being based on two different instantiations. If we consider the first one, in which* x *and* y *take the values* u_2 *and* u_3*, respectively, then we do not observe an error because we obtain the expected output* $(!o', u_3)$. *However, this is not the case when the variables are instantiated with the values* u_3 *and* u_1. *In this case we skip* $(?i, u_2)$ *and start matching with the second action of the sequence. Then, we also skip* $(!o', u_3)$ *because we are only interested in checking the behaviour of* u_1*, due to the fact that* y *is the only variable involved in the set of actions included in the property. We observe an unexpected output for* u_1*,* $(!o, u_1)$. *In this case, we should return failure.*

As the previous example shows, we have to deal with different automata in parallel, with the number of automata being the number of instantiations of the property that appear in the trace. In order to deal with the set of required automata we use a *list*. Each element of the list represents an automaton and stores the set of *pending states*; for each of these states, we store the current values of the variables that appear in the property. The pending states are those states of the automaton related to the property, that have been reached during the matching of the observed trace and the automaton. The variables of the transitions traversed in the path from the initial state to each pending state are associated with this path. The values assigned to these variables correspond to the users that appear in the trace associated with the actions that match the labels of the transitions. In our current framework we do not consider the analysis of several sessions corresponding to the same user because we only record the user id not the id of the session.

We present an algorithm (see Algorithm 2) to decide whether an automaton, representing a given property, detects a possibly erroneous behaviour in an observed trace. Essentially, we traverse all the (action,user) pairs of the trace and for each we analyse if the actions match any

Algorithm `Validation_Trace_Property`$(channel, aut)$
/*$aut = (S, \mathcal{A}ct, Tr, X, C, cu, nu, s_0, s_f)$ */

$correct \leftarrow$ `true`;
$Aut \leftarrow \emptyset$;
while $channel.connected \wedge correct$ **do**
  `Read`$(channel, a, u,)$;
  foreach *instance of the automaton* $InstAut \in Aut$ **do**
    foreach *pending state* $pdSt \in InstAut$ **do**
      $Aux \leftarrow \emptyset$;
      $T_c \leftarrow \{s | (pdSt.state, (a, cu), s) \in Tr \wedge s \neq s_0\}$;
      $T_c \leftarrow T_c \cup \{s | (pdSt.state, (a, u), s) \in Tr \wedge s \neq s_0\}$;
      $T_c \leftarrow T_c \cup \{s | (pdSt.state, (a, nu), s) \in Tr \wedge s \neq s_0 \wedge u$ is not assigned to any variable of $pdSt \wedge u \notin C\}$;
      foreach s *in* T_c **do**
        if $s = s_f$ **then**
          $correct \leftarrow$ `false`;
        else
          $Aux \leftarrow Aux \cup$ `Create_pendState`$(pdSt.var, s)$;
        end
      end
      if $u \notin C$ **then**
        if u *is not assigned to any variable of* $pdSt$ **then**
          $T_x \leftarrow \{(s, x_i) | (pdSt.state, (a, x_i), s) \in Tr \wedge s \neq s_0 \wedge x_i \notin pdSt.var\}$;
        else
          $T_x \leftarrow \{(s, x_i) | (pdSt.state, (a, x_i), s) \in Tr \wedge s \neq s_0 \wedge pdSt.var(x_i) = u\}$;
        end
        foreach (s, x_i) *in* T_x **do**
          if $s = s_f$ **then**
            $correct \leftarrow$ `false`;
          else if $x_i \in pdSt.var$ **then**
            $Aux \leftarrow Aux \cup$ `Create_pendState`$(pdSt.var, s)$;
          else if u *is not assigned to any variable associated with* $pdSt$ **then**
            $Aux \leftarrow Aux \cup$ `Create_pendState`$(pdSt.var \cup \{x_i = u\}, s)$;
          end
        end
      end
      `Delete_pendState`$(InstAut, pdSt)$;
      if $Aux \neq \emptyset$ **then**
        `Add_pendState`$(InstAut, Aux)$;
      end
    end
    $T_0 \leftarrow \{(s, x_i) | (s_0, (a, x_i), s) \in Tr \wedge s \neq s_0\}$;
    foreach (s, x_i) *in* T_0 **do**
      `Create_autIns`$(Aut, newInsAut)$;
      `Add_pendState`$(newInsAut,$ `Create_pendState`$(\{x_i = u\}, s))$;
    end
    $T_c \leftarrow \{s | (s_0, (a, u), s) \in Tr \wedge s \neq s_0\}$;
    foreach s *in* T_c **do**
      `Create_autIns`$(Aut, newInsAut)$;
      `Add_pendState`$(newInsAut,$ `Create_pendState`$(\emptyset, s))$;
    end
  end
end
return$(correct)$;

Algorithm 2: Correctness of a trace with respect to a property.

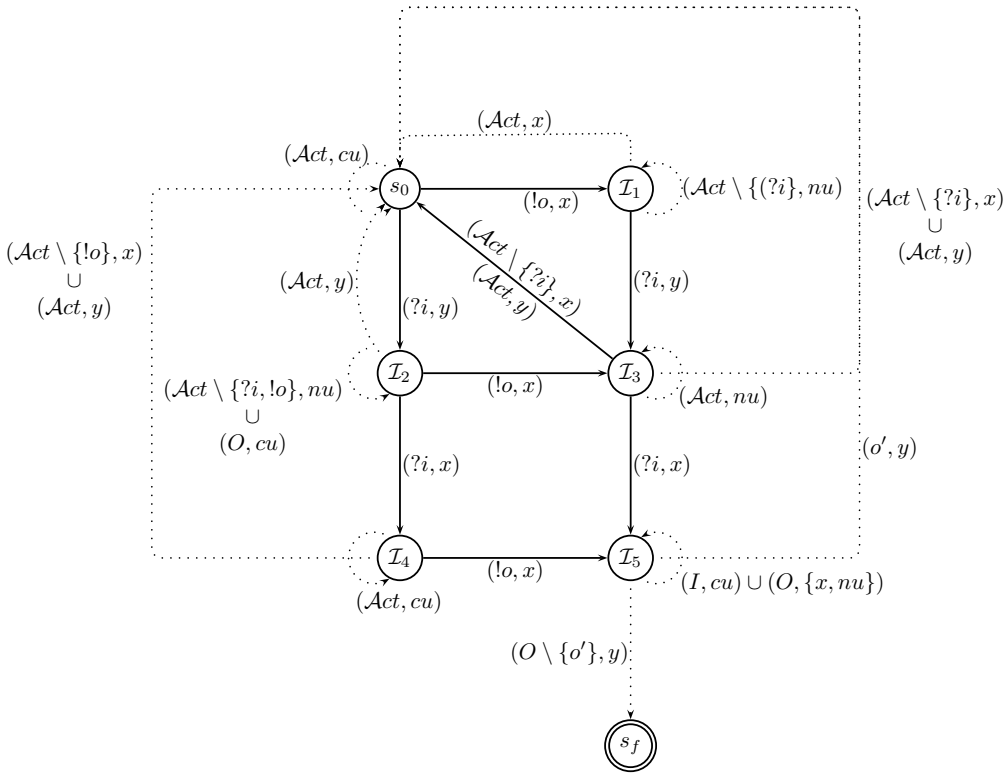

Figure 6: Automaton $\mathcal{A}_{\mathcal{U}}(((!o, x)(?i, y)(?i, x), \{(!o', y)\}))$

transition of the automaton outgoing from the last reached states. If so, we distinguish between the transitions labelled by the variable cu, the ones labelled by nu and the rest of the transitions. The transitions labelled with cu do not impose restrictions over the user associated with the action. In this case, we only record the change of state. Regarding the transitions labelled by nu, they can only be triggered if the current user in the (action,user) pair being considered is not assigned to any of the variables in the automaton. If this requirement is fulfilled, then the transition will be triggered and the change of state recorded. We do not need to store the user in the system. A transition labelled by a specific user u only can be fired if the user in the current (action,user) pair is equal to u. If the label of a transition contains a variable y then there are two conditions under which this transition can be followed: either the value of y coincides with the user u of the current (action,user) pair or y has no value assigned and the current user u is not assigned to any other variable. In the first case, we only update the state. In the second one, we also need to register the new value of the variable y, which becomes u. However, if the value of the variable y is not equal to the current user u, then the transition cannot be triggered. In order to reduce the number of elements in the list, we delete all pending states corresponding to s_0. In this way, we avoid the need to analyse elements that cannot be modified by the pairs that appear in the rest of the trace. If an automaton in the list reaches the final state, then an error is reported and the algorithm stops.

Example 8. *Consider that we observe the trace*

$$(?i, ip_1)(!o, ip_2)(?i, ip_2)$$

and we have the property $((!o, x)(?i, y)(?i, x), \{(!o', y)\})$. *Next, we explain how our algorithm works. First, the set of associated ideals is:*

$$\begin{array}{lcl} \mathcal{I}_0 & = & \{\} \\ \mathcal{I}_1 & = & \{(!o, x)\} \\ \mathcal{I}_2 & = & \{(?i, y)\} \\ \mathcal{I}_3 & = & \{(!o, x), (?i, y)\} \\ \mathcal{I}_4 & = & \{(?i, y), (?i, x)\} \\ \mathcal{I}_5 & = & \{(!o, x), (?i, y), (?i, x)\} \end{array}$$

The automaton associated with the property is depicted in Figure 6. Initially, the algorithm processes $(?i, ip_1)$ *and searches for all the transitions outgoing from the initial state that are labelled by* $?i$. *In this case, the only transition of the automaton that can be triggered is* $(s_0, (?i, y), s_2)$. *A new instance of the automaton,* A_1, *is created and a new pending state,* $ps_{11} = ((x = null, y = ip_1), s_2)$, *is generated and associated with it. The pending state corresponds to the final state of the transition and stores the value associated with the variable. In our case, the variable* y *labels the transition and, therefore, the current user,* ip_1, *is assigned to it. For each pair (action, user) processed by the algorithm, the set of pending states will be updated. In our*

example, when the next pair $(!o, ip_2)$ *is received, the algorithm determines if any of the pending states of* A_1 *must evolve taking into account the new action. Given that the only pending state,* ps_{11}*, is associated with the state* s_2*, the algorithm looks for transitions labelled by* $!o$ *and outgoing from* s_2 *that can be triggered. As we can observe, two transitions fulfill these conditions:* $(s_2, (!o, cu), s_2)$ *and* $(s_2, (!o, x), s_3)$*. The first one is associated with the cu variable and, in this case, the initial and final states coincide. Therefore, no changes are applied to* ps_{11}*. In contrast, the second transition can be applied if either the variable* x *has not been assigned any value in the pending state and none of the other variables have been assigned the value* ip_2*, or the value assigned to the variable* x *corresponds to* ip_2*. In this case the first condition holds and a new pending state is associated with* A_1*,* $ps_{12} = ((x = ip_2, y = ip_1), s_3)$*. Note that for each pair (action, user) received by the algorithm, the algorithm evaluates whether a new instance of the automaton, corresponding to a new instantiation of the property, must be generated. To do this, the algorithm considers the initial state of the automaton and, as happened with the first (action, user) pair of the processed trace, it checks if any transition can be triggered. In this case, the transition* $(s_0, (!o, x), s_1)$ *can be applied and, therefore, a new instance of the automaton,* A_2*, is created and a pending state* $ps_{21} = ((x = ip_2, y = null), s_1)$ *is associated with it. The application of the new pair* $(?i, ip_2)$ *updates the set of pending states,* ps_{11} *and* ps_{12}*, of* A_1 *to* $ps_{11} = ((x = ip_2, y = ip_1), s_4)$ *and* $ps_{12} = ((x = ip_2, y = ip_1), s_5)$*, respectively. In the case of the pending state* p_{21} *the only transition that can be triggered leads to the initial state. This means that the instantiation corresponding to this pending state does not match the property and we do not need to continue analysing it. Therefore, the algorithm removes* p_{21}*. If any of the transitions that can be triggered when a pair (action, user) is processed reaches the final state* s_f*, then an error has been found and the algorithm stops.*

Algorithm 2 is correct in the sense that if the observed trace, which is a proper permutation of the actual trace produced by the SUT, does not satisfy the property represented by the automata, then the error state will be reached. The proof follows the same lines as the proof of soundness in our previous work (Theorem 1 [19]). Essentially, if we have a subsequence of the observed trace matching the first part of the property (once irrelevant actions are removed) and failing to produce an expected output then this subsequence will also lead from the initial state to the error state. We only need to take into account the fact that there can be irrelevant actions, inputs observed *in advance*, due to delayed outputs, that are not part of the property. These will be appropriately skipped thanks to the loops included in the states of the automaton.

Concerning the complexity of the algorithm, let l be equal to the length of the observed trace. Let $\mathcal{O}(f(l))$ be the complexity of checking, in the classical sense, whether an automaton accepts the trace. In the worst case we have to check all the suffices of the trace, that is, we may have to check at most l traces of lengths $l, l-1, l-2, \ldots 1$, respectively. Therefore, in the worst case, the complexity of checking the trace is in $\mathcal{O}(f(l)^2)$.

5. Case study: Vulnerabilities in WebDAV

In this section we show the applicability of our methodology in the context of the *Exchange Store Web Distributed Authoring and Versioning* (WebDAV) protocol [51] in the open source communication platform NextCloud [36]. First, we introduce the new version of PTTAC, *Passive Testing Tool for Asynchronous Communications* [5]. We have extended the tool in order to implement the methodology presented in this work. Next, we will describe the experimental setup used in our experiments for detecting vulnerabilities, explain the properties that we considered for each of the scenarios and report on the obtained results.

5.1. PTTAC Tool

The testing phase of the experiments has been carried out by using a new version of the PTTAC tool. We have extended PTTAC in order to implement and automate the passive testing methodology proposed in the framework presented in Section 3. PTTAC allows users to define properties that an SUT must satisfy, it automatically generates the corresponding automata, it is able to capture the packages that are sent in a specific network and is able to test both online and offline captured traces. Unlike the previous version of PTTAC, which only allowed us to capture the actions performed by one actor, the new release allows us to capture and analyse the traces performed by several users.

The overall architecture of PTTAC is shown in Figure 7 and its main components are:

- *Properties manager.* It allows users to manage properties associated with a system.
- *Automata generator.* It implements the algorithms to generate automata associated with the properties of the systems.
- *Trace grabber.* It is the module where online traces of a network are captured.
- *Testing manager.* It performs the process of checking the correction of the captured traces with respect to a specific property.

The *properties manager* allows users to define properties that must be fulfilled by the SUT. The *automata generator* is the key component of the tool because it puts into action our methodology. After the user has added a property of interest, the tool constructs the automaton

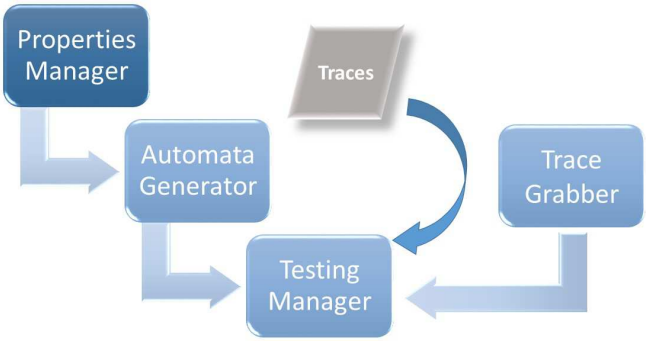

Figure 7: PTTAC architecture

that will be used to check captured traces against the corresponding property (see Algorithm 1). The *trace grabber* is responsible for the capture of online traces, using a specific communication protocol. Currently, the tool provides three different options: HTTP, FTP and TCP. The trace grabber is built on top of the network monitor of the web developer features provided by the Firefox web browser. The last component, the *testing manager*, allows the testers to either capture or load observed traces and check them with respect to different properties. This process uses the previously generated automaton and reports the final verdict (see Algorithm 2).

PTTAC provides a user friendly graphical interface. Its GUI presents different areas that allow the user to easily access and use the functionalities of the tool. Figure 8 shows different screenshots of the PTTAC interface.

5.2. Experimental Setup

The experiments were performed on three computers equipped with the following features:

- PC 1: Intel(R) Core(TM) i7-6500U CPU @ 3.10GHz, 16 GB RAM with Windows 10 OS.
- PC 2: Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 12 GB RAM with Windows 10 OS.
- PC 3: Intel Celeron N2840 Dual Core 2.16GHz, 4 GB RAM with Kali Linux v2016.1 OS.

In order to apply our methodology in the considered scenarios, we followed the next steps:

- *Design of properties.* We defined several representative properties of the behaviour of the WebDAV protocol.
- *Traces capture.* The traces were captured from two PCs using our trace grabber. These traces had to be processed offline and it was necessary to format them.
- *Traces formatting.* Once the protocol communications that we wanted to analyze were captured, and stored as plain text files, it was necessary to transform these files into files that can be accepted by the testing manager of PTTAC. In order to do it we designed a Java application, called TraceTranslator, for translating plain text files (TCP, HTTP, HAR, etc) into XML files with the format that can be managed by our tool. In the case of the WebDAV protocol, the traces captured by each computer generate HAR files with the actions corresponding to each user. These files had to be merged to generate a global trace. With this goal in mind we developed MergeLogs, a Java application for generating an XML file from different HAR files compatible with PTTAC. Both tools are released as open source software and included in the previously mentioned repository. An example of the translation is showed in Figure 9.
- *Traces analysis.* Finally, we tested the traces against the automata generated by PTTAC from the previously defined properties.

5.3. Scenarios and Properties

In this work we have used two protocols in our experiments. However, the goals that we tried to reach with each of them were different. We used FTP as the basis of the running example used in the presentation of our theoretical framework. FTP is a standard network protocol of the application layer used to transfer resources between a server and a client. The original specification was published in 1971 and it has been improved over time [12]. Therefore, FTP is currently very stable and it is very unlikely that new errors can be found. However, this protocol allowed us to define properties and check traces against them for checking the validity and usefulness of our approach in a real scenario. The selected properties were related to the control and management of files and their content. The experiments were conducted in different scenarios in which the testers tried to lead the operation of the system. For example, a user tried to rename a file while its content was being modified by another user, different users tried to download a file that was being simultaneously modified or tried to create new directories and files with the same name. The package of traces that we studied had around 5000 FTP messages but, as it was expected, the behaviour of the protocol did not exhibit an error. Therefore, we decided to analyse a more recent protocol, WebDAV, an extension of the Hypertext Transfer Protocol (HTTP). WEbDav has been extended with new methods and headers for the performance of remote web content operations, such as namespace management, overwrite protection, maintenance of file properties and author properties. The initial version dates from 1996, but the latest one was concluded in 2007. On the contrary to FTP, WebDAV has less updates and it is more likely to find vulnerabilities. In fact, there are some references to previously detected vulnerabilities due to Man-In-the-Middle

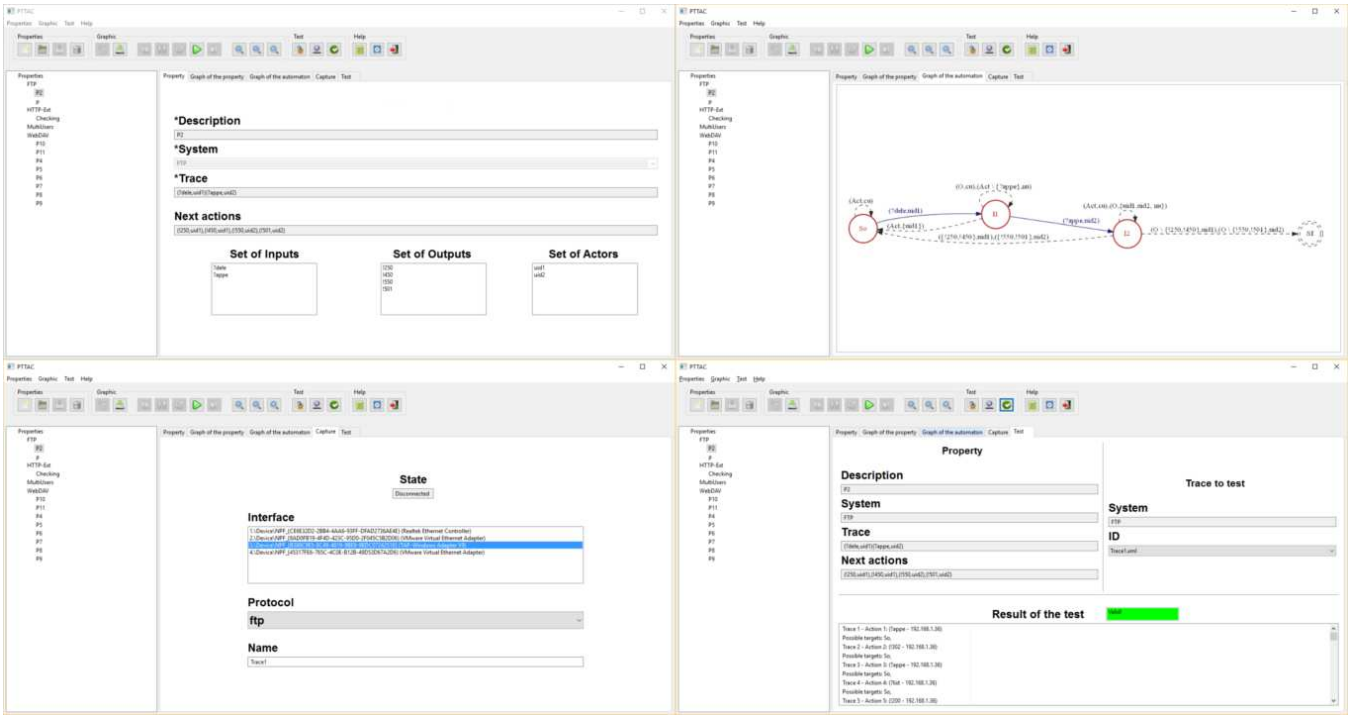

Figure 8: PTTAC Graphical User Interface

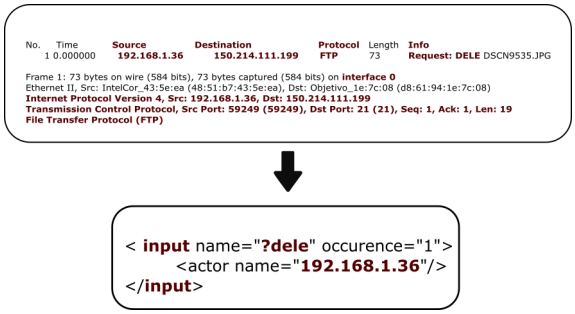

Figure 9: Example of trace translation

(MITM) [11, 9, 30] attacks. We were able to simulate an attack and show the effectiveness of our methodology for detecting this kind of threats. There exist several platforms that support WebDav. Among them, we decided to use Nextcloud due to the recent release of a new product: Nextcloud Box [34]. Nextcloud Box provides a private cloud that allows to store and synchronize data between local and remote devices. Although we did not have access to the full version at the time of the experiments, we were able to use the demo of the software, available at the official website [35]. In our experiments, we used two computers to run Nextcloud desktop clients and the third computer was running PTTAC.

In order to facilitate the understanding of the studied properties, in Table 2 we present the WebDAV requests and responses that we have used in their design. WebDAV responses are grouped in series, similar to the HTTP protocol responses: the 200s series includes successful responses, the 400s series corresponds to client failure responses and the responses gathered in the 500s series are related to server failures.

We defined 5 properties to capture different behaviours of the protocol in the communication between clients and server in the Nextcloud platform. The first four properties deal with the behaviour of the system when different users simultaneously access shared resources. The fifth property was designed to validate the behaviour of the server when it does not give access to users.

The first property represents the behaviour of the system when two users, uid_1 and uid_2, try to update the content of a shared common file. The first user (uid_1) that accesses the file has priority over the second one when they try to modify the content of the file. Therefore, the second user should receive a failure response.

$$P_1 = \left(\begin{array}{l} (?GET, uid_1)(?GET, uid_2)(!200, uid_1) \\ (!200, uid_2)(?PUT, uid_1)(?PUT, uid_2) \\ (!200, uid_1), \\ \{(!404, uid_2)\} \end{array} \right)$$

The next property captures the situation in which a user (uid_1) moves a file and another user (uid_2) tries to get some information from that file using an obsolete location. Again, the second user should receive a failure response.

$$P_2 = \left(\begin{array}{l} (?MOVE, uid_1)(?GET, uid_2)(!201, uid_1), \\ \{(!400, uid_2), (!404, uid_2)\} \end{array} \right)$$

In the next property, the first user (uid_1) renames a

Requests	GET	Request data from a specified resource
	PUT	Submit data to a specified resource
	MOVE	Move a resource to a specific location
	DELETE	Delete a resource
	MKCOL	Create a new collection, for example a folder
	PROPFIND	Retrieve properties for a resource
Status codes	200	Successful response for methods that do not require the creation of new resources
	201	The collection was created/moved correctly
	204	Successful response for methods that require the copy, move or delete of a resource
	400	Malformed syntax
	404	Resource not found
	503	Service unavailable

Table 2: WebDAV requests and status codes used in the paper

folder and the second one (uid_2) tries to access it using the old location.

$$P_3 = \left(\begin{array}{l} (?MOVE, uid_1)(!201, uid_1)(?GET, uid_2) \\ (?GET, uid_1)(!200, uid_1), \\ \{(!400, uid_2), (!404, uid_2)\} \end{array} \right)$$

Property P_4 describes the behaviour that the system should show in the case that a user (uid_1) deletes a folder and another one (uid_2) tries to rename it. Due to the priority of the first request, the only possible response to user uid_2 is a failure.

$$P_4 = \left(\begin{array}{l} (?DELETE, uid_1)(?MOVE, uid_2), (!204, uid_1), \\ \{(!404, uid_2)\} \end{array} \right)$$

The last property considers the situation when two users try to access the content of a file. If the first one cannot access it due to a failure in the server (a !503 error message), the second one should receive the same response.

$$P_5 = \left(\begin{array}{l} (?GET, uid_1)(?GET, uid_2)(!503, uid_1), \\ \{(!503, uid_2)\} \end{array} \right)$$

As a final remark, the definition of P_5 was motivated by the experience obtained during the use of the platform. After a period of time, the server eventually restarts the connection with the user and requires to log in again. The length of the periods of time were always very similar. This was due to the fact that the demo version of the platform restores the contents of all the files to their default versions, even if they were deleted, when the connection is restarted. Checking that this fact happened for all the users connected to the platform at the same time, we decided to explore the possible scenarios that could happen with these conditions.

5.4. Capturing and Checking Traces

We performed 6 experiments from which we collected the corresponding traces. The five first traces were captured during the performance of general actions such as copy and moving files between directories, checking their properties, adding comments to the different versions of the files, and deleting or restoring files. However, the obtained results did not give us any relevant data about possible vulnerabilities or suspicious behaviours of any of the actors involved. For this reason, we decided to perform a last experiment inspired on the previously commented detected vulnerabilities. Specifically, we decided to simulate an MITM attack with a third computer in the same network as the one to which the users were connected. While the users were performing general actions, like in the previous experiments, we put in practise the MITM attack adding delays to the hacked communication and changing the order of the reception of messages. As a consequence, the user under attack received the corresponding messages later than the user that has not being hacked with our intervention.

Table 3 presents some data related to the traces collected during the experiments. In addition to the number of packages registered during each experiment, the table also indicates if any attack was simulated during the performance of the experiment and the number of packages corresponding to the most relevant messages of the study.

The system detected only one error and it was in the trace captured during the last experiment. However, a simple inspection of Table 3 does not show a possible vulnerability during any of the experiments because the responses obtained for the series 100s, 200s, 300s and 400s seem to be quite similar in all the experiments. They represent correct interactions between the users and the server, information derived from the exchange of information and correct or incorrect reception of requests in the server. In addition, the relation between the number and type of requests and the obtained responses suggests a correct behavior. Nevertheless, if we take into account that we were studying the interactions of two users with the server and the messages of the series 500s indicate an error in the server, it looked suspicious to us that there were only one package of this series in Experiment 6. If the server was not properly working during a period of time

Experiment	Total	Attack performed	?GET	?PROPFIND	?DELETE	?MOVE	?MKCOL	?PUT	100s	200s	300s	400s	500s
1	547	No	264	7	5	21	0	3	8	266	4	3	2
2	825	No	385	12	13	6	31	6	0	406	4	3	0
3	817	No	399	6	9	11	0	0	11	217	192	0	0
4	903	No	427	15	0	17	21	4	0	252	196	4	0
5	711	No	342	8	17	42	14	0	0	350	4	2	0
6	910	Yes	439	10	31	35	7	0	4	387	75	0	1

Table 3: Data regarding captured traces: occurrences of each different package per experiment

and both users were interacting with it without interruption, the message of the 500s series should have been sent to both users, not only to one of them. In order to check that the capture was not interrupted during the sending of a "possible" second 500s message, we manually checked the XML file of the trace. We wanted to discard that the capture of the trace had been interrupted and this package was one of the last actions stored in the trace. The last registered action of the server did not correspond to this message. Trying to determine the reason of the incorrect behaviour of the system, we looked for the package in the trace and we detected that it was delivered at the same instant at which the MITM attack was perpetrated. Checking the log of the MITM attack confirmed our suspicion: the hacker delayed the 500s response delivered to the second user. Therefore, other responses were sent before a system failure was received by the user, leading to make him believe that the service was properly working. In order to confirm our hypothesis, we analysed the trace registered during the first experiment because it also contains 500s messages. In the first experiment, the number of responses indicating a failure is an even number (there are 2 different 500s messages). In addition, the information stored in the HAR files corresponding to each of the users was produced with a difference of milliseconds. Such a small difference does not indicate an erroneous behaviour. Actually, the difference can be due to the distance between the users and the server. Consequently, in this case, the observed trace did not show a possible error of the server side. However, if we analyse the moment when both users sent a GET request, we see that the user that is not being attacked received a 500s response while the user that is being attacked received a different message. This is a direct consequence of the MITM attack previously indicated.

In conclusion, the application of our methodology allowed us to detect a vulnerability in the performance of the WebDav protocol in the Nextcloud platform. We think that this shows the effectiveness of our approach in detecting faults and that our approach has a compelling value to formally test applications in asynchronous environments with multiple users. However, it must be emphasised that, in general, failing a property such as P_5 does not allow us to claim that a vulnerability has been revealed. For example, the fail might be due to an improper network latency. Therefore, once an error is found we need to perform a subsequent analysis, as the one previously explained, in order to confirm that we have detected a real vulnerability.

5.5. Threats to Validity

We have analysed different types of threats to the validity of the results of our experiments.

Threats to *internal validity* consider uncontrolled factors that might be responsible for the obtained results. In our study, the main threat to internal validity is the possible faults in the implementation of our approach and the misleading results that could be derived as a consequence. In order to reduce the impact of this threat, we checked our tool with critical cases during the development of PTTAC. Regarding the experiments, we manually checked the correctness of the obtained results.

Threats to *external validity* consider those conditions that allow us to generalize our findings to other situations. We identified external threats associated with the state of the networks that we used in our communications and with the number of users connected to them. These are uncontrollable aspects for testing the reliability of the system. In order to reduce the impact of these two threats, we performed the experiments with different networks, at different times of the day, different geographical locations and with different number of users, with the aim of checking the same conditions in all these cases. Due to the extensive use of the service offered by Nextcloud, another external threat that we were not able to control was the availability of the servers of the platform. There are different factors that affect the servers such as the quality of the network, the number of concurrent users and the use of special dates (e.g. software update releases). The state of the platform service can also affect the experiments. In order to reduce the impact of these threats, we captured traces at different times of day and, also, in different days.

Finally, threats to *construct validity* are related to the reality of our experiments, that is, whether our experiments reflect real-world situations. In this case, the main threats are the numerous scenarios that can be developed using Nextcloud and the *representative* group of packages that we used as a general population to represent them. In order to reduce its impact, we used different properties and scenarios in order to have various experiments that reflect daily experiences of their users (sending comments, sharing files, deleting files and changing file properties).

6. Conclusions and Future Work

In this paper we have introduced a complete framework, supported with tools, to perform passive testing of complex systems with two main features that strongly complicate its design and implementation: asynchronous communications and several users interacting with different objects. In addition to presenting the theoretical framework, including all the algorithms implementing its features, we have modified our tool PTTAC to implement the new framework. In order to practically show how our framework works we have detailed our experiments in a real, non-trivial system. It is very difficult to unmask errors and vulnerabilities in real, already running systems, but we were able to find a previously unknown malfunction in the WebDAV protocol.

There are several lines to extend the work reported in this paper. First, it would be interesting to complement the theoretical framework with characteristics that can increase its power to detect errors. Most notably, it would be interesting to distinguish those swaps between inputs and outputs that are due to delays and those due to an erroneous behavior of the SUT. In this line, we are considering to add probabilities to quantify the likelihood of such a swap to be admissible. In order to have a more realistic framework, and if the information is available, we would like to use time information to discard some potential swaps. We can consider our recent work [29] as an initial step. Another line of work to improve our formalism is to add data in the messages. It is well known that finite domains can be encoded without explicitly representing data but the extension would allow users to express properties in a more concise way. An alternative approach would be to introduce the notion of *role* when defining the framework. So, in addition, to the properties, we will have a set of roles assigned to each user. An action can be performed by a user only if it has a specific role associated with the action. Therefore, the process of checking traces against properties would have two phases: no unexpected action is observed and no user performs an action without having the associated role. From the practical point of view, we would like to perform more experiments with other protocols so that we can unmask hidden vulnerabilities. We are aware that this is a time-consuming activity, because it is not easy to find these errors, but it is very satisfying to see that our theoretical framework is useful in practice. Finally, it would be interesting to extend the framework to test the *rollback transitions* of a user in a multiple user scenario.

Acknowledgements

We would like to thank the anonymous reviewers of this paper for the careful reading and the detailed comments and suggestions. The changes proposed by the reviewers have certainly strengthened the paper.

This work has been supported by the Spanish MINECO/FEDER (grant number TIN2015-65845-C3-1-R); and the Region of Madrid (grant SICOMORo-CM of the program S2013/ICE-3006).

[1] C. Andrés, M. G. Merayo, and M. Núñez. Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012.

[2] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: Application to the WAP. *Computer Networks*, 48(2):247–266, 2005.

[3] R. V. Binder, B. Legeard, and A. Kramer. Model-based testing: where does it stand? *Communications of the ACM*, 58(2):52–56, 2015.

[4] C. Braunstein, A. E. Haxthausen, W.-L. Huang, F. Hübner, J. Peleska, U. Schulze, and L. V. Hong. Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In *16th Int. Conf. on Formal Engineering Methods, ICFEM'14, LNCS 8829*, pages 380–395. Springer, 2014.

[5] M. A. Camacho, M. G. Merayo, and I. Medina-Bulo. PTTAC: Passive Testing Tool for Asynchronous Systems. In *10th Int. Conf. on Signal-Image Technology & Internet-Based Systems, SITIS'14*, pages 223–229. IEEE Computer Society, 2014.

[6] A. R. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837–852, 2003.

[7] A. R. Cavalli, T. Higashino, and M. Núñez. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, 70(3-4):85–93, 2015.

[8] A. R. Cavalli, E. Montes de Oca, W. Mallouli, and M. Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *12th IEEE/ACM Int. Symposium on Distributed Simulation and Real-Time Applications, DS-RT'08*, pages 315–318. IEEE Computer Society, 2008.

[9] Cisco. Microsoft windows WebDAV SSL information disclosure vulnerability. https://tools.cisco.com/security/center/viewAlert.x?alertId=40282.

[10] C. Colombo, G. J. Pace, and P. Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.

[11] Security Focus. Microsoft windows WebDAV cve-2015-2476 man in the middle information disclosure vulnerability. http://www.securityfocus.com/bid/76234/info.

[12] FTP: File Transfer Protocol. `https://tools.ietf.org/rfc/rfc959.txt`.

[13] M.-C. Gaudel. Testing can be formal, too! In *6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, pages 82–96. Springer, 1995.

[14] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

[15] O. Henniger. On test case generation from asynchronously communicating state machines. In *10th Int. Workshop on Testing of Communicating Systems, IWTCS'97*, pages 255–271. Chapman & Hall, 1997.

[16] R. M. Hierons. Implementation relations for testing through asynchronous channels. *The Computer Journal*, 56(11):1305–1319, 2013.

[17] R. M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009.

[18] R. M. Hierons, J.P. Bowen, and M. Harman, editors. *Formal Methods and Testing, LNCS 4949*. Springer, 2008.

[19] R. M. Hierons, M. G. Merayo, and M. Núñez. Passive testing with asynchronous communications. In *IFIP 33rd Int. Conf. on Formal Techniques for Distributed Systems, FMOODS/FORTE'13, LNCS 7892*, pages 99–113. Springer,

2013.

[20] R. M. Hierons, M. G. Merayo, and M. Núñez. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1):408–424, 2017.

[21] J. Huo and A. Petrenko. Transition covering tests for systems with queues. *Software Testing, Verification and Reliability*, 19(1):55–83, 2009.

[22] I. Hwang, A. R. Cavalli, M. Lallali, and D. Verchère. Applying formal methods to PCEP: an industrial case study from modeling to test generation. *Software Testing, Verification and Reliability*, 22(5):343–361, 2012.

[23] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu, and X. Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, 14:424–437, 2006.

[24] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *5th IEEE Int. Conf. on Network Protocols, ICNP'97*, pages 113–122. IEEE Computer Society, 1997.

[25] M. Liyanage, J. Okwuibe, I. Ahmed, M. Ylianttila, O. López Pérez, M. Uriarte Itzazelaia, and E. Montes de Oca. Software defined monitoring (SDM) for 5G mobile backhaul networks. In *IEEE Int. Symposium on Local and Metropolitan Area Networks, LANMAN'17*, pages 1–6. IEEE Computer Society, 2017.

[26] A. Mammar, A. R. Cavalli, W. Jimenez, W. Mallouli, and E. Montes de Oca. Using testing techniques for vulnerability detection in C programs. In *23rd Int. Conf. on Testing Software and Systems, ICTSS'11, LNCS 7019*, pages 80–96. Springer, 2011.

[27] A. Mazurkiewicz. Traces, histories, graphs: Instances of a process monoid. In *11th Symposium on Mathematical Foundations of Computer Science, MFCS'84, LNCS 176*, pages 115–133. Springer, 1984.

[28] A. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995.

[29] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing (in press)*, 2018.

[30] Microsoft. Ms15-089: Vulnerability in WebDAV could allow security feature bypass: August 11, 2015. https://support.microsoft.com/en-us/kb/3076949.

[31] G. Morales, S. Maag, A. R. Cavalli, W. Mallouli, E. Montes de Oca, and B. Wehbi. Timed extended invariants for the passive testing of web services. In *8th IEEE Int. Conf. on Web Services, ICWS'10*, pages 592–599. IEEE Computer Society, 2010.

[32] P. Mouttappa, S. Maag, and A. R. Cavalli. Using passive testing based on symbolic execution and slicing techniques: Application to the validation of communication protocols. *Computer Networks*, 57(15):2992–3008, 2013.

[33] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 2nd edition, 2004.

[34] Nextcloud box. https://nextcloud.com/box/.

[35] Nextcloud. Demo of nextcloud software. https://demo.nextcloud.com/index.php/login.

[36] Nextcloud. https://nextcloud.com/.

[37] N. Noroozi, R. Khosravi, M. R. Mousavi, and T. A. C. Willemse. Synchrony and asynchrony in conformance testing. *Software and Systems Modeling*, 14(1):149–172, 2015.

[38] J. Peleska. Industrial-strength model-based testing - state of the art and current challenges. In *8th Workshop on Model-Based Testing, MBT'13, EPTCS 111*, pages 3–28, 2013.

[39] J. Peleska. Translating testing theories for concurrent systems. In R. Meyer, A. Platzer, and H. Wehrheim, editors, *Correct System Design - Symposium in honor of Ernst-Rüdiger Olderog on the occasion of his 60th birthday, LNCS 9360*, pages 133–151. Springer, 2015.

[40] S. Sandberg. Homing and synchronization sequences. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems, LNCS 3472*, chapter 1, pages 5–33. Springer, 2005.

[41] T. Scheffel and M. Schmitz. Three-valued asynchronous distributed runtime verification. In *12th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign, MEMOCODE'14*, pages 52–61. IEEE Computer Society, 2014.

[42] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[43] N. Shahmehri, A. Mammar, E. Montes de Oca, D. Byers, A. R. Cavalli, S. Ardi, and W. Jimenez. An advanced approach for modeling and detecting software vulnerabilities. *Information and Software Technology*, 54(9):997 – 1013, 2012.

[44] G. Shu and D. Lee. Message confidentiality testing of security protocols - passive monitoring and active checking. In *18th Int. Conf. on Testing Communicating Systems, TestCom'06, LNCS 3964*, pages 357–372. Springer, 2006.

[45] A. Simão and A. Petrenko. Generating asynchronous test cases from test purposes. *Information and Software Technology*, 53(11):1252–1262, 2011.

[46] M. Tabourier and A. R. Cavalli. Passive testing and application to the GSM-MAP protocol. *Information and Software Technology*, 41(11-12):813–821, 1999.

[47] Microsoft TechNet. Anatomy of a DoS attack that exploits WebDAV vulnerability in Apache Web server. https://blogs.technet.microsoft.com/nettracer/2011/09/08/anatomy-of-a-dos-attack-that-exploits-webdav-vulnerability-in-apache-web-server/.

[48] K. Toumi, W. Mallouli, E. Montes de Oca, C. Andrés, and A. R. Cavalli. How to evaluate trust using MMT. In *8th Int. Conf. on Network and System Security, NSS'14, LNCS 8792*, pages 484–492. Springer, 2014.

[49] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[50] H-E. Wang, K-H. Tu, J-H. R. Jiang, and N. Kushik. Homing sequence derivation with quantified boolean satisfiability. In *29th IFIP WG 6.1 Int. Conf. on Testing Software and Systems, ICTSS'17, LNCS 10533*, pages 230–242. Springer, 2017.

[51] Exchange store web distributed authoring & versioning (WebDAV) protocol. http://www.ietf.org/rfc/rfc4918.txt.

[52] B. Wehbi, E. Montes de Oca, and M. Bourdellès. Events-based security monitoring using MMT tool. In *5th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'12*, pages 860–863. IEEE Computer Society, 2012.

[53] M. Weiglhofer and F. Wotawa. Asynchronous input-output conformance testing. In *33rd Annual IEEE Computer Software and Applications Conference, COMPSAC'09*, pages 154–159. IEEE Computer Society, 2009.