FEBA: An Action-Based Feature Extraction Framework for Behavioural Identification and Authentication

***Abstract*—Behavioural authentication methods are gaining more attention day by day. This is due to the fact that they provide higher levels of security and accuracy. However, to the best of our knowledge, there is not an open-source framework in the literature to be adopted by the interested research community. In this paper, we provide the first open-source Feature Extraction Based on Action (FEBA) framework, for behavioural identification and authentication. Our framework is composed of several components for data processing, which provides the user with a means to define application-specific actions. Our framework recognizes user actions based on the received raw data, and extracts the action-specific features. The output of our framework could be used along with the existing learning, and classification algorithms in order to perform user authentication or identification. We built a complete implementation of this framework, and made it available online to facilitate future research in such context. In order to prove the performance of our proposed framework, we provide an experimental evaluation of a use case scenario, i.e., mouse movements pattern. Our thorough experimental results validate the efficacy of our proposed framework. We believe that FEBA will pave the way for researchers and developers to design and implement novel behavioural authentication mechanisms.**

***Keywords*-Behavioural Authentication; Feature Extraction; Biometrics.**

I. Introduction

Today's Internet connected world calls for more ingenious, reliable, and secure user authentication methods than the traditional password or token mechanisms. There is a rich literature on numerous issues of the password-based, and token-based authentication approaches [1]–[6]. The research community has therefore found it convenient to compliment (or even replace) traditional authentication methods with biometric-based ones [7]–[10]. The term "biometric" refers to two different types of an individual's characteristics that are supposed to be uniquely associated to a person [9], [11]: *physiological characteristics*, and *behavioural characteristics*.

A common limitation of physiological biometric authentication methods, such as facial or fingerprint recognition, is the need for special hardware devices for data collection. In contrast, behavioural biometric authentication methods are based on how the user behaves over time, and have two advantages compared to the physiological biometrics: i) each user's behaviour can be measured using existing peripherals, such as mouse, keyboard, or touch screens; and ii) user's behaviour can be actively monitored to continuously verify the user during her interaction with the system.

In order to deploy a behavioural-based authentication system, a pattern recognition framework is required, which learns the specific properties (features) of known data, and recognizes which category the new unknown data belong to. With the advances in pattern recognition tools, biometric-based authentication, and in particular, behavioural authentication methods are becoming a promising technique to improve security without impacting the user experience. There is a fairly rich literature on behavioural user authentication methods, such as [7]–[10], [12], [13], just to mention a few. Among the existing approaches, action-based user authentication methods, [14], provide a higher level of accuracy, while reducing the required time to authenticate the user.

Contribution: Despite the growing interest in the adoption of behavioural authentication systems, to the best of our knowledge there is not an open-source pattern recognition framework, which could be utilized in real-world scenarios. This motivated us to concentrate on action-based behavioural authentication methods, and provide the first open-source framework for Feature Extraction Based on user Actions, in short, *FEBA* [15]. Indeed, we focus on the *feature extractor* component of pattern recognition frameworks, since it is one of the most important components. In the case of action-based behavioural authentication systems, the feature extractor component extracts user-specific features from the user action-specific data, which itself is extracted from the raw dataset. In particular, we provide two basic components in FEBA: i) action extractor, and ii) feature extractor, while for the learning, classification and verification process we adopt some already existing algorithms.

We believe that our action-based framework, i.e., FEBA, has two important features: (i) it allows extraction of customized features specific to user actions, and (ii) it allows developers to create a more structured action hierarchy. In order to show the performance of our proposed framework, we considered a use case example: mouse-dynamics behavioural authentication, and implemented it employing FEBA. Our thorough experimental results, which we report in the paper, confirm the efficiency and ease-of-use of our framework.

II. Background and Related Work

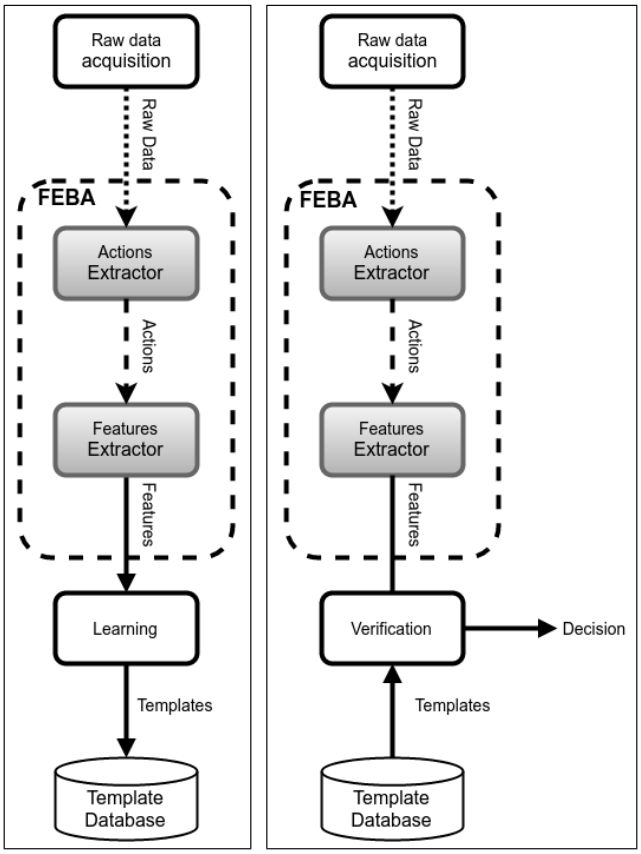
In this section, we first provide some background knowledge on pattern recognition and behavioural authentication systems. We further review some research studies in the context of behavioural authentication. Finally, as our specific focus in this context is on "action-based" pattern recognition methods, we review the most related work in the literature.

Pattern recognition refers to automated machine recognition of patterns of interest (e.g, objects, signals or images) by observing the environment, and making decision based on a given set of parameters [16]. A pattern recognition system basically consists of the following components [17]: (i) data acquisition and pre-processing, (ii) data representation, and (iii) decision making. As reported by Jain et al. [17], in order to have a successful pattern recognition system, we need to pay attention to the following issues: "pattern definition and representation, feature extraction and selection, learning and design of classification algorithms, training, and performance evaluation".

A behavioural authentication system is basically a pattern recognition system which consists of several components [14]: data acquisition, feature extraction, classifier, and database. The data acquisition component captures raw data from the interaction between the user and computer through input devices. The feature extraction component extracts a set of variables that defines behavioural characteristics of a specific user. The classifier component generates a user verification model based on the trained data (i.e., user-specific extracted features from user's past behaviour). Finally, the database stores each user's behavioural template (signature).

In the classic user authentication methods, usually the user starts the interaction with the system through a standard login page. However, since the user verification occurs just once, an attacker who hijacked a genuine user's session is not verifiable. To overcome this limitation, having a continuous verification/authentication method along the whole session is desirable. In such a scenario, behavioural authentication systems could provide a continuous authentication mechanism, in order to continuously validate the user [14]. This means that the user behaviour can be monitored silently in the background, both to acquire training information, and to check whether the current behaviour matches with the authenticated user behaviour.

Continuous authentication has attracted attention from researchers over the last decade. Several researchers proposed various methods considering different kinds of inputs, such as mouse dynamics [7], [8], [14], [18]–[20], keystroke [11], [21], [22], and touch screen [23]–[25]. Despite most of the existing continuous authentication methods that are histogram-based, Feher et al. [14] proposed an interesting action-based authentication method, from which we took inspiration for the proposed framework. Their proposed approach verifies a user based on each individual mouse action. In a histogram-based approach [18], the identification of a user is based on construction of a histogram from a large number of mouse activity events, which should be obtained over a long time. However, the proposed action-based method requires a fewer amount of action events to be collected in order to perform an accurate user verification. Moreover, the verification time is shorter than the histogram-based approaches. Furthermore, in order to characterize the mouse



(a) Registration phase (b) Verification phase

Figure 1. FEBA architectural diagrams, showing the registration and verification phases separately (the components provided by FEBA are coloured in grey)

actions, the authors create an action-hierarchy, in which the lowest level corresponds to the basic mouse events (e.g., left click). While the higher levels of the action-hierarchy are a combination of basic actions (e.g., mouse-move followed by a double click).

Our contribution compared to the method proposed in [14] is twofold: (i) we propose a general feature extraction framework, which could be adopted for every kind of behavioural authentication methods; (ii) our framework is publicly available to be used in real-world application scenarios [15].

III. Proposed Framework Architecture

In this section we describe the architectural design of the FEBA, Action-Based Feature Extraction Framework. We aim to provide both basic components and a common procedure through which a behavioural authentication system based on the user's interaction with a device can be implemented. Figure 1 shows the main components involved in the registration and verification phases. The components provided by FEBA are coloured in grey. During the registration phase, the system is trained to recognize the user's behaviour, while during the verification phase, the system recognizes the user's behaviour, e.g., authorization purposes.

The *raw data acquisition* module is a domain-specific

component that is responsible for collecting the user's behavioural data. These *raw data* are then fed to the *actions extractor* module whose aim is to derive *actions* based on patterns in the provided data. The actions extractor module can also in turn combine actions into higher level actions. The *features extractor* module is responsible for the computation of features from the previously derived actions. The features are then used during the registration phase to train the system by deriving, through the learning module, *templates*, which are stored in the *template database*. During the verification phase the templates are retrieved from the template database and are employed for comparison with the features extracted from the raw data. In the following sections we describe these components in further detail.

A. Raw Data Acquisition

The *Raw Data Acquisition* module aims at measuring and capturing the lower level data describing the user's behaviour. It is expected to provide as output a sequence of raw data objects with a fixed structure.

This module depends on the specific application domain and on the devices through which data are captured from. The implementation of such a module is out of the scope of the FEBA framework.

B. Action Extractor

The acquired sequences of raw data, obtained from the *Raw Data Acquisition* module, are fed to the *Action Extractor* module. Based on a state machine and possibly on temporal information, the raw data is processed in order to derive actions.

An action a is recursively defined as a container of either a raw data sequence $[r_1, ..., r_n]$, or actions sequence $[a_1, ..., a_m]$, which matches a specific pattern p. In particular, p specifies the conditions needed to be satisfied by the sequence, in terms of types and number of raw data elements or actions, to assert that the sequence belongs to the action a. The pattern p defines also the *type* of its associated action, i.e., if action a_i is associated to pattern p_i, and action a_j is associated to pattern p_j, then a_i is of the same type of a_j iff $p_i = p_j$.

An advantage of this general definition of the action concept is that it allows developers to create a more structured action hierarchy being still compliant to the action definition. Higher level actions can in fact be defined starting from lower level ones already defined. As an example, suppose to have defined two actions: $\{a_1^0$, $a_2^0\}$, associated to two patterns p_1^0 and p_2^0, respectively, where the top 0 means zero level. Then, considering specific combinations of zero level actions, others first level patterns $p_1^1, p_2^1, \ldots$ can be defined, and first level actions $a_1^1, a_2^1, \ldots$ can be extracted.

The *Action Extractor* module takes a sequence of raw data as input and yields a sequence of actions as output, according to the action patterns. This component just needs setting up specifying which actions could be encountered.

The *Action Extractor* engine algorithm processes raw data one by one and determines if the encountered subsequence complies with an action pattern. For each action type, the algorithm checks:

- the local conditions over each collected raw data,
- the global conditions over the whole collected subsequence.

The final decision is made by combining all the checking results in order to determine which is the right action, whose pattern matches the subsequence.

C. Features Extractor

A feature is defined as an individual measurable property of a phenomenon being observed [26]. Features are extracted from individual actions by the *Features Extractor* module. One of the main benefits of the action based approach is that customized features can be extracted based on the specific characteristics of each action, making the system modular and extensible.

Moreover, our recursive actions definition allows building actions based on sequences of other actions. In this case, the feature extractor can measure parameters taking into consideration the relations between the actions, instead of action specific features. Examples of cross-actions features can be: statistics about time between consecutive actions, or the number of occurrences of a specific type of action in a given time interval.

The *Features Extractor* engine algorithm processes a sequence of actions (previously generated by the *Action Extractor* module), extracts action specific features and finally generates K different datasets, where K is the number of different types of actions encountered in the sequence. The generated datasets, which we consider it to be a matrix, can be employed for learning and classification purposes as we will discuss in the Section III-D.

D. Learning

The goal of the *Learning* module is to create a behavioural model for each user based on training data. To this aim, it applies classification algorithms in order to create templates, which describe the user model. The template specification depends on the employed classification algorithm.

In what follows, we describe the learning procedure workflow applied on a specific action type. First the features related to the considered action types are retrieved. If the features have been stored in separate databases associated to different users, a merging operation is needed. Note that in general, considering a group of N users, the number of samples of the i-th user, i.e., P_i, varies from one user to another user. This is because each user might have different amount of training data collected in the acquisition phase. After the merging step, a unified structure, so-called *dataset*, holding both feature values and class labels values, is available. We assume the dataset for the action k to be represented by a $P \times F_k$ matrix, where $P = \sum_{i=1}^{N} P_i$ and F_k is the number of features

characterizing the k-th action type. Dataset class labels are instead held in a list of length P in which the j-th element identifies the user related to the j-th sample on the data matrix.

Finally, a learning algorithm is run on the merged dataset. In this final phase, a pattern recognition technique is applied in order to learn how classes (and so users) are distributed over the feature space. The output is a behavioural template for each user, that is stored in the *Template Database*.

The system runs the learning procedure described above for each action, creating K different templates for each user, where k is the number of different actions.

Since we followed the action-based approach, the framework does not create just a single template, instead, it computes a different template for each action type.

E. Verification

The verification procedure aims to make a decision about the user's identity, based on new user data given as input. These new data are processed by the Actions Extractor and Features Extractor modules in order to obtain their features.

The classification algorithm takes a dataset holding the samples that belong to the action and compares them against the template of the user, previously created and stored in the learning phase.

As in the learning case, the verification acts both at an action specific level and at a global level. Indeed, the system firstly runs a separate verification operation for every type of action and finally makes a definitive decision.

We assume that the output of the classification matching is a *score*, according to the probability that samples belong to the observed user. Therefore, we employ a *soft* classifier, i.e. a classifier which estimates the class probabilities explicitly. In contrast, a *hard* classifier would directly estimate the class based on defined class boundaries.

The last classification step aims to combine all the scores evaluated by each action based classifier in order to make a final binary *decision* on the user's identity.

IV. Framework Implementation

We have implemented the FEBA framework as a set of classes in the Python (version 2.7) language. Figure 2 shows the UML diagram of the framework. We have made the source code freely available at [15].

The *Action* class is the main class the whole framework revolves around. This abstract component takes care of handling a valid sequence of raw data, according to a given raw data schema, or a sequence of other lower hierarchic level actions. Furthermore, it provides methods to validate the action just specifying simple conditions over the data sequence, i.e., the action pattern. We implemented both a static validation and a dynamic one. The static validation operates over a fixed data sequence when the action is initialized. The dynamic one validates instead a partial sequence when provided raw data point by point (through the *add data* method), and computes at each step whether

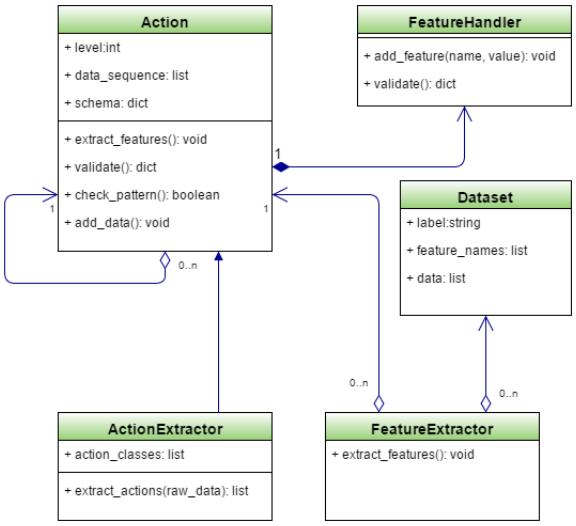


Figure 2. UML class diagram of the framework elements

the current sequence is a complete action, and/or if it is still compliant with the action pattern.

This abstract definition of an action can be specialized by developers according to the application and domain specific data. As a matter of fact, different device interaction scenarios like the mouse dynamics, the typing pattern based on keystroke dynamics or the touch screen behaviour case, have different raw data types from which different action types can be defined.

To deploy a domain specific action, developers have to firstly specify the structure of the base raw data using a *JSON schema* format. Then, they can override the *check pattern* method that specifies conditions over the data sequence, and the *extract features* method. The latter is used to compute new features customized for the specific action. Features are managed by a dedicated internal object called *FeaturesHandler*, that takes care of storing, organizing and retrieving the extracted features.

The *Action Extractor* class needs a list of action classes in order to be initialized. The *extract actions* method implements the main operation of this component. Giving a raw data sequence as input, this method adds each data to all the action objects and determines whether a specific action has to be created and stored in the output list. The advantage of having implemented the main action validation logic inside the *Action* class, is that it makes the *Action Extractor* completely independent from the patterns which identify the actions. Modifying or adding actions does not affect the *Action Extractor* engine. Figure 2 shows the relation between the action extractor component and the others entities of the framework.

The *Feature Extractor* component is the higher level component of the whole framework because it is completely unrelated with the domain specific characteristics of the application. Indeed, at the implementation level, each action instance has its own feature extraction method that is responsible for correctly extracting features from the action raw data. Therefore, the role of the Feature

extractor component is to call the feature extraction method of every action it has to process and organize the extracted features in a dataset structure. This means that this component does not need to be specialized based on the application and it is ready to be used. Indeed, once the process is completed, the collection of datasets can be stored in a database and become available for learning and classification purposes.

A *Dataset* is basically a handler of samples. A sample is a point in the N dimensional feature space extracted from an action and associated to a target (the user who performs the action). A *Dataset* instance manages and accesses the following information:

- data samples: an $M \times N$ matrix containing the feature values. N is the number of the features while M is the number of samples;
- feature description: in the simplest form it can be an N dimensional array in which the i-th element specifies the name of the ith feature in the data matrix;
- action label: a string that uniquely identifies the action type which data refer to;
- targets: an M dimensional array holding labels that specify the class the i-th sample belongs to. This attribute is optional since classes could also be unknown. Furthermore, the user id, and then the target labels, can also be managed at the database level.

Figure 3 shows how this component works and which data is processed. The primary input is a sequence of action objects. The only restriction is that each action must inherit the abstract action class defined by the FEBA framework. In the figure, three types of actions are considered as an example, each of them identified by a different color.

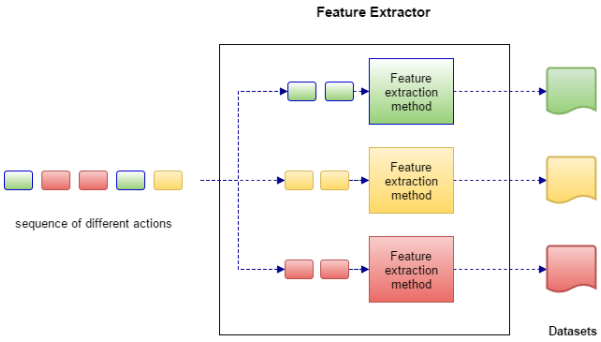


Figure 3. Feature Extractor component. It takes a sequence of different actions (indicated with different colors) and outputs a dataset for each action

As explained earlier, learning and classification modules are outside the scope of the FEBA framework. However, in order to carry out classification and learning tests, we integrated the machine learning utility "Scikit-learn" [27] in our Python framework. *Scikit-learn* is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems.

A large variety of learning algorithms can be used to create the users' templates. In this paper, we used a Random Forest classifier [28]. The simple strategy deployed in our implementation is to define a threshold, compare the scores with it and finally perform a majority vote. Formally, given the list of action scores $S = \{s_1, s_2, \ldots, s_k\}$, where k is the number of actions, the system firstly checks whether each score is above the defined threshold λ, applying equation (1) to associate S to the binary list B.

$$B = \{b_1, b_2, \ldots b_k\} : b_i = \begin{cases} 1, & \text{if } s_i > \lambda \\ 0, & otherwise \end{cases} \quad (1)$$

The final decision is made by majority vote computing $\Delta = \sum_{i=1}^{k} b_i$ where $b_i \in B$, and then applying the rule defined by equation (2).

$$\text{decision} = \begin{cases} \text{accept}, & \text{if } \Delta > \frac{k}{2} \\ \text{reject}, & otherwise \end{cases} \quad (2)$$

V. Mouse Dynamics Use Case

Here we explain our special use case example which is *mouse dynamics* scenario.

Our approach is similar to what Feher at al. [14] proposed. Indeed, we define a set of actions starting from lower level mouse events. Each action is characterized by a name and a sequence of raw data that matches a specific pattern. However, we simplify the action hierarchy proposed in [14] by identifying just the basic types of actions, without considering the combination of the *Move* action and the other ones (i.e., *Move* and *Left Click*, *Move* and *Drag and Drop*) as depicted in Table I. We considered the basic set in order to provide a general case.

Table I
Actions and pattern of raw events

Action type	Events pattern
Move	sequence of move events
Left click	left down, optional move, left up
Right click	right down, optional move, right up
Drag and drop	left down, move, left up

Figure 4 shows the mouse action hierarchy derived from basic mouse events. In what follows, we further describe how the general FEBA framework can be employed in the mouse dynamics scenario. These steps can be followed also to employ FEBA in other application scenarios.

1) *Define the raw data structure*. The structure of an atomic raw data has to be defined using a JSON schema format. A JSON schema allows to easily define the properties of data in terms of names, types, allowed values and other useful information, using a common language.
2) *Create the root action class*. The root action class is the class that directly inherits from the *Action* class of FEBA. This class just needs the raw data

schema previously defined to be initialized, and will be the parent class for all the actions represented in the considered application scenario. In this way, the implementation of the actions employed in the same scenario does not need to take into account the raw data structure, making the system more consistent. The root action in the mouse dynamic scenario was called *MouseAction*, as shown in Figure 5.

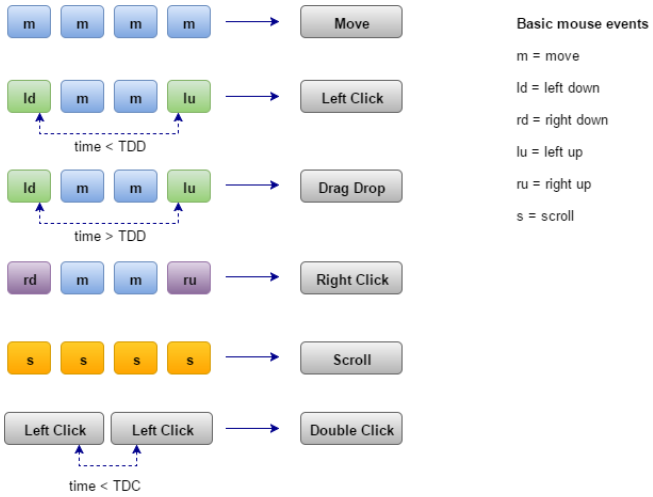


Figure 4. Mouse action hierarchy from basic mouse events

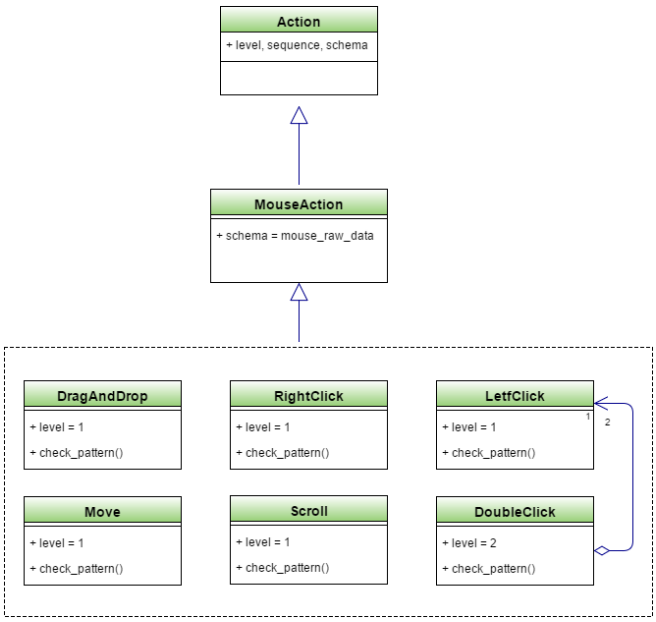


Figure 5. Mouse dynamics action classes

3) *Define the employed actions*. Extending the root class, the actions which will be concretely used during the data analysis process can be created. Three elements need to be configured: the hierarchic level, the pattern conditions and the features. In our mouse dynamics use case we considered the classes shown in Figure 5. For instance, the *LeftClick* action is a level one action in our hierarchy whose pattern can be visualized in Figure 4. Features will be better described below.
4) *Initialize the Action Extractor class*. After extending the *ActionExtractor* FEBA class, it should be initialized by providing it a list with the references to all the defined action classes.

Following the above-mentioned steps, the system is ready to process raw data, recognize actions, extract features and make datasets available.

In what follows, we present the features considered in the developed mouse behavioural recognition system. Moreover, we explain the action dependent methods implemented to measure and collect features, for each defined mouse action.

We adopted a similar approach to the one proposed in [14]. Two main feature categories were considered: move features, and click features. Click features obviously concern the click based actions that are *Left Click*, *Right Click* and *Double Click*. Table II explains how click features are computed, starting from the available attributes that characterize the click action raw data:

- t_d, t_u stand for the timestamps of the *down* and *up* events respectively;
- (x_d, y_d), (x_u, y_u) stand for the coordinates of the *down* and *up* events respectively.

Table II
FEATURES OF THE MOUSE CLICK ACTIONS

Feature name	Definition
click time	$ct = t_u - t_d$
click distance	$cd = \sqrt{(x_u - x_d)^2 + (y_u - y_d)^2}$

In Table II, the click time measures the duration of the click action and the click distance between the down and up events. The second feature might assume a zero value in most cases, but it is useful to characterize the "tremor" of the user while she is performing a click action. The *Double Click* action is a special case, as it is characterized by the two mentioned features for each *Left Click* action it is made of, and an additional pair of features considering the events between the clicks.

Move features are extracted from a sequence of raw move events and are used to characterize both the *Move* and *Drag Drop* actions. Giving a sequence of k move events, the following three sequences of attributes are involved in the feature computation task:

$$x = \{x_i\}_{i=1}^{k}, y = \{y_i\}_{i=1}^{k}, t = \{t_i\}_{i=1}^{k}$$

Where x holds for the horizontal coordinates, y the vertical ones and t the timestamp values of the move sequence elements. Pairs of consecutive elements are considered for each list of attributes, and the difference, δ, between the v values is computed by providing three additional lists:

$$\delta = \{v_{i+1} - v_i\}_{i=1}^{k-1}, \forall v \in \{x, y, z\}$$

The δ values are used to compute the basic move features shown in Table V. The basic features aren't the final ones used to represent the *Move* action. Indeed, the basic features are statistically analysed and five statistic descriptors for each basic feature are computed: average, standard deviation, maximum, minimum and range (difference between maximum and minimum). As a result, $6 \times 5 = 30$ features are collected. Moreover, three additional features are extracted and a total of 33 features are finally used. Table III gives the complete summary of the considered move features. Similar to the *Move* action, also *Drag Drop* is characterized by move features, considering the movement activity from the mouse down to the mouse up events. Finally, Table IV summarizes the number of features extracted for each different type of action.

Table III
COMPLETE MOVE FEATURES

Feature name	**Definition**
avg, std, max, min, max - min	the specific statistic of basic features in V
duration	$d = t_k - t_0$
traveled distance	$Td = \sum_{i=1}^{k} \sqrt{{\delta_x}_i^2 + {\delta_y}_i^2}$
Straightness	$\frac{\sqrt{(x_k - x_0)^2 + (y_k - y_0)^2}}{Td}$

Table IV
NUMBER OF FEATURES EXTRACTED

Action	**Features**	**Number of features**
Move		33
Left Click	click time, click distance	2
Right Click	click time, click distance	2
Drag Drop	move features from left down to left up event	33
Double Click	features of the two clicks, time and distance between clicks,	6

Table V
BASIC MOUSE MOVE FEATURES

Feature name	**Definition**
Angles	$\theta_i = \arctan \frac{\delta_{x_i}}{\delta_{y_i}}$
Vertical velocity	$v_y = \frac{\delta_y}{\delta_t}$
Horizontal velocity	$v_x = \frac{\delta_x}{\delta_t}$
Velocity	$v = \sqrt{v_x^2 + v_y^2}$
Acceleration	$a = \frac{\delta_v}{\delta_t}$
Jerk	$J = \frac{\delta_a}{\delta_t}$

VI. RESULTS

This section presents the results of an experiment based on the use case described in SectionV. The goal of the experiment is to demonstrate that FEBA can be successfully employed in real use case scenarios.

A. Data Collection

The first step of our experiment was to collect data through a mouse dynamics acquisition module implemented as an extension of the Google Chrome Web Browser. The extension was distributed among a group of ten users asking them to install it in their own laptop. Before starting the actual data capture, users where prompted for some general information by a form provided by the extension.

Tester users were asked to use the mouse, interacting with their laptop and computing their normal tasks. Furthermore, each user was asked to play with the online game accessible through the plugin for at least twenty minutes. Clicking on the blue button, the extension redirects to the web page running the game. We took advantage of the free online game "Don't cross the line" [1]. The game consists in a reverse puzzle that requires moving the lines in the right order to solve the interwoven pattern. We chose this specific game as it encourages the user to use drag and drop, move and click actions and for letting the different users interact with a common environment and perform similar tasks.

At the end of the data collection phase, raw mouse data belonging to ten users were uploaded to a remote database. Each user had an interaction of at least twenty minutes with the browser game, collecting an average of 30000 consecutive raw events.

B. Feature Extraction

The experiment goal is to emulate an authentication scenario in which users attempt to prove their identity through the mouse dynamics.

A set of ten users is considered. The set is randomly split in two subsets: a subset of *internal users* U_{int}, and a subset of *external users* U_{ext}. In our experiment, eight internal users and two external ones were considered.

Internal users are assumed to be registered into the system, and their data is used to create the templates in the training phase, while external users are considered as unknown by the system. This means that the system is not aware of their mouse behavioural profile and doesn't include it in the template creation during the training phase.

Three different authentication cases are considered:

1) A genuine user $u_0 \in U_{int}$ attempts to authenticate herself;
2) An internal impostor user $u_i \in U_{int}, i \neq 0$, tries to authenticate herself claiming to be the genuine one;
3) An external impostor user $u_e \in U_{ext}$ tries to authenticate herself claiming to be the genuine one

Figure 6 shows the main operations which the test procedure is made up of. First of all, the available raw data collected are divided into two sets: training data, and test data. The training set is created from the first 66% of total raw data, while rest of data composes the test set.

[1] available at http://play.famobi.com/dont-cross-the-line

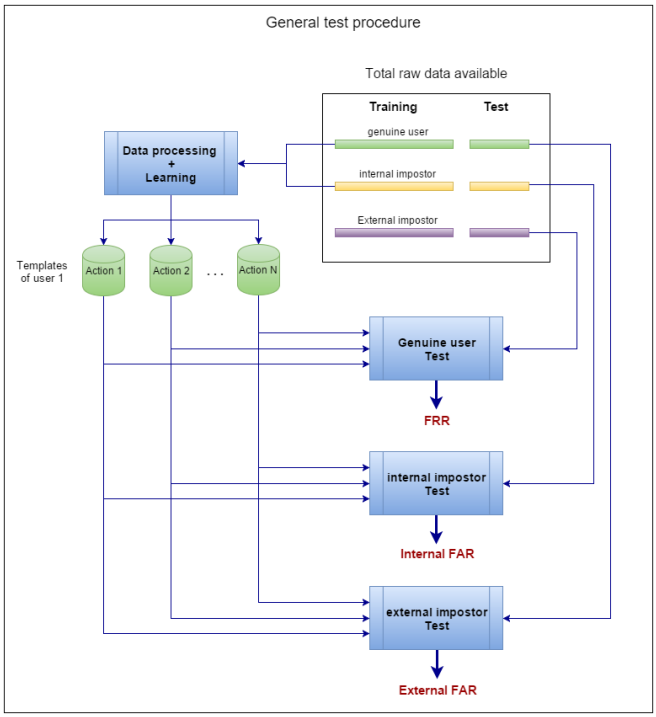


Figure 6. General test procedure

Training data belonging to the selected genuine users along with the training data of other internal users are processed to create a genuine user template for each different type of action. In order to accomplish this task, each raw data sequence is processed by the data processing module which extract actions and features. Finally, features extracted from every internal user are processed by the learning module and action based templates for the genuine user are created and stored. The learning algorithm employed was the *Random Forest* [28].

In contrast, the training data of external users aren't involved in the template creation. However, since raw data are time ordered, we split the raw data sequence of external users in the same way of the internal ones to have an homogeneous test data.

Once the training phase is completed, the test phase can start. The test procedure is iterated over a list of threshold values, used in the decision making step. In our experiment we use a static threshold applied to every action but other dynamic approaches can be employed. For a given threshold value, we test each of the cases described in 3. Furthermore, three action types are involved: *Left Click*, *Move*, *Drag Drop*, which are the main actions executed by the users when playing with the browser game. Table VI shows the number of actions extracted from 30000 raw data points, i.e. the average size of the training set, for each action type.

Genuine users raw data test are used to simulate a login attempt. Action and feature extraction is accomplished as usual, and for each action the system makes a comparison against the user template and estimates a score. Finally, scores are compared with the threshold and a reject/accept

Action type	Number of actions
Move	3000
Left Click	700
Drag Drop	2200

Table VI
AVERAGE NUMBER OF ACTIONS EXTRACTED FROM 30000 RAW DATA

decision is made, as described in section IV. This operation is performed varying the target genuine user and considering a number of 10 input data tests per round. Based on the number of rejected tests over the total ones, the False Recognition Rate (FRR) is finally computed.

The same testing procedure is applied in the internal and external impostor cases, evaluating the number of times in which the impostor is wrongly accepted as if she were the genuine user. In this way, the internal False Acceptance Rate (FAR) and the external FAR are evaluated. In the next section we present the obtained results.

C. Performance

The test procedure described in the previous section was applied to get a preliminary evaluation of the system performance.

Table VII shows the variation of FAR and the FRR compared to the score threshold used in the decision making phase. Furthermore, the corresponding ROC curve are shown both for the internal impostor case, in figure 7, and the external one, in figure 8. The horizontal axis has the label FPR(t) that stands for *False Positive Rate* for the given threshold t; it represents the FAR (external or internal) in table VII. Instead, the vertical axis TPR(t) stands for *True Positive Rate* and is related to the FAR in table VII: TPR$(t) = 1 - \text{FAR}(t)$. This means that TPR(t) indicates the number of the attempts of a genuine user that were correctly accepted by the system.

Obviously, the lower the threshold value is, the lower the FRR is, because the system access condition is more relaxed. This means that the variability of the genuine user features as respect to the known user's pattern is more tolerated. In contrast, the FAR is higher, because the distance between the impostor user features and the genuine user's pattern is allowed to be higher too. As a consequence, it's easier for an impostor to be confused as the genuine user.

A common parameter employed to have an idea of the system performances is the AUC. The internal impostor test shows a AUC equal to 0.7556 while in the external impostor test it's equal to 0.7175. These values reveal that the system is not yet ready to be deployed in production. As a matter of fact, good AUC values should be greater that 0.8 to make the system suitable to be used in practice. However, although the performance in terms of error rate seem not to be excellent, the results show that a discrimination among different users is accomplished, so that the system doesn't behave like a random selector.

Indeed, the test shows that the developed prototype works in the right way. As a matter of fact, it can be seen that the external FAR is a bit higher than the

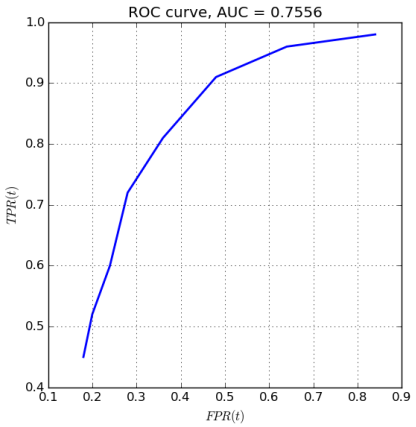


Figure 7. ROC curve in the internal impostor case

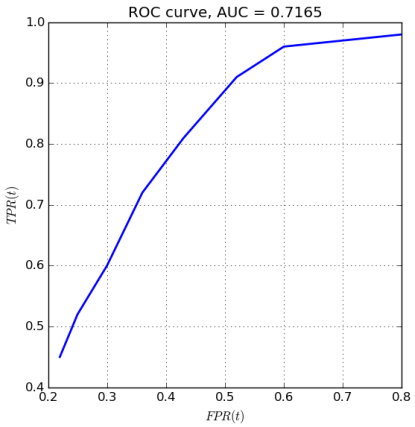


Figure 8. ROC curve in the external impostor case

Threshold	FRR	int FAR	ext FAR
0.1	0.02	0.84	0.8
0.15	0.04	0.64	0.6
0.2	0.093	0.48	0.52
0.25	0.19	0.36	0.43
0.3	0.28	0.28	0.36
0.5	0.4	0.24	0.3
0.7	0.48	0.20	0.25
0.8	0.55	0.18	0.22

Table VII
VARIATION OF FRR AND FAR WITH THRESHOLD

corresponding internal FAR for a given threshold. This highlights that the learning procedure works correctly, since the test data of internal users are more accurately discriminated thanks to the previous use of training data. Test data of external user are instead completely unknown by the system, and can therefore be wrongly classified more frequently. Furthermore, it also emerges that the difference between the external FAR and the internal one is enough contained. Indeed, both the values can be controlled, carefully choosing the threshold value.

The test results also have to be put in the context of our experiment. The group of ten users considered in our test provided to us preliminary results. In order to increase the accuracy of the experiments, the number of users in the experiment should be significantly higher. Moreover, more discriminant features and distinctive actions should be investigated in order to efficiently improve the classification process. FEBA allows to easily extend the feature extraction operation of a given action without the need to change neither the system structure or the other implemented components. An additional parameter that can be optimized in order to obtain better performance is the threshold value. In this experiment the threshold was statically imposed, and the same threshold value was used for every action type. A better approach could consist in determine the threshold dynamically, based either on the specific user's pattern and the action type. This method would need the introduction of an additional validation phase in which the threshold should be evaluated from known data.

VII. Conclusions and Future Work

In this paper we proposed an authentication framework that can be employed in the development of a behavioural authentication system based on human to device interaction. An action based approach was proposed. Moreover, the approach was analyzed and generalized to make it usable in more than one application context. Furthermore, a mouse pattern based behavioural authentication system was implemented, adopting the general architecture of the designed framework. We described the framework components and the mouse pattern based implementation focusing on the data work flow, from the acquisition of raw data, to the learning and classification based on extracted features.

As a future work, we aim to provide a complete user authentication protocol. In particular, supplement FEBA with other required components, such as learning and classification modules, in order to provide a complete ready-to-use building block for an authentication system.

Finally, we further propose some possible future work:

- Take advantage of the framework infrastructure to create other implementation based on different behavioural characteristics, as the mobile touch behaviour;
- Improve the mouse pattern based behavioural authentication system already implemented, adding more complex actions and new features with a better discrimination accuracy;
- Spread the data collection procedure out in order to have a larger and more various data-set on which more accurate tests can be performed;
- Adapt the implemented mouse based system to the continuous authentication use case and make it usable not only for testing purposes but even in a real application scenario.

References

[1] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring password guessability for an entire university,"

in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS'13. ACM, 2013, pp. 173–186.

[2] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. S& P'12. IEEE, 2012, pp. 538–552.

[3] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson, "Password managers: Attacks and defenses," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 449–464.

[4] M. Raza, M. Iqbal, M. Sharif, and W. Haider, "A survey of password attacks and comparative analysis on methods for secure authentication," *World Applied Sciences Journal*, vol. 19, no. 4, pp. 439–444, 2012.

[5] L. O. Gorman, "Comparing passwords, tokens, and biometrics for user authentication," *Proceedings of the IEEE*, vol. 91, no. 12, pp. 2021–2040, 2003.

[6] E. G. Lebovitz, "A Survey of Authentication Mechanisms," Working Draft, IETF Secretariat, Internet-Draft draft-iab-auth-mech-07.txt, Feb. 2010.

[7] N. Zheng, A. Paloski, and H. Wang, "An efficient user verification system via mouse movements," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS'11. ACM, 2011, pp. 139–150.

[8] C. Shen, Z. Cai, X. Guan, Y. Du, and R. A. Maxion, "User authentication through mouse dynamics," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 1, pp. 16–30, 2013.

[9] K. O. Bailey, J. S. Okolica, and G. L. Peterson, "User identification and authentication using multi-modal behavioral biometrics," *Computers & Security*, vol. 43, pp. 77–89, 2014.

[10] S. Mondal and P. Bours, "A computational approach to the continuous authentication biometric system," *Information Sciences*, vol. 304, pp. 28–53, 2015.

[11] P. S. Teh, A. B. J. Teoh, and S. Yue, "A survey of keystroke dynamics biometrics," *The Scientific World Journal*, vol. 2013, 2013.

[12] N. Zheng, A. Paloski, and H. Wang, "An efficient user verification system using angle-based mouse movement biometrics," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 3, p. 11, 2016.

[13] Z. Sitov, J. ednka, Q. Yang, G. Peng, G. Zhou, P. Gasti, and K. S. Balagani, "HMOG: New behavioral biometric features for continuous authentication of smartphone users," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 5, pp. 877–892, 2016.

[14] C. Feher, Y. Elovici, R. Moskovitch, L. Rokach, and A. Schclar, "User identity verification via mouse dynamics," *Information Sciences*, vol. 201, pp. 19–36, 2012.

[15] "Feba - feature extraction framework for behavioural identification and authentication source code," 2016. [Online]. Available: https://www.dropbox.com/sh/om7ux81zdvbpikg/AAAc50tgqwXxPSxjkhpW9yu3a?dl=0

[16] R. Polikar, "Pattern recognition, in wiley encyclopedia of biomedical engineering," *Ed. Akay, M., New York, NY: Wiley*, 2006.

[17] A. K. Jain, R. P. Duin, and J. Mao, "Statistical pattern recognition: A review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4–37, 2000.

[18] A. A. E. Ahmed and I. Traore, "A new biometric technology based on mouse dynamics," *IEEE Transactions on dependable and secure computing*, vol. 4, no. 3, p. 165, 2007.

[19] Z. Jorgensen and T. Yu, "On mouse dynamics as a behavioral biometric for authentication," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 476–482.

[20] S. Mondal and P. Bours, "Continuous authentication using mouse dynamics," in *Biometrics Special Interest Group (BIOSIG), 2013 International Conference of the*. IEEE, 2013, pp. 1–12.

[21] M. Karnan, M. Akila, and N. Krishnaraj, "Biometric personal authentication using keystroke dynamics: A review," *Applied Soft Computing*, vol. 11, no. 2, pp. 1565–1573, 2011.

[22] P. Kang and S. Cho, "Keystroke dynamics-based user authentication using long and free text strings from various input devices," *Information Sciences*, vol. 308, pp. 72–93, 2015.

[23] M. Frank, R. Biedert, E.-D. Ma, I. Martinovic, and D. Song, "Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 1, pp. 136–148, 2013.

[24] A. Jain and V. Kanhangad, "Exploring orientation and accelerometer sensor data for personal authentication in smartphones using touchscreen gestures," *Pattern Recognition Letters*, vol. 68, pp. 351–360, 2015.

[25] G. Kambourakis, D. Damopoulos, D. Papamartzivanos, and E. Pavlidakis, "Introducing touchstroke: keystroke-based authentication system for smartphones," *Security and Communication Networks*, 2014.

[26] C. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer, 2006. [Online]. Available: https://books.google.it/books?id=kTNoQgAACAAJ

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[28] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.