SN Computer Science (2022) 3:450
https://doi.org/10.1007/s42979-022-01293-9

ORIGINAL RESEARCH

Plascua: Programming Language Support for Continuous User Authentication

Julius Muganji[1] · **Engineer Bainomugisha**[1]

Received: 25 May 2021 / Accepted: 30 June 2022 / Published online: 20 August 2022
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2022

Abstract

Compared to traditional user authentication methods, continuous user authentication (CUA) provide enhanced protection, guarantees against unauthorized access and improved user experience. However, developing effective continuous user authentication applications using the current programming languages is a daunting task mainly because of lack of abstraction methods that support continuous user authentication. Using the available language abstractions developers have to write the CUA concerns (e.g., extraction of behavioural patterns and manual checks of user authentication) from scratch resulting in unnecessary software complexity and are prone to error. In this paper, we propose new language features that support the development of applications enhanced with continuous user authentication. We develop `Plascua`, a continuous user authentication language extension for event detection of user bio-metrics, extracting of user patterns and modelling using machine learning and building user authentication profiles. We validate the proposed language abstractions through implementation of example case studies for CUA.

Keywords Continuous user authentication (CUA) · Explicit user authentication (EUA) · Language abstractions · Machine learning

Introduction

Nowadays mobile computing devices provide support for various application services including e-banking, e-mail, personal health, social media, conferencing and so on. This has made the mobile computing devices to be more vulnerable to various security threats because of the confidential information processed and stored in them [2, 22, 34]. A lot of research has been carried out to mitigate the underlying causes where methods such as explicit authentication have been implemented to provide support of possible security measures on to these devices. Currently mobile devices are largely authenticated using the explicit methods such as passwords, pins, unlock patterns, access codes, fingerprints and others.

✉ Engineer Bainomugisha
baino@mak.ac.ug

Julius Muganji
muganji93@gmail.com

[1] Department of Computer Science, Makerere University, Kampala, Uganda

Although explicit user authentication methods provide a high level of security, their effectiveness is not guaranteed due to the pitfalls associated with them, for example, users find difficulties in using explicit authentication methods [24, 28, 46], are not completely secure [46], are intrusive [28] and intermittent [47] in nature, are easily forgotten by users [46], are subjected to brute force attacks [46], sometimes sensors fail to work [6], hardware and sensor dependency [15] and to some extent they are disabled by users [24]. Furthermore, the explicit authentication methods cannot detect intruders after one-time authentication has been done.

Continuous or implicit authentication methods have emerged as the way to address the recurring pitfalls associated with explicit authentication approaches through being non-intrusive and adding second security layer on top of the existing explicit methods. These measures continuously monitor an authenticated user throughout the interactions with the device or the specific application service [20]. The system keeps on checking the authenticity of the user through comparing live biometric data with already known data. Although research has been carried out investigating on the various approaches and implementation of continuous user authentication [11, 15, 16, 23, 24, 28, 32, 46],

surprisingly less work on defining reusable abstractions for continuous user authentication has been done. To the best of our knowledge, there is no existing work on providing support of continuous user authentication at language level. For example, Chowdhury et al. [13] and Kiyani et al. [25] focus on the frameworks and the properties for CUA but less on providing the appropriate language abstractions to enable developers incorporate continuous use authentication in their software applications, leaving end-users vulnerable to unauthorised access and attacks.

In this paper, we lay foundational work for reusable abstractions for continuous user authentication at the programming language level. We investigate possible design dimensions that enhance continuous user authentication at a programming language level. Our goal is to model a language extension coupled with newly developed language abstractions necessary for expressing continuous behaviour-based authentication.

Our contribution is a language extension that serves as a proof-of-concept for continuous user authentication. We propose a number of reusable methods that can be leveraged by developers in modelling applications coupled with continuous user authentication. We designed a demonstration app using the new language features that we let users interact with to experiment the effectiveness of the proposed and implemented abstractions.

Motivating Scenarios

To illustrate the need for new programming language abstractions for continuous user authentication, we introduce a series of scenarios. We adopt scenarios as defined in previous research [3].

Online Assessments

The COVID-19 pandemic has meant that many schools and institutions of learning have had to adopt online learning and teaching. One of key challenges faced in these circumstances is student authentication, particularly for assessments. This has renewed research into novel authentication methods for online learning, teaching and assessment. Consider a scenario where a student is enrolled into an end of semester examination. A traditional e-learning system verifies the student at the start of the exam, for example, with a password or two-level authentication methods. It remains a difficult task to verify whether the student was authenticated at the start of the exam is the same student who completed the exam. In such scenarios, threats occurs when a substitute student completes an exam on the behalf of a valid student who was verified at the start. This kind of vulnerability is referred to as an insider threat [31]. Therefore, the system need to continually verify the identity of a student throughout the entire examination session.

An Emailing System

Consider a scenario of an emailing system in a company. A traditional emailing system verifies a user at the initial stage of accessing a client application, for example with a password and email address or two-level authentication methods. Issues arises when the emailing system fails to verify whether the user who was authenticated at the start is the same user who typed and sent an email. In such scenarios, a threats occurs when an intruder types and sends an email using a session of the valid user who was verified at the start. This kind of vulnerability is referred to as an outsider threat [31] and mostly occurs when a valid authenticated user leaves an active session without logging off and an intruder takes ownership of the valid session and completes activities on the behalf of the valid user. The system would consider such activities to be coming from a valid user. This scenario can be manifested in many other systems, for example, online banking systems. A valid customer can initiate a session after going through an authentication process. The communication channels between user transactions and bank servers are secured to avoid security. A security breach may occur at a user level where a valid authenticated customer leaves an active session without logging off. An intruder may take ownership of the valid session and completes transactions on behalf of the valid user.

On Device Broken User Authentication

Computing devices (e.g., smartphones, tablets, laptops and ATMS) are associated with a lot of data processing applications including bank applications, e-commerce tools, email applications, personal health, social media and others. These applications contain sensitive information, of which securing it using the existing explicit security measures alone still remains a challenge in the current state. This has created room for developing applications enabled with continuous user authentication where users are continuously validated their authenticity as they utilize resources of smart devices. Consider a smart device that permits the current user to use email applications, online banking, e-commerce and others. To continuously secure the devices with its associated applications, such smart devices or applications can be enhanced with continuous user authentication to continuously grant access to the resources of the smart devices to the genuine user as well as denying impostor users.

The Need Language Support for Continuous User Authentication

The above scenarios show the need for CUA in modern software applications for enhanced security and user experience. While the CUA functionality is appealing, it is not straightforward to implement in the underlying language. For example, the developer would need to decide which events to monitor to derive conditions for CUA, implementing logic to continuously check whether the valid is logged in or not, implement concerns of what happens to the application state when the user is no longer authenticated, among others. The above scenarios and the resulting technical challenges motivate the need for dedicated language to implement CUA behaviour the same way there are dedicated language abstractions and models to deal with application concerns such as concurrency and distribution. Without dedicated CUA language abstractions, developing effective continuous user authentication applications using the current programming languages is a daunting task leading to the well known phenomena of accidental complexity in Software Engineering [12]. Using the available APIs and libraries developers have to manually write the CUA concerns (e.g., extraction of behavioural patterns and manual checks of user authentication) from scratch resulting in unnecessary software complexity and are prone to error. We present our CUA language model in the next section.

Overview of the Plascua Language Model

In this section we present Plascua, a programming language extension that supports implementation of continuous user authentication concerns in modern software applications. The Plascua model builds upon the Flute programming language [9], that models programming executions as first-class citizens that are constrained to run only when certain contextual conditions are true and can be seamlessly interrupted or resumed depending on the developer specified contextual conditions. Plascua extends this concept to model user behaviours as conditions that must be continuously satisfied for the user to be authenticated and given access to a software system.

The main properties of the Plascua language extension are: authentication behaviour events, authentication user patterns, authentication user profiles, computational interruptions and continuous authentication.

Authentication Behaviour Events

CUA depends on the events that describe the behaviour of the user. These events could include for example, the typing pattern from the keyboard or device orientation patterns from an accelerometer sensor. Therefore, sensor data recording serves as the fundamental key feature in implementing continuous user authentication in Plascua language. With this feature, we provide appropriate abstractions so that user biometrics can be derived from the underlying sensors without the developer being concerned about the underlying implementation details. This abstraction is provided as a layer on top with various low-level recording APIs including keyboard recording API [18], mouse recording API [19] and others to fetch user events from sensors. The main objective of this feature is to listen and record user biometrics required for authentication of users.

Authentication User Patterns

Two categories of features are extracted from user raw data. The idea behind this is to improve the performance of user authentication process and to reduce the amount of data required for enrolment and authentication of users [5]. The extracted features include **Timing** and **non-conventional** features [5]. In total we extract 34 features from a typing task where 25 are timing features [4] and 9 non-conventional features [5] that we fuse to form one dataset that we subject to machine learning algorithms for training to generate a user profile for further authentication processes.

Authentication User Profile

User profiling in this context refers to building of a valid user authentication profile that can be used to further validate the authenticity of user. In Plascua, we build a user profile where the extracted and fused features from Sect. 3.2 are subjected to machine learning algorithms to train a prediction model and later deploy and saved as a valid user profile for further authentication purposes. We use supervised machine learning algorithms to train authentication models as these have been studied and proven to be effective in various research papers [4, 5, 14, 21, 29, 41].

Computational Interruptions

Computational interruptions refers to the ability for the programming environment to temporarily stop an ongoing execution in response to external event while remaining with the ability to resume the execution when certain conditions are satisfied. This is unlike program exceptions where program execution can be stopped but cannot be resumed from the same point in case there is a need. Resumption of a stopped or interrupted execution is important in the context of CUA because it is assumed that the application execution can be stopped at any moment in case authentication conditions are not satisfied. Catering for such scenarios in Plascua to prevent termination of the service, we define

various interruptions that may arise through context predicates that are checked periodically as the service runs. As in Flute programming language [9], if the service realizes any violation of the context predicate, the program execution is captured and the service is either suspended, aborted, resumed or restarted depending on what state the executing process may be.

for validity of the current user. The result is continuously checked through the application execution.

We explore different supervised learning algorithms including KNN classifier, SVM classifier, Random Forest classifier, Voting classifier, Decision Tree classifier, Extra Trees classifier and One-Vs-Rest classifier in the scikit-learn library [30]. The usage of different machine learning models

```
1
2 def startRecordingEvents(self,sensor=None,wait_time=None):
3         sensorToUse = None
4         waitTime = None
5         if not sensor:
6             sensorToUse ='keyboard'
7         else:
8             sensorToUse = sensor
9         if not wait_time:
10            waitTime = 30
11        else:
12            waitTime = wait_time
13        results = self.decapitalizeSensorName(sensorToUse)
14        if results =='mouse':
15            mouse.hook(self.record_mouse_events)
16            time.sleep(waitTime)
17            mouse.unhook_all()
18        elif results == 'keyboard':
19            keyboard.hook(self.record_keyboard_events)
20            time.sleep(waitTime)
21            keyboard.unhook_all()
22        else:
23            raise ValueError('Sensor supplied is not supported. The value of sensor was: {}
    '.format(results))
```

Listing 1 `startRecordingEvents`

Continuous Authentication

Continuous authentication in Plascua is the ability to constantly check the user patterns and authentication conditions to ensure the validity of the user who is accessing a service. The flow of events starts with recording authentication behaviour events from configured sensors, then extract authentication of user patterns from the recorded authentication behaviour events, subject the authentication user patterns to machine learning algorithms for prediction was introduced to compare model performance and choose the best given the nature of the authentication user patterns subjected to them. We use machine learning to train user profile using user patterns extracted from behaviour events observed from sensors. Once the model has been trained and tested, we then deploy the model as the authentication user profile for a current user to be used to continuously verify the same user through prediction.

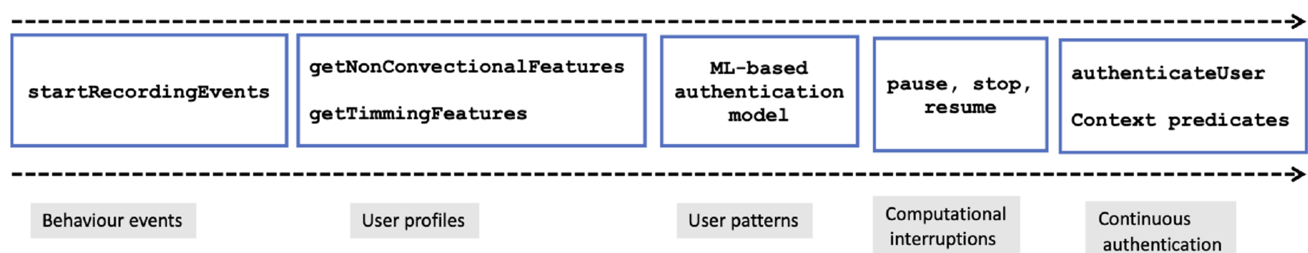

Fig. 1 Plascua abstractions for continuous user authentication

Implementation of Plascua as a Language Extension

In this section, we present the implementation of the Plascua model as a Python language extension and the abstractions it provides for developers to implement CUA concerns in new or existing software systems. Figure 1 presents a summary of the key abstractions in Plascua.

Abstractions for Behaviour Events

User bio-metrics serves as the key feature in continuous user authentication and these are fetched from the underlying sensors. On sensor data recording we limit our scope and focus mainly on user bio-metrics from the keyboard sensor. We collect data from various sensors including keyboard, mouse, application usage, touch screens, GPS and others. In Plascua, we leverage data from the existing sensors such as keyboard, mouse, GPS and we further leverage the existing recording APIs such as [18] and [19] to fetch data from these sensors.

In Listing 1 we define `startRecordingEvents` that takes in two optional parameters including **sensor** and `wait_time`. Both parameters are set to none by default implying that if nothing is parsed on the calling of the method, the function loads the default configurations where keyboard is taken as the sensor and 30 s are considered as the waiting time. We use the waiting time parameter in the recording method to control on the volumes of data that may clog the application.

Feature Extraction

In listing 2 we define `getNonConventionalFeatures` and `getTimingFeatures` to achieve the task of extracting features from user raw data. Both methods must be supplied with user raw data in an array format as the argument on invoking them.

```
1  def getTimingFeatures(self,data):
2          data,data1,data2,data3,data4 = self.adjacentNamePairs(data,'up','down')
3          adjacent = []
4          secondadjacent =[]
5          thirdadjacent=[]
6          fourthadjacent=[]
7          noneadjacent = []
8          for i in data:
9              adjacent.append((i[1][1],i[1][2],i[1][3],i[1][4],i[1][5]))
10         for k in data1:
11             secondadjacent.append((k[1][1],k[1][2],k[1][3],k[1][4],k[1][5]))
12         for w in data2:
13             thirdadjacent.append((w[1][1],w[1][2],w[1][3],w[1][4],w[1][5]))
14         for t in data3:
15             fourthadjacent.append((t[1][1],t[1][2],t[1][3],t[1][4],t[1][5]))
16         for q in data4:
17             noneadjacent.append((q[1][1],q[1][2],q[1][3],q[1][4],q[1][5]))
18         if len(secondadjacent) and len(thirdadjacent) and len(fourthadjacent) > 0:
19             ad,sd,td,ft,non = np.array(adjacent),np.array(secondadjacent),np.array(
       thirdadjacent),np.array(fourthadjacent),np.array(noneadjacent)
20             adMean,sdMean,tdMean,ftMean,nonMean = np.mean(ad,axis=0),np.mean(sd,axis=0),np.
       mean(td,axis=0),np.mean(ft,axis=0),np.mean(non,axis=0)
21             q =np.vstack((adMean,sdMean))
22             w = np.vstack((q,tdMean))
23             z = np.vstack((w,ftMean))
24
25             return pd.DataFrame(data=np.vstack((z,nonMean)),columns=['HKey1','HKey2','DD','
       UU','UD'])
26         else:
27             self.getTimingFeatures(data)
```

Listing 2 getTimingFeatures

The getNonConventionalFeatures extracts both the semi-timing features [5] and editing features [5] in the fetched user raw data from the sensor. The method takes data as its parameter and data in this context represents authentication behaviour events and returns a 1 by 9 array of features that are arranged in this order negUpDown,negDownDown, capsLockPercentage, errorPercentage,wpm, rsaPercentage, lsaPercentage, rasPercentage, lasPercentage respectively. In listing 3 we demonstrate how we extract the non conventional features from user data. The method takes data as the parameter which is of an array data type. In the body of the method we extract the total number of words a user has typed in the current session, we further extract all the total errors that are recorded as the user types, shift and caps lock keys instances are as well counted and lastly, we count all the Up-Down and Down-Down negatives associated with the current session. In the rest of the method we compute the percentages, append every result into a list and return the results in an array format for further processing.

Building a User Profile

Data Preparation and Processing

To build a user profile, we subject the collected data to processing techniques such as sampling and scaling. The former is applied to address the issues of data imbalance that is associated with the fetched events on training. To improve the purity of data we employed principal component analysis(PCA) using sklearn PCA library [39] and Standardization using sklearn StandardScaler [40]. PCA involves casting data from a higher dimension to a lower dimension which involves selecting a few features that have relevant information from a huge database for training [39]. Standardization involves transforming data with a uniform distribution of mean as 0 and a standard deviation as 1 [40]. Listing 5 shows demonstrates how we process data with PCA and StandardScaler.

```
1  def getNonConventionalFeatures(self,data):
2          features = []
3          totalWords = self.countAllEvents(data,'down','space')
4          totalErrors = self.countAllEvents(data,'down','backspace')
5          capsLock,rsa,lsa,ras,las = self.countShittsAndCapsLockKeys(data)
6          negUpDown,negDownDown  = self.countNegatives(data)
7          rsaPercentage = (rsa/totalWords) * 100
8          lsaPercentage = (lsa/totalWords) * 100
9          rasPercentage = (ras/totalWords) * 100
10         lasPercentage = (las/totalWords) * 100
11         capsLockRate = capsLock/totalWords
12         capsLockPercentage = capsLockRate * 100
13         errorRate = totalErrors/totalWords
14         errorPercentage = errorRate * 100
15         wpm = ((totalWords - totalErrors)/totalWords)/60
16         features.append(
17             (negUpDown,negDownDown,
18              capsLockPercentage,errorPercentage,
19              wpm,rsaPercentage,lsaPercentage,
20              rasPercentage,lasPercentage))
21         nonConvectionFeatures = np.array(features)
22         return nonConvectionFeatures
```

Listing 3 getNonConventionalFeatures

In a typing session of a user, we extract two sets of features timing and non-conventional features as we explain in our previous sections. We further proceed with fusing the datasets to have one huge dataset that we subject to machine learning algorithm as we illustrate in listing 4. The ultimate goal of fusing is to improve on the performance of final authentication model [5, 8, 42].

```
1  def featureFusing(self,data,data2):
2          return np.concatenate((np.array(data).reshape(1,-1),np.array(data2).reshape(1,-1)),
       axis=1)
```

Listing 4 featureFusing

```
df_minority_upsampled = resample(negative_data, replace=True, n_samples = 52,random_state
    =123)
df_upsampled = pd.concat([positive_data,df_minority_upsampled])
x = df_upsampled.drop('Subject', axis=1)
Y = df_upsampled['Subject']
y = df_upsampled['Subject'].values
data = {}
data['total'] = pd.DataFrame(StandardScaler().fit_transform(x))
data['pca3'] = pd.DataFrame(PCA(n_components=3).fit_transform(data['total']))
data['pca5'] = pd.DataFrame(PCA(n_components=5).fit_transform(data['total']))
data['pca10'] = pd.DataFrame(PCA(n_components=10).fit_transform(data['total']))
data['pca20'] = pd.DataFrame(PCA(n_components=20).fit_transform(data['total']))
data['pca25'] = pd.DataFrame(PCA(n_components=25).fit_transform(data['total']))
```

Listing 5 Sampling, PCA and Scaling

Training Authentication Model

We extract numerous authentication user patterns from the authentication behaviour events. We fuse the authentication user patterns to form a single consolidated dataset and subject them to machine learning algorithms to predict the validity of the current user. Given the user behaviour of the current user, we predict the likelihood that he/she is the valid user. User bio-metrics such as keystroke dynamics have been studied under both supervised [4, 5, 14, 21, 29, 41] and unsupervised [1, 45] machine learning algorithms and we focus on supervised machine learning algorithms with keystroke dynamics.

Supervised machine learning refers to the building of algorithms that can detect patterns and hypotheses using externally supplied instances to predict the fate of future instances [38]. For example, in [10], supervised machine learning is described as a classification technique where class labels are assigned to data objects based on the relationship between the data items with a pre-defined class label. The goal of supervised machine learning is to build a brief and comprehensive distribution of class labels in terms of predictor features [26]. In Plascua, machine learning serves two purposes, the machine learning algorithms are used to learn the various user patterns from the extracted user biometric features through training. On learning, an authentication is trained, tested and deployed which later is used to authenticate users. Machine learning is as well used in the authentication method to make predictions. The authentication method takes a set of newly extracted user biometric features from a live typing session and makes a prediction against the existing user profile. On building and deploying authentication user profiles in Plascua, we apply supervised machine learning algorithms in learning the various user patterns in the extracted user biometric events for users. Machine learning algorithms in Plascua serves as building block for modeling valid user profiles that can be used to authenticate users. With building an authentication user profile, we train the authentication model with 34 features extracted from authentication behaviour events. The features are categorized into two, timing and non-conventional features as we describe in "Feature Extraction" section.

On evaluating the machine learning models, we used Receiver operating characteristic (roc_curve) to compute the **True Positive Rate (tpr), False Positive Rate (fpr)** and **Thresholds**. We further used these metrics to compute the Equal Error Rate (EER) metrics which means if the False Positive Rate is equal to True Positive Rate and the lower EER is the higher the accuracy of the model. The figure below shows the various accuracy scores and the evaluation metric scores we obtained from every algorithm we tested with user bio-metrics.

Building and Deploying the Model

We deploy the trained authentication model as a dump a serialized format using the `pickle` library [33]. The serialized object can be consumed by the authentication method we defined earlier for validating the authenticity of a user. Loading the serialized model, we used pickle to load the dumped user authentication model to make predictions. In listing 6 we demonstrate how we deploy and load the authentication user profile.

```
tuple_obj = (X_train,rf_best)
filename = 'UserProfile.pkl'
pickle.dump(tuple_obj, open(filename, 'wb'))
model = pickle.load(open('UserProfile.pkl',
    'rb'))
```

Listing 6 Deploying and Loading of Authentication Model

Implementation of Computational Interruptions

In this section, we explain we implement computational interruptions in Plascua. The conditions for computational interruptions are expressed in terms of predicate methods that we explain below.

Predicate Methods

Predicate methods describe the valid execution context that fulfils the expected user behaviour on a device. Predicate methods return true or false, true indicating the validity of the execution content, false otherwise. We consider a number of possible constraints that have to be fulfilled to ensure smooth continuous user authentication. In Plascua, we focus on three predicate methods that we believe to provide a valid context of the executing process and these include the lock status of the hosting device, idleness of the current session and the typing status of the configured sensor. In listing 7 we demonstrate how we defined the lock predicate method in Plascua.

```
def isWorkStationLocked(self,predicate=None) -> bool:
        predicateToUse = None
        if not predicate:
            predicateToUse = (ctypes.windll.user32.GetForegroundWindow() == 0)
            if predicateToUse:
                return True
            else:
                return False
        else:
            if predicate:
                return predicate
            else:
                return predicate
```

Listing 7 isWorkStationLocked Method

Interruptible Executions

Continuous user authentication is a background service that runs repeatedly while interacting with user operations at various levels. The authentication process is a context-dependent service where interruptions may as well be triggered by the dynamic execution contexts the program interacts with during its execution. Change of the execution context through which the program is executing may be caused by either user operations or system behaviours. These abrupt interruptions may subject the service to unplanned crashing and freezing. Managing such interruptions at an early stage guarantees a smooth running of the authentication service without crashing and freezing. We extend our language with managing executions where various interruption strategies are reused as defined in [9]. Some of the functions we develop include pause and stop, the former suspends the execution of the current process if the specified context condition is not satisfied. The Pause function further saves the state of the execution. On saving and resuming the execution context, we leverage the existing library called dill [17] that offers various functionalities. In Plascua, we focus on two functions from dill including the dump_session and load_session. The former saves the current session of a user and the later reloads the saved session from disk to resume the execution. The later aborts the executing process if the specified context condition is not satisfied. The stop function in our language, terminates the current running process and no chances of resuming the terminated process. For an execution to be suspended or aborted, specific context conditions have to be satisfied. In listing 8 we demonstrate how the implementation of interruptible executions in Plascua.

Model	Accuracy Score	EER
VotingClassifier	86%	0.168
KNN Classifier	95%	0.083
Random Forests	90%	0.153
DecissionTrees	81%	0.196
OneVsRestClassifier	90%	0.099
SVM Classifier	95%	0.083

Fig. 2 Accuracy scores of tested learning algorithms

```
1 def pause(self,*context_conditions):
2         if not context_conditions:
3             if not self.isRunning():
4                 if path.exists('.userSession/UserSession.pkl'):
5                     filename = '.userSession/UserSession.pkl'
6                     _pickle.dump_session(filename)
7                 else:
8                     os.mkdir('.userSession')
9                     filename = '.userSession/UserSession.pkl'
10                    _pickle.dump_session(filename)
11
12            else:
13                pass
14        else:
15            for predicate in context_conditions:
16                if predicate:
17                    if path.exists('.userSession/UserSession.pkl'):
18                        filename = '.userSession/UserSession.pkl'
19                        _pickle.dump_session(filename)
20                    else:
21                        os.mkdir('.userSession')
22                        filename = '.userSession/UserSession.pkl'
23                        _pickle.dump_session(filename)
24
25                else:
26                    pass
```

Listing 8 pause

Resumable Executions

In Plascua, interruptions are managed by defining various functions that either suspend or abort a process when the predefined context predicates are not satisfied. Since the interruptions occur when the context predicates are not satisfied, there is a room for these predicates to be satisfied again and the suspended process has to either resume or restart. According to [9], the resumable executions are categorized into two including **The interrupted execution is resumed** and **The interrupted execution is restarted**. The former involves resuming the suspended process from where it stopped from. This is only possible when the context predicate regains its validity and the suspended process is resumed from where it stopped from. The later involves restarting either a suspended or aborted process. In the listing 9 we demonstrate how we implement the resumption of a suspended context in Plascua.

```
def resume(self,*context_conditions):
        """
        Resumes the process if the isPaused predicate is True.
        """
        if not context_conditions:
            if self.isPaused():
                filename = '.userSession/UserSession.pkl'
                with open(filename,'rb') as file:
                    _pickle.load_session(file)
            else:
                pass
        else:
            for predicate in context_conditions:
                if predicate:
                    filename = '.userSession/UserSession.pkl'
                    with open(filename,'rb') as file:
                        _pickle.load_session(file)
                else:
                    pass
```

Listing 9 `resume`

Language Methods For Authenticating Users

On authenticating users with Plascua, we define `authenticateUser` method that makes prediction of newly fetched extracted user details against already existing authentication user profile for validity. In listing 10 below we demonstrate how `authenticateUser` was implemented in Plascua.

```
def authenticateUser(self,profile,data) -> bool:
        """
        Returns True if the user profile predicts the supplied data to 1 else false.
        Profile: UserProfile
        data: new data
        """
        if profile.predict(data) == 1:
            return True
        else:
            return False
```

Listing 10 `authenticateUser`

Validation and Testing

Considering the motivational scenario in "An Emailing System" section we explain a simple mail continuous user authentication application developed using our Plascua language. As introduced in the previous sections, the mail authentication application is built to continuously authenticate users while interacting the mailing system across different platforms. For this particular example we develop a mail authentication application that is constructed as a service composed of all the ingredients pf continuous user authentication as we explain in third section. The recording of sensor events layer manages the fetching of user sensor details from the hosting device. The extraction of user patterns layer manages the extracting of user features from the fetched user events. The authentication layer predicts the extracted features against the saved authentication user profile. In addition, the mail continuous user authentication application manages interruptions from various contexts while the application is running. We break down individual ingredients of continuous user authentication and demonstrate how we reuse them to model the mail user authentication application. In listing 11 below we demonstrate how we implement the listening and recording of sensor events using Plascua.

```
1  import sys
2  import pickle
3  import PlascuaEvaluator as Plascua
4  import time
5  _Plascua = Plascua.Plascua()
6
7  def recordEvents():
8      if not isContextValid():
9          time.sleep(2)
10         recordEvents()
11     else:
12         _Plascua.startRecordingEvents()
13         hasRecorded = True
14
15 def isContextValid() -> bool:
16     if _Plascua.isRunning() and _Plascua.isTyping():
17         return True
18     else:
19         return False
20
21 def getUserData():
22     return _Plascua.getSensorData()
```

Listing 11 `recordEvents`

Feature Extraction

```
1  def getCurrentData(data):
2      TimingFeatures= _Plascua.getTimingFeatures(data)
3      nonTimingFeatures = _Plascua.getNonConventionalFeatures(data)
4      results = _Plascua.getTrainingData(TimingFeatures,nonTimingFeatures)
5      hasComputed = True
6      return results
```

Listing 12 `getCurrentData`

Listing 12 demonstrates the extraction of features from sensor data in Plascua language extension. The method takes in sensor recorded events and extracts both Timing and non-Timing features which later are transformed into one huge dataset and returned as the result. In listing 13 we demonstrate how authenticating of users could be achieved using the methods in Plascua language extension.

```
1  def authenticateUser(profile,data):
2      if _Plascua.authenticateUser(profile,data):
3          hasAuthenticated = True
4          print("The user is valid")
5          main()
6      else:
7          print("Invalid user")
8          _Plascua.lockWorkStation()
```

Listing 13 `authenticateUser`

The `authenticateUser` represents the validation of user authenticity in Plascua language extension. The method takes in a current saved authentication user profile and the newly extracted authentication user patterns from the recorded events. The method performs a prediction of the new submitted features against the saved authentication user profile and if the submitted authentication user patterns matches with authentication user patterns in the saved authentication user profile, the current user continues to manipulate his or her tasks on the authenticating device while silently repeating the authentication process in the background else the work station is locked. In Fig. 2, we demonstrate what response is printed on the screen given the

```
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======= RESTART: D:\Latest Engineer\Engineer\plascua\testingLibrary.py ======
The user is valid
The user is valid
Invalid user
>>> |
```

Fig. 3 Valid and invalid user response

different authentication behaviour events that are subjected to the existing authentication user profile of the authenticating application or device (Fig. 3).

The Fig. 2 contains two valid activities and one invalid activity. The `authenticateUser` method validates the authenticity of the user and prints on the screen "The user is valid" if true and "Invalid user" is false. On situations where the user is valid, the continuous authentication continues to grant access to resources while the invalid user, the work station of the validating user is locked instantly. In listing 14 we summarize how all the stated processes above are added into the main program of the mail continuous authentication application.

Discussions

The main goal of this study was to model a new language coupled with reusable methods that aid in developing applications coupled with continuous user authentication in smart devices. The newly developed language is rich in various features including the support to manage computation interruptions where the authentication can be suspended, resumed, restarted or stopped depending on the context under which the application is being conducted. Although the Plascua language is meant to be used by various developers to come up with applications coupled with continuous user authentication on any smart device, our greatest challenge on supporting most of the available sensors that aid in continuous user authentication still remains. Plascua currently supports continuous user authentication through keyboard sensor on any smart device.

```
1  def main():
2      recordEvents()
3      if hasRecorded and not isContextValid():
4          pauseAuthentication()
5          print('Paused and saved the context....')
6      elif processPaused and _Plascua.isPaused():
7          resumeAuthentication()
8          print('Paused process resumed......')
9      elif restartableValidContext():
10         restartAuthentication()
11         print('Process restarted......')
12     elif stoppableValidContext():
13         stopAuthentication()
14         print('Process stopped......')
15     else:
16
17         recordedData = getUserData()
18         testdata = getTestingData(recordedData)
19         clearHistory(recordedData)
20         validateUser(_userprofile,testdata)}
```

Listing 14 Main

The listing 14 represents the entire flow of the events involved in the continuous user authentication process as defined in the Plascua language extension. The process starts with capturing sensor data, then different predicate functions are checked to validate if the ongoing process should proceed, resume, restart or stop as configured. If neither of the above predicate functions is valid, the program continues to execute the next commands as aligned in the method. The recordedData variable in the method represents the retrieving of all the captured user sensor data and which are submitted to the getTestingData method which extracts all the required features and stored under testData variable for further processing.The clearHistory method represents the resetting of the testData variable making it possible to be assigned a new value on every call that is made on the main program. Lastly the authenticateUser method does the validation of the new testData against the saved and loaded user authentication profile.

Future Work

This work is a first step towards support of continuous user authentication at a language level. In this regard, our research opens way for a number of interesting topics for future research. In this section we discuss some of the future directions resulting from this work.

Extension of the Recording Sensors

As discussed in our previous chapters, we mainly focus on supporting keyboard sensor in our new language. This has been considered to be the most fundamental as a lot of research has been done that has proven the keyboard

sensor to be the most ideal in implementing continuous user authentication. The question of what more sensors can be supported in our language that can aid in continuous user authentication still remains open for future research.

Cross-Platform Usage of the User Profile Model

In our approach to supporting continuous user authentication at language level, we generate a user authentication model from various extracted user patterns from different sessions. We test that the generated model works across the same hardware platform. The question of how the generated authentication model can work across different smart devices with the same sensors still remains open for future work. e.g a user profile to be able to authenticate the same user on a computer and a smart phone.

Context Environment for Continuous User Authentication

In our language we define predicate functions that determine valid under which continuous user authentication can be conducted. A few predicate conditions have been implemented that we believe to achieve the desired goal of defining a valid context environment. The challenge with these predicate functions is, some are dependant on the supported sensor. The question of what more predicate functions can be implemented that can define valid context environment under which continuous user authentication can be conducted with other sensors still remains open for future research.

Related Work

The work presented in this paper, builds upon the state of the art on language support for uninterruptible context-dependent executions [9], user authentication approaches and methods in literature [35]. The remainder of this section provides a review of those approaches.

Interruptible Context-Dependent Executions

The interruptible execution context paradigm has recently been proposed for the development of context-dependent applications whose interruptions can be managed at different levels of execution [9]. The language is built on two strategies including interruptible executions and resumable executions. The former involves aborting and suspending an execution if the defined execution context becomes violated and the later involves resuming a suspended execution and restarting when the execution context regains its validity.

Continuous User Authentication

Various methodologies have been proposed to support continuous user authentication and these include behavior-based authentication methods and graphical and geo-location-based authentication methods. Behavioral-based authentication methods leverage human interface devices (HIDs) to identify users by their mouse/keyboard operating characteristics. HIDs such as mouse and keyboards are used to record data that uniquely identify a user when operating with smart devices [35]. These methods can as well be categorized as sensor-based authentication methods which leverage biometric characteristics of a user recorded by inbuilt sensors of smart devices for authentication. The concept of these authentication systems lies on the fact that users perform gestures in a unique way that depends on how they hold the phone, and on their hand's geometry, size, and flexibility and it is these gestures that are transformed into password signatures that uniquely identify the user in the authentication process. Sensor-based password authentication systems mainly focuses on the geometry and biokinetics of the user's hand in the gestures and some of the sensors involved include the touch screen, gyroscope and accelerometer sensors [27, 44]. Some of the other works that demonstrate the implementation of continuous user authentication through behavior-based methods include [6, 11, 15, 16, 23, 24, 28, 32].

Graphical and Geo-location-Based Authentication Methods

These are authentication methods that leverage already existing known maps and images to generate a password to uniquely identify a user in the authentication process. With these methods, a visual map or an image is presented to a user and various locations or points are selected by a user which are transformed into a unique password signature that identifies a user. The logic of selecting different locations on a map or points on an image varies depending on the approach used for example, geopass [37] uses an online map and only provides an option of selecting one geographical location from the entire map for a password and the location includes latitudes and longitudes co-ordinates, smartpass [36] provides multiple locations to be selected by a user on a world map to generate a unique password and passpoints [7] works by generating a password from a sequence of clicks of points a user selects from an image. In geo-location based password authentication methods, specific services are processed with in authorized locations and this is accomplished by leveraging of the GPS sensors that synchronize locations within a specified time range while enabling and disabling of application services [48]. Passmap [46] also another

implementation of continuous user authentication, this is built on the concept of landmarks and justifies this with a human memory being good at remembering landmarks on a well-known journey [43].

Acknowledgements The authors would like to acknowledge the feedback received from faculty in the Department of Computer Science at Makerere University.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Ahmed AA, Traore I. Biometric recognition based on free-text keystroke dynamics. IEEE Trans Cybern. 2014;44(4):458–72. https://doi.org/10.1109/TCYB.2013.2257745.
2. Alotaibi E, Albar A, Hoque M. Mobile computing security: issues and requirements. J Adv Inf Technol. 2016;7(1). https://doi.org/10.12720/jait.7.1.8-12.
3. Alsolami E. An examination of keystroke dynamics for continuous user authentication. Ph.D. thesis, Queensland University of Technology; 2012. https://eprints.qut.edu.au/54730/.
4. Alsultan A, Warwick K. Keystroke dynamics authentication: a survey of free-text methods; 2013. www.IJCSI.org.
5. Alsultan A, Warwick K, Wei H. Improving the performance of free-text keystroke dynamics authentication by fusion. Appli Soft Comput J. 2018;70:1024–33. https://doi.org/10.1016/j.asoc.2017.11.018.
6. Araujo L, Sucupira L, Lizarraga M, Ling L, Yabu-Uti J. User authentication through typing biometrics features. IEEE Trans Signal Process. 2005;53(2):851–5. https://doi.org/10.1109/TSP.2004.839903.
7. Ashwini J. Authentication for attacks in graphical passwords pass points style. Int J Adv Comp Sci Cloud Comput. 2013;1(11):2321–4058.
8. Ayotte B, Huang J, Banavar MK, Hou D, Schuckers S. Fast continuous user authentication using distance metric fusion of free-text keystroke data. In: 2019 IEEE/CVF conference on computer vision and pattern recognition workshops (CVPRW); 2019. p. 2380–2388. https://doi.org/10.1109/CVPRW.2019.00292.
9. Bainomugisha E, Vallejos J. Interruptible context-dependent executions: a fresh look at programming context-aware applications; 2012. p. 67–84. https://doi.org/10.1145/2384592.2384600.
10. Bhavsar H, Ganatra A. A comparative study of training algorithms for supervised machine learning. Int J Soft Comput Eng (IJSCE). 2012;2:1–8.
11. Brocardo ML, Traore I, Woungang I. Toward a framework for continuous authentication using stylometry. In: Proceedings—international conference on advanced information networking and applications, AINA; 2014. https://doi.org/10.1109/AINA.2014.18.
12. Brooks FP. No silver bullet essence and accidents of software engineering. Computer. 1987;20(4):10–9. https://doi.org/10.1109/MC.1987.1663532.
13. Chowdhury MA, Light J, McIver W. A framework for continuous authentication in ubiquitous environments. In: 2010 Sixth international conference on wireless communication and sensor networks; 2010. p. 1–6. https://doi.org/10.1109/WCSN.2010.5712289.
14. Davoudi H, Kabir E. A new distance measure for free text keystroke authentication. In: 2009 14th international CSI computer conference; 2009. p. 570–575.
15. Deutschmann I, Lindholm J. Behavioral biometrics for DARPA's active authentication program. IEEE; 2013. p. 1–8.
16. Eremin A, Kogos K, Filina A. A concept of continuous user authentication based on behavioral biometrics. In: 2017 20th conference of open innovations association (FRUCT); 2017. p. 62–68. https://doi.org/10.23919/FRUCT.2017.8071293.
17. Foundation PS: dill . PyPI. https://pypi.org/project/dill/.
18. Foundation PS: keyboard . PyPI. https://pypi.org/project/keyboard/.
19. Foundation PS: mouse. PyPI. https://pypi.org/project/mouse/.
20. Frank M, Biedert R, Ma E, Martinovic I, Song D. Touchalytics: on the applicability of touchscreen input as a behavioral biometric for continuous authentication. IEEE Trans Inf Forensics Secur. 2013;8(1):136–48. https://doi.org/10.1109/TIFS.2012.2225048.
21. Gunetti D, Picardi C. Keystroke analysis of free text. ACM Trans Inf Syst Secur. 2005;8(3):312–47. https://doi.org/10.1145/1085126.1085129.
22. Jang-Jaccard J, Nepal S. A survey of emerging threats in cybersecurity. J Comput Syst Sci. 2014;80(5):973–93. https://doi.org/10.1016/j.jcss.2014.02.005 (**Special issue on dependable and secure computing**).
23. Juola P, Noecker J, Stolerman A, Ryan M, Brennan P, Greenstadt R. Towards active linguistic authentication. IFIP Adv Inf Commun Technol. 2013;410:385–98.
24. Kayacık H, Just M, Baillie L, Aspinall D, Micallef N. Data driven authentication: on the effectiveness of user behaviour modelling with mobile device sensors. mobile security technologies; 2014. p. 1–11. http://web.cs.dal.ca/~kayacik/papers/MOST14.pdf.
25. Kiyani AT, Lasebae A, Ali K, Rehman MU, Haq B. Continuous user authentication featuring keystroke dynamics based on robust recurrent confidence model and ensemble learning approach. IEEE Access. 2020;8:156177–89. https://doi.org/10.1109/ACCESS.2020.3019467.
26. Kotsiantis SB. Supervised machine learning: a review of classification techniques. Informatica. 2007;31:249–68.
27. Lee WH, Lee RB. Implicit smartphone user authentication with sensors and contextual machine learning. proceedings—47th annual IEEE/IFIP international conference on dependable systems and networks, DSN 2017; 2017. p. 297–308. https://doi.org/10.1109/DSN.2017.24.
28. Lin CC, Chang CC, Liang D, Yang CH. A new non-intrusive authentication method based on the orientation sensor for smartphone users. In: Proceedings of the 2012 IEEE 6th international conference on software security and reliability, SERE; 2012. p. 245–252.
29. Messerman A, Mustafić T, Camtepe SA, Albayrak S. Continuous and non-intrusive identity verification in real-time environments based on free-text keystroke dynamics. In: 2011 international joint conference on biometrics, IJCB 2011; 2011. https://doi.org/10.1109/IJCB.2011.6117552.
30. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: machine learning in python. J Mach Learn Res. 2011;12:2825–30.
31. Pusara M. An examination of user behavior for user re-authentication. Theses and dissertations available from ProQuest; 2007. https://docs.lib.purdue.edu/dissertations/AAI3291194.
32. Pusara M, Brodley CE. User re-authentication via mouse movements. In: Proceedings of the 2004 ACM workshop on visualization and data mining for computer security—VizSEC/DMSEC '04; 2004. p. 1. https://doi.org/10.1145/1029208.1029210.

33. Python Software Foundation: pickle—Python object serialization. https://docs.python.org/3/library/pickle.html.
34. Roman R, Lopez J, Mambo M. Mobile edge computing, Fog et al.: a survey and analysis of security threats and challenges. Future Gener Comput Syst. 2018;78:680–98. https://doi.org/10.1016/j.future.2016.11.009.
35. Shen C, Cai Z, Guan X. Continuous authentication for mouse dynamics: a pattern-growth approach. In: Proceedings of the international conference on dependable systems and networks; 2012.https://doi.org/10.1109/DSN.2012.6263955.
36. Shin J, Kancharlapalli S, Farcasin M, Chan-Tin E. SmartPass: a smarter geolocation-based authentication scheme. Secur Commun Netw. 2015. https://doi.org/10.1002/sec.1311.
37. Shivhare B, Sharma G, Kushwah SPS: A study on geo-location authentication techniques. In: Proceedings—2014 6th international conference on computational intelligence and communication networks, CICN 2014; 2014. https://doi.org/10.1109/CICN.2014.161.
38. Singh A, Thakur N, Sharma A. A review of supervised machine learning algorithms. In: 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom); 2016. p. 1310–1315.
39. sklearn.decomposition.PCA—scikit-learn 0.23.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.
40. sklearn.preprocessing.StandardScaler—scikit-learn 0.23.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.
41. Tappert CC, Villani M, Cha SH. Keystroke biometric identification and authentication on long-text input. In: Behavioral biometrics for human identification: intelligent applications. IGI Global; 2009. p. 342–367. https://doi.org/10.4018/978-1-60566-725-6.ch016.
42. Teh PS, Andrew Teoh BJ, Ong TS, Neo HF. Statistical fusion approach on keystroke dynamics. In: Proceedings—international conference on signal image technologies and internet based systems, SITIS 2007; 2007. p. 918–923. https://doi.org/10.1109/SITIS.2007.46.
43. TheMindTools.com: The Journey Technique - Memory Skills Training from MindTools.com. https://www.mindtools.com/pages/article/newTIM$_05.htm$.
44. Wang H, Lymberopoulos D, Liu J. Sensor-based user authentication. In: Proceedings European conference on wireless sensor networks, vol. 8965; 2015. p. 168–185.
45. Xiaofeng L, Shengfei Z, Shengwei Y. Continuous authentication by free-text keystroke based on CNN plus RNN. In: Procedia computer science, vol. 147. Elsevier B.V.; 2019. p. 314–318. https://doi.org/10.1016/j.procs.2019.01.270.
46. Yampolskiy RV. User authentication via behavior based passwords. In: 2007 IEEE long island systems, applications and technology conference, LISAT; 2007. p. 82–89. https://doi.org/10.1109/LISAT.2007.4312636.
47. Yazji S, Chen X, Dick RP, Scheuermann P. Implicit user re-authentication for mobile devices. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 5585. Berlin, Heidelberg: Springer; 2009. p. 325–339.
48. Zhang F, Kondoro A, Muftic S. Location-based authentication and authorization using smart phones. In: 2012 IEEE 11th international conference on trust, security and privacy in computing and communications. IEEE; 2012. p. 1285–1292. https://doi.org/10.1109/TrustCom.2012.198.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.