TOWARD THE SYSTEMATIZATION OF ACTIVE AUTHENTICATION RESEARCH

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Daniel Fleming Gerrity

June 2015

COMMITTEE MEMBERSHIP

TITLE: Toward the Systematization of Active Authentication Research

AUTHOR: Daniel Fleming Gerrity

DATE SUBMITTED: June 2015

COMMITTEE CHAIR: Zachary N J Peterson, Ph.D.
Assistant Professor of Computer Science

COMMITTEE MEMBER: Philip Nico, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Foaad Khosmood, Ph.D.
Assistant Professor of Computer Science

ABSTRACT

Toward the Systematization of Active Authentication Research

Daniel Fleming Gerrity

Authentication is the vital link between your real self and your digital self. As our digital selves become ever more powerful, the price of failing authentication grows. The most common authentication protocols are static data and employed only once at login. This allows for authentication to be spoofed just once to gain access to an entire user session. Behaviometric protocols continuously consume a user's behavior as a token of authentication and can be applied throughout a session, thereby eliminating a fixed token to spoof. Research into these protocols as viable forms of authentication is relatively recent and is being conducted on a variety of data sources, features and classification schemes. This work proposes an extensible research framework to aid the systemization and preservation of research in this field by standardizing the interface for raw data collection, processing and interpretation. Specifically, this framework contributes transparent management of data collection and persistence, the presentation of past research in a highly configurable and extensible form, and the standardization of data forms to enhance innovative reuse and comparative analysis of prior research.

ACKNOWLEDGMENTS

A special thanks to John Gerrity for assisting with data visualizations.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1 - Introduction

Authentication is the bridge between availability and two other major goals of information security [1], confidentiality and integrity. The difficulty of constructing effective authentication protocols contributes to frustrating situations which seemingly admit only two out of the three goals. Information etched on a titanium plate, encased in concrete and sunk to the bottom of the sea possesses great integrity and is highly confidential, but is not available. The same plate displayed on a public monument is available and has integrity, but is no longer confidential. One's own memory is both confidential and available, but compared to artificial records, its capacity to preserve significant quantities of information intact is very limited. With the advent of the information age, the need to maintain vast digital records securely has exploded. Thus the need for effective authentication methods has become even more acute [2, 3].

The primary approach to authentication has been, in essence, to replace the original security problem with another, smaller and more controlled one. The clearest form of this is the yoking of the record's confidentiality and integrity to that of some arbitrary, but easier to secure, object or record. This method of authentication is commonly referred to as authentication *by what you know* or *by what you have*. Smart cards, social security numbers, proximity badges, and the infamous password [4], are all examples of this technique. In order to secure buildings and banks accounts we would convert the problem to that of securing someone's pocket and remembering 15 characters or your favorite aunt's dog's name. Arbitrariness is most advantageous when a virtually unlimited number of credentials is needed, or when compromise requires a reset. However, it is also the

method's greatest weakness. An impostor can just as truly enter a password or present a proximity badge as any authorized person.

The opposite of an arbitrary mapping of authentication to authorized personnel is an essential mapping, which is dominated by protocols based on *what you are*. A series of technologies, such as biometrics, aims at such a mapping to respond to the shortcomings of arbitrary secrets. Most practical applications exploit essential features of the user that are static, such as thumb prints, iris patterns, DNA, facial features, or vocal passwords. These offer the promise of eliminating the arbitrary nature of the smaller security problem, immediately enhancing its availability and confidentiality. However this comes at a cost, mostly in the form of overhead in both time and money to support specialized interfaces that can accurately assess these features.

Applying these protocols to a work flow is far from transparent. Furthermore, while it is more difficult to spoof someone's thumb print than to copy a password, any static feature is a complete form at all times and so may be isolated and cloned (e.g. lifted prints). What is sorely needed is an authentication protocol that is essential to the user, transparent to the workflow, and of a strictly incomplete form.

Behaviometrics [5] is another branch of biometrics, but its subjects are just such incomplete forms. Behavior is the "how" of a more abstract action. Unlike the form of an iris pattern, which at all times is present, characteristic behavior is not. Moreover, it is fundamentally bound to other forms, which precludes a simple cloning procedure to replicate it. We can never simply act nervousness or write complicatedness - we must be doing some concrete thing nervously or writing with complexity. Behavior is essentially

bound to both our actions and our identity. Even if an attacker could capture a representation of our behavior, he would have the problem of forging desired actions. Behavior as an authentication token offers a protocol that is essential, transparent, and resists counterfeiting. Given today's advancements in computational power, this form of embedded authentication opens the door to a novel benefit of complete transparency: continuous authentication.

Continuous authentication, also known as active authentication (AA), is a relatively new field of serious investigation [6]. It intends to ascertain the practicality of systems to continuously execute authentication protocols based on behaviors rather than static forms. Since every user-action contains input for the authentication protocol, the protocol need not interrupt the user. Instead, continuous authentication both relieves the user and eliminates the single temporal point of failure, typical of static methods.

Several behavior modalities have been tried [7, 8, 9, 10], including keystroke dynamics, pointing device movements, writing styles, web-browsing habits, gait analysis, and even grammar corrections. Many of these areas show promise, and some may have the potential to reach static biometric accuracy and beyond. One study of mouse-movements has achieved error rates of less than 2% with only 20 events [9]. But the overarching drawback is that none of these modalities is continuously active. Multi-modal systems are presently being designed to handle the bursty nature of mono-modal sensors and allow for an easily extensible system to fuse the input of such sensors [10, 11, 12]. However, replication and systemization of work in the field is hampered by the incredible diversity of modalities, features, classifiers, architectures, and test sets employed.

We propose a multi-modal framework for the Linux input subsystem to aid research in this field by standardizing the interface for raw data collection, mono-modal modules and central information fusion algorithms. This framework provides three distinct levels of standardized interfaces to accelerate the consistent improvement of both mono-modal and multi-modal classifiers:

1) Raw Resource Layer - a simple set of interfaces presenting raw data streams specifically optimized for active authentication for future feature development. Active authentication requires access to fine-grained behavioral data to extract robust patterns that are unique to individuals. Operating systems do not always aggregate or present the level of detail required for behavioral features used in active authentication in convenient forms. Collecting and presenting raw data streams for each input source in a standard interface provides a consistent basis to build and compare classifiers. A universal interface of raw inputs also allows the framework to be easily extensible.

2) Mono-Modal Layer - a standardized interface for each mono-modal system to comply with in order to expose minimum and useful metrics to the multi-modal system. Since active authentication's goal is to eliminate single points-of-failure in time by providing continuous authentication, the ability to integrate many separate modalities of behavior is critical to overcome gaps of input in any single modality. Establishing a uniform interface that mono-modal authentication systems present as output, allows easy aggregation of multiple modalities. A standard interface also allows easy addition of novel mono-modal systems.

3) Fusion Layer - an extensive interface aggregating all resources and loaded modal modules to be presented to whatever central fusion algorithm is desired. There is yet to be discovered an optimal algorithm by which to fuse the work of multiple mono-modal systems advantageously into a multi-modal system. The fusion interface presents all possible resources to researchers from the two previous layers. Computing the optimal authentication decision from multiple modalities may require more supporting information for each modality such as raw data flow statistics, historical accuracy, time since last decision, etc. beyond the simple authentication acceptance or rejection.

This framework allows researchers to add or replace modules at any of the three levels, thereby allowing for the aggregation and refinement of diverse attempts. Resource preprocessing, mono-modal features, and multi-modal fusion systems can now be cross tested rapidly, as demonstrated by the four resource modules, three mono-modal modules, and two multi-modal modules created to seed the frame work.

In this work, we describe the design and engineering challenges of this system. Our main contributions are 1) convenient library-style access to fine-grained data from the Linux input subsystem, 2) standardization of data representation and serialization specially designed for the challenges and goals of active authentication, 3) an extensible repository of feature extractors, mono- and multi-modal systems, and 4) rapid configuration of previous and new modalities from comparative experimentation and hybridization.

Ultimately, such a framework could become a standard feature of modern operating systems. Various module configurations would be optimized for various hardware

platforms and then “compiled” for that system, dispensing with any extraneous data flows to optimize performance. Hopefully, active authentication will one day be a common security option for every device.

Chapter 2 – Background

The problem of authentication is timeless. On the one hand it poses an intuitively simply problem any child could understand. On the other hand, it demands concrete and operational mechanisms to capture one of the most elusive notions in human thought: identity. Often the full blown philosophical notion of human identity is not required. Often a reasonably artificial and constrained definition regarding a certain role or privilege is all that must be established. However, as the digital age ever more perfectly envelopes our lives, authenticating to digital systems will increasingly need an ever more absolute proof of identity. This chapter considers some of the current challenges of digital authentication, the role of active authentication in meeting them, and its enabling technologies.

Methods of Authentication

The development of electronic information systems has been incredibly precipitate and organic, often leaving proactive planning orders of magnitude behind the curve. The exhaustion of the IPv4 address space is a good example of core functionality being eclipsed by unplanned expansion. Security, no less, has seen similar trends in swift obsolescing of once adequate protocols. Modern cipher security is quite literally an inverse function of computational power – which is ever-increasing. The past two decades are littered with broken security protocols.

However, unlike core functionality, whose obstacles must be addressed to obtain any functionality at all, security can often be placed aside, while the primary function carries on. The digital world is a very large place and perhaps insecure operations will avoid disaster by mere obscurity for a significant time. Alternately, it may be that the cost of insuring a security failure is less than properly addressing the vulnerability. Today, the large breaches of confidential financial information by a for-profit ecosystem of malicious actors, would seem to indicate that we are about to exhaust the economic wisdom of insecurity.

Not all the worst failures of security are failures of authentication, but some of the most popular and lucrative involve authenticating users to empower them to effect actions (usually financial actions). The obsolescence of common authentication protocols has become a subject of serious investigation. The venerable username and password may have been appropriate to keep track of which faculty in a computer science department used the mainframe. But today, many are doubting the duo's ability to secure bank accounts, medical records, payment vehicles, trade secrets, personal devices, or legal identity against the planet's array of malicious actors.

In 2012, Bonneau et al. [13] conducted an extensive survey of alternative authentication schemes which explored the difficulties of displacing the password, and the security vulnerabilities inherent to its operation. Ironically, while passwords persist as the most feasible protocol, their security is almost perfectly opposed to their usability. Quite simply, the more character-types and the longer a password, the more secure it is. But it is precisely these qualities that make them difficult for humans to maintain.

However, computers specifically outperform humans in terms of remembering many small arbitrary details and rapidly comparing them. Therefore, while humans show a propensity to use horribly insecure passwords, dictionary attacks and more sophisticated attacks are easily within computers' reach; in short, passwords are made for computers not humans.

The survey considers biometrics in the static forms of fingerprints and iris scans, and the dynamic form of voice patterns. It acknowledges that biometrics offer certain benefits as essential authentication media (e.g. it is not possible to forget voice patterns and not likely to lose fingerprints), but it also notes their limitations. As static (and so complete) forms, fingerprints may be lifted and iris scans or voice password digitizations may be replayed by an attacker. Furthermore, authenticating with such dedicated media has a high cost in both hardware and time.

It is precisely these drawbacks that free-from behavior biometrics overcome. Behaviors are incomplete dynamic forms bound with innumerable actions from which behavioral patterns are extracted. If such patterns can be extracted from everyday usage of standard devices, the expense and usability issues disappear. Furthermore, since it is everyday uses and not a targeted input token – a certain phrase to speak or password to write – there is no reason why authentication cannot happen continuously throughout the user's session. This continuity of authentication resolves the replay attack that static and targeted dynamic forms suffer from. With continuous authentication an attacker would need to craft an entire session of useful actions to replay – a much more difficult task than simply spoofing the login.

But it is yet to be demonstrated that such behavior patterns can be found to reliably indicate distinct users. Most recent investigation into such patterns rely upon viewing the problem as whether an automated system can consistently classify samples of a behavior patterns as belong to a certain user. If such a system can be built, then the type of patterns selected is validated as sufficient for authentication.

Authentication as a Classification Problem

Deciding whether or not a user is authorized can be characterized as a simply binary classification problem of dividing authentication inputs into "authorized" and "not authorized" categories. The evaluation of binary classification systems generally employs the terms of false positive and false negative rates to measure qualitative performance.

However, in the context of authentication, the false positive rate is referred to as the false acceptance rate (FAR) and the false negative rate is referred to as the false rejection rate (FRR) since the system is accepting or rejecting a user based on their authentication input.

Generally, classification problems create an inverse relationship between improving the FAR and FRR about some third parameter or group of parameters (e.g. cost, time, complexity, etc…). Intuitively, making a more lax authentication system will reduce the FRR, but increase the FAR, and a stricter system will do the opposite. Thus, the essential performance of a system with tunable parameters affecting the FAR/FRR tradeoff is often measured in terms of the equal error rate (EER) which is the rate of misclassifications,

accepting or rejecting, when the system has been tuned to make the FAR equal to the FRR.

In the context of security, tuning to EER may or may not be appropriate, depending upon the situation. Under certain conditions, the consequence of falsely accepting an imposter is far worse than rejecting a legitimate user and so the authentication system is tuned to a much lower FAR than FRR. Nevertheless, the EER is typically proportional to the system's penalty in FRR for maintaining a target FAR and vice-versa. This tradeoff is discussed more fully in Chapter 3.

Machine Learning Approaches to Classification

Many of the attempts to identify viable behaviors for active authentication exploit fine-grain detail, measurable by standard devices (e.g. timing data down to the millisecond). Characterizing behaviors at this level of detail supports the collection of enough samples to drive machine learning algorithms. Such algorithms can automatically optimize the discrimination of patterns if given a large enough data set.

Machine learning is broadly categorized into supervised and unsupervised learning. In supervised learning, a training set of data is tagged with the correct classification for each unit to be classified. The machine learning algorithm then automatically builds a correlation function between the data and the given classifications. In unsupervised learning, no classifications are given, but rather a number (or minimum/maximum) of desired classes is specified and the machine learning algorithm builds a clustering

function to classify the data according to the specified constraints. Active authentication is primarily concerned with supervised learning insofar as the problem is to classify between “authorized” and “unauthorized” – for which the correlation to each is different for each individual authorized user. Since the classes are known a priori, unsupervised learning is unnecessary. Therefore, we will only concern ourselves with briefly sketching the supervised learning process. Deploying a machine learning algorithm for supervised learning requires a three step pipeline:

First, the raw data must be divided into units to be classified. Usually this follows logically from the nature of the classification. For example, to classify the price of a house, the unit of classification is all data related to one house, which may include data common to two houses (e.g. the average weather data for a county). However, when the goal is to classify a user, the unit of classification needs to capture data most likely to exhibit values unique to the user. What these data are is still being researched, though some excellent candidates are discussed in Chapter 3.

The second step is to devise functions that isolate the most informative/discriminatory forms of data from each unit of classification – often known as feature extraction. Feature selection allows human intuition to point learning algorithms in at least a reasonable direction. Continuing the house example, if the outside temperature for every hour was known, it could reasonably be intuited that only the annual or monthly high and low temperatures would be needed to gauge the impact of temperature on the house’s value. A function defined to take the hourly data and compute the extremes or average would be one feature extractor for temperature – potentially one among many. Similarly, other functions could take a blueprint and extract relevant features such as the number of

bedrooms and bathrooms, etc… This application of human intuition to capture divisive qualities is the heart of machine learning problems. Unsurprisingly, this step is a hotbed of research evolving diversely and rapidly.

The final step is the selection of a machine learning algorithm with which to classify the features selected. Nearly every machine learning algorithm known to man has been used for validating candidate behaviors for active authentication. Currently the most successful attempts make use of at least one form of machine learning, if not many.

The field of supervised machine learning is large, even considered in abstraction from the problems it is applied to. Some algorithms, like Decision Trees, are intuitive and whose mode of operation is simple in theory even if specific implementations and optimizations are complex. Others, like Back Propagation in Neural Networks, require advanced knowledge of mathematics to implement and deploy correctly. It is not possible to give an adequate introduction of each algorithm referenced here, but a short description and reference to fuller exposition for the major algorithms follow:

1. Naïve Bayes – a simply probabilistic method which (naively) treats all features as independent and attempts to classify by applying Bayes' rule of conditional probability [14].
2. Decision Tree – a method which organizes the features' weights by how decisive they were in the training set [15].
3. Maximum Entropy – Similar to decision trees, this method organizes a model of constraints based upon probabilities of features in the training data [16].
4. Linear Regression – the simple fitting of a linear function to the training data [17].

5. Logistic Regression – the fitting of the logistic function to the training data [18].
6. Support Vector Machine (SVM) – an optimization of logistic regression which maps features to a higher dimension space to support linear separation of classes [19].
7. Neural Network – a model of neuron input/output allowing for non-linear correlation functions to be built by tuning each "neuron's" function [20].

Some algorithms can handle real-value features, others require quantized values, some are more efficient with smaller training sets than others, and all have different capacities for handling the usual space-time trade-off in performance.

The decisions determining the specific steps of the learning pipeline impact not only the standard classification metrics of precision, recall, FAR and FRR, but also the secondary metrics related to security, such as response time, and time coverage (i.e. the percentage of typical session time a user can be authenticated by the system). The ideal system will minimize FAR, FRR, response time, the amount of data needed and time to train, and it will maximize the time coverage.

Chapter 3 – Related Work

Considering behavior as a form of identification is not new. The specific application of this idea to the problem of computer-user identification is likewise been under study for some decades now. Nevertheless, automating this identification restricts the kinds of behavior and performance available in the technology of the day – and that has changed problem space significantly. In 2004, a survey of biometric identification identified the following properties that candidate behaviors should possess [21]:

1. Universality: the behavior is exhibit by typical users.
2. Distinctiveness: any two users must be differentiable based on it alone.
3. Collectability: the behavior must be easily quantized
4. Permanence: the behavior must be consistent over a significant period of time
5. Performance: the sensors and environment necessary to capture the behavior must be economic
6. Acceptability: user must consent to and be willing to engage the sensors
7. Circumvention: the behavior must be difficult to generate by malicious actors.

As early as the 1970's [22], researchers speculated that a user's typing habits could be used for identification. In the past decade, there have been numerous efforts to exploit typing and other human-computer-interaction (HCI) behavior spaces. In general there have been three roughly distinct areas of research supporting the construction of active authentication systems.

Mono-Modal Systems

First, there are Mono-modal systems, that is, systems relying on one form of HCI to analyze behavior. The development of discriminating features in the various forms of HCI is the foundation for proving the ability of behavior to distinguish individuals.

By far the most popular modality for active authentication is keystroke analysis. Initially, static keystroke analysis, that is, analyzing how the user type pre-specified text, was investigated for use in authentication [23]. However, to enhance usability (and to prevent replay attacks) free-form text systems began to be investigated [24, 23, 25]. Popular features of keystrokes include key transition latencies (also known as digraphs) and dwell times of keys. From these absolute features, a number of derivative features have been explored such as n-grams of relative timing, such n-grams relative to certain words or n-grams of relative time values. Studies using such features have obtained very high accuracies in identification over a data set of tens of users [26]. An excellent overview of development in this field is given by [25].

Mouse and other pointing device dynamics do not have as large a body of research as keystrokes, but nevertheless have shown significant progress in recent years. Popular features include click latencies, velocity, acceleration, jerk, angle of straight line from beginning to end of motion, and rates of curvature. As with keystrokes, many derivative features have been tried in various combinations. A good overview of current efforts is given by [27].

However the most promising pointing device results we found were three curve-based metrics invented in 2011 [9]. Raw mouse data coordinates were grouped into actions

based on temporal proximity terminating in a click event. The first metric, denominated simply as “direction”, computes the angle between the horizontal and the straight line between each pair of consecutive points in an action. The second, labelled “angle of curvature”, is the interior angle formed by every three consecutive points in an action. The last, termed “curvature distance”, is the ratio of the straight-line length from the first point to the third point, to the perpendicular length from the middle point to the straight-line length, for every three consecutive points in an action. With only 20 actions, an SVM based classification system was able to obtain an EER of less than 2% on a corpus of 1000+ users.

Other attempts have tried more sophisticated media for recording behavior such as accelerometer, face-tracking, clothes-color tracking and voice patterns. However this work is only concerned with active authentication via the standard HCI functions of pointing and keystrokes. A major part of active authentication’s fundamental advantage in usability is lost if the system relies on specialized input sensors whose use is not essential to malicious HCI.

Multimodal-Systems

The second major area of research is the field of Multi-modal systems which combine multiple sources of behavior metrics to produce an identification decision. The proper weighting of diverse forms of HCI, and the methods of combining feature or decisions made from each are addressed by these systems.

In 2011, DARPA announced serious interest in promoting active authentication research, launching a four year program recruiting multiple academic and commercial research institutions to make robust active authentication a reality [24]. The program is broken into three phases: 1) 2011-2013 – discover viable biometric modalities that can, without the addition of special hardware, serve as the basis of mono-modal systems, 2) 2013-2015 – expand the discovery of biometric modalities to mobile platforms and begin integrating with DARPA's clients. Design the final authentication platform, which is planned to provide open API's to allow wide application support for both client software and novel input sensors, 3) 2015 – combine the modalities identified and fuse them into a robust multimodal system supporting the authentication platform specified in phase 2.

To the best of our knowledge, the most advanced work on multi-modal systems is being conducted by one of the DARPA participants, Drexel University. In early September 2014, they released a preprint [28] to Computers & Electrical Engineering, detailing unprecedented performance in the field. Their solution combines both strictly behavioral (*how* the user performs an action) features with features measuring more complete, intentional forms of behavior (*what* the user is doing). Eleven strictly behavioral features used included the outstanding mouse movement features of [9], the popular keystroke metrics of key-dwell time, and the delay between keys. Eighteen more complete forms from stylometry included: typing habits such as the preference to backspace repeatedly, lexical features such as the most frequent character bigrams and average word-length, syntactic features, such as the most frequent part-of-speech (POS) trigrams, and semantic features such as word bigrams.

The data was fed to classifiers in time windows that varied from 10 to 1800 seconds. They used different Naïve Bayes classifiers to classify each of the mouse and keystroke features separately and a single SVM to classify all the stylometric features. They relied upon the WEKA Machine Learning Java [29] library for their classifier implementations.

Their evaluation data set was an impressive 67 users-worth of data, generated over a period of 16 weeks of roughly defined blogging and topic-specific writing tasks. A new set of 5 users was selected each week, each contributing 40 hours of session time. All data was collected in identical hardware and software environments, to maximize the detection of only human differences.

The authors decided to combine the decisions of the classifiers according to the Chair-Varshney optimal decision fusion rule [30]. The fusion of decisions allows the classification of each feature to occur entirely independently, only requiring the final binary decision to be forwarded to the central decision algorithm. This grants the system modularity and scalability since little data needs to be forwarded from the classifiers. The Chair-Varshney formulation aims at optimizing the accuracy of the multimodal decision by taking into account the characteristic error rates of the individual mono-modal decisions. Besides the classification decision itself, the optimization of the decision fusion only required the characteristic FAR and FRR rates of each mono-modal classifier. This defines a multi-modal system with an incredibly lightweight interface for each future mono-modal module to comply with.

To obtain characteristic FAR and FRR rates, they conducted a 4-fold cross-validation of each classifier on 80% (time-wise) of their data. The FAR and FRR rates thus obtained

were used to test the multi-modal fusion rule on the final 20% of data. Multiple experiments were run for differently sized time-windows, and the results measured in terms of Time-to-Decision (window size) vs. FAR, Time-to-Decision (window size) vs. FRR, and EER vs Time-to-Decision. Inside of 30 seconds their system was able to achieve an EER of less than 1%.

The most similar multi-modal framework to the work proposed here is the Transparent Authentication Framework (TAF) in Java, which was proposed explicitly as an extensible framework for mobile devices [31]. The TAF uses multiple mono-modal biometric modules to render classification decisions on events and fuses their decisions with a history of explicit (entering a password) attempts. Two biometric modalities were used, keystrokes and voice verification. Both used the JaDTi decision tree classifier to render decisions. The authors integrate the training of the classifier into the normal operation of the system to allow multiple options for re-training to follow changes in user behavior in controlled ways.

The framework allows alternate configurations of mono-modal decision fusion, mono-modal classifiers besides decision trees, biometric feature collectors, and explicit authentication methods. The work proposed here, besides being targeted for personal computers and not mobile devices, aims at providing a far more loosely coupled framework. TAF accepts complete mono-modal feature collectors for customizable classification. The work proposed here extends the frame work one more level to offer a Raw Resource Layer providing a foundation for experimental feature design. Further differences of fusion policy extensibility are discussed in the next section on policy.

Policy

Finally, there is investigation of polices to adapt the decisions made by classification systems, to the security goals of authentication. Correctly relating the goals, acceptable performance standards, and desired outcomes of acting upon a decision, are necessary to construct a practical system. But beyond sheer practicality, considering security as the peculiar client of such systems also influences the correct manner of evaluation.

As discussed in Chapter 2, the problem of authentication can be treated as a binary classification problem, and many approaches to active authentication treat it as such. Thus, most studies adopt the standard FAR/FRR/EER evaluation metrics to validate their systems. However, achieving robust *active* authentication in a continuous manner may not be verified best by the traditional application of these metrics.

As an extension of his mono-modal work with keystroke features [24], Bours presented an alternate evaluation arrangement specifically to address the peculiar goal of active authentication [7]. In static authentication, he admits that it is important to measure how often the wrong decision is made. But in continuous authentication, the proper performance evaluation is not to see *if* the imposter is detected on a given classification as much as *how fast* is the imposter detected. To support this shift in priority, he introduces the notion of "trust" in the user as a metric which is increased or decreased based upon classification decisions made on very small units of input. By setting a threshold for minimal trust, the trust value acts as a buffer for the FRR of a system, only locking out the user when a set number of consecutive negative decisions have been

made. The number of units of input needed to reliably detect an imposter, is now the prime measure of performance, instead of the EER.

In a way, this is nothing more than multi-modal system expanded over time rather than across feature spaces. The trust value allows for the temporal fusion of multiple classification decisions. The specific manner of this fusion is governed by a penalty/reward function which may asymmetrically adjust the trust value based on classification decisions made. Unlike reporting an EER for such a system, it is possible to set a much higher bar for measurable performance with a trust metric. By setting the trust threshold appropriately, the FRR can be fixed at 0% for all training data, and the resulting average number of inputs to detect imposters is now the single metric of performance. Indeed for some test sets, the system may require more inputs than are in the test set before it detects any imposter. Nevertheless, for truly usable security, this evaluation converts the tradeoff of FAR / FRR to the more useful metric of time-to-detection with 0% FRR.

An alternate method of evaluation is that used in [28], which instead of rendering many decisions and varying the manner of decision fusion, varies the time-window of input data to render decisions on. The performance of the system is reported as the EER for a given size of window[1].

Finally, in TAF from [11], events are classified with a probability that it belongs to the authorized user. Classified events are stored in buffers per biometric along with a buffer for the outcome of explicit authentication attempts, allowing for a short history of

[1] Of course it should be noted that number of classifiable features, and not time per se, influences the EER.

classifications to contribute to the device's overall confidence. The device's overall confidence is computed on demand from the individual buffers when the user attempts a new task. The authors computed the confidence of each buffer by weighting each event to favor the younger events and computing the arithmetic mean floored to 0.5 in the absence of events. The overall device confidence was simply taken to be the greatest probability of all buffers. Various tasks could require different overall device confidence thresholds as configured by the user. If the device confidence is below the specified threshold for an attempted task, the user will be required to explicitly re-authenticate.

This embedded use of buffers is one among many design decisions that could be made regarding multi-modal decision-fusion. The Fusion Layer of this work embeds no such data flow choice, but rather offers researchers the largest possible potential for experimenting with any policy, including the three described here. The benefit of constructing them on top of the same Fusion Layer interface, is that a strict, consistent comparison of policies can be made.

All the current research efforts described above used different data sources, different collection techniques, and custom code bases. Replicating, comparing and otherwise extending their work is restricted by the diversity of their implementation ad data representation choices. Even the abstract design choices are difficult to treat uniformly unless re-implemented in a common environment. The work in [9] gathered data from web forum activity in Javascript, used a SVM and majority voting fusion rule. The work in [28] used custom desktop software to gather data, used diverse classifiers and a Bayesian risk minimization fusion rule. Supporting the comparison of these serve as the proper requirement motivations for this work, which are defined in Chapter 4.

Chapter 4 – Requirements

The fundamental contribution a framework should provide any problem is the masking of tedious, research-generic and system-specific tasks. A good framework will accomplish these tasks without compromising flexibility proper to the problem space of the user. The targeted users for this type of framework are software developers. As such the framework is intended to be used in the construction of research experiments by directly calling library-like functions in scripts or fully developed applications.

As has been shown in Chapters 2 and 3, active authentication research software has usually been organized as a pipeline of data processing, beginning with raw human interaction data and ending with some sort of binary classifier. Since the desired time-scale of active authentication is short, it is assumed that managing many small data points quickly and efficiently is incumbent upon a good framework. Within and beyond this assumption of data and time scales, there are a few peculiar problems arising in this field which impact the desired qualities of a framework. This Chapter exposes the concrete form framework goals take on in the realm of active authentication as well as some of the field specific factors that must be considered.

Extensibility

A general problem of developing good software is its ability to support continued development. This work considers the users of the framework to be familiar with basic programming paradigms and extension patterns. Therefore, the goal of creating an extensible framework ought to support some well-known software paradigm, allowing

future researchers to easily collect and extend the functionality of any aspect of the research.

Concretely, this means that the framework should not bind either functionality or configuration parameters that could conceivably be desired separately. For example, a researcher should be able to augment an existing feature extractor and configure it to his specific experiment's parameters without having to modify or clone the framework's existing code. Similarly, a researcher should be able to easily configure the input sources for a pre-existing mono-modal system to his experiment's environment without having to override the internal default sources in multiple locations throughout the mono-modal system's dependencies.

Each unit of input sources, data processing modules, classifiers, and decision systems, ought to be independent units of functionality, each of which may be extended separately.

Maximal Data Integrity

The overarching intuition behind active authentication, is that there exists a digit fingerprint hidden in the minute peculiarities of human interaction. Preserving this potential fingerprint is of the utmost importance to a multimodal framework. To that end, several qualities are needed:

a) No loss of time data at any point of processing.

   Different hardware architectures present different physical limitations on how often and accurate data can reliably be produced. But, however good or bad the

data offered is, the framework must handle requesting and propagating the data in the best possible manner to avoid delay and loss. To support live experimentation or recording for future analysis, the framework must give the researcher the best level of data integrity.

b) No out of order data from any module to any module.

Considering that the same data source may be processed in multiple ways, and the outputs be merged back together for classification Figure 1, care must be taken to prevent one classification-path from receiving data from a significantly different time, due to varying processing latencies.

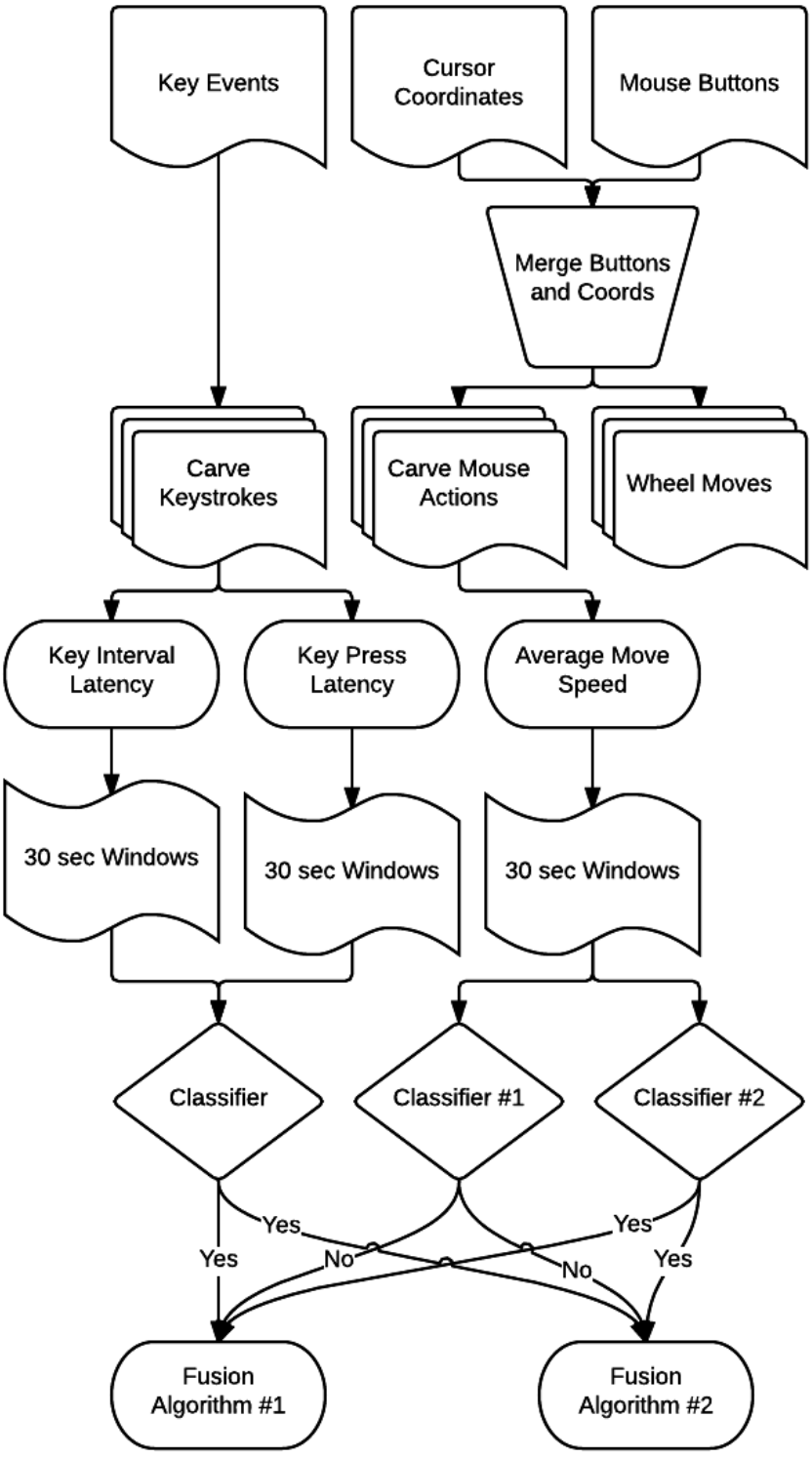


Figure 1 – Framework Example Instance

Decoupled Interpretation of Data

Data may be digitized into units from a relatively continuous stream with several different analysis goals Figure 1. First, it may be broken into logically related blocks (Carving). For example, the stream of mouse data comprising one intentional move by the user, should be gathered into one data structure for the sake of more intuitive feature analysis. Second, data may be grouped into fixed quantities simply for aggregate or arbitrary feature dependent analysis (Feature Extraction). For example, every ten double-clicks may be gathered to update a running average of double-click latency features. Third, data may be broken into temporal windows of a finite duration to regularize classification in time (Windowing). Finally, the classification scheme used to render a decision can use a variety of interpretive models.

Decoupled Collection and Carving

Various hardware setups will produce data with various levels of detail. The envisioned framework will abstract away overly peculiar formats of raw data, but it must leave as much low-level detail as possible for innovation to take place. Therefore, the framework will preserve the smallest unit of data available and refrain from binding its collection to any algorithm carving it into larger data structures.

Decoupled Carving and Feature Extraction

Carving refers to the gathering of raw data (e.g. button-down and –up events) into collections representing intentional actions which intuitively would contain user specific traits (e.g. the user has typed "A"). Just as biometric matching analysis expects to be applied to one (or at least the fragment of one) fingerprint and not multiple fingerprints jumbled together, so the analysis of human interaction ought to consider movements constituting one intentional mouse motion or keystroke. For this reason, carving should occur prior to feature extraction in most cases.

Decouple Feature Extraction and Windowing

The obvious approach to classification is to devise features whose samples should be classified one at a time. But more subtle differences may be hidden by the amount of noise gathered with a single feature sampling. A more robust characterization of a feature may be gained by building a frequency distribution of its values over time or a fixed number of samplings or some other logical unit of data. This aggregation of feature values forms a window in time or data quantity for classification. But the parameters of such windowing, while potentially related to, can have very different criteria from the feature itself. The temporal division of data is useful for managing the tradeoff of response time and accuracy. Intuitively, the more data presented to the classifier, the more accurate the classification will be. In terms of active authentication, the nature of this window plays an important role in balancing FRR and FAR, as well as minimum

data quantity thresholds for attempting a classification decision. Gathering more data also takes more time, extending the window of interaction without authentication. Therefore, the configuration of such windowing should be decoupled from the extraction of any given feature.

Decoupled Windowing and Classification

In the usual machine learning pipeline, there is always some division between forming the unit of data for classification and the classification itself. While preparation of a classification unit can often be specifically tailored to the type of classification chosen (e.g. quantizing for Naïve Bayes), the envisioned framework will not bind any particular form of unitization to the type of classification used.

Comprehensive Serialized Data Collection and Playback

A major principle of empirical science is reproducibility. The medium of digital information offers the researcher no excuse but to be able to show exactly the data and methodology used in experimentation. While trivial compared to most other data forms, the preservation of digital data in an efficient and organized manner can nevertheless be tedious and fraught with arbitrary formatting decisions. As entirely generic in its goal, the ability to save and load raw and processed data from any point during an experiment is a perfect candidate for the framework's responsibility. Standardizing, abstracting and hiding the operation of saving experimental configurations and test corpora is a great aid in forming uniform reproductive and comparative research, as well as accurately

demonstrating incremental improvements. Therefore, the framework will provide a transparent system of saving and loading data from any arbitrary unit of processing that may be native or extended from it.

Logical Consistency of Raw Data Playback and Live Streaming

Similar to the above requirement for comprehensive saving and loading, the manner of loading the data must accurately reproduce the operation of the researcher's processing logic, regardless of whether the input is loaded from a saved corpus or streamed live, or a mixture of both. In general, machine learning is very reliant upon the ability to speed up experimentation by decoupling the collection of data from its classification. Multi-modal authentication, however, is properly targeting a live classification system. To support this essential use case, the framework must not lead researchers astray by offering logically exploitable differences between the playback of recorded raw data, and the live acquisition of the same. Knowing when data is exhausted, the length of a corpus, or depending upon a certain number of data to be acquired in a certain time are all examples of details that are strictly unknown in live acquisition and should not be depended upon by the framework when playing back corpora.

Rapid Experiment Setup

A further goal of a good research framework is to organize the setup of experiments with a minimum amount of effort. As the framework is extended with more data-sources,

preprocessing modules, classification algorithms and entire mono- and multi-modal systems, the systematic empirical comparison of old, new and hybrid techniques will bolster consistent progress in the field. To this end, the framework should define a standard method of configuring experiments with a minimum of additional logic and without directly altering the code of past research. New and hybrid modules should either explicitly extend old ones or the configuration of existing ones should be easily contained in one high-level perspective, and not require scattered tweaks.

Chapter 5 – Design

Following the lead of research in the field, the Active Authentication Research Framework (AARF) proposed in this work follows the pattern of a generic machine learning pipeline of data classification, with auxiliary modules handling data collection, serialization, deserialization, and comparative experimentation. At all stages of research the framework endeavors to imbue qualities satisfying the requirements outlined in Chapter 4. This chapter addresses the design techniques and patterns used pervasively throughout the framework, as well as the generic architecture of the framework's core organization to fulfill the requirements.

Modularity

It is generally recognized that modular design enhances the extensibility software and promotes maintainability and conciseness of code. For a research framework, modularity is particularly beneficial for establishing inter-purpose boundaries that standardize the scope of functionality and thereby allow for different approaches to be mixed and matched with a minimum of effort. To achieve the desire level of modularity, several more concrete designs are called upon.

Object Oriented (OO) Paradigm

To undergird the requirements of extensibility and data management, the OO paradigm will be used to encapsulate the units of functionality, along with their data definitions. This technique will achieve the required decoupling of core components described in the Chapter 4 section 3.

Publisher/Subscriber Pattern

All core parts of the machine learning pipeline share the common requirement of receiving and sending streams of data to other parts. Add to this the requirements for arbitrary save/load functionality at any point in the pipeline, chronological data integrity and the management of stochastic input, and the need for an abstract, asynchronous publisher becomes apparent. Thus, every component of the pipeline will inherit a suite of functions for publishing. These functions will define a generic interface for the creation of, consumption of, and subscription to the component's data. This will give the user maximum freedom to organize any set of subscribers to subscribe to different or the same publishers at any time. Further, the publisher will provide functions to save its publication to file, and to stream a stored publication from file.

Recursive Default & Top-Down Customizable Configuration

Each publisher will recursively instantiate its own input sources with default configurations to hide the details of earlier pipeline stages from later ones. In this way, each stage of processing will be the user of earlier stages and will be used by later ones. This allows future development to target any point in the pipeline, without being concerned with the configuration of all prior stages. However, customized experimentation may require non-standard configurations at any arbitrary point throughout the pipeline. Researchers could meticulously alter the construction arguments for each stage throughout the recursive decent of instantiation. However, it would be much easier if at the stage of interest, all the needed configuration changes to earlier stages could be executed in a simple list of instructions. This is accomplished via the Singleton pattern, Figure 2, whereby the prior stages may be configured in a simple list of instantiations that occur before the recursive default constructors are called. After the novel configurations are instantiated, the recursive default instantiation will simply retrieve the existing instance and not create the default instance. This allows the researcher to override arbitrary configurations throughout the pipeline without having to alter the code within each component, while maintaining the default hiding of prior stages. Thus, the best of both worlds is obtained: minimal required configuration and maximal flexibility to re-configure a custom run, all in one place.

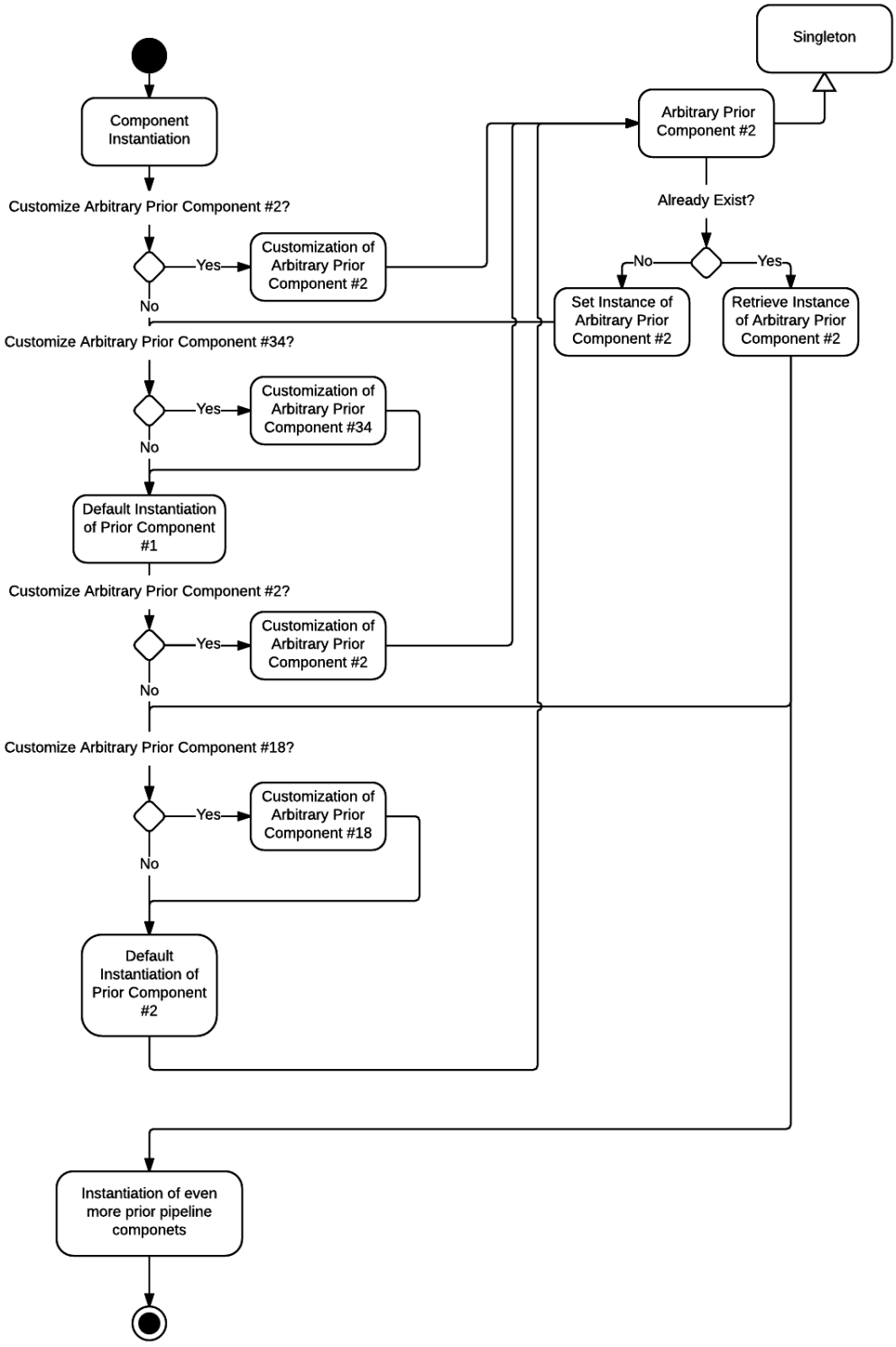


Figure 2 – Recursive Default and Top-Down Custom Configuration of Components

Extensibility

Perhaps the primary contribution of this entire work is its offer of an extensible repository of research so that the field may systematically progress. Re-invention of the wheel and a general lack of good comparative studies have been a problems in computer science. As noted by Hamming "*Indeed, one of my major complaints about the computer field is that whereas Newton could say, "If I have seen a little farther than others, it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.*" [32] While this work does not claim to have replicated a significant amount of research in the field, it hopes to lay the ground work for a more effortless self-systemization of the field. Therefore, the following techniques were selected to achieve the goal of making everything as extensible as possible.

Inheritable Types

A popular aspect of the OO paradigm is the notion of inheritable object types. The pattern of inheritance is nearly identified with the notion of extensibility (e.g. Java "extends" keyword). This supports the general purpose of a framework to abstract pervasive functionality into one module form which others inherit. The specific details of

which functionality are deemed pervasive in this design are described below in section #. Beyond propagating the common functions of the framework's core, requiring new modules to abide by this same typed structure (as opposed to procedural libraries or scripts), will insure that future additions can always be extended in the same manner.

Aggregation

Another benefit to extensibility from the OO paradigm is the design pattern of aggregation. Aggregation allows for further optimizations of the more complex components by defining auxiliary objects aggregated into a single larger object. This allows complex stages of the pipeline itself to be independently improved, further expanding opportunities for innovation. This pattern is used in the publisher to decouple saving and loading, allowing alternate file I/O adapters to be made compatible with AARF.

Interfaces Layers

Three main interfaces define the major points of standardization in the pipeline. It is speculated that these present the most popular division of effort regarding active authentication.

a) Raw Resource Interface

The peculiarities of collecting data on various hardware platforms and from various devices are almost incidental to the goals of feature extraction, therefore, the Raw Resource interface allows the search for features to abstract from the problems of hardware differences, if the research so chooses. The raw resource interface will mandate two functions: 1) the acquisition of a raw data stream, 2) the conversion of each unit of data to a standard raw event format. The interface will define the raw data format to include a minimum of the event's value and the time stamp in either a relative or absolute format.

b) Mono-modal Interface

A mono-modal system is comprised of the desired preprocessing (Carving and Windowing) feature extraction from sample data and the classification of the extracted features. Because there is some potential for features to be mixed and matched with different classifiers, a sub-interface defining the functionality of a feature is included in this layer. The mono-modal interface will mandate three functions: 1) the acquisition of a feature-vector and classifier(s), 2) the grading of a single feature-vector's sufficiency for classification (e.g. contains enough events to merit a decision), 3) the measurement of characteristic FAR and FRR, given a training and characterization data set. The interface will also define the format of an extracted feature as including a minimum of the feature value and a label that uniquely identifies the feature inside AARF. To support the standardization of feature generation, the Mono-modal Layer will rely upon on feature extractor modules to implement a sub-interface.

c) Feature Sub-interface

The feature sub-interface will mandate three functions: 1) the acquisition of a single sample, 2) the extraction of a feature value from a single sample, 3) the provision of a unique feature-Label. The interface will also provide the definition of an optional function to provide a list of possible feature-values on which to base frequency distributions of the feature.

d) Multi-modal Interface

Finally, the multimodal interface allows the highest level perspective of extensibility, abstracting from even the mono-modal system's internals. As more mono-modal systems are added, the multi-modal interface will allow researchers to easily extend existing multimodal systems either in quantity of modalities, or in quality of logic via inheritance. These extensions will retain the same interoperability with experimental evaluators and policy applications if they consume this same interface. The multi-modal interface defines two functions: 1) a simple getter to return a list of MonoModalSystems to be used as input, and 2) a fusion Algorithm to be used to fuse the output of the input systems into one single binary decision.

Performance

The anticipated performance challenges of this framework do not immediately warrant special attention (an average stream of cursor movement events yields less than 5MB/hr.). Nevertheless, as this framework intends to act as a repository of future efforts, it must be considered that more data intensive inputs (e.g. cameras) may be explored with

it. Anticipations of future possibilities such as this, as well as some peculiarities of the field itself, lead to the following design decisions regarding performance.

Singleton Pattern

At each point in the pipeline, it may occur that multiple subsequent components need to use the data produced at that layer. To conserve CPU time, it is beneficial that all subsequent components simply register for their own subscription rather than instantiate a new instance of the producing component. Otherwise, each publisher will instantiate prior components it may depend on for its input and re-process the data to provide the same output for each subscriber. Therefore, the primary components of the pipeline should be Singletons.

Transparent File Management

As noted above, the amount of data needed to run active authentication schemes does not yet require special storage handling for scaling purposes. However, each publisher will inherit a suite of file management functions to handle the automatic compression and serialization of data. Furthermore, since multi-modal systems are likely to conduct experiments on a corpora of multiple raw data sources, it is highly desirable that the framework automates the process of saving and loading multiple files in the same session. And so, a Session recorder and Session loader are required to relieve the development of the pipeline from these issues of storage optimization.

Multi-threading

In order to ensure not only good performance, but even the correct handling of stochastic inputs, the framework will executionally isolate the publication of data from its consumption. This is required to fulfill the requirement of logical consistence. Consider a multimodal system fusing several mono-modal systems. If the a raw data source for one of the mono-modal systems blocks waiting for an event, its mono-modal system should register a null vote or otherwise signify that it has nothing to contribute to the multi-modal system. Therefore, a single-threaded execution is not sufficient to properly construct a multi-modal system. To simplify the amount of multithreading management used, the suite of publishing functions will also contain the thread management logic. Each publisher will dispatch a worker thread to process its input sources and output as many copies of the publication as needed to give each subscribing component its own.

Generic Architecture

Contemplating the deployment of the described design techniques gave rise to the core generic architecture of AARF. At the center of the framework is the notion of a publisher which takes other framework components as input streams, applies arbitrary logic to transform the input into a publication, and serves the publication to an arbitrary number of subscribers. Furthermore, each publisher should optionally be able to serialize its own publication to file, as well as play a serialized publication back from the file, just as if it

was being computed for the first time. Finally, to prevent duplication of effort and to maximize scalable performance, each type of publisher ought to be instantiated as a Singleton, and spawn a worker thread to produce its publication.

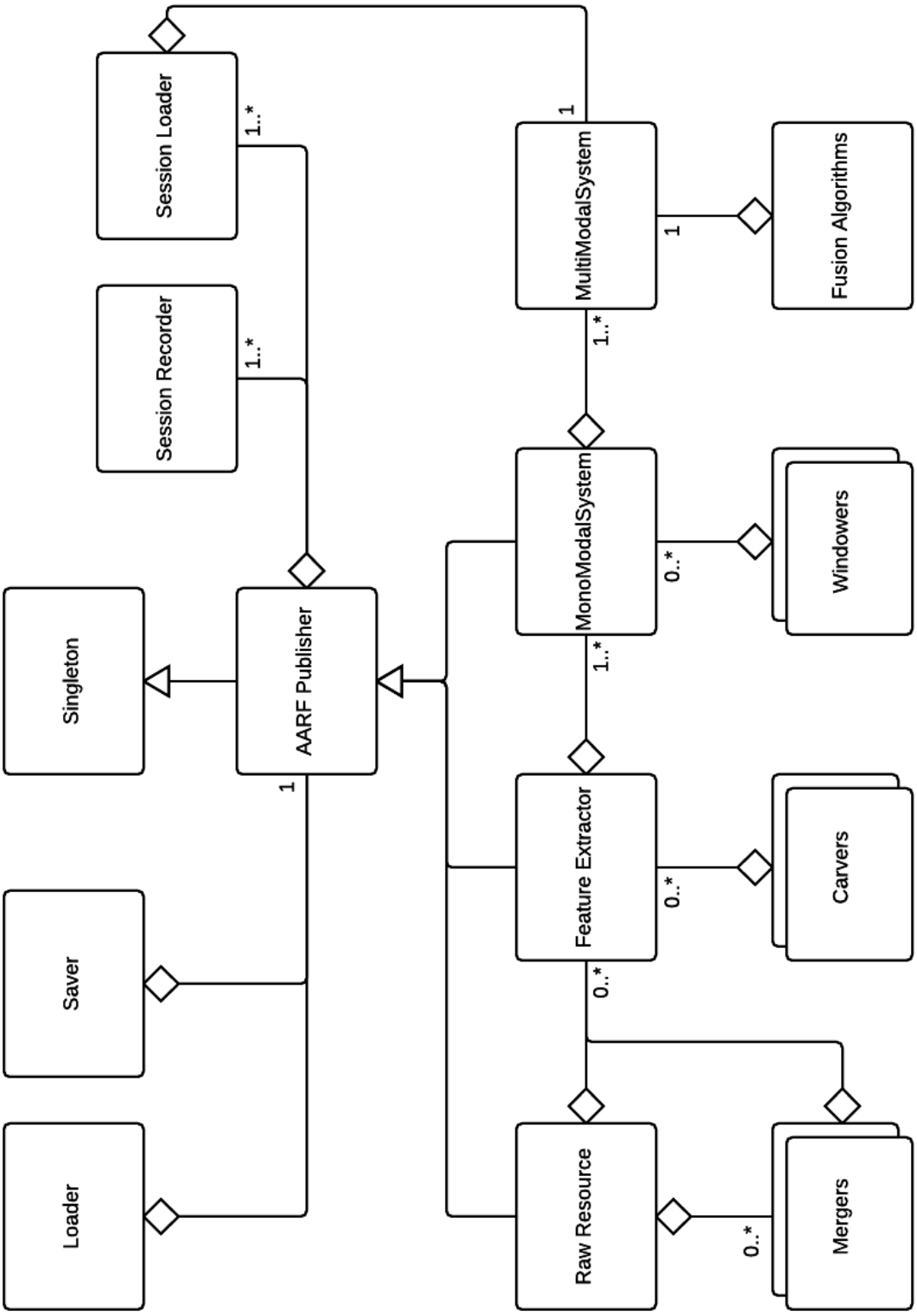


Figure 3 – Generic Architecture

Inheriting from this generic publisher, are specialized publisher types to provide simplified and concrete APIs for extending and improving research efforts. Chief among these types are the abstract classes defining the layered interfaces. While targeting the Linux input subsystem, the Raw Resource Interface may potentially host more operating systems in the future. For now it will abstract the generic form of a Linux input device and reduce as much common logic from the collection and decoding of data as possible. The Mono-modal Interface will support the generic tasks of integrated multiple feature-types, manage the training of classifiers, and the computation of characteristic FAR and FRR metrics. The Multi-modal Interface will manage the automation of training and experimenting with Mono-Modal systems as well as the integration of user defined fusion algorithms.

Beyond the core machine learning pipeline components, auxiliary modules including Session, Carving, Windowing, and Fusion Algorithms provide non-Singleton data aggregators, adapters, and transformers that may be injected at multiple points throughout the framework as desired. Because their tasks are either very generic (e.g. persists data to files, or dividing into 30 sec windows) or orthogonal to the pipeline, they do not merit the full blown performance and extension management of the publisher pattern. Concrete examples of the rationale for these decisions are provided in Chapter 6.

Chapter 6 – Implementation

The Implementation of AARF began with two pilot projects, one creating a mono-modal system for mouse movements and the other attempting to classify StackOverflow posts by author. Both uncovered valuable implementation techniques and data representations for the active authentication realm, as well as some of the inconvenient and unsustainable practices of ad-hoc scripting of research. While the AARF code base was written entirely new, some of the logic of the two pilot projects was used to seed the AARF system with example modules. It was precisely this experience of translating research logic from a mass of scripts to a reusable and extensible framework that guided the implementation of AARF. The implementation of the AARF framework, Figure 4, executes the generic architecture put forth in Chapter 5. This Chapter details this implementation in terms of the underlying technologies employed and the further detailed design and formatting decisions made.

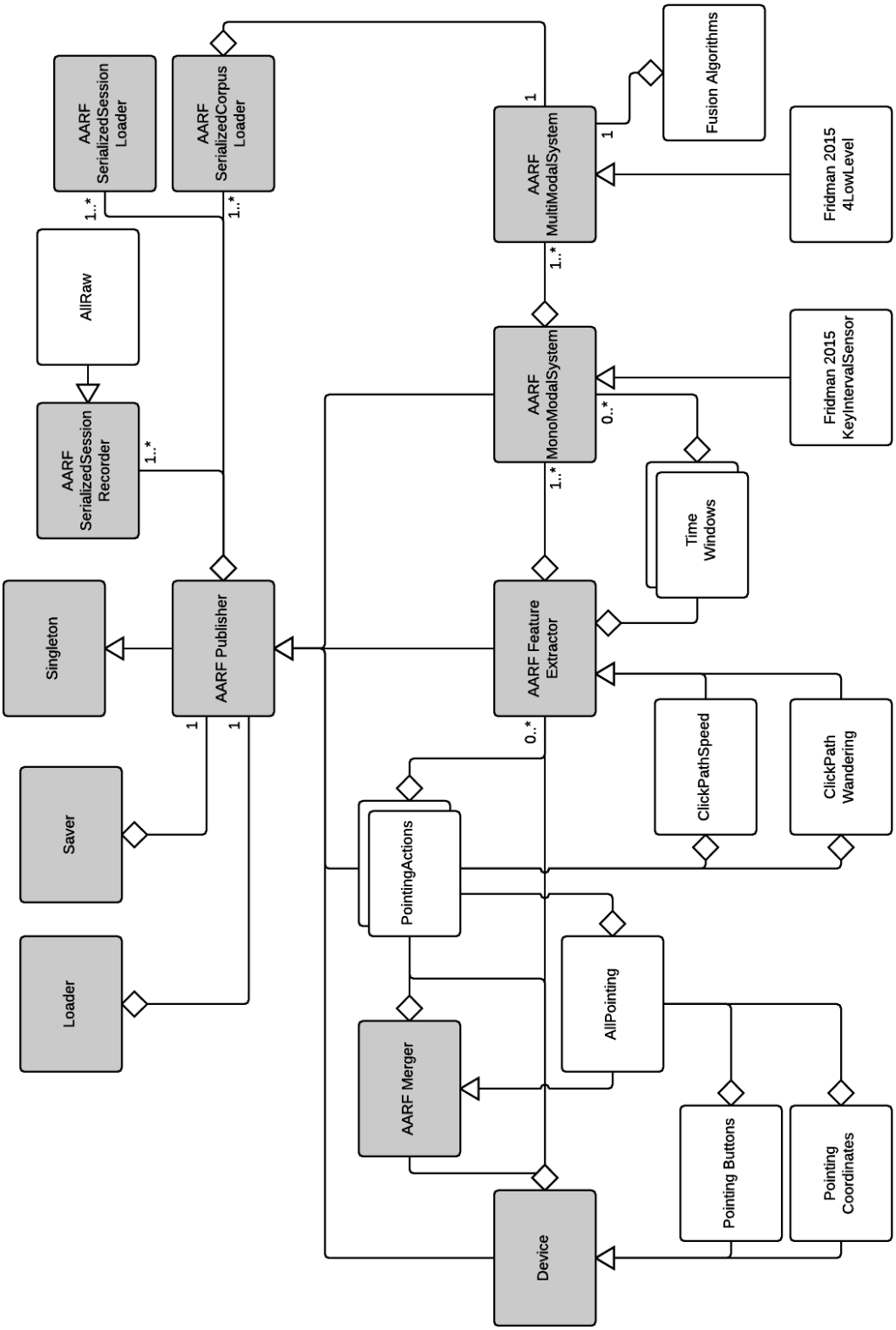


Figure 4 – Core AARF

Language

The choice of language was made in consideration of the requirements of a framework, the peculiar requirements of the field, the targeted level of user expertise, and the availability of library resources. The Python language meets a great deal of these criteria and was chosen for the following reasons:

a. Python is a relatively compact, high-level language which supports rapid prototype development common in research. It has a minimum of peripheral syntax and uses a nearly pseudo-code style.
b. Python supports a form of object-oriented development completed with inheritance, virtual methods, and class/instance distinctions.
c. As an interpreted language, Python is highly portable, allowing for its widespread use in both experimentation and data collection.
d. There exist substantial Python libraries for the analysis of human interaction including natural language processing (NLTK) and machine learning in general (Sci-kitLearn).
e. Python contains a native threading library.
f. Python offers a closure type of iteration pattern called “generators”. A generator renders each element of a collection when called, while remembering an entire execution context surrounding the formation of the collection. This allows for pipelines of data processing to be written in an intuitive top-down consumer perspective.

g. Python supports the mechanics of optional arguments. This feature is ideal to offer the researcher invisible default management of configurations, while still allowing targeted customization on demand.
h. Python supports a built-in unit testing framework.

To support the longevity of AARF and some details regarding the management of scoping, we selected Python 3.4. All development occurred under Visual Studio Professional 2012 and 2013 using Python Tools for Visual Studio 2.1. This works expects a basic working knowledge of the Python language, including its scoping, class structure and module management. For more information regarding these topics, we refer the reader to the Python documentation [33].

AARF Publisher

The AARF Publisher is the heart of the framework's implementation and, as such, is tasked with making each product of the pipeline available to an arbitrary number of subscribers, savable, and accessible in a non-redundant, global fashion. The AARF Publisher accomplishes global, non-redundant access by inheriting from Singleton, thereby allowing any module to instantiate an AARF Publisher and simply receives a subscription of the singleton instance rather than truly creating another instance with all the requisite processing threads and dependencies.

The AARF Publisher has a very simple constructor possessing only two optional arguments: `savePath=None, loadPath=None`, specifying the file path to which to

serialize the publication and from which to load it. AARF Publisher is an abstract class requiring two methods to be overridden by its subclasses: `generateInput()`, and `getUniquePublisherLabel()`. Upon construction, the AARF Publisher instantiates its input sources via the first required overridden method `generateInput()`, and adds a new Python queue to its list of subscriptions Figure 5. The method `generateInput()` is expected to return a Python generator of input data from which the publisher will compute its publication data. Construction does not actually begin publication. A consumer must both construct, and then subscribe to an AARF Publisher instance.

This distinction between construction and subscribing is vital to create a logical window in which multiple consumers register their interest in a publication before the publisher actually begins consuming its input in a separate thread. During live capture, the late arrival of a few subscribers could be a trivial matter. But if the data is being streamed from files, a significant part of a corpus may have passed in the time an asynchronous subscription is made. For the sake of live addition subscriptions, the AARF Publisher simply begins sending late subscribers whatever data it is currently processing – it has no notion of what the "first" datum of input data was, and so cannot bring a late subscriber up to speed.

Not only does this window simplify the initialization logic, it also prevents the actual recursive instantiation of the publisher's inputs until a subscriber actually needs the publication. This is a helpful distinction made use of by auxiliary services (see Session below) which may indiscriminately construct publishers, but not actually activate them all. This distinction will also allow for future a validation process to execute after construction, but before the actual execution of pipeline. An exception to this distinction,

occurs when the AARF Publisher is constructed with a `savePath`, in which case it begins streaming its publication to file immediately. Presumably this means the publisher is not also loading its data from a file and so the problem of late subscribers does not arise. During live capture, the late arrival of a few subscribers could be a trivial matter. But if the data is being streamed from files, a significant part of a corpus may have passed in the time an asynchronous subscription is made. For the sake of live addition subscriptions, the AARF Publisher simply begins sending late subscribers whatever data it is currently processing – it has no notion of what the "first" datum of input data was, and so cannot bring a late subscriber up to speed.

Calling the `subscribe(useGen=True)` method initiates the publisher's consumption of input data and production of publication data in a separate worker thread Figure 6. This thread calls the optionally overridden method `processInputUnit()` which transforms the input data into publication data. By default, `processInputUnit()` does nothing.

Subscribing to a publisher returns a Python generator which supplies the publication data. Optionally, a Python queue may be returned (`useGen=False`), which allows the consumer to directly manage reading the subscription queue. This option is crucial to allow consumers to not block on the delayed publication of data or to set custom timeouts when acquiring publication data.

The second required overridden method, `getUniquePublisherLabel()`, is simply getter that returns a string uniquely identifying the publisher. This name will be used to annotate data saved to file and to load data. Although the class name could have been

used, this offered the subclass designer flexibility in augmenting the name with module-load time configuration parameters. Since each publisher is a singleton, runtime parameters will not differentiate publishers in the same runtime environment, but this allows for specific configuration information to be saved when data is saved.

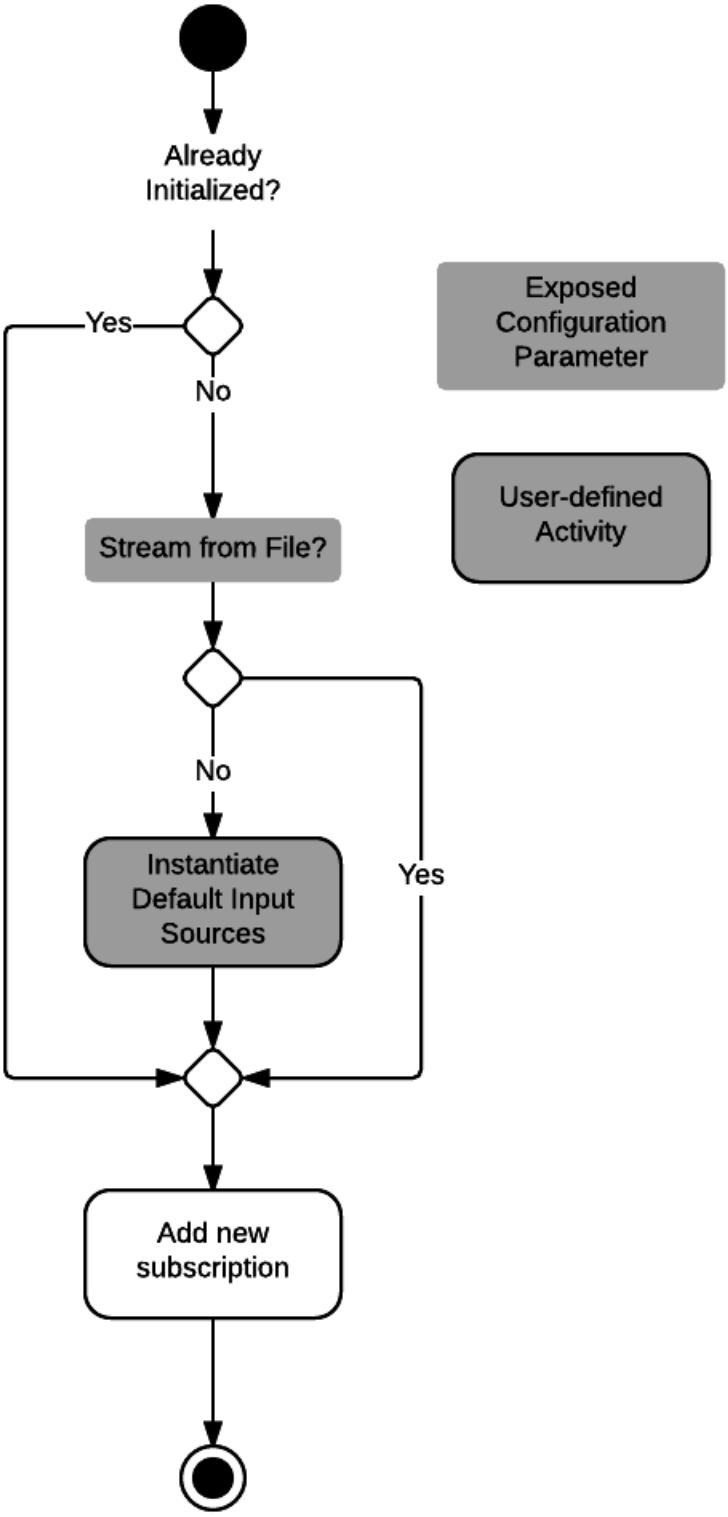


Figure 5 – Publisher Instantiation

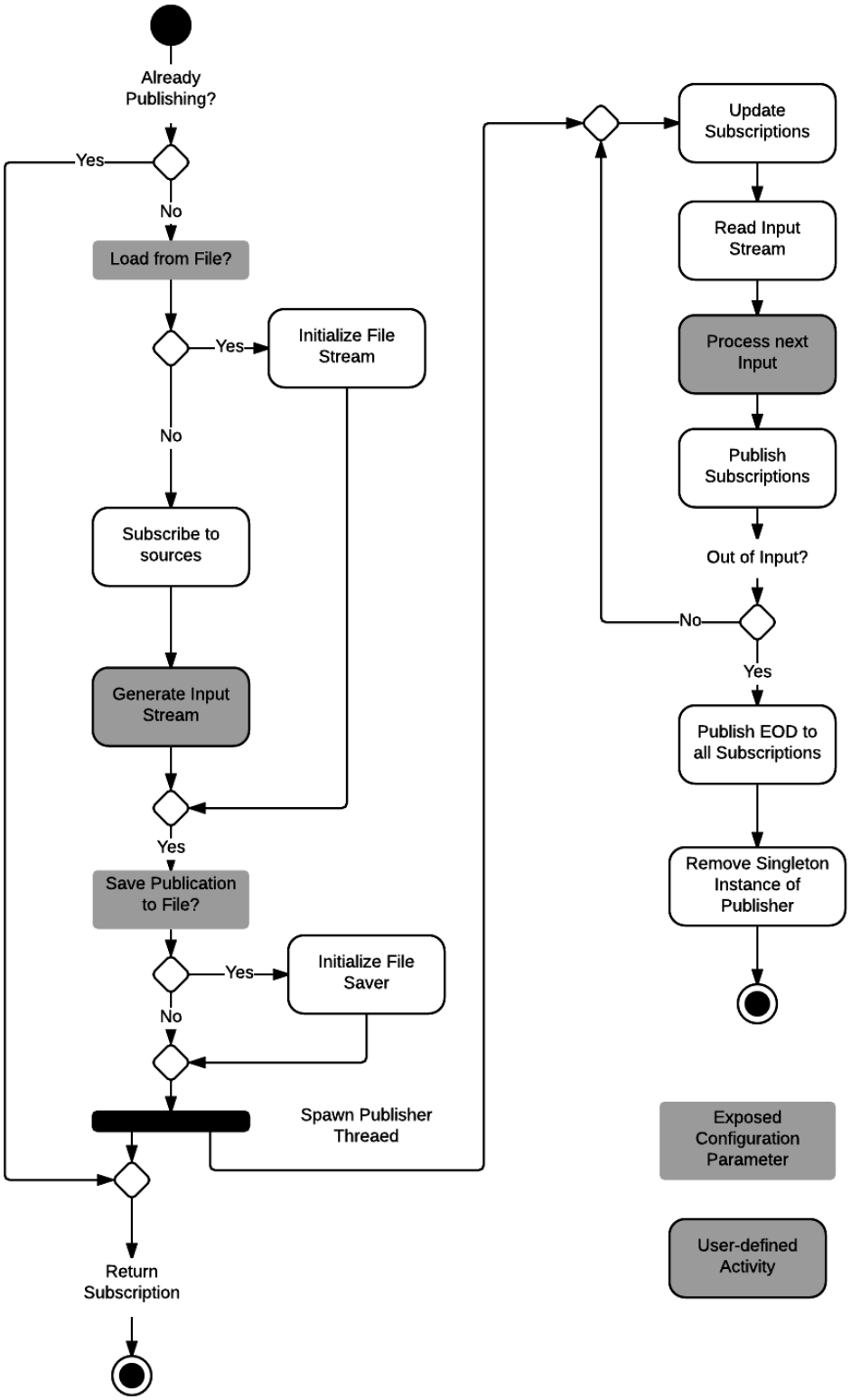


Figure 6 – Subscribing to Publication

Saver

The AARF Publisher aggregates two auxiliary classes: Saver and Loader. By simply registering itself as a subscriber, the Saver mimics the Python queue, thereby tapping into the AARF Publisher's worker thread's execution to drive the serialization process. The Saver class implements a simple serialization pipeline shown in Figure 7, using the lzma compression library and Python's pickle serialization. Data recorded in an AARF session is formatted as compressed text rather than the original binary structure of numeric values used by Linux. Text is human readable when uncompressed and it is less susceptible to platform specific data formats. This makes AARF corpora more portable and more accessible to manual inspection.

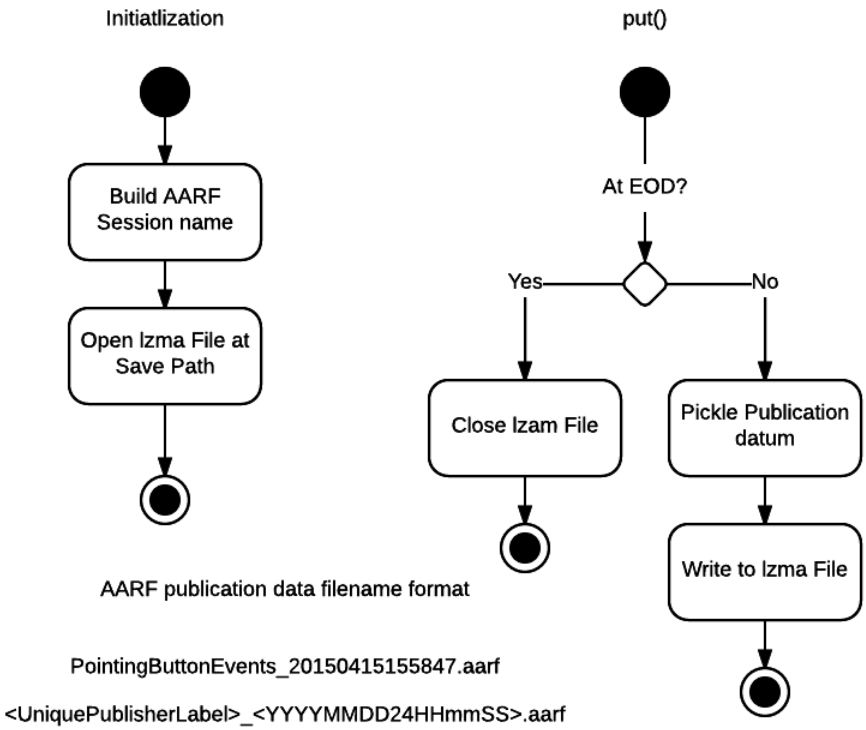


Figure 7 – Saver Dataflow

Loader

The Loader class assists AARF Publisher by opening serialized AARF data at the file location specified by the publisher's load path argument. It simply inverts the serialization of the Saver, first decompressing the lzma file and then unpickling each data object. The Loader automatically detects if the load path points to a single AARF file, an AARF Session load directory (discussed below in Session) containing multiple AARF files, or a directory containing multiple AARF Session load directories. In each case, the Loader searches for AARF files bearing the name of its publisher. If multiples are found, it will load that data from each file in turn.

Input Device

The primary embodiment of the Raw Resource Interface is the InputDevice module. We decided not to call it AARF Raw Resource or anything more generic, because it is specifically designed to access the Linux input subsystem. The Device class is an AARF Publisher that interfaces with the Linux input event files to decode input events from HCI peripherals such as mice, touchpads, keyboard, touchscreens, trackballs, etc. The Device class is easily extended by simply constructing a subclass with the desired input event file descriptor. The InputDevice module also acts as the framework's repository of Linux specific data format and constant values, enabling any subclass of Device to convert the binary event structure of type, code, value, second and microseconds into an easily

manipulated Python dictionary. As an AARF Publisher, any of Device's subclasses may be loaded or serialized and many serve an arbitrary number of subscribers.

Session

An AARF Session refers to a group of AARF files saved from publishers running concurrently. AARF supplies two generic modules for session management: SerializedSessionRecorder, and SerializedSessionLoader. The SerializedSessionLoader class has one abstract method `startSessionSources(self, folderPath)` , which the subclass uses to define which publishers to include in the session recorded. The folderPath argument specifies the savePath to construct each of the publisher instances being returned by `startSessionSources.` The framework is seeded with one Serialized Session Recorder "AllRaw" which serializes a session containing output from the KeyEvents, PointingCoordsEvents and PointingButtonEvents or PointingCombine if PointingCoordsEvents and PointingButtonEvents are unavailable. SerializedSessionRecorder also expects to find a Serials.txt file containing any number of newline delimited 10-digit serial numbers. These serial numbers are used to tag each session with an associated authentication identity. When a SerializedSessionRecorder is launched, it presents a simply GUI to accept a serial number typed by the user. If the typed serial number is found among the preloaded numbers, the session recording begins. The user may stop session recording through the GUI at any time. Once a session is completed, the SerializedSessionRecorder collects all the generated AARF Files and adds

them to a single archive tar archive, which, in turn is also compressed with lzma. This allows corpora to be easily built as a simple collection of AARF Session Archives.

The SerializedSessionLoader class reverses the process of the Recorder, but first creates an AARF Session load folder to which it extracts all the AARF files contained in the session. Since AARF publishers have no knowledge of serial numbers, the SerializedSessionLoader tags all extracted AARF files with the session's serial number to prevent confusion.

A more powerful version of SerializedSessionLoader is the SerializedCorpusLoader class, which decompresses multiple AARF Session archives and discovers all sessions and unique serial numbers. Once the SerializedCorpusLoader has decompressed a corpus, it may be queried to load all sessions belonging to a target serial number or all sessions not belonging to a target serial number. This assists the automation of cross fold validation with the minimal amount of file rearrangement.

Carvers

There is nothing special about the implementation of a Carver in AARF, other than it inherits from AARF Publisher. Notionally, Carvers should aim at supplying the framework with intuitive units of data that would be suitable subjects for feature extraction. But the concrete form such units should take on is so varied that no common interface was defined for them. It is arguable that Cravers must always provide a standard timestamp to identify their publication, but even this was left for future thought. The

framework was seeded with two Carvers: PointingActions and Keystrokes. A pointing action is a single intentional group of pointing device events (e.g. a deliberate motion and click to close a window). A keystroke is the constellation of events required to print a given character to the screen. For most characters, this is a combination of SHIFT, CAPS LOCK, or NUM LOCK, and a down event paired with an up event for the targeted key. The implementation of both Carvers is far from comprehensive for rendering a complete palette of possible intentional groups, but they cover most common actions.

Feature Extractors

Feature extraction is the core human contribution to the machine learning process. As such, it is expected that the majority of future research in active authentication will revolve around the invention and testing of features. To support this effort, the AARF Feature Extractor sub interface attempts to relieve the researcher from the handling of feature labeling, representation and integration with classification logic, as shown in Figure 8. Ideally the AARF Feature Extractor only requires the researcher to define two methods: `getSamples(self)` and `extract()`. The former defines the framework sources providing the samples to extract from, while the latter defines how to extract a feature value from a sample.

Optionally, the researcher may inform AARF that all extraction and labelling will be accomplished in `getSamples(self)`. However, since method is bound to the instance of the feature extractor and expects external input source, unit testing extraction will require a heavy weight setup. Defining `extract()`, on the other hand, was specifically

designed not to be a bound instance method. Therefore, unit testing the correctness of extraction logic will not require the simulation of the entire feature extractor object, but will simply test individual samples. All AARF Feature Extractors support a flag to request every feature value to be tagged with a timestamp. If researchers do not override `extract()`, it is their responsibility to tag the Linux event style time to each feature value published.

Finally, the researcher may optionally define a `getPossibleValues()` method to return a list of values the feature should be quantized into if windowing into frequency distributions is to be supported.

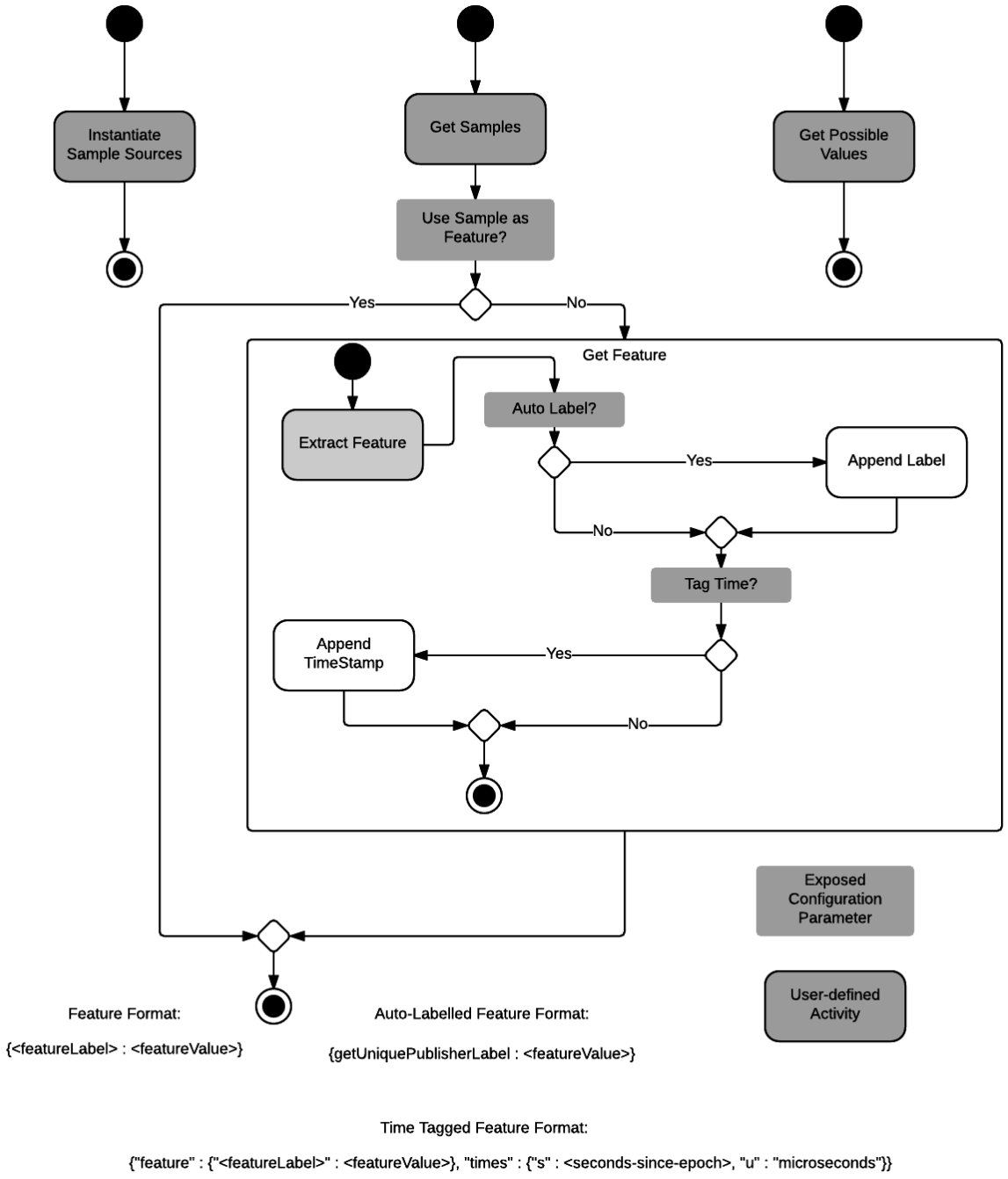


Figure 8 – Feature Extractor

Windowers

Windowing modules simply take an AARF publication as input and generate a stream of lists contain nothing but data from the original publication. Each list contains a "window" of data defined by some windowing metric. We have seeded AARF with a temporal windowing module, but the metric by which to window may be arbitrary. The only design restriction that window modules must abide by is that they must accept an AARF Publisher as input and generate Python lists as output. Our temporal windower requires that input data bear time information as defined by the AARF Feature Extractor. Windowers are not AARF Publishers since their logic is generic and does not represent a unique or novel contribution to the field. Furthermore, since they may be interposed between feature extractors and mono-modal systems or, in principle, any consecutive pipeline components, meaningful loading of their serialized output would require knowing both the publisher they were applied to as well as their own configuration. This is a level of sophistication in the loading of serialized AARF data than was deemed presently feasible.

Classifiers

AARF is not intended to compete with machine learning libraries as such. Many sophisticated, well maintained, and powerful libraries, frameworks, and even GUI applications are readily available at no cost. WEKA [29], NLTK [34], SCIKIT-Learn [35], JStylo [36], and others provide a vast amount of resources to research in the way of feature management, classifier tuning, experimentation and pre-fabricated end-end

machine learning systems. AARF intends to be the client of such resources and provide a repository of past experimentation built with the services such libraries provide. However AARF also protects the researcher from two extremes.

First, there is the tyranny of choice involved with powerful but complex libraries such as WEKA and SCIKIT-Learn. For those willing to take on a higher learning curve or already having expertise in machine learning, nothing stops them from dropping their optimized classifier into their own Mono-model System. But for those more interested in rapidly experimenting with tweaking features already classified by prior research, a lighter-weight minimum configuration is desirable.

Second, there is the overly targeted pre-arrangement of features, classifiers and evaluation techniques. GUI tools like JStylo offer wonderful simplification of pipeline configuration for writing analysis, but at the same time, the end user is not expected to be a software designer. While both the more complex and more simplified libraries allow for their own forms of simplification or code-level customization, AARF intends to provide the middle ground in one consistent framework. Of all the surveyed machine learning frameworks, we deem NLTK to have struck this balance best with its wrappers for the SCIKIT-Learn library. Of course NLTK is targeted for natural language processing, which, while a valuable piece of active authentication, is narrower than the data types considered by AARF. Therefore, we decided to model our integration of classifiers after that of NLTK. Specifically, the Mono-modal Interface integrates with the NLTK interface for classifier interaction: training is handled internally by the classifier and feature vectors are expected to be labelled as Python dictionaries.

Mono-Modal Systems

The AARF Mono-modal System module is an AARF Publisher that supports the creation of a trained classifier according to the NLTK style wrapper discussed above. The researcher is responsible to define the framework source from which to acquire feature vectors, the classifier which the system uses, and how to compose a feature vector from the feature sources defined. After specifying these things, the researcher may load the system with positive and negative training samples and proceed to train the system, as shown in Figure 9.

In keeping with the requirement for logical consistency, positive and negative training samples are loaded through the framework itself. This requires either a live capture to be coordinated between the call of `loadPositive()` and `loadNegative()`, or more conveniently, load AARF Session(s) of positive data, followed by a load of AARF Session(s) of negative data. Since these separate loads will regenerate all affected publishers in AARF, this operation ought to be coordinated at the final stage of the framework: either in a Multi-modal System or in a script governing the use of the entire AARF instance.

Once the system is loaded with positive and negative samples, the `train(self, train=3, crossVal=0, test=1, foldCycle=0)` method is used to specify how to train the system and whether to compute characteristic data on a cross-validation fold. The values for the `train`, `crossVal`, and `test` arguments specify the relative quantities of data dedicated to each set. Only `train` must be a non-zero integer. The rest may be zero or some other positive integer. The `foldCycle` argument specifies the ordering of

folds throughout the three sets. If all the data were divided into 4 folds, fold-cycle 0 would allocate folds 1-3 to the training set and fold 4 to the test (nothing to cross validation since its argument is 0). Fold-cycle 1 would allocate folds 2-4 to the training set and fold 1 to the test. Fold-cycle 2 would allocate folds 3, 4 and 1 to the training set and fold 2 to the test etc… This cycling of folds is not guaranteed to exhaust all permutations, but it is deemed a suitable formula for fairly assessing the system on the entire corpus.

After acquiring a trained classifier with the training set, the system checks if the cross-validation set is nonempty. If so, a characterization test of the system's FAR and FRR are computed on the cross-validation set. After characterization, the system is ready to accept subscriptions to its publication of decisions on the test set if present, or if not, defaults to classifying live data. The FAR and FRR results of characterization are cached in the system's instance field for the use of reporting or multi-modal logic.

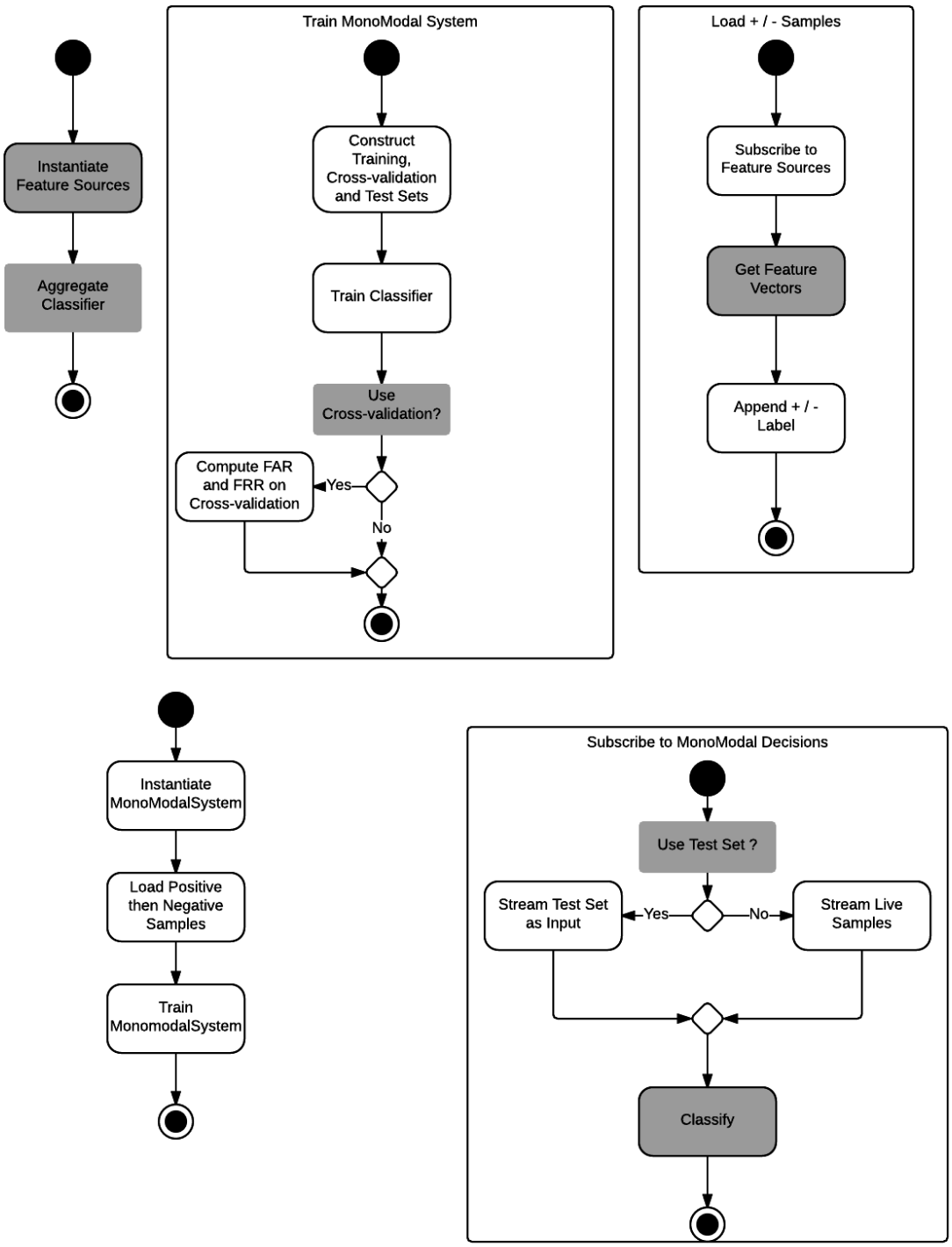


Figure 9 – MonoModal System

Multi-Modal Systems

The AARF Multi-Modal System class defines a single authentication system relying on the output of several mono-modal systems. Since the output is relatively simple, this class does not merit inheriting from AARF Publisher. However, it is defined as an abstract class so that configurations will be preserved in subclasses.

To use an AARF Multi-Modal System, the researcher specifies the location of an AARF corpus which is merely a folder containing multiple serialized AARF Session Archives, as a super constructor argument. The mono-modal systems to instantiate are defined in the overridden method `getMonoModalSystems()`. The desired distribution of data into train, cross-validation and test sets as super constructor arguments, and the fusion algorithm to employ is defined in the overridden method `fuseDecisions()`.

Since AARF is targeting the development of multi-modal approaches, training session data maybe defined at the multimodal level. This way a single AARF Session can be used to train all mono-modal systems in parallel on the same training corpus, as shown in Figure 10. The automated initialization of the system extends the training of all the mono-modal systems to train and compute characteristic FAR and FRR values for all fold cycles for all user serial numbers, each chosen in turn to be the "authentic" user. Once this comprehensive testing is complete, the fully trained and characterized system publishes its decisions on the test set if present or, if not, on live data.

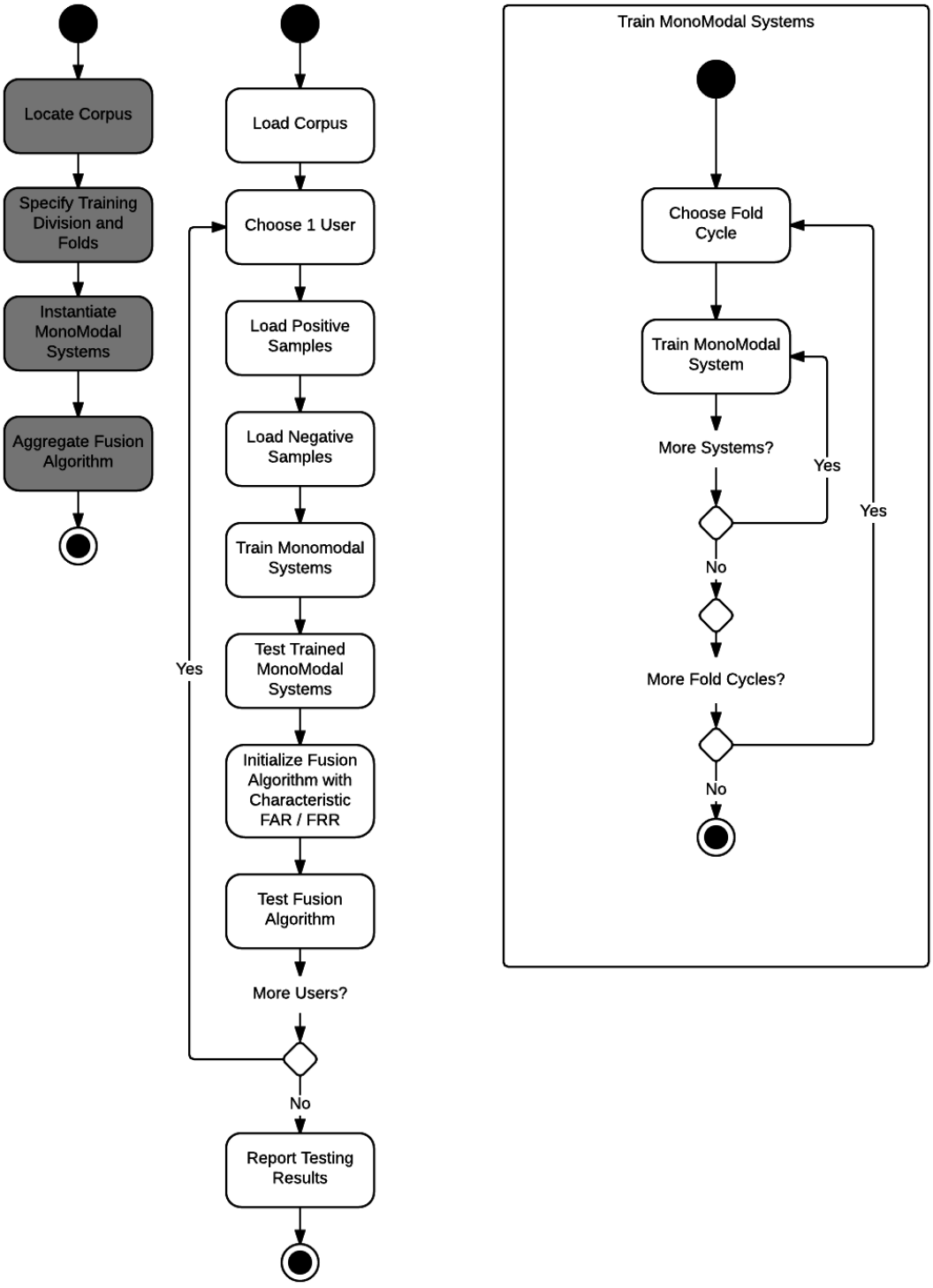


Figure 10 – MultiModal System

Chapter 7 – Validation

The validation of this work's contributions comes in two forms. First, there is the validation of the design of the framework itself, in light of the requirements specified in Chapter 4. Second, there is the practical validation of the implementation of the design, both regarding its fidelity to the design as well as the usual metrics of correctness and performance. This chapter documents the validation techniques and tests applied to the AARF framework.

Requirement Validation

Chapter 4 lays out the requirements we set for ourselves as needful in the field of active authentication and proper to the scope of responsibility of a generic framework. Therefore, we shall consider each in turn and assess the merits and limitations of the proposed design.

Extensibility

AARF deploys the OO-paradigm to support the extension of core framework modules as well as any prior research inheriting from the core. To evaluate the execution of the design, each major AARF extension point is here considered from the perspective of an AARF contributor.

```python
import AARF.AARFPublisher

class MyAARFPublisher(AARF.AARFPublisher.AARFPublisher):
    """This is a template of an instantiable AARF publisher.
       Copy this boiler plate and remove/replace every "myXXX" identifier.
       Use ctrl-F "my" case-insensitive to make sure you do not leave any placeholders"""

    def __new__(cls, myReqArg1, myReqArg2, savePath=None, loadPath=None, myOptArg1=None, myOptArg2=None):
        #MUST PUT YOUR CLASS NAME IN THIS super CALL
        return super(MyAARFPublisher, cls).__new__(cls, savePath, loadPath)

    def __init__(self, myReqArg1, myReqArg2, savePath=None, loadPath=None, myOptArg1=None, myOptArg2=None):
        if not hasattr(self, "INITIALIZED"):
            # do your intialization logic prior to parent initialization here
            super().__init__(savePath, loadPath)
            # do your intialization logic after to parent initialization here

    # MUST IMPLEMENT
    def getUniquePublisherLabel(self):
        """Returns a descriptive string that uniquely identifies this publisher"""
        raise NotImplementedError

    # MUST IMPLEMENT
    def generateInput(self):
        """Make generator of input data from other publishers and/or files.
        To be implemented by subclasses."""
        raise NotImplementedError

    # OPTIONAL METHOD - will be called on every datum generated by generateInput()
    def processInputUnit(self, inputUnit):
        """"Define the processing to be done on each input unit"""
        return inputUnit # default: do nothing
```

Figure 11 – Extending AARF Publisher Example

To add a new pipeline interface, serializable data source, or Carver, contributors ought to extend the AARF Publisher class. As shown in Figure 11, there is a fair amount of boilerplate Python code handling the construction and initialization of a subclass. However, there is only one required addition to the construction code: forwarding the subclass name in the call to `super()` inside `__new__()`. Otherwise, the contributor is free to add as many required or optional construction arguments[2], and specify initialization code either before or after AARF Publisher's initialization or both. The contributor must specify a unique string identifying the new publisher to be returned by `getUniquePublisherLabel()`. Arbitrary logic to collect input data for the publication is placed inside `generateInput()`, which must return a Python generator which generates each unit of input data. Optionally, `processInputUnit()` may be defined to be automatically invoked on each input datum generated. Overall this extension interface provides effective extensibility of the framework with only a moderately complex inheritance process.

To add a new Linux input device to AARF, the contributor should extend the Device class. The modification of boilerplate inheritance code is trivial, as highlighted in Figure 12. First, the contributor must pass the name of the subclass in three places, two inside the constructor `__new__()` and one in the initialization function `__init__()`. Second, the `deviceNumber` argument should be exposed (for future auto-detection), and preset to the default input event file number for the new device. This default should be set as a constant value at the module level and ideally should be imported from the InputDevice

[2] The arbitrary addition of construction and initialization arguments is valid for any extension discussed here and so will not be illustrated again. The only requirement is that the construction and initialization argument lists be identical and that default-valued arguments come last.

module so that all input constants reside in one location. Lastly, if possible, the `touch()` method should be implemented to support programmatic generation of events. This optional method is helpful for diagnostic and recording purposes.

```
import AARF.RawData.InputDevice

MYDEVICE_NO = None #should be added to the InputDevice module

class MyDevice(AARF.RawData.InputDevice.Device):
    """Publishes My devices events

    deviceNumber -- the event file number (usually = ?)"""

    def __new__(cls, deviceNumber=MYDEVICE_NO, savePath=None, loadPath=None):
        return super(MyDevice, cls).__new__(cls, "MyDevice" deviceNumber, savePath, loadPath)

    def __init__(self, deviceNumber=MYDEVICE_NO, savePath=None, loadPath=None):
        if not hasattr(self, "INITIALIZED"):
            super().__init   "MyDevice" deviceNumber, savePath, loadPath)

    def touch(self):
        """Programmatically generate an input datum"""
        #TODO
        pass
```

Figure 12 – Extending Device

To add a new feature extractor to AARF, the contributor should extend the AARF Feature Extractor class. The contributor has two major options for generating a feature value. First, all input sources, feature computation, and feature labelling may be handled directly in the `getSamples()` method. If this option is chosen, a `self.sendSample` flag should be set to True to inform AARF that the resulting sample should be treated as feature values. Additionally, the `self.autoLabel` flag should be set to false if custom feature labels are to be applied.

The second option is to implement `getSamples()` to only generate sample data which must, in turn, have the feature values extracted from them. This is automatically accomplished by implementing the `extract()` method. The benefit of splitting the extraction process into two methods is two-fold. First, it allows AARF to take responsibility for packaging the feature value into the NLTK feature vector format as a Python dictionary. The default label used by AARF is the unique publisher label. Second, it allows for easier unit testing since `extract()` is not an instance method. This way the core logic of the feature extraction may occur by simply invoking `extract()` on the class object.

Arguably these options complicate the extension process beyond an optimally usable framework. While this interface presents a flexible and rough start, it needs further refinement to meet the ease of extensibility originally intended.

```
class MyFeature(AARF.AARFPublisher.AARFPublisher):
    """Provide Logic to extract a feature"""

        def __new__(cls, savePath=None, loadPath=None, tagTime=False)
        return super(MyFeature, cls).__new__(cls, savePath, loadPath)

    def __init__(self, savePath=None, loadPath=None, tagTime=False)
        if not hasattr(self, "INITIALIZED"):
            self.sendSample = ?
            self.autoLabel = ?
            super().__init__(savePath, loadPath)

    def getUniquePublisherLabel(self):
        return "MyFeature"

    def getSamples(self):
        """Define input source and formation of a sample to extract from."""

    def extract(sample):
        """Define logic to extract feature value from sample."""

    def getPossibleValues(self):
        """List of all possible values that could be returned.
        This is used for building frequency distributions."""
        values = []
        return values
```

Figure 13 – Extending AARF Feature Extractor

To add a new mono-modal system, a contributor should extend the AARF MonoModalSystem class. The contributor need only forward the subclass name and the classifier of choice to the parent's constructor and initialization, shown in Figure 14. Classifiers should comply with the NLTK style interface providing, at a minimum, methods to train a new classifier on a list of labelled feature vectors, and a method to classify a list of unlabeled vectors. The only special method to be implemented is `getFeatureVectors()`, which defines the feature sources and returns a generator of feature vectors to be classified.

```
import AARF.MonoModalSystems.AARFMonoModalSystem
Class MyMMS(AARF.MonoModalSystems.AARFMonoModalSystem.AARFMonoModalSystem):
    """A classification system based on related modalities of HCI"""

    def __new__(cls, savePath=None, loadPath=None):
        return super(MyMMS, cls).__new__(cls, nltk.classify.NaiveBayesClassifier,
savePath, loadPath)

    def __init__(self, savePath=None, loadPath=None):
        if not hasattr(self, "INITIALIZED"):
            super().__init__(                                  , savePath, loadPath)

    def getUniquePublisherLabel(self):
        return "MyMMS"

    def getFeatureVectors(self):
```

Figure 14 – Extending AARF MonoModalSystem

To add a new multi-modal system to AARF, the contributor extends the AARF MultiModalSystem class. Again, the required information is minimal. Since AARF MultiModalSystem does not inherit from AARF Publisher, there is no requirement to specify explicit constructors or initialization logic as shown in Figure 15. However, if the constructors are overridden, they should call their super counterparts after executing custom logic. There are only two methods to implement. The first is `getMonoModalSystems()`, which defines the mono-modal systems to draw classifications from. This is also the place to override any component of the AARF framework. This method will be called on every session load made during the training of the mono-modal systems, ensuring that custom configurations remain in place. The second method to be overridden is `fuseDecisions()`. This method defines the fusion algorithm to be invoked on the stream on mono-modal classifications from the test set or from live data.

```
import AARF.MultiModalSystems.AARFMultiModalSystem

class MyMuMS(AARF.MultiModalSystems.AARFMultiModalSystem.AARFMultiModalSystem):
    """A system combining multiple modes of HCI classification for the same
       Decision."""

    def getMonoModalSystems(self):
        """Choose the Mono-Modal Systems you want to include.
        return as a list of AARFMonoModalSystems"""
        systems = []

        return systems

    def fuseDecisions(self):
        """Combine the output of all input systems
        to render a single binary decision per input unit"""
        raise NotImplementedError
```

Figure 15 – Extending AARF MultiModalSystem

As demonstrated at each of these extension points, the requirement of extensibility is met with a moderately complex interface. However, the benefits of retaining research logic in an interchangeable framework is worth the overhead presented here. As with any design execution, the ability to extend AARF could be refined, but its initial state satisfies a first attempt.

Maximal Data Integrity

The AARF design of forming a continuous pipeline from the raw data collection to classification is naturally conducive to a simple series of audit points along the path of processing to ensure data integrity. However, the windowing algorithm, seeded with the framework, uncovered a special case which bears mentioning.

Windowing by time will not account for latency of prior processing stage. To allow the researcher maximum freedom to window at any stage, the TimeWindows module may intercept data after an arbitrary number of processing steps have already occurred. The windowing algorithm does not have any framework-wide knowledge of when raw data was collected. The original collection timestamp is not observed until the data's effect reaches the stage at which the windower was inserted. Therefore, sufficient processing latency may cause data to miss the window it belongs to based on its collection timestamp. To avoid this, all temporal windowing modules should be made aware of the collection times of raw data as soon as they are acquired by AARF. This instrumentation is left for a future effort. The validation of AARF's implementation

considered below does not carve windows from positive and negative data shuffled together, thereby avoiding this drawback.

Other than special aggregation functions like windowing, the handling of data is primarily in the hands of the research logic, except during serialization and loading. As mentioned above, serialization is based on compressed text data. Automated detection of publisher shutdown warns the Saver to properly close open files, and Python thread locks are used to protect file I/O from asynchronous disruption. Similar locks are used to ensure that subscribers are guaranteed to receive an End-Of-Data signal no matter when they successfully register with a publisher.

Decoupled Interpretation of Data

The ease of acquiring a Python generator from any point in the pipeline, along with the ability to customize each is illustrated by six example use cases in Figure 16. The core inheritance from AARF Publisher ensures that raw data, carved data, feature values, windows of data, and mono-modal classifications can all be streamed via generator.

```
# EXAMPLE #1 Acquire stream of live keyevents in 2 lines
ke = AARF.RawData.Keyboard.KeyEvents()
keystrokes = ke.subscribe() #get generator

# EXAMPLE #2 Save a live feature stream to file - 1 line
pd = AARF.FeatureExtractors.Keys.KeyDurations.PressDuration(savePath="./MyDurations")

# EXAMPLE #3 Load carved data from file - 2 lines
charPublisher = AARF.Carvers.Keystrokes.Characters(loadPath="./Characters_20150422151011.aarf")
characters = ch.subscribe() #get generator

# EXAMPLE #4 Acquire stream of 30-sec windows of characters 1 line plus 1 above
windows = TW.TimeWindows(charPublisher,duration=30).subscribe()

# EXAMPLE #5
#Load a session into AARF - 1 line (pre-loads multiple publishers to read from files)
session = AARF.Session.SerializedSessionLoader.SerializedSessionLoader("angrybirds_serial_.tar.xz")

# EXAMPLE #6
#Train a Multi-modal system on a given corpus with specifc parameters
#After training the cross-validation will report trained accuracy
#Since there is no test set, the system will begin classifying live data after training is complete
system = F159L.Fridman_2015_9LowLevel(ASSSL.SerializedCorpusLoader("C:\\RawData"),train=3, crossVal=1, test=0)
```

Figure 16 – User's Perspective of AARF

This is the level of abstraction originally intended for AARF: access to any prior research at any point in the pipeline with one or two lines of code. If the corpus of raw data sources, feature extraction logic, and classifier deployment could be grown in AARF, the effort to assemble new composite systems, compare similar performance, and systematize prior research would be trivial.

Comprehensive Serialized Data Collection and Playback

Also, as shown in Figure 16, any AARF publisher may be instructed to read from or save to a file. Furthermore, the ability to coordinate the saving and loading of entire sessions offered by the Session module makes test corpus creation and experimentation trivial. In this respect, the intended function is well developed. The only drawback of the current implementation is the inability to serialize windowed data, as such, for reasons discussed in Chapter 6. This is especially inconvenient for research focused on mono-modal systems, since the input is often windowed.

Logical Consistency of Raw Data Playback and Live Streaming

In keeping with the requirement for logical consistency, any publisher loading data from files publishes it to its subscribers directly, bypassing any processing logic. One drawback to this is that publishers simply trust that the loaded data actually belongs to them. This is loosely enforced by the Loader's search for files bearing the name of the publisher, but can easily be circumvented.

The Session modules, and the AARF Multi-modal System ensures consistency by reloading all needed publishers from the ground up when loading session data. This prevents publishers from detecting file I/O latencies between session loads. The Loader also ensures one continuous generation of input data when loading from multiple files by nesting Python generators and intercepting the End-Of-Data signal when one file or directory is exhausted. This allows a seamless concatenation of data from the publisher's perspective.

Rapid Experiment Setup

As described in Chapters 5 and 6, AARF's primary simplification of experiment configuration lies in leveraging the Singleton design pattern. By making each pipeline component recursively construct its input components, a default channel of configuration hides pre-tuned setups. However, if the researcher wishes to alter any exposed configuration parameter at runtime, the desired components may be constructed prior to the default recursion channel. Since components are Singletons for the life of an AARF session, all special construction may be executed in any code scope convenient to the researcher. This alleviates the congestion of massive configuration objects being passed down the default channel, and eliminates the need for scattered compile-time tweaks to existing code.

The validation of this design, in theory, is presented by a use case illustrated in Figure 17. Consider the most complex object in AARF: an AARF Multi-modal system. Natively, the AARF Multi-modal system offers the researcher the choice of mono-modal

systems to include a fusion algorithm to employ. However, the performance of the system is dependent upon the entire machine learning pipeline. Suppose the researcher wanted to customize a carving operation of the raw data sources. Prior to instantiating the mono-modal systems, which in turn, recursively execute the default instantiation of prior components, a custom construction of the carver can be executed. Mergers, carvers and feature extractors could all be customized prior to default initialization. Once a component is initialized, its Singleton is merely retrieved by later initializations.

There are two limitations of this design. First, the researcher must have prior knowledge of which components will be used by the mono-modal systems selected. A more streamlined process will provide automated discovery of prior components. Second, the customized initialization of components must proceed from most prior to least prior. If a later stage of the pipeline is customized before an earlier stage, only the later stage will be customized, since its customization will recursively trigger the default initialization of the earlier stage. Once initialized, the attempt at customized construction will be ignored, as the Singleton instance already exists.

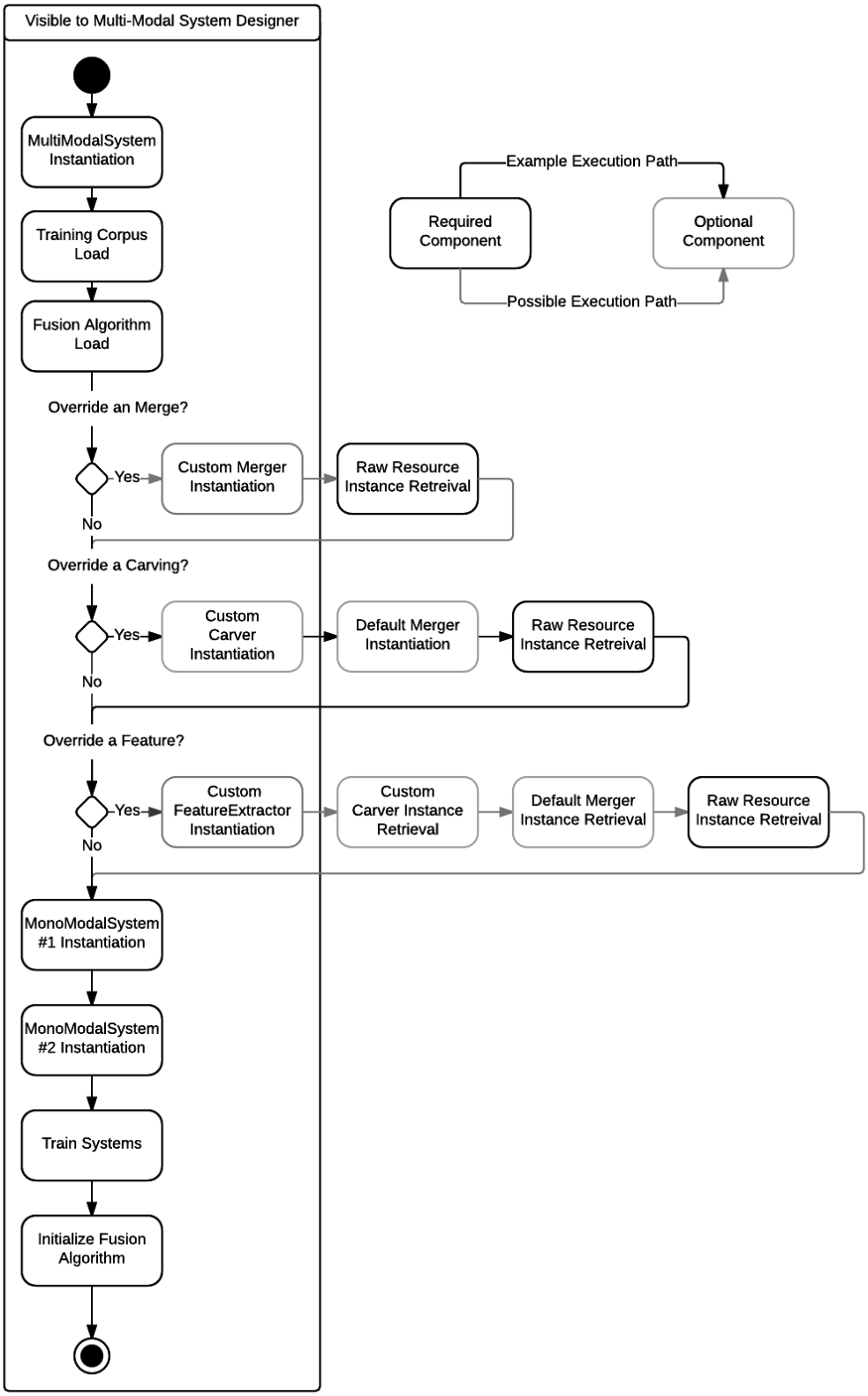


Figure 17 – Example Experiment Configuration

Implementation Validation

The validation of AARF's implementation is also two-fold: unit tests and a small experiment. A modest suite of 39 unit tests was developed, which primarily targets core AARF functionality, but also includes inspection tests for seed modules. Table 1 summarizes the purpose and extent of tests targeting core AARF modules. Net line coverage is a conservative estimate of the true line coverage, because it only reports the maximum line coverage obtained in any single test run. Some abstract interface modules were tested indirectly via seed modules which inherit from them.

Tested Module	Dedicated Tests	Net Line Coverage[3]
Singleton	3	100%
AARFPublisher	7	65%
InputDevice	0	63%
AARFFeatureExtractor	0	64%
AARFMonoModalSystem	3	43%
AARFMultiModalSystem	1	29%

Table 1 – Core Unit Test Extent

For a fuller end-to-end integration test, a proof of concept experiment was conducted on a small corpus, using a multi-modal system which partially replicated the work of [28].

[3] While no dedicated unit tests were crafted for certain modules, their classes were tested via subclass unit tests, thus net line coverage considers coverage from the entire AARF test suite.

We solicited user data from upper division computer science students, in accordance with the following IRB approved criteria:

- No selection or recruitment based on gender or ethnicity was used. Any student interested in participating could have.
- Students were presented with a consent form, and given ample opportunity to ask questions or leave, should they so choose.
- A specially designed data collection tool, with clear indicators of when the tool is and is not collecting data, was installed on each computer used.
- Students were given specific instructions on when to start and stop data collection as well as a set of specific tasks that should be conducted during the experiment.
- Subjects were asked to not enter any personally identifiable information or visit any personally registered services or sites (e.g. Facebook).

Twelve students volunteered. All volunteers were male computer science students of typical college ages, with similar computer experience[4]. No effort was made to maximize, minimize or otherwise arrange demographic, physical, or knowledge differences among the volunteers. Each student was assigned a unique serial number to identify his data, but no demographic data was bound to it. We did not record a mapping of serial numbers to students in any form, but relied upon the students to remember their serial number should they wish to contribute multiple sessions.

The corpus consisted of Linux input events for keyboard and mouse, containing an average of 2500 complete (down+up) keystrokes and 150k pointing device events per

[4] All these characteristics were assessed by mere inspection and were not quantified or systematically measured.

individual. However, it should be noted that these averages included spacing events and other meta-events used by Linux, so the true averages were somewhat less. The construction of the corpus was subject to the following independent constraints:

- All data was gathered from identical WYSE thin client workstations using identical USB unified keyboards and mice.
- During a continuous 30 minute session, the user was asked to perform three tasks: 1) Play 20 classic levels of Bejeweled (a graphical 3-match game) or play for 15 minutes, whichever was shorter, 2) summarize two news articles in 200-400 words, and 3) complete the Xrite color-vision test made up of dragging colored tiles into hue order. Each of the tasks was chosen specifically to test heavy skewing of the performance of each mono-modal classifier within the decision fusion module.
- An experiment data set for each user was constructed from all classification units for the user (positive samples) and all classification units from all other users (negative samples). Both sample sets were divided into 5 folds of equal time duration; 3 for training, 1 for cross-validation and 1 for testing.
- All data from the training folds were used to train the classifiers, while equal numbers of positive and negative samples were used to construct the cross-validation and test sets. This ensured that a baseline of most common tag accuracy would be fixed at 50%.

Some students recorded multiple sessions, however most only contributed one. One user's data had to be dropped entirely due to an unexpected interruption, which was later discovered to have corrupted the session archive.

To classify the data, we constructed a multi-modal system to partially replicate the work of [28], using four of their nine low-level sensors: the two keystroke mono-modal systems and mono-modal systems based on the curvature angle and distance features of [9].

The first keystroke feature, termed KeyInterval, measured the time between the up-event of one key and the down-event of the next key. Since nothing prevents a second key from being pressed before the first is released, this interval can take on negative time values. Furthermore, the intervals of interest occur when the user is actively typing and not the arbitrary intervals between active typing. To ensure that this feature only reported such intervals, we only considered interval values falling between -1 and 2 seconds. Figure 18 shows the composite relief graphs for two example users. The graph is composed of the average distributions of values over time-windows varying from 30 to 1800 seconds with a bin size of 0.01 seconds. Each graph averages the frequencies to integer values, thereby filtering out outliers. For this reason, the total number of samples recorded in the final average graphs is significantly less than the total 2500 keystrokes per user. Although, the actual classification of windows did not filter the distributions, this method of filtering helps visualize the most determining values which rise above the noise. The graphs presenting this and the remaining features use the same filtering technique, but users chosen as examples may vary.

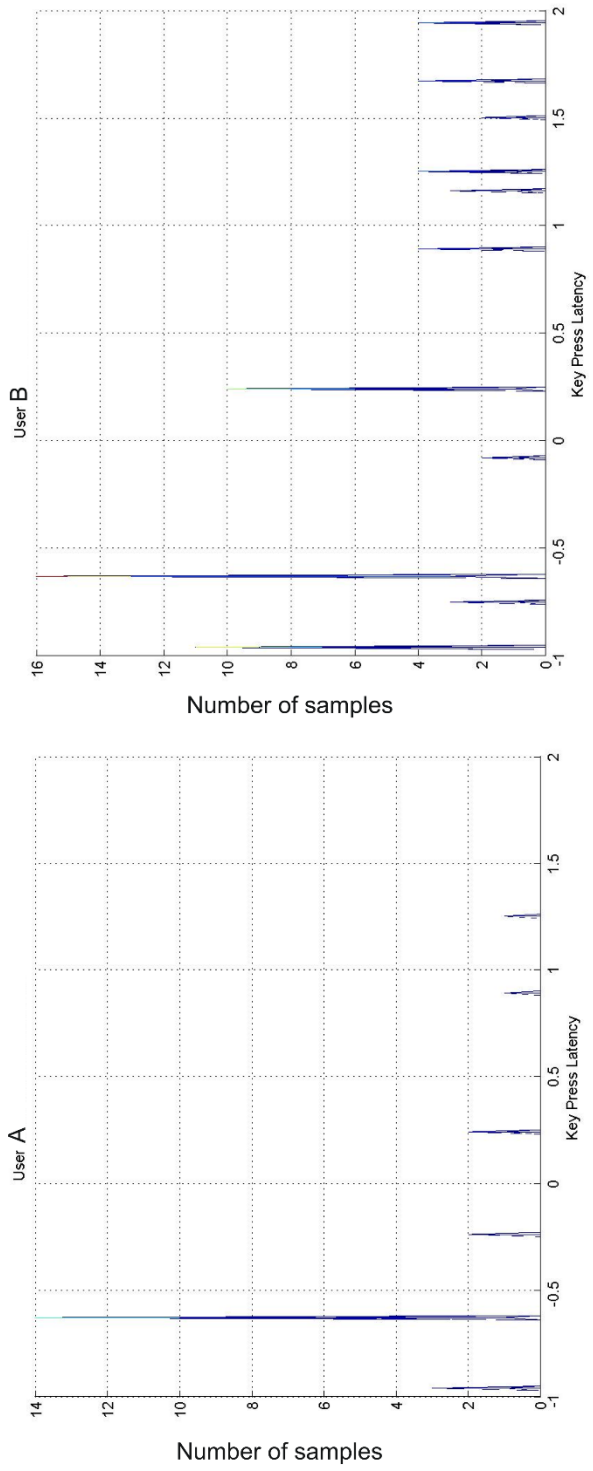
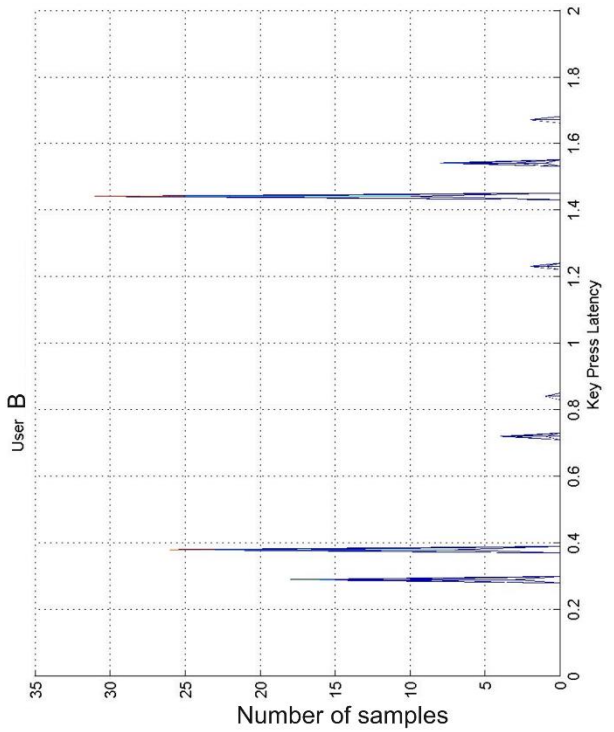


Figure 18 – Example Key Interval Distributions.

The second keystroke feature, KeyPress, was the latency of how long each key was held down. Linux already incorporates auto-repeat detection, but we ignored this and simply computed the difference between the down- and up-events. Similar to KeyInterval, we filtered the extracted values to only consider the latency range between 0 and 3 seconds. Figure 19, shows the composite relief graph of two users' average KeyPress distributions over their entire session duration.



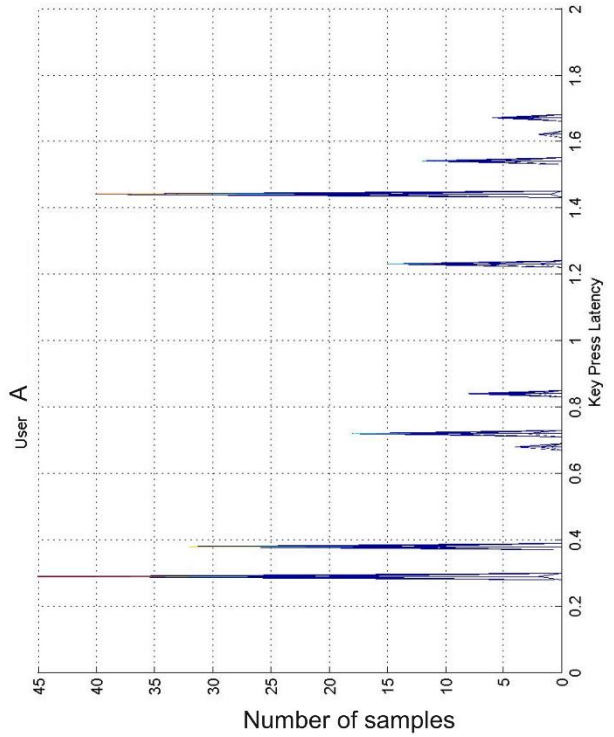


Figure 19 – Example KeyPress Latency Distributions

The first mouse feature we chose, denoted Angle of Curvature, considered the interior angle of every triple of x-y coordinates associated with a given mouse movement as illustrated in Figure 20. As an explicit dependent upon carved data, this feature extracts a distribution of such angle values from a carved pointing action. We defined a pointing action as a series of x-y coordinates consecutive in time and terminated by either a button event, or a pause of 0.4 seconds. The pointing actions begun within a classification time window, contributed to an average distribution which was rendered as the final unit of classification. Figure 21 shows the composite graph of such average distributions using the same filtered visualization as the keystroke feature visualizations above. Since most actions form reasonably straight lines, only values between 135° and 180° were collected in distributions with a bin size of 1°.

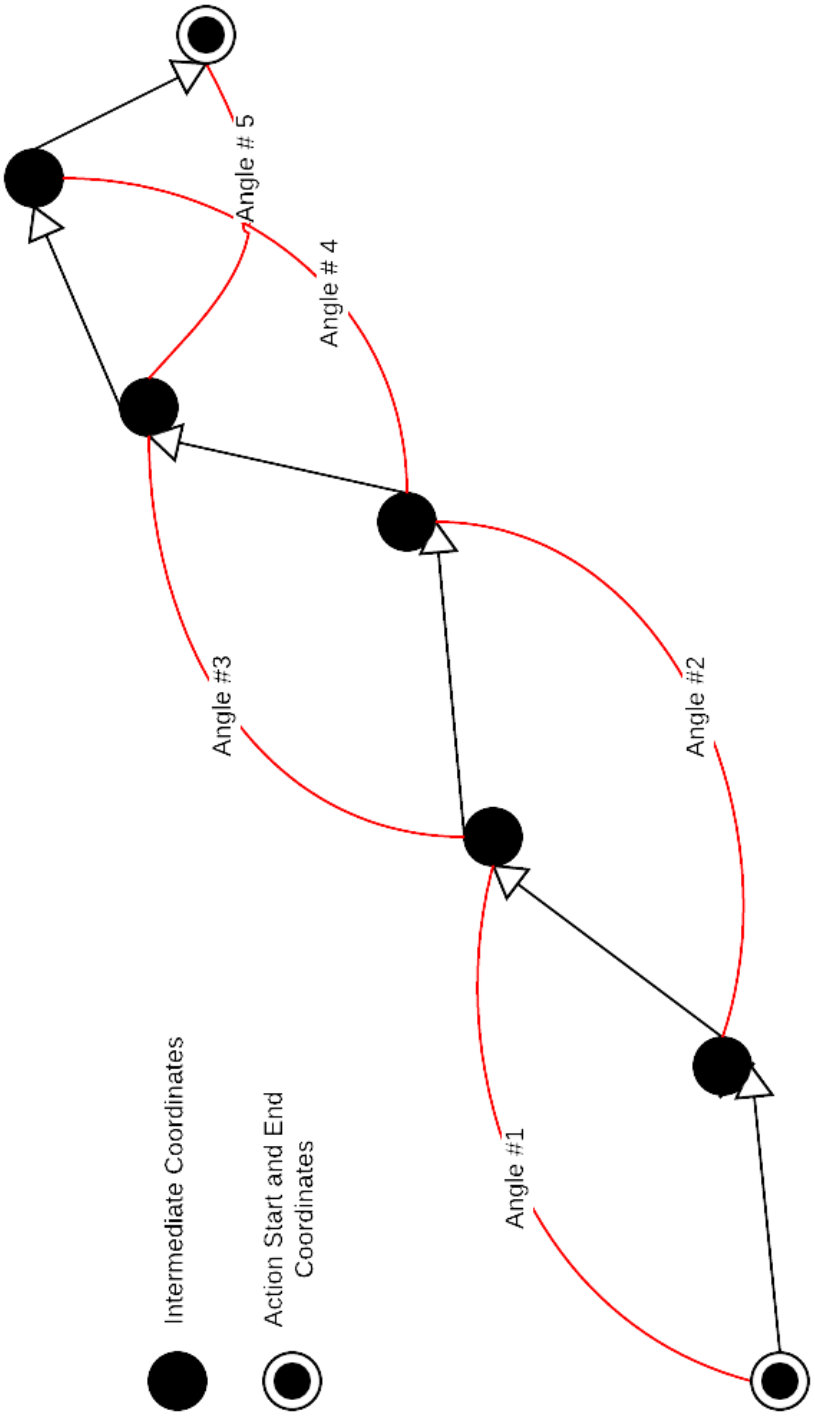


Figure 20 – Angle of Curvature Feature

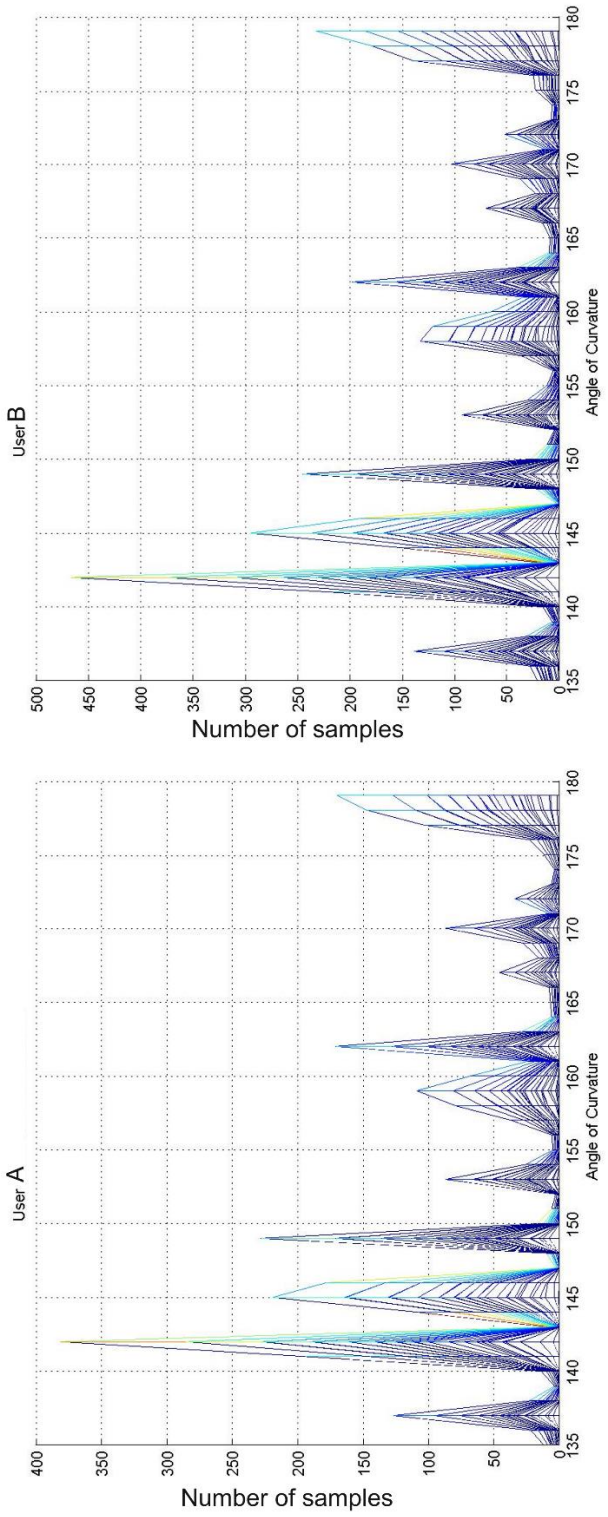


Figure 21 – Example Angle of Curvature Distributions

The final feature we used, Curve Distance, characterized cursor movements with a similar metric, but used a distinct set of calculations. Rather than taking the interior angle of every coordinate triple, Curve Distance computes the ratio of the line joining the first and last points to the line drawn through the middle point and perpendicular to the line joining the first and last, as shown in Figure 22.

Curve Distance and Angle of Curvature are related, but are not convertible features. It is easily shown that it is possible for two coordinate triples to have equal angles of curvature but different curve distances and vice-versa. As illustrated in Figure 23, consider three triples ABC, DEF, and DE′ F. Let B lie on the circumference of the circle whose diameter is the line AC, while E lies outside the circumference of the circle whose diameter is DF such that the line through E and perpendicular to DF meeting at Y is equal to the line through B and perpendicular to AC meeting at X. Further let AC equal DF. We know from [37] that the angle <ABC is equal to 90° while the angle <DEF is less than 90°, while the ratios EY : DF and BX : AC are equal since EY = BX and AC = DF. Conversely, let EY be reduced to the point at which the circumference of the circle with diameter DF cuts it at point E′. Since E′ Y is shorter than EY, the ratios E′ Y : DF and BX : AC are not equal. However since E′ now lies on the circumference, both angles <ABC and <DE′ F equal 90°. Therefore, it is possible for either feature to discriminate while the other fails.

Figure 24 shows example distributions of Curve Distance values from the corpus. Manual inspection of the data determined the distance values in the range of 0 – 0.4 as appropriate to construct distributions from. The bin size was set to 0.01.

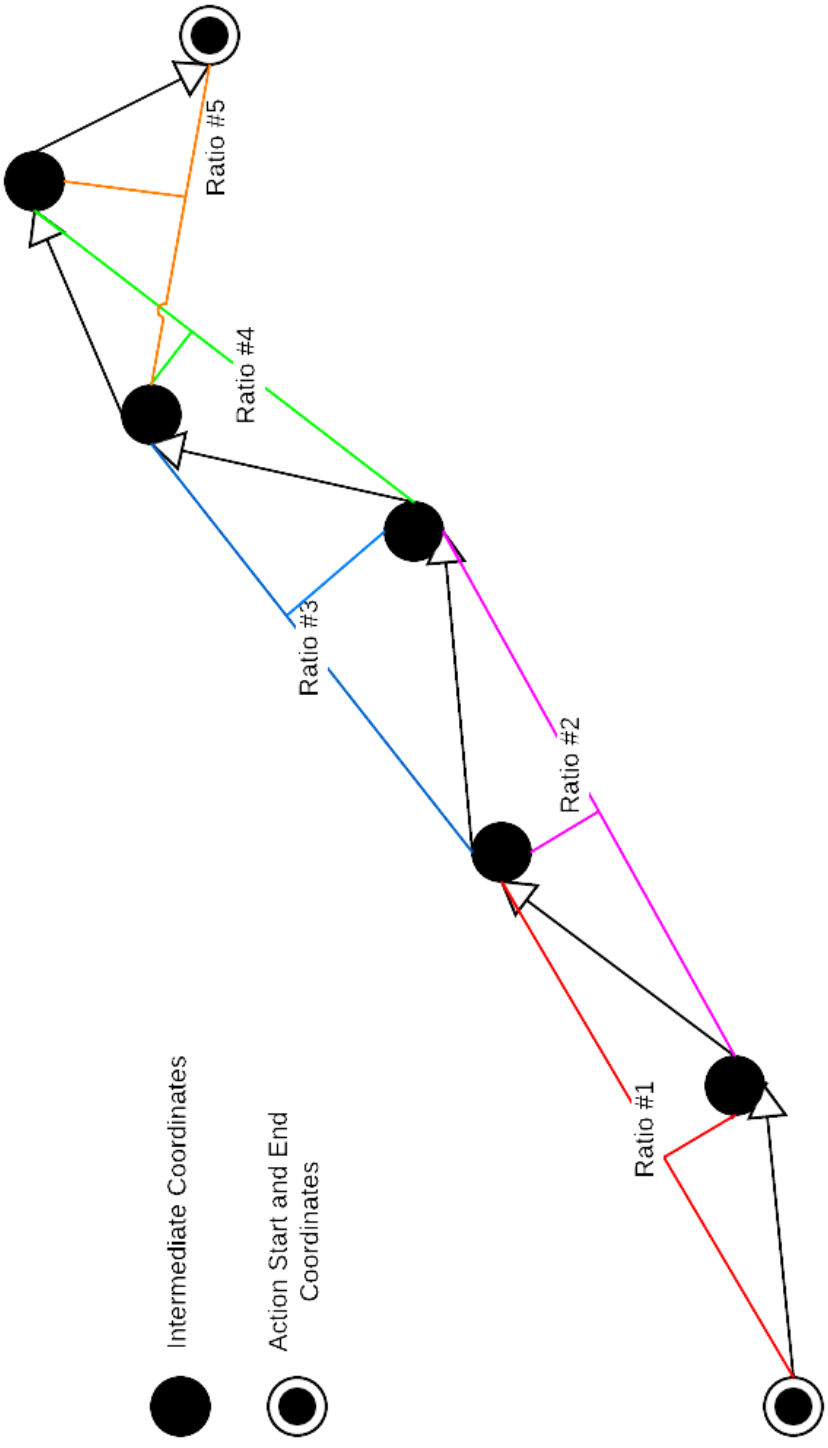


Figure 22 – Curve Distance

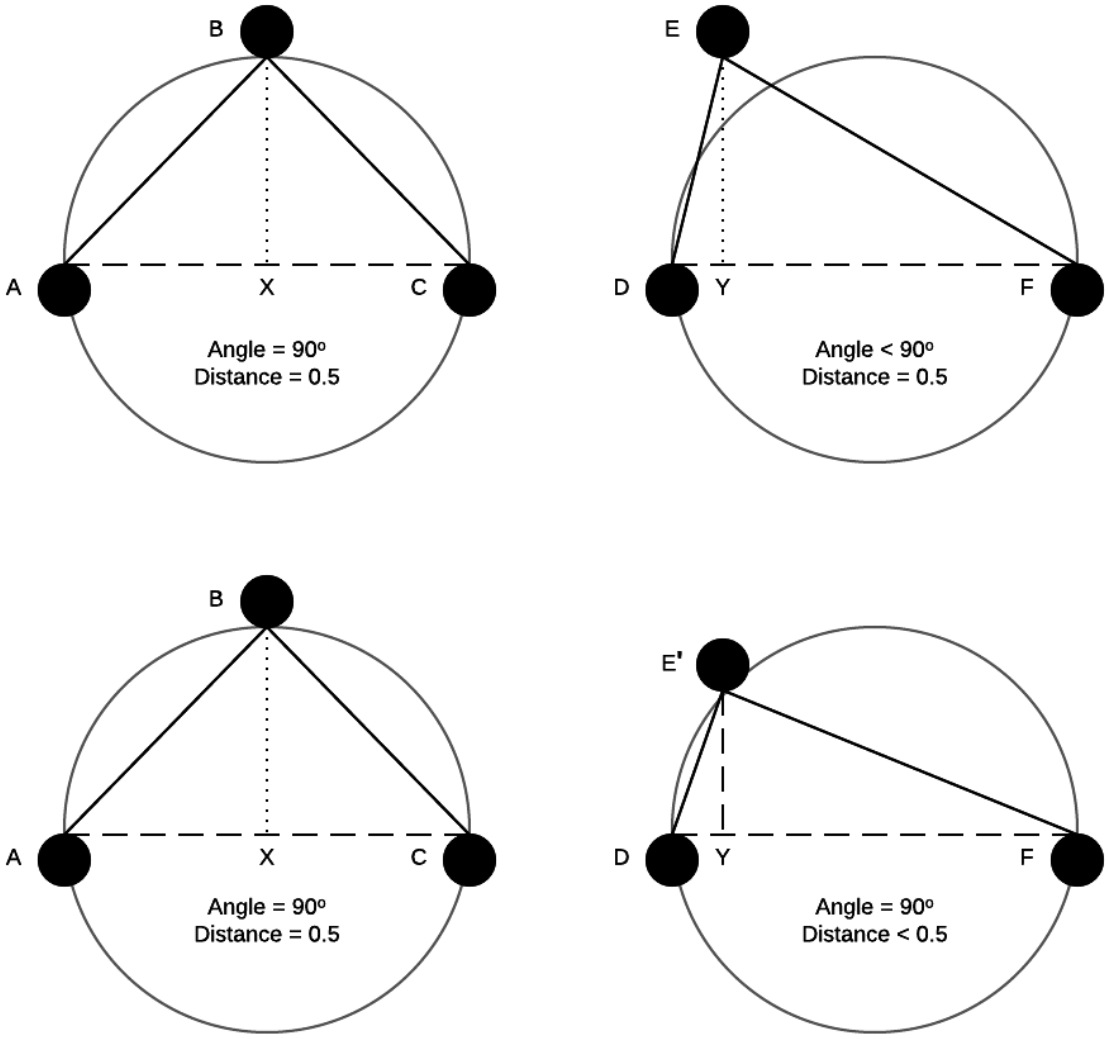


Figure 23 – Angle of Curvature vs. Curve Distance

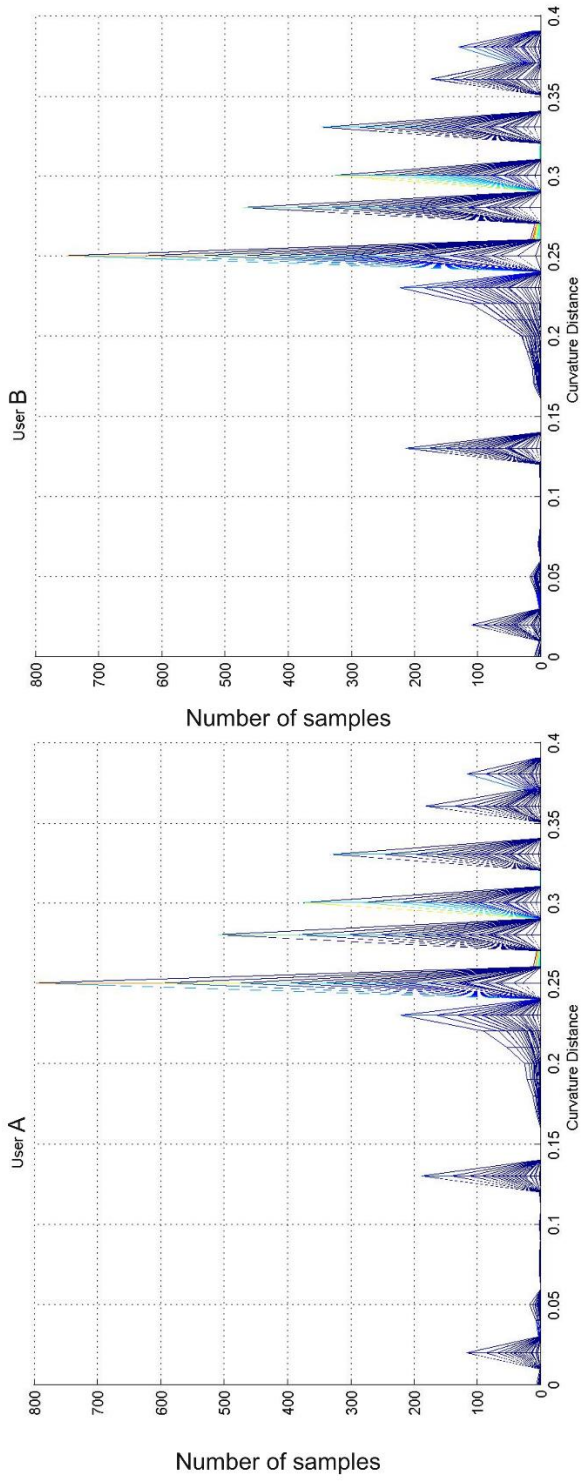


Figure 24 – Example Curve Distance Distributions

The classification of these features assumed the presence of subtle differences in the temporally cumulative distributions of different users. To maximize the effect of these differences, two parameters of each mono-modal system were considered: window duration and distribution bin size. The duration of a time window controls both how many data points on average will be included in a distribution and the response time of the system. Figure 25 shows how the distribution of each feature takes shape from data collected in windows varying from 0 – 1800 seconds in length. As with the two-dimensional visualization above, these three-dimensional graphs show the average distribution for time windows, rounding frequencies to integer values. For all four features, it appears that the characteristic distribution has acquired all it peaks by ~250 sec., and uniformly scales thereafter.

Bin size controls the granularity of value discrimination with distribution. The only drawback of reduced bin sizes is the increased processing power/time required to render the distributions. This is not a concern for the live deployment of active authentication systems, since real-time delay of acquiring input data eclipses the processing latency, but for running experiments on our saved corpus, processing a net 6 hours of session time is considerable. For our experiment, we picked the finest level of binning we could render in a reasonable time, namely 0.01 seconds for the keystroke features, 1° for the Angle of Curvature and 0.01 ratio differences for Curve Distance.

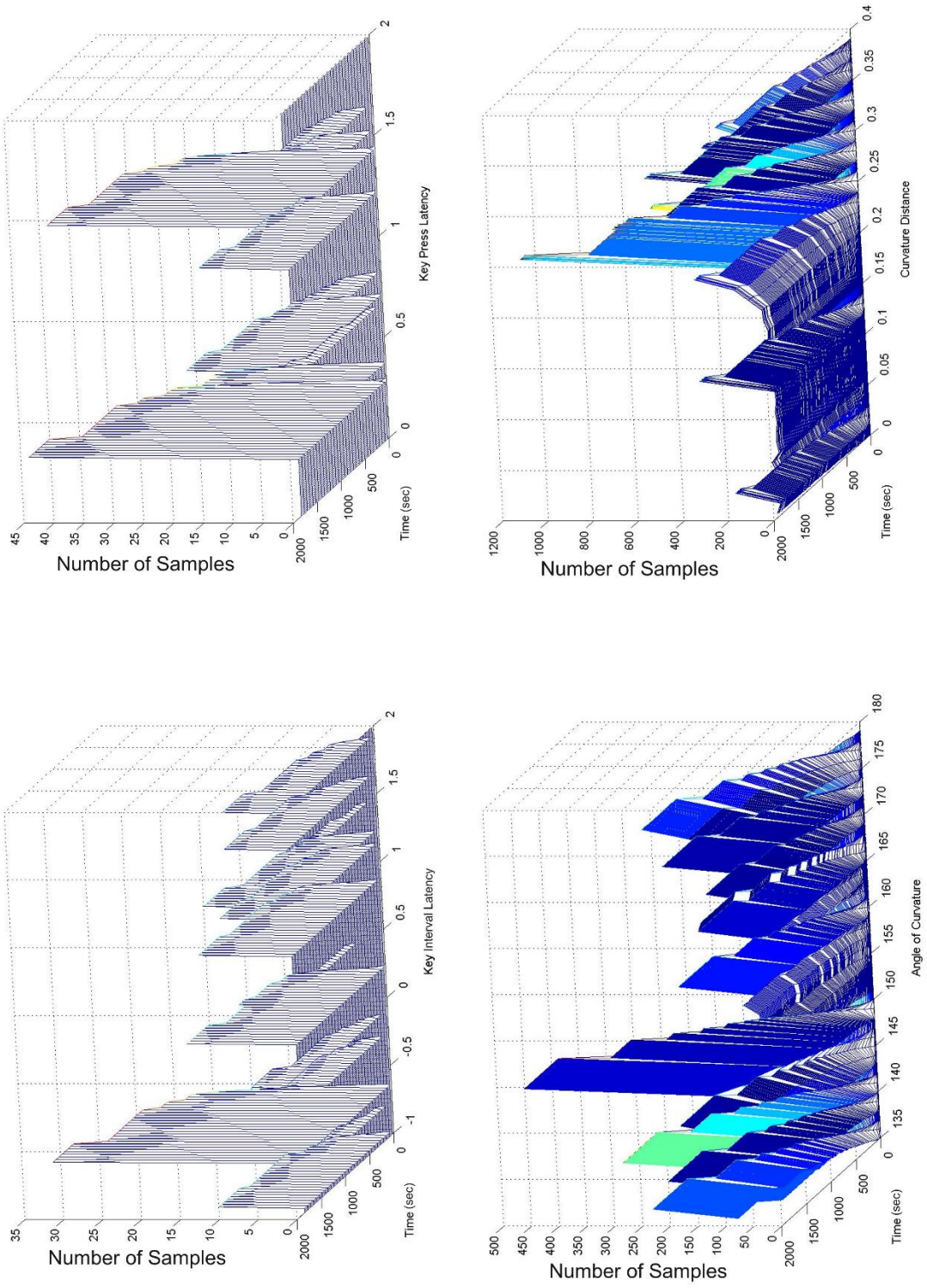


Figure 25 – Growth of Distributions with Window Duration

The classification unit for all mono-modal systems was defined as the distribution of feature values over 150 sec windows requiring a minimum of 30 values per window for a decision. Following the methodology of [28], naïve bayes was used as the classifier for all modalities. Each mono-modal system's FAR and FRR for each user was pre-characterized by a four-fold cross-validation process. The training data for each user was divided into four folds, assigning 3 folds to a pre-training set and the remaining fold to a cross validation set. The folds selected for cross-validation and training were cycled without re-dividing the data so that every possible combination of consecutive folds for training and cross-validation was executed. The average FAR and FRR for the mono-modal systems was computed over all fold cycles for each user. The classifier trained for the last fold cycle was used to classify the test set and send decisions to the multimodal system for decision fusion.

We employed the application of the optimal fusion rule given by [30] and applied in [28], weighting each mono-modal system's decision with its pre-characterized FAR and FRR, according to:

$$f(u_1 \dots u_n) = \begin{Bmatrix} 1, & if\ a_0 + \sum_{i=0}^{n} a_i u_i > 0 \\ -1, & otherwise \end{Bmatrix} \qquad (1)$$

where the vector $u_1 \dots u_n$ is comprised of the decisions of the mono-modal systems, with 1 meaning authentic and -1 meaning inauthentic. The weights $a_0 \dots a_n$ are computed from each mono-modal system's characteristic FAR and FRR according to,

$$a_0 = \frac{P_1}{P_{-1}} \qquad (2)$$

and,

$$a_i = \begin{cases} log\frac{1-P_i^{FRR}}{P_i^{FAR}}, \ if\ u_i = 1 \\ log\frac{1-P_i^{FAR}}{P_i^{FRR}}, \ if\ u_i = -1 \end{cases} \qquad (3)$$

where P_1 and P_{-1} are the *a priori* probabilities of the authentic and inauthentic cases, both of which we defaulted to equal 50%, and where P_i^{FAR} and P_i^{FRR} equal the characteristic FAR and FRR rates for the *i*th mono-modal system respectively.

It was hypothesized that the combination of modalities would maintain a higher accuracy than any single modality. Validation of this hypothesis consisted of a 5-fold cross-validation experiment. The standard accuracies of each mono-modal system and the multimodal system were used to demonstrate improved performance. The folds selected for testing and training were cycled without re-dividing the data so that every possible combination of consecutive folds for training and cross-validation was executed. Since each user's test was balanced with equal numbers of positive and negative samples, accuracy is a valid metric to consider using 50% as a most common tag baseline. Figure 26 shows the classification accuracy of the multi-modal system and all four modalities for each user.

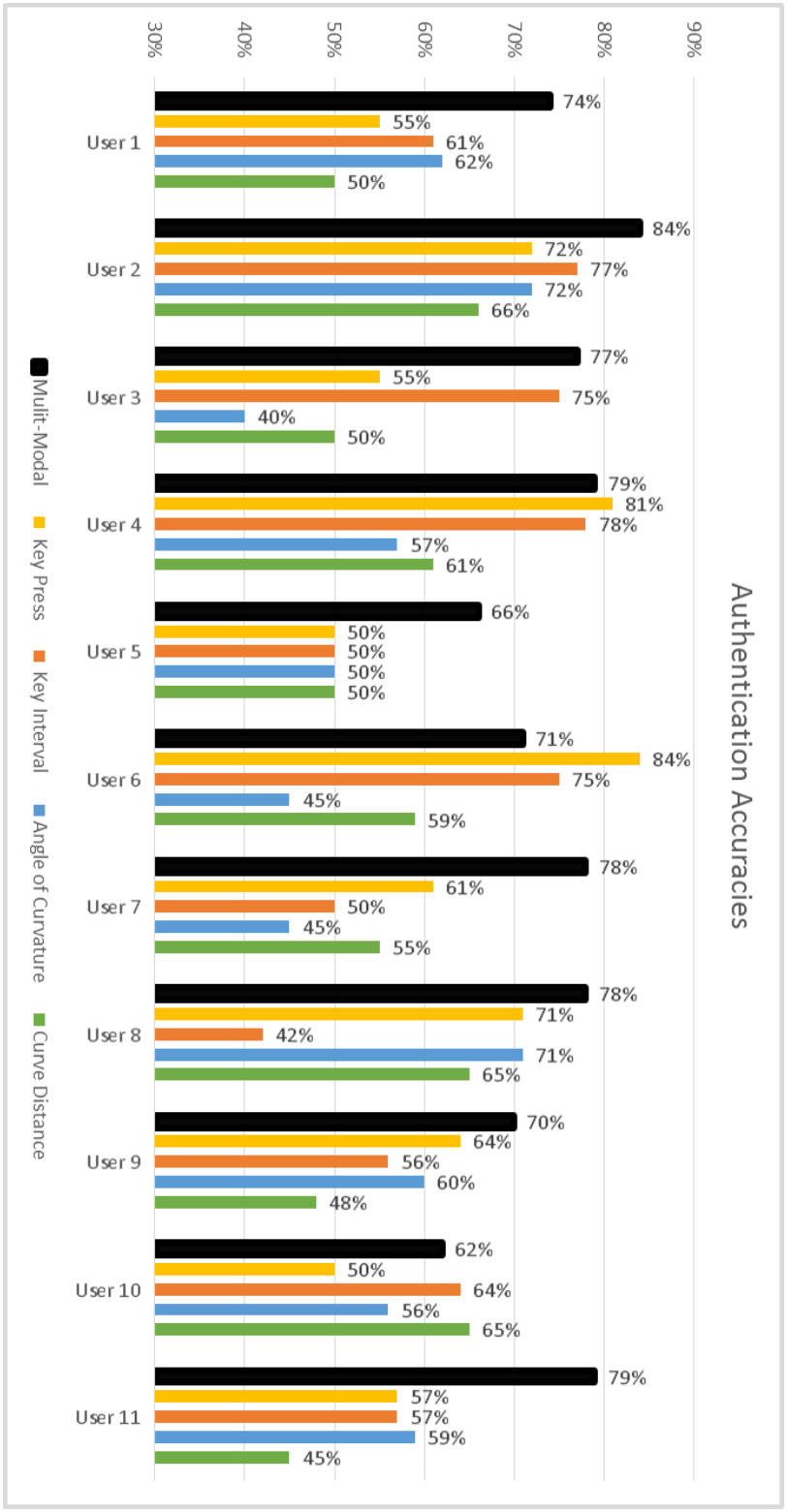


Figure 26 – Authentication Accuracies

While the multi-modal system did not always outperform the best mono-modal system, it was the best for 8 out of 11 users. Of course these accuracy results are far less than the state of the art, but that was entirely expected, considering the limited size of the corpus and distribution granularity. Furthermore, since each window is non-overlapping and therefore strictly consecutive, there is no consistent transition from authentic to inauthentic data. This is a strict standard since each window is classified in isolation, and cannot be affected by surrounding window data or decisions.

Nevertheless, these results demonstrate the power of the fusion rule to optimize the performance of multiple mono-modal systems in a single system. As shown in Figure 27, the multi-modal system achieved an average 10% improvement over the best mono-modal system average.

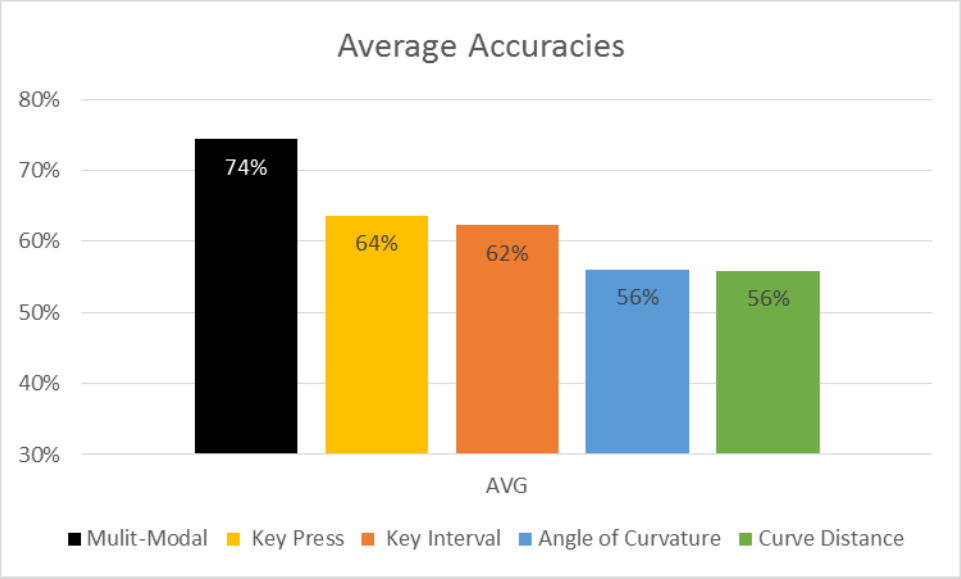


Figure 27 – Average Authentication Accuracies

The key to this performance optimization is the information gained by the fusion algorithm through the characteristic FAR and FRR acquired in training. As shown in Figure 28, every user displayed a variety of FAR and FRR values for each feature in training. When one modality performs more accurately overall or only for accepting or only for rejecting, the fusion algorithm leverages that information to give deference to the most reliable modalities. In a sense, this converts the modality performance itself into features specific to the user.

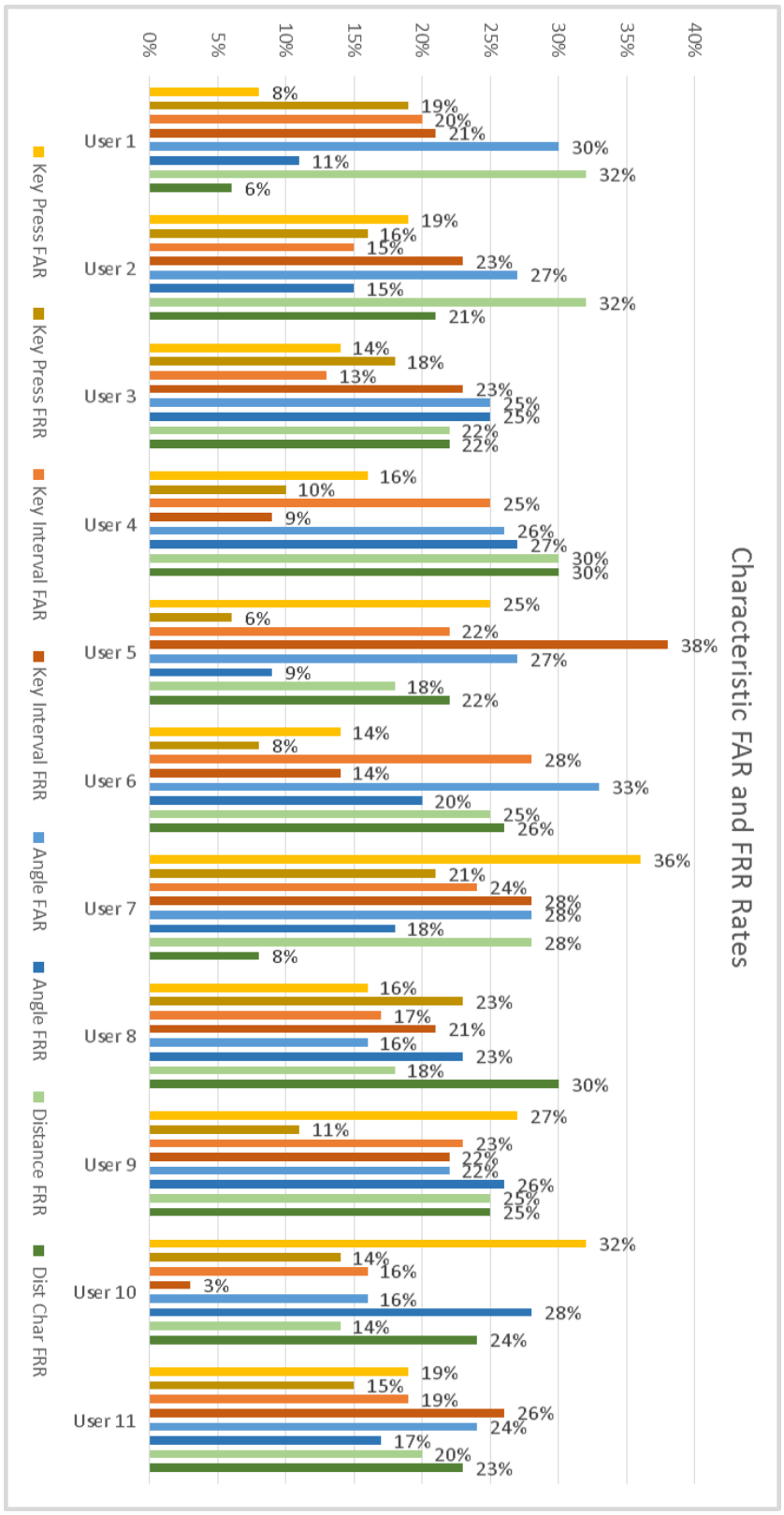


Figure 28 – Characteristic FAR and FRR

This experiment could be improved if rolling decision windows were used in place of strictly consecutive windows. That would considerably improve the response time of the system while testing the ability of the system to detect polluted windows. A further extension would be the simulated live performance over a session involving real user transitions.

Originally we intended to investigate the precise reaction of the multimodal system to tasks heavily skewed to certain modalities. Unfortunately, we only gathered one session of data per user, which was minimal for a cross-fold validation. If a corpus could be built with multiple sessions per user, training and cross-validation could use some sessions while leaving an entire session intact for testing. That way, one continuous test of the performance could be recorded over a data set sometimes skewed to one modality, sometimes skewed to another. However, these improvements exceed the goal of validating AARF with a simple proof-of-concept experiment.

Though modest, these results show that AARF has the foundation for future refinement. Given the benefits of abstraction and alleviation of trivial data management, improved results should forevermore only depend upon the quality of the research ideas themselves, and not so much on the ability to merely manage data and ad-hoc scripts.

Chapter 8 – Future Work

As with any software tool, the AARF framework has a virtually unlimited capacity for improvement in terms of compatibility, performance and convenience features. However, there are a few features that are peculiar to the demands of this type of research.

Many of the raw data modules will need to be manually updated as new input sources arrive. Furthermore, differences in hardware, can change the required device files providing the same data. A fully automated auto-configuration of inputs, making AARF truly plug and play, would be most desirable.

Currently, a hardcoded list of raw input publishers informs the loading modules of all the available publishers to attempt to load. A comprehensive reflective discovery of available publishers would allow contributors to immediately have AARF's session management be aware of new additions.

While raw data corpuses have the most potential for innovative research, it may still be needed to produce corpora of semi-processed data or feature values to save time during experiments. Another core addition to AARF would be a generic corpus transformer tool which takes a less processed corpus and produces a more processed corpus according to the processing modules specified.

Windowing and feature extraction often go hand in hand when computing feature vectors as frequency distributions over the feature-value space. But while windowing is a generic operation that only needs a timestamp or event count to compute, building frequency distributions require knowledge of a specific feature's typical value space. A

standardized interface for the AARF Feature Extractor may be desirable to enforce the provisioning of floor, ceiling, and bucket size values to allow any AARF feature to be quantized on demand.

Since response time is a critical performance element of active authentication, it seems reasonable that mono-modal systems should be forced to incorporate a windowing system. Adding such a requirement to the mono-modal interface would significantly reduce the flexibility of timestamp management and mono-modal systems design in general. Considering the central role mono-modal systems play, the wisdom of this design decision was left for future consideration. Perhaps there are decisions spaces that should not be immediately bound to a regular time interval.

Thus far the object oriented design pattern deployed in AARF has aimed at encapsulating pipeline operations and processing modules. However, the data itself passing through the pipeline remains relatively primitive, Python dictionaries and tuples and are not represented by a class. Throughout the testing of AARF it became apparent that the Linux event format, statistical manipulation functions, and timestamp handlers would be good candidates for a data object class. However, the proper variety of generic data classes and associated logic was not immediately clear. Furthermore, once the data is encapsulated, the manual inspection and handling requires maintaining a whole new interface. What this interface should be at each stage of the pipeline requires further thought.

Chapter 9 – Conclusion

Usable security remains one of the greatest challenges in the digital world. Indeed, one of the most frequent breaches of security is the failure of usable authentication protocols. While popular approaches involving arbitrary secrets (usable in the sense that tens of millions of people trust their digital lives to them), are not truly usable when considering the inverse relationship their ease of use bears to their security. Non-arbitrary secrets have begun to reverse this relationship, but static tokens, such as fingerprints and iris scans, still bear the problem of spoofing. Dynamic forms, such as characteristic typing or cursor movement, would, in principle, bind the form of authentication to the useful action of the device. This marriage of use and authentication through behavior at once eliminates usability as a concern, and allows authentication to proceed throughout the entire session rather than just at login.

The goal of truly continuous, active authentication has been pursued for some decades in a variety of modalities and on a variety of hardware technologies. However, only in the past three years has a serious effort been launched by DARPA to make active authentication schemes truly practicable. Contemporary research is still quite diverse and is conducted on a wide variety of data sources, collection mediums and classification logic. The need for systematization in this nascent field was taken up by the Active Authentication Research Framework (AARF), proposed by this thesis. AARF organizes research efforts broadly into extensible modules handling data collection, serialization, pre-processing, feature extraction, classification and multi-classification management. It

further offers the organization of mono-modal and multi-modal systems whose parts are standardized to be as interchangeable as possible.

This framework provides a basis for building a repository of active authentication research, thereby enhancing the reproducibility, traceability, and reuse of prior work. Hopefully this will aide accelerating open progress in the field and one day make active authentication a default capability of personal digital devices.

BIBLIOGRAPHY

[1] G. Stoneburner, C. Hayden and A. Feringa, "Engineering Principles for Information - Technology Security (A Baseline for Achieving Security), Revision A," NIST Special Publication, 2008.

[2] M. K. Evans, "Consumer Security," ECT News Network, Inc, 3 April 2006. [Online]. Available: http://www.ecommercetimes.com/story/49731.html. [Accessed 21 October 2014].

[3] Identity Theft Resource Center, "ITRC DATA BREACH REPORT," ITRC, 2014.

[4] F. Stajano, P. C. v. Oorschot, C. Herley and J. Bonneau, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *IEEE Symposium on Security and Privacy*, 2012.

[5] M. Nisenson, I. Yariv, R. El-Yaniv and R. Meir, "Towards Behaviometric Security Systems: Learning to Identify a Typist," in *Knowledge Discovery in Databases: PKDD 2003*, Heidelberg, Springer Berlin, 2003, pp. 363-374.

[6] DARPA, "Broad Agency Announcement," 13 January 2012. [Online]. Available: https://www.fbo.gov/utils/view?id=65da86bb52c0f992d9631447f2a6e357. [Accessed 21 October 2014].

[7] P. Bours, "Continuous keystroke dynamics: A different perspective towards biometric evaluation," *Information Security Technical Report*, vol. 17, no. 1-2, pp. 36 - 43, 2012.

[8] I. Deutschmann, P. Nordstrom and L. Nilsson, "Continuous Authentication Using Behavioral Biometrics," *IT Professional*, vol. 15, no. 1520-9202, pp. 12-15, 2013.

[9] 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*, Chicago, Illinois, USA, 2011.

[10] A. Fridman, A. Stolerman, S. Acharya, P. Brennan, P. Juola, R. Greenstadt and M. Kam, "Decision Fusion for Multimodal Active Authentication," *IT Professional*, vol. 15, no. 4, pp. 29-33, 2013.

[11] H. Crawford, K. Renaud and T. Storer, "A framework for continuous, transparent mobile device authentication," *Computers & Security,* Vols. 39, Part B, no. 0, pp. 127-136, 2013.

[12] A. Azzini and S. Marrara, "Toward trust-based multi-modal user authentication on the Web: a fuzzy approach," in *Fuzzy Systems Conference, 2007. FUZZ-IEEE 2007. IEEE International*, London, UK. , 2007.

[13] J. Bonneau, C. Herley, P. C. v. Oorschot and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," University of Cambridge Computer Laboratory, Cambridge, 2012.

[14] D. D. Lewis, "Naive (Bayes) at forty: The independence assumption in information retrieval," in *Machine Learning: ECML-98*, Springer Berlin Heidelberg, 1998, pp. 4-15.

[15] S. R. Safavian and D. Landgrebe, "A Survey of Decision Tree Classifier Methodology," *IEEE Transactions on Systems, Man, and Cybernetics,* vol. Vol. 21, no. 3, pp. 660-674, 1991.

[16] A. L. Berger, S. A. D. Pietra and V. J. D. Pietra, "A Maximum Entropy Approach to Natural Language Processing," *Computational Linguistics,* vol. 22, no. 1, pp. 39-72, 1996.

[17] G. A. F. Seber and A. J. Lee, Linear Regression Analysis, Hoboken: John Wiley & Sons, 2003.

[18] J. David W. Hosmer and S. Lemeshow, Applied Logistic Regression, Hoboken: John Wiley & Sons, 2000.

[19] S. Tong and D. Koller, "Support vector machine active learning with applications to text classification," *The Journal of Machine Learning Research,* vol. 2, no. 3/1/2002, pp. 45- 66, 2002.

[20] S. Haykin, NEURAL NETWORKS: A Comprehensive Foundation, Upper Saddle River: New Jersey, 2004.

[21] J. AK and P. S. Ross A, "An Introduction to Biometric Recognition.," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 14, no. 1, pp. 4- 20, 2004.

[22] G. Forsen, M. Nelson and J. R. Staron, "Personal attributes authentication techniques," Rome Air Development Center, New York, 1977.

[23] D. Umphress and G. Williams, "Identity Verification through Keyboard Characteristics," *Int'l J. Man-Machine Studies*, vol. 23, no. 3, pp. 263-273, 1985.

[24] P. Bours and H. Barghouthi, "Continuous Authentication using Biometric Keystroke Dynamics," in *Norwegian Information Security Conference (NISK)*, Trondheim, Norway, 2009.

[25] Y. Zhong, Y. Deng and A. K. Jain, "Keystroke Dynamics for User Authentication," BAE Systems, Burlington, 2012.

[26] A. Messerman, T. Mustafic, S. Camtepe and S. Albayrak, "Continuous and non-intrusive identity verification in real-time environments based on free-text keystroke dynamics," in *Int'l Joint Conf. on Biometrics (IJCB)*, Washington DC, 2011.

[27] B. Sayed, I. Traor´e, I. Woungang and M. S. Obaidat, "Biometric Authentication Using Mouse Gesture Dynamics," *IEEE SYSTEMS JOURNAL*, vol. 7, no. 2, pp. 262-274, 2013.

[28] A. Fridman, A. Stolerman, S. Acharya, P. Brennan, P. Juola, R. Greenstadt and M. Kam, *Multi-Modal Decision Fusion for Continuous Authentication*, Preprint submitted to Computers & Electrical Engineering, 2014.

[29] M. Hall, "The WEKA data mining software," *ACM SIGKDD Explorations*, vol. 11, no. 1, pp. 10- 18, 2009.

[30] Z. Chair and P. Varshney, "Optimal data fusion in multiple sensor detection systems," *IEEE Transactions on AES-22*, vol. 1, no. 310699, pp. 98-101, 1986.

[31] H. Crawford, K. Renaud and T. Storer, "A framework for continuous, transparent mobile device authentication," *c omp u t e r s & s e c u r i t y*, vol. 39, no. May, pp. 127-136, 2013.

[32] R. Hamming, "1968 Turing Award lecture," *Journal of the ACM*, vol. 16, no. 1, pp. 3-12, 22 Feb 1969.

[33] Python Software Foundation, [Online]. Available: https://docs.python.org/3/.

[34] C. Joakim, "Explore Python, machine learning , and the NLTK library," *IBM Developer Works,* pp. 1-13, 2012.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel and B. Thirion, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning,* vol. 12, pp. 2825-2830, 2012.

[36] A. Caliskan and R. Greenstadt, "Translate Once, Translate Twice, Translate Thrice and Attribute: Identifying Authors and Machine Translation Tools in Translated Text," in *Semantic Computing (ICSC),* 2012.

[37] Euclid, "Euclid's Elements," p. Book III Proposition 31.

[38] R. P. Guidorizzi, "Security: Active Authentication," *IT Pro,* pp. 4-7, July/August 2013.