Co-Authentication

A Probabilistic Approach to Authentication

Einar Jónsson

Kongens Lyngby 2007
IMM-MSC-2007-83

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC: ISSN 0909-3192

Summary

All authentication mechanisms have a failure probability that is usually left implicit. Consider a password system that is presented with a valid password. The system cannot know whether the password was entered by its rightful owner or an impostor who has guessed the password, and despite that it is commonly known that some passwords are easily guessed, the password authentication system does not differentiate between weak passwords that are easily guessed and stronger passwords. Ignoring the failure probability, we risk silent authentication failures, e.g., an impostor is authenticated based on an easily guessed password. We believe that ignoring these failures leads to false security assumptions. Therefore, we propose to make the failure probabilities in the authentication method explicit, similar to what is now done in some biometric verification systems.

In this thesis we propose a probabilistic model of authentication, called *Co-Authentication*, which combines the results of one or more authentication systems in a probabilistic way. This model may, in some ways, be seen as a generalization of information fusion in biometrics, which has been shown to reduce the failure rates of biometric verification. We show that Co-Authentication increases flexibility in system design and that it reduces authentication failures by combining multiple authentication probabilities. The proposed model has been implemented in a prototype Co-Authentication framework, called *Jury*.

Preface

This thesis was prepared in the department of Informatics Mathematical Modelling, at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in Computer Systems Engineering.

The thesis deals with user authentication, and how authentication systems are subject to failures that are generally ignored. The main focus is on how authentication systems can be combined in a way that increases the reliability of the authentication result, but also how existing authentication systems can be adapted to the probabilistic scheme proposed in the thesis..

Lyngby, September 2007

Einar Jónsson

Acknowledgments

I would like to thank the following individuals for helping me towards my degree:

First of all, I want to thank my supervisor, Christian D. Jensen for his patience, constructive input, and motivation.

My parents, Jón Einarsson and Guðrún Þorsteinsdóttir, My girlfriend, Ósk Ólafsdóttir, and her parents, Ólafur Daðason and Helga Ingjaldsdóttir, are all thanked for their love and support, and without whom these past two years would have been significantly harder.

Contents

Chapter 1

Introduction

Identification and authentication comes naturally to human beings. Recent research suggests that we learn to recognize our mothers voice even before we are born [20]. We subconsciously use a combination of a persons physical attributes, such as their voice, behavior and other characteristics to authenticate those we know. and generally do not need elaborate authentication schemes when we speak to our friends on the phone, since we simply recognize their voices.

Computer systems on the other hand, are devoid of this ability. The reason we log into our systems in the morning is because our mere presence is insufficient for these systems to be able to identify us. Systems require us to perform some sequence of actions to identify ourselves and verify the claimed identity. In order to provide computer systems with the ability of authentication we use formal protocols, which typically require user interaction. We can generalize the authentication scenarios involving a computer system into three scenarios, namely *human-computer*, *computer-computer* and *human-computer-human* authentication [44]. In this thesis we will limit the discussion to the first category, which we refer to as user authentication.

User authentication can be defined as *"the process of verifying the validity of a claimed user"* [44], and the methods used for verification are typically divided into three categories: *something we know*, *something we have* and *something we are*. A password is an example of the first category, since we assume that

it is a secret known only by the legitimate user. A smart card is an example of *something we have*, namely a physical token which is assumed to be in the possession of the legitimate user. Finally, biometrics are an example of the last category. By allowing the system to scan our iris or fingerprint we provide it with data that is, at least for all practical purposes, unique to us.

The authentication process is commonly performed in two steps: *identification* and *verification*. In the identification mode, the system needs to establish our identity, which typically involves the user providing a username. An example of a more elaborate scheme, is a biometric face recognition system which identifies the person who is sitting in front of the terminal in an unobtrusive manner, and relays the identity information to the authentication system. Once the system has been provided with an identity, it needs to verify the authenticity of it. In most cases this means that the users have to provide a password which has been associated with their account. Other means of verification include providing a fingerprint or a physical token, such as a smart card. All of these methods have drawbacks that can seriously affect the security they provide. A password can be shared, guessed or forgotten, a physical token can be stolen, and a fingerprint can be forged [35]. In other words, each authentication method is susceptible to different types of attacks.

The verification can be based either on an exact match or a probabilistic match of the verification input. For an exact match, the input value is compared to a stored value and rejected unless they are identical. An example of this are password systems, where a hashed value of the password is stored on the system. The password provided by the user is then hashed and compared to the stored value and the authentication fails if there is any difference between the two values. In other words, a password system will not distinguish between an input of a correct password with a single typographic error and a string which has no characters in common with the genuine password. Similarly, it will treat all matching inputs the same way, even if they provide significantly different levels of security. In this paper we will use the term *binary authentication systems* for systems that employ an exact match verification.

For a probabilistic match, the input value is typically compared to stored data and is ranked based on the similarity between the input and the stored data. The similarity score is then compared to a pre-configured threshold, and if it exceeds the threshold the authentication is successful, but fails otherwise. Examples of such threshold-based authentication systems are biometric authentication systems. In biometric systems, the user input is a sample of the user's specific biometric traits, which is compared to a stored template that was registered during the user's enrollment in the system.

Threshold based systems have two complementary error rates, namely a false

accept rate (FAR) and a false reject rate (FRR). A false accept is when an impostor is accepted as a genuine user, whereas a false reject is when a genuine user is rejected as an impostor. The threshold value for such a system determines the balance between FAR's and FRR's. A low threshold will increase FAR's and decrease FRR's, and a high threshold will do the opposite. For example, a fingerprint scanner identifies its input as belonging to John in accounting, with a 0.52 match score. If the threshold is above 0.52, the authentication will be rejected regardless of whether the sample input is authentic or not. Similarly, it will be accepted if the threshold is below the match score, even if the sample belongs to an impostor.

A considerable amount of work has been done in the field of biometrics to decrease these error rates by combining multiple biometrics into so called multi-biometric systems. The general idea is that the results of multiple biometric systems are combined, into a single result, and such systems have been shown to have lower error rates than any of the participating systems [28]. One method of combining these systems is to use the individual match scores. By running a score fusion algorithm, such an arithmetic mean function, on the match scores we can compute an overall score, which serves as a probability of a genuine authentication.

Binary authentication systems, such as password systems, are also subjects to false accepts and false rejects. An impostor might guess the password of a legitimate user and thereby cause a false accept. Similarly, long and complicated passwords are likely to increase the frequency of input errors, causing false rejection of legitimate users. In this particular example, the password complexity policy can be seen as a threshold, striking a balance between the FAR with regard to guessing attacks and FRR. Unlike their biometric counterparts, binary authentication systems leave these error rates implicit and typically ignore them. Normally all passwords are seen as equal, and entering the correct password is seen as sufficient proof that the presented identity is authentic. However, not all passwords are equal. Publicly available password crackers [22, 41] will crack some passwords in a matter of seconds, while it can take weeks, months or even years to crack others [34]. This suggests that passwords can be assigned a strength indicator, which can be seen as a probability of a genuine authentication. While the password remains an exact match, the strength indicator allows us to implement a threshold-based user authentication system, i.e., the authentication of users with strong passwords is considered stronger than the authentication of users with weaker passwords.

Similarly, we have different levels of confidence towards different systems. For instance, a credit card transaction made using a magnetic stripe is considered less secure than if the transaction were made with a Chip & PIN technology. We believe that it is a good idea to quantify these confidence levels and take them

into account when making authentication decisions. By combining these levels with other authentication factors we can make better informed decisions about whether or not to authenticate a particular principals, be it users, transactions or something else.

In this thesis we propose *Co-Authentication*, which allows multiple authentication systems to combine their probabilistic results and reach a unified threshold-based decision. It allows us to combine static confidence levels, biometric match scores as well as any other authentication factors that can be expressed in probabilistic terms, and compute an overall authentication score. We have developed a generic Co-Authentication framework, called *Jury*, which allows us to combine these scores using different statistical methods. The Jury framework provides a generic platform that allows organizations to gradually adapt existing security infrastructure to the Co-Authentication scheme. Finally, we show that the framework performs well enough to be applicable in real scenarios, and give examples of how existing binary authentication systems can be adapted to a threshold-based scheme, and how they can benefit from Co-Authentication.

The rest of the thesis is organized as follows: We discuss the background and current state of the art in Chapter 2 and motivate our work by showing where these current approaches are lacking. The notion of *Co-Authentication* is introduced and analyzed in Chapter 3. We present the requirements, design and implementation of the Jury framework in Chapters 4, 5 and 6 respectively. Our work is evaluated in Chapter 7 where we also show how existing authentication schemes can be adapted to Co-Authentication by integrating them with the Jury framework. Finally we summarize our work and conclude the thesis in Chapter 8.

Chapter 2

Background and Motivation

Systems are often described in terms of strength, i.e., a system is either *strong* or *weak*. This is a relative and abstract scale, where a system is considered to be strong if it is impractical to break it, i.e. it is not worth it, given the cost of the attack. Similarly, a system is considered weak if it is either easy to break, or if the cost required to break it is considered to be acceptable with regards to the potential gains of having access to that system. The cost of attacking a system can refer to money, time, overall effort and risk.

To put the above discussion in a concrete example, let us imagine that we are evaluating data security solutions to protect industrial trade secrets that are to be used in a product that will be released in two years time, and is expected to give a profit of $\$100\dot{0}00$. Given that the time and revenue estimates are accurate, we can automatically reject all data protection solutions which will cost more than the expected profit. Similarly, solutions such as data encryption mechanisms, which are expected to take more than two years to break are acceptable since the protected data will become public knowledge in two years.

The discussion above is summarized in Definition 2.1, and we will refer to it throughout this thesis.

Definition 2.1 (System Strength) "*A strong system is one in which the cost of attack is greater than the potential gain to the attacker. Conversely,*

a weak system is one where the cost of attack is less than the potential gain. Cost of attack should take into account not only money, but also time, potential for criminal punishment, etc.“ [44]

A system with a weak component is considered to be weak, even if it enforces strong security in other areas. For instance, imagine a server which is stored in a highly secure building, but which allows remote access using weak passwords. We must assume that an attacker will target the most vulnerable point within the system, e.g., the remote access in the example above. If a house has thick concrete walls, reinforced steel doors and a few fragile and unprotected windows, it is obvious that a smart burglar will enter through the windows, and the same principle applies to computer systems. In other words the system is only as strong as its weakest link. Pfleeger and Pfleeger summarize this well in their *principle of easiest penetration* [46], which is shown in Definition 2.2. We will also refer to this definition in subsequent chapters.

Definition 2.2 (Principle of Easiest Penetration) “*An intruder must be expected to use any available means of penetration. The penetration may not necessarily be by the most obvious means, nor is it necessarily the one against which the most solid defense has been installed.*” [46]

In the following sections we will present common authentication systems and combinations thereof. It is helpful to keep the above two principles in mind when we discuss the weaknesses of various authenticators, to determine whether the weaknesses are of real concern in a particular context.

2.1 Authentication Systems

We will now give an overview of various commonly used authentication systems. In particular we will provide a detailed analysis of password systems, since they are the most commonly found systems.

2.1.1 Analysis of Password Security

A password is a secret sequence of characters, generally only known by a single user, and it is the oldest authentication scheme used in computer systems. Passwords are typically assigned to a user identifier, also known as a *username* in such a way that each user has her own password, that she inputs along with

her username to authenticate herself to the system. The system then looks up the stored password which is associated with username, and compares it to the input password to determine whether they match. If they match the user is successfully authenticated, but otherwise the authentication fails, i.e., the user is not authenticated. A match means that every character in the input matches the character in the same position in the stored password, and that the two strings are of the same length.

Password systems typically do not store the password itself, but its *hash* value. The *hash* is the result of performing a one-way cryptographic hashing function, such as MD5 [50] or SHA-1 [13], on the password, which results in a password storage that is hard to reverse. When a user logs in, the password is hashed using the same hash function as was used when the password was set. If the computed hash and the stored hash match, then the password was correct. In some cases the password string is used as a key in a one-way hash function, which then hashes a constant.

2.1.1.1 Keyspace and guessing probabilities

Passwords are an example of so-called *Knowledge-based* authenticators, which means that users must keep their passwords secret, and in order for a password to provide sufficient protection, it has to be hard to guess. We call the number of possible password combinations a *keyspace*. A random password from a large keyspace is theoretically harder to guess than a random password from a smaller keyspace. A password of length n from a character set of size c, will have a keyspace size of $k_p = c^n$ [44]. Therefore we can either make a password longer, or use more characters in order to increase the size of the keyspace.

As an example, suppose we have a 4 digit password which gives us a keyspace of $10^4 = 10000$ possible passwords. By increasing the length of the password to 6 digits we obtain a keyspace of $10^6 = 1000000$ whereas keeping the same length but increasing the allowed input characters to include all lowercase alphabetic characters and digits yields a keyspace of $(26+10)^4 = 16791616$. Assuming that passwords are evenly distributed over the keyspace, the probability of a random guess matching a password – given that the guess is of the right length – is then:

$$P(correct\ guess) = \frac{1}{k_p}$$

.

The above examples all show the keyspace for a single password of length n.

Plaintext	Salt	Salted Plaintext	MD5 Hash
JohnyBGood	ab	*abJohnyBGood*	16bfd9df440f758cc93b87ba0016fc14
JohnyBGood		*JohnyBGood*	ef242b0321979b00330c4cc82177697a
JohnyCGood	ab	*abJohnyCGood*	fec79a705eba0ae76cafe0967d6b1d1b

Table 2.1: This table shows how much effect changing a single letter or changing the salt has on the outcome of the hash function. The first line is our baseline, a simple password and a salt. The second line is the same password but without the salt. Finally, the third line is the same as the first except a single character, 'B' has been changed by one value, to 'C'. Both of these small changes cause significant changes the output value of the hash function.

Normally however, a password systems does not enforce a fixed length, but rather a minimum and a maximum length. The probability of a random guess for a password being correct, where the minimum password length is n and the maximum length is m, and the number of available characters is c is then:

$$P(correct\ guess) = \frac{1}{\sum_{i=n}^{m} c^i}$$

For instance a password of a length between 6 and 8, consisting of randomly chosen characters from a set of 95 printable characters yields a guessing probability of approximately $(6.7 * 10^{15})^{-1}$. To further increase the keyspace, password systems commonly use salts. The *salt* is a randomly chosen value which is prepended to the password, after which we refer to it as a *salted password*. Adding a salt of length s to a password of length n increases the keyspace to $c^{(s+n)}$, assuming that the salt uses the same character set as the password. This increase in keyspace makes offline guessing attacks, which we will describe further in the next section, significantly harder.

The use of salts generally changes the password storage such that instead of storing just the hashed password, it now stores the hashed salted password, along with the salt in plaintext. Similarly, the login procedure is slightly altered such that when a user logs in, the salt is prepended to the password he enters, and the result is hashed using the same hash function as was used when the password was set. Again, the authentication depends on whether the two values match. Table 2.1 shows examples of plaintext passwords, salts and their MD5 hash function outcome.

2.1.1.2 Offline Guessing Attacks

While the large keyspace of passwords offers good security in theory, experience shows that this does not always hold true in practice. The problem is that users typically choose passwords from a small and predictable subset of the keyspace [44, 34, 49]. These passwords are often variations of the username, names of pets, names of cartoon characters, common dictionary words [34], or names of fictional characters from literature or films. This allows a malicious attacker to use more efficient techniques to significantly reduce the time it takes to find passwords, since he can now focus on these categories of passwords rather than the entire keyspace. Such focused attacks are known as dictionary attacks, since the attacker often has dictionary files which contain common passwords.

If the password is random, the attacker can still avoid exploring the entire keyspace, for several reasons. First, if the attacker is trying to guess a password by brute force, i.e., trying every possible character combination, he will succeed after exploring half the keyspace on average. Secondly, if the attacker is only trying to find a single password, i.e., not a particular one, out of a list of passwords, the probability shifts further to his favor. For a password list for n_u users, where the passwords are distributed evenly within the keyspace, the attacker only has to explore the first $\frac{k_p}{n_u}$ segment of the keyspace, on average. Finally, if the attacker knows many personal details of the user, he can often find the password simply by guessing, e.g., if the password is the title of the users' favorite movie.

Finding a single password is often all the attacker needs to compromise a system, even if the compromised user account has few privileges on the system. Once the attacker is logged into the machine she can utilize other attacks such as exploiting vulnerable software, to elevate her privileges. This combined approach was for instance used in the infamous Morris worm [56].

A password guessing attack is typically performed offline, i.e., not on the machine which the attacker is trying to compromise. To perform an offline guessing attack, the attacker obtains a copy of the file, or the set of files, which contain the user information, hashed passwords and salt values. She can then run a so-called password cracker program on these files to obtain user passwords. There are several popular password crackers which are publicly available, such as John the Ripper [22] and Crack [41].

The above mentioned programs share common approaches. They generate a list of words each of which is appended to the salt, and the result is hashed using the same hashing function as the system which is being attacked. The word lists are found by creating various permutations of the user information found

in the password files, such as the username, the full name, department name or domain name of the host. Similar permutations are then run on each word from one or more dictionary files, such as an English dictionary or a dictionary of commonly used passwords. Finally, the remaining passwords are guessed by exhaustively trying all character combinations from the keyspace. Note that this exhaustive search does not have to be in an alphabetic sequence. John the Ripper for instance utilizes trigraph frequencies for each character position and length of the password, in order to find as many passwords as it can in a limited amount of time [22].

While the password keyspace can be quite large, password crackers can be very efficient at guessing passwords. For instance, Teracrack [45] used word lists generated from the Crack [41] utility, and managed to pre-compute hashes for over 50 million passwords in about 80 minutes. While they used a High Performance Computing environment, these numbers are not completely dismissible. Guessing passwords is a task that is very well suited for parallel computing, and many average user PCs can be utilized to find passwords very quickly [45]. It is widely known that many malicious attackers have access to so-called botnets [36], i.e., a network of infected user PCs that are at the attackers disposal, usually without the knowledge of the computers rightful owner. While these botnets are known to be used for sending SPAM and perform distributed denial-of-service (DDos) attacks, there is nothing that prevents these botnets to be used as a distributed password cracking mechanism.

2.1.1.3 Mitigating Password Attacks

Several approaches have been suggested in the literature to address the attacks described above and reduce the risk of compromised passwords. Most of them focus on increasing the active keyspace of the passwords, i.e., preventing users from using simple passwords such as dictionary words in favor of more random character sequences. One such approach is to check passwords at the time they are set by the user, and rejecting them if they do not fulfill the complexity [56]. This approach was for instance used by Bishop and Klein [18] where they combined it with messages which educated the user about password security by explaining why their password was rejected. Another approach to increase the complexity of passwords is to assign randomly generated passwords to users. This ensures that the passwords are properly distributed across the keyspace.

While these methods succeed in increasing the active keyspace and making offline guessing attacks harder, they are not perfect solutions. Increasing password complexity has the side-effect that users find it more difficult to memorize them, which in turn causes them to bypass the security measure, for instance by writ-

ing the password down on a piece of paper and stick it on the monitor, or to skip logging out of the system when they leave. This sort of behavior has for instance been observed in environments where employees are highly mobile and need frequent access to machines, i.e., frequently need to log in and out [16].

The use of passphrases has been suggested to address the problem of memorizing strong passwords [47]. Passphrases are essentially normal sentences, and the idea is to allow the users to create much longer passwords that are easy to remember and yet hard to guess. For instance, *"Mary had a little lamb"* is a 22 character passphrase that is easy to remember[1]. Recent research indicates however, that these passphrases are as hard to commit to memory as traditional stringent passwords and have a higher input error rate due to their length [31].

A completely different approach which does not involve password complexity, is to force users to change passwords periodically. In order for this method to work, the password expiration time has to be shorter than the time it takes for an attacker to guess the password. Similarly, the user cannot be allowed to change his password back to a previously used password, since an attacker may have obtained previous password files, in which case she has had plenty of time to crack them. This means that the password system has to include a list of previous password hashes and salts, to compare to the new password. This method has two obvious drawbacks. First, frequent changes cause similar memory problem as the complexity solutions [54]. Second, unless it is combined with complexity requirements, the time interval has to be too low to be practical due to the short time it takes to guess weak passwords. A weak password can be found in as little as a few seconds using a popular password guessing program, which is far to short to be a reasonable expiration time for a password.

Finally, some solutions aim at making the guessing process slower. For instance, Morris and Thompson replaced the encryption program used to create the password hashes in UNIX, with a slower program [40]. The slower program implemented the DES encryption algorithm, whereas the former had been an emulation of an hardware cipher machine. If the algorithm itself is slower, as opposed to just a particular implementation of it, this has the effect of slowing down the guessing since it takes a longer time to compute each hash. If, on the other hand, the lower performance is limited to a particular implementation of the algorithm, the attacker can use his own optimized version to obtain faster results. So for instance, inserting a delay into the encryption implementation on the machine we want to protect offers no additional defense against offline attacks. The idea of having faster hashing implementations in guessing programs is already in use. As an example, John the Ripper "has its own highly optimized

[1] Although it is a bad candidate since it is very well known, and is therefore a likely dictionary guess-phrase

modules for different hash types and processor architectures" [22]

2.1.1.4 Summary

In theory, passwords can provide great protection, given a large keyspace. Passwords that are easy to remember are also a comfortable authentication mechanism, at least for normal office environments. However, we have identified various weaknesses of passwords. Essentially, password policies have to strike a balance security and usability, i.e., between enforcing cryptic and secure passwords, and weaker passwords that its users can remember. Forcing strong passwords on users is likely to cause them to circumvent the security measures, which may render the authentication mechanism as little more than a false sense of security.

Further the weakness of password technology is not limited to guessing attacks, since passwords can be forgotten or shared. In the former case, an administrator has to reset the password and if this is a frequent event it can be quite costly. In the latter case there is no way for the authentication system to know that the password has been shared. The activity of sharing passwords is such a common office practice, that in a biometric user acceptance study, the subjects complained about that they were unable to transfer biometric characteristics, as they commonly do with passwords [14]. When a login system is presented with a username and the correct corresponding password, it cannot treat an impostor any differently than a legitimate user. Even worse is that the rightful owner of a compromised password typically has no way of noticing that his password has been compromised, and thus cannot report the breach and change his password.

O'Gorman states that *"a fundamental property of good authenticators is that they should not easily succumb to guessing attacks or exhaustive search attacks"* [44]. Due to a potentially large keyspace, it is clear that passwords fulfill this property in theory. Given the way in which normal users treat passwords however, it is clear that passwords do not fulfill this property in practice. Despite decades of research it is still unclear how we can provide secure passwords in a way that will not cause users to circumvent the technology due to poor usability.

2.1.2 Tokens

Authentication with a token is an example of *something you have*, also known as *Object-Based Authentication*. A token is a physical object which typically has some unique identifying properties, and ownership of such a token is normally

seen as sufficient proof of authenticity. If the token is unique for each person it can be used as a proof of identity, whereas if a token is unique for a group it can be used as a proof of membership of that group, e.g., ID-cards and membership cards.

Authentication tokens have been around for a long time, and have been actively used since long before the invention of the computer. For instance, wearing a sheriff star was commonly seen as sufficient proof of the wearers authority, i.e., the star was a well known group authenticator. Similarly, the practice of sealing letters with a token is as old as writing itself. An example of a sealing token is a signet ring, which is used to make an unique – and hard to forge – impression on the seal. Since the impression is unique to the token owner, a recipient can inspect the seal to determine if it is authentic, i.e., if it is truly from the claimed sender and not a forger. To some degree, the signet rings are similar to the private cryptographic keys we use for digital signatures today.

These tokens often have physical manifestations, but they can also be virtual (digital), e.g., digital certificates. The possession of a certificate allows the holder to perform operations which can be verified by others. Digital signatures are an example of this, where the sender can sign a message using a private key which corresponds to the public key in the certificate. The message recipient can then verify the signature using the certificate, given that she trusts the certificate authority, i.e., the issuer. If the senders private key is truly held private, forging his digital signature is as very hard.

Today we use physical keys to open doors and to start our cars. Similarly, we often use swipe cards or smart cards to enter different sections of our workplace. Smart cards in particular have found their way into computing systems and are sometimes used in authentication as a supplement to, or replacement of the traditional login system. A smart card is a small plastic cards which include a processor and memory. Such a card can be combined with password protection, so that it cannot be used without the correct password, which is equivalent to having a complete password protection system on the card itself. Once activated – by entering the right password – it can provide either a static passcode or generate a one-time passcode [44]. Since the user does not have to remember the passcodes provided by the card, they can be made long and random[2]. As a result the passcodes are more robust against guessing attacks since common dictionaries are no longer of any real use. This forces the attacker to use brute-force methods which are unlikely to produce a match within a practical time frame, due to the large keyspace of these passcodes.

[2]We will use the word random a bit liberally, since these computers generally cannot produce truly random numbers but merely pseudo-random

The main disadvantage of tokens is that they can be lost or stolen, and consequently found, or otherwise obtained, by an impostor. In that case, the impostor can use it to gain the same level of access to all the same places and systems as its rightful owner had, given that these systems solely rely on the token as an authenticator. In other words, token based systems authenticate users if presented with a correct token, regardless of whether it is carried by a rightful owner or an impostor. Moreover, the theft of a token is often an easier task than gaining access to the password files of a system, since it can be obtained by traditional pickpocketing. They do however have the advantage over passwords that if the token is lost or stolen, the owner sees evidence of this, i.e., that she no longer has the token, and can notify the appropriate administrators or authorities of the breach.

This advantage does not apply however, if the impostor creates a replica of the token. It is very possible that the attacker can acquire a token, forge it and return it before the token-holder notices it. The time it takes to forge a token naturally depends very much on the technology and design of the token. Magnetic stripe cards for instance, can be cloned in a few seconds with cheap consumer hardware. However, a smart card with well designed and properly implemented encryption mechanisms, may be sufficient to make card cloning a less attractive attack vector.

2.1.3 Biometrics

Human beings recognize other peoples faces, and we have used signatures for authentication for a long time. In recent times, these types of authentication methods have found their way into computer systems, and are called *biometrics*.

Biometrics is a set of methods to automatically identify a person based on their physiological or behavioral traits [28], such as fingerprints [27, 28], face recognition [27, 28, 42, 33, 19], keystroke dynamics [42], voice recognition [33], signature recognition, or speaker recognition [19]. This is normally done by comparing an input to a database of stored templates, e.g., comparing an image from a fingerprint scanner to a database of fingerprint images.

Biometric systems operate in either *identification* mode or *verification* mode [30]. In identification mode the goal is to identify the person, which is normally done by comparing a given sample to the entire template database, i.e., it is a one-to-many comparison. In verification mode we know who the sample owner claims to be, and need to verify that claim. In this case we only need to compare the sample to the templates stored for the claimed identity, i.e. a one-to-one comparison. The identity is typically claimed via a user name or a smart card

[30]. In terms of processing time, the one-to-one comparisons are much faster. In fact one-to-many comparison for large databases for some biometric traits can result in unacceptable execution times [27].

Biometric authentication is not an exact science. The final decision is generally based on a so called *match score*, which represent how well the input sample matches the stored template. A system which uses the biometric authentication system is configured with a certain match score *threshold*. A match score below the threshold means that the user is not authenticated, whereas a score equal to, or above it, means that the user is authenticated. In biometrics, a *false accept*, also called a *false match* is when the system mistakenly believes two samples from two different persons to be from the same person [30]. Contrary, a *false reject*, also called a *false non-match* is when the system mistakenly believes two samples from the same person to be from two different persons [30]. The false accept rate (FAR) and false reject rate (FRR) are both functions of the threshold and configuring the threshold can be seen as a trade-off between FAR and FRR [30]. A low threshold means that the system is more tolerant of noise and input variations, which increases FAR, while a high threshold means that the system is less tolerant and more secure but increases the FRR.

While biometric systems are in some sense the latest and most advanced authentication technology, they are not without flaws. Forging a biometric trait is not always as difficult as one might think. Matsumoto et al. [35] demonstrated that they can easily create artificial fingers with forged fingerprints, which are sufficient to fool fingerprint recognition systems, and Sandström [53] repeated the experiment in 2004, where she fooled several fingerprint systems at the CeBIT trade fair in Germany. For some systems, such an approach is unnecessarily complicated for the attacker. In the popular TV-show, Myth Busters [8], the hosts demonstrated how they could bypass a fingerprint system simply by presenting it with a paper printout of a valid fingerprint. That particular scanner claimed not just to use the thumb-print pattern, but also pulse, sweat and temperature, and was also claimed to have never been broken. Although this particular scenario involved a bad scanner which was likely configured with a very low match score threshold, it demonstrates that biometric systems cannot be treated as a perfect authentication solution.

There are other problems with biometrics. While they are, for most practical purposes, unique identifiers, they are not secrets [55]. We leave fingerprints on things we touch, and our eyes, hands etc. can all be observed. This is a real concern, especially with regards to fingerprints, since attempts to forge fingers from lifted fingerprints have generally been successful [35, 59]. Another related problem is that biometrics cannot be revoked as easily as passwords, and cryptographic keys. If a users thumbprint is compromised, it cannot be considered to be secure, ever again. Moreover, while it is generally advocated

that people use different passwords and keys for different applications, this does not translate well to biometric applications. If the user needs access to multiple applications which all require authentication via iris recognition, the user has no choice but to re-use his iris. The more such applications the user is enrolled in, the less secure the trait becomes, since only one of these systems needs to be exploited to gain access to the biometric information.

Finally, introducing biometric solutions can be challenging in terms of user acceptance. Users often consider them to be invasive, both in terms of effort, i.e., having to stare into a retina scanner, and in terms of privacy [14]. The privacy concerns include questions about which data is registered, how it is protected and who has access to it.

2.1.4 Attacks against Authentication Systems

We have discussed the advantages and shortcomings of various common authentication technologies. From this discussion it is clear that each type of system is susceptible to some attacks. Moreover, all of the above authentication systems fail if the attacker manages to compromise the authentication system itself, as opposed to just the authentication factor. For instance, if a biometric reader is tampered with in such a way that all decisions are reversed, i.e., that authentic users are not authenticated whereas others are, gives an attacker unrestricted access using his own fingerprint. If the device is not properly tamper resistant, this attack can be as simple as switching two wires.

In addition, authentication systems and other technological security solutions are generally of little use against Social Engineering attacks. These attacks involve using psychological tricks to manipulate legitimate users of the system to give the attacker access or confidential information [39].

2.2 Combining Authentication Systems

Each type of authentication systems has its own strengths and weaknesses. For instance, a password can be guessed while a physical token cannot, and similarly, a token can be counterfeited while it is normally hard to forge biometrics[3]. Since the strengths and weaknesses of these systems differ, it makes sense to try to combine multiple systems into a unified authentication scheme in such a

[3] although in some cases, such as with fingerprints, it can be really easy, as we have previously discussed

way that the strengths of one system complement the weaknesses of another. These schemes typically fall into one of two categories, namely *multi-factor* authentication and *multibiometric* systems. We will now give short descriptions of these.

2.2.1 Multi-factor Authentication Systems

In the beginning of this chapter we described the three factors of authentication, i.e., *Something we know*, *Something we have* and *Something we are*, also known as *Knowledge-based*, *Object-based* and *ID-based* authenticators respectively. Each of these factors is subject to different attacks as shown in Table 2.2 on the following page.

Table 2.2 provides a good overview of different authentication systems and common attacks against them. This allows us to take the weaknesses into account when choosing an authentication method for a system, since some of the drawbacks may be irrelevant or of little concern in a given application context or environment. Second, by clearly stating the strengths and weaknesses of each method we can combine methods in a complementary way that addresses known weaknesses of the individual authenticators and thus strengthen the overall system.

An example of this is a two-factor authentication where a smart card that contains large keys and passwords, is protected using a single password. Such system is considered to be more secure than either a smart card or a password implemented separately. In order to break such system, the attacker can either try to obtain the keys stored on the smart card, or the smart card itself. Obtaining the keys should be very hard without access to the card, and obtaining the card is of little use unless the attacker can get the accompanying password. Thus, to defeat the system the attacker has to steal the smart card, guess its password and launch his attack, before the theft is discovered and the system access restricted accordingly. Compared to a stand-alone password, the attacker now has to access a physical token and break its password within a short time frame, which is considerably more secure than just having to crack the password offline without any significant time constraints. Similarly, compared to a stand-alone smart card, it is no longer sufficient to obtain the card, since the attacker needs the password to be able to use it. In other words, the combined system offers better security than its individual components, but not better authentication.

The scenario described above is well recognized by many, since it has been used in the banking world for a while. In order to withdraw money from an ATM,

Attacks	Auth.	Examples	Typical Defenses
Client Attack	Password	Guessing, exhaustive search	Large entropy, limited attempts
	Token	Exhaustive search	Large entropy; limited attempts; theft of object requires presence
	Biometric	False match	Large entropy; limited attempts
Host Attack	Password	Plaintext theft, dictionary/exhaustive search	Hashing; large entropy; protection (by administrator password or encryption) of password database
	Token	Passcode theft	1-time passcode per session
	Biometric	Template theft	Capture device authentication
Eaves-Dropping, Theft and Copying	Password	"Shoulder surfing"	User diligence to keep secret; administrator diligence to quickly revoke compromised passwords; multi-factor authentication
	Token	Theft, counterfeiting, hardware	Multi-factor authentication; tamper resistant/evident hardware token
	Biometric	Copying (spoofing) biometric	Copy-detection at capture device and capture device authentication
Replay	Password	Replay stolen password response	Challenge-response protocol
	Token	Replay stolen passcode response	Challenge-response protocol; 1-time passcode per session
	Biometric	Replay stolen biometric template response	Copy-detection at capture device and capture device authentication via challenge-response protocol
Trojan Horse	Password, token, biometric	Installation of rogue client or capture device	Authentication of client or capture device; client or capture device within trusted security perimeter
Denial of Service	Password, token, biometric	Lockout by multiple failed authentications	Multi-factor with token

Table 2.2: This table shows different types of attacks, examples of how they are executed against different authentications (passwords,tokens and biometrics) and lists common defenses against these attacks. Source: [44]

.

we have to provide both our card and the accompanying PIN. However there are other combinations of factors that can be used. We can combine tokens and biometrics by storing our templates in a tamper resistant smart card. That way, the biometric system knows that the sample belongs to the legitimate card holder, given that the input sample matches the stored templates well enough. A common use of a token combined with a biometric is found in identification cards which contain a photo of the card holder, e.g., a drivers license.

Knowledge-based authenticators can be combined with biometrics, such as when a computer system requires the user to input both a password and a biometric sample. Finally, all three methods can be combined, such as when a smart card stores biometric samples that are encrypted using a key that is created from the users password. In this case a biometric authentication system cannot read the templates from the card unless the user enters the correct password.

It is worth mentioning that some multi-factor solutions introduce additional time-constraints to further secure the system. For instance RSA SecurID [10] provides a physical token that generates one-time passwords that are only valid for 60 seconds. This reduces the chance of an attack where a previously generated key is used to gain access, i.e., when an attacker gains temporary access to the token to generate a password, or a sequence of passwords, which she can write down or memorize for later use.

While multi-factor systems can increase security, they also decrease user convenience. All combinations that include a token factor require the user to carry the token, and in a company which relies on tokens, forgetting the token at home may prevent an employee from doing his job, until he has retrieved the token. Similarly, systems with a knowledge-based factor require the user to memorize a password, and if the password complexity policy is strict, it might increase the number of password resets performed by the organizations technical support. Finally, biometric factors normally require the user to provide a biometric sample which can be very inconvenient, e.g., staring into a retina scanner for a few seconds. In other word, each factor comes with some inconvenience, and combining factors also combines the inconveniences of each factor, e.g., a password protected token combines the inconvenience of having to carry the token and having to remember the password. Therefore, multi-factor authentication systems are typically less convenient than a single-factor authentication [44].

We must take this inconvenience and other usability factors into account when we design security infrastructures, since a too inconvenient authentication mechanism provides poor usability, which can cause its users to revolt and find ways to circumvent it. This type of user behavior has for instance been observed in hospital environments, where the required logins are too frequent, take too long and cause other inconveniences [16]. In the study, the hospital workers were ob-

served bypassing the security of an electronic patient record (EPR) system, for instance by creating universal account which was shared with all the workers, and for which the username and password was written on monitors throughout the ward. Clearly, the strict policies and access control mechanisms dictating which employee can read which patient record, were scrapped in favor of an environment where people can carry out their work without constant interruptions in the name of security. If the security mechanisms are designed such that they are easy to use and do not hinder the users from doing their job, the users are less likely to seek ways to bypass the security measures, and therefore we can obtain a more secure system.

2.2.2 Multi-biometric systems

Multi-biometric systems, as the name suggests, are systems which combine multiple biometrics to make a unified authentication decision. Ross et al. provide a very good overview of the field in their book, Handbook of Multibiometrics [52]. This section is largely based on material found in that book, which we summarize here for the sake of completeness.

The benefits of combining multiple biometric systems are numerous. The unified decision can offer a significant improvement in accuracy and can achieve reduced FAR and FRR simultaneously. Another benefit is that the more biometric traits we request the harder it is to spoof them, especially if we use a challenge-response protocol where we request a random subset of the traits. Multibiometrics also reduce the problem of noisy input data, such as from a sweaty finger or a drooping eyelids, since if one input is very noisy, the other biometric systems might still have samples of sufficient quality to make a reliable decision. This can also be seen as *fault tolerance*, i.e., if one system breaks down or is compromised, the others might suffice to keep the authentication system running and producing accurate results. There are many ways in which biometrics systems can be combined, and we will now discuss some of them briefly.

A typical biometric system reads a biometric input sample from the user, extracts features that describe the sample and compares them to a set of templates to produce a match score. The match score indicates how well the extracted features from the sample match a given template, and it is compared to a threshold to determine if the authentication succeeded. If the match score is below the threshold the authentication fails, but succeeds otherwise. These processing steps indicate that we combine biometric systems at different levels of abstraction.

The lowest level of abstraction is to combine the raw data from the sensors, and is only possible for samples of the same biometric trait, e.g., it can be used to combine multiple samples of the same finger, but not to combine fingerprint and a retina recognition systems. The next level is to combine feature vectors that are extracted from the input sample. A feature vector contains a simplified description of a biometric sample. If the samples are of the same type, e.g., two samples of the right thumb, the features can be combined into a single more reliable feature vector. If on the other hand the samples are of different types, e.g., fingerprint and face recognition photo, the feature vectors can be concatenated into a more descriptive feature vector. The next level of abstraction is combining at the match score level, where each system calculates their match score independently, and the scores are then combined into a single score using some mathematical algorithm. Another method at this level of abstraction, is to match at the rank level, where biometric systems return a list of top n candidates, i.e., an ordered list of n elements that best match the input sample. Rank level fusion is concerned with combining such lists from different systems to produce a reliable overall result, and is generally only applicable to identifications. Finally we have matching at the decision level, where each system has its own threshold and delivers only their final decision. The fusion then consists of merging these decisions into a single decision, such as by majority voting or boolean AND/OR rules. Of these methods, match score fusion is the most commonly used since match scores are generally easy to access and there are numerous methods of combining them, some of which are very easy to implement. Moreover, match scores offer rich information about the input, second only to feature vectors. They do however, suffer from the fact that some commercial biometric systems only provide access to the final authentication decision.

To be able to combine biometric data we need to decide how to obtain it, i.e., what data sources to use. We will use the term *biometric data* for any level of abstraction, i.e., it can mean a feature vector, a match score or a decision. There are several methods for obtaining biometric data from multiple sources. The first one is *multi-sensor systems*, which create multiple images of the same biometric trait, where each image is obtained by a different sensor. For instance, face recognition images from a thermal infrared camera and a visible light camera. Another method is to process the same data with *multi-algorithm systems*, i.e., each algorithm produces independent results which are then used in the unification. *Multi-instance* systems are concerned with using multiple instances of the same biometric trait, e.g., the left eye and the right eye. *Multi-sample systems* read multiple samples of the same trait in order to either decrease the effect of input variance, or to construct a better representation of the trait. *Multimodal systems* combine biometric data from different traits or systems, e.g., the results from a fingerprint recognition system and a speaker recognition system, and combining uncorrelated traits, such as fingerprints and voice, is expected to give better performance then correlated traits, e.g., voice and speaker

recognition. Finally, we have *hybrid systems* which combine two or more of the methods described above, e.g., a system which uses multi-sensor fingerprint system combined with a face recognition system, which effectively makes it a multimodal system.

Once we have gathered the data from the various sources, we must decide in what order we will process it. It can be beneficial to process the data in a sequence, for instance if we want one system to narrow down the choices to a limited number of candidates, which a second system can then verify. If the former system scales very well but has a high FAR, while the second system is slow but with a low FAR, this approach can offer high accuracy with acceptable performance. If however, we just want to combine the match scores of several different systems, we should aim for a parallel input acquisition. Figures 2.1 and 2.2 show a parallel system and a sequential system respectively.

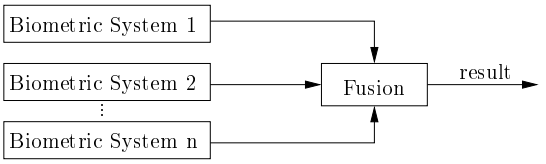

Figure 2.1: A parallel processing of biometric input.

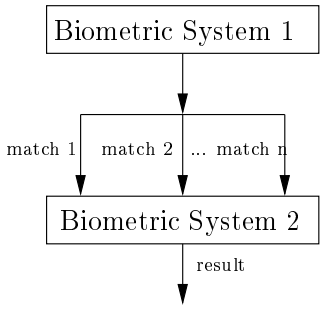

Figure 2.2: A cascading processing of biometric input.

The combination of biometrics has been shown to increase reliability, i.e., where the combined system provides more reliable results than the participating systems individually. For instance, Hong et al. [27] showed this with a cascading system that uses a face recognition system to identify the top n users, which are then further verified by a fingerprint scanner. They showed that the integrated system provided lower false rejection rates, compared to the individual systems, for several different FAR values. For instance, for a false acceptance rate of 1%, the face recognition system had a FRR of 15.8%, the fingerprint scanner had a FRR of 3.9%, while the integrated system only had a FRR of 1.8%.

2.2.3 Attacks against combined systems

Multi-factor and multibiometric authentication systems offer better protections against some attacks. We must however remember the principle of easiest penetration, which we mentioned from definition 2.2. In particular, the combination of authentication mechanisms is of little use if the decision point is weak. For instance, if we have a parallel multibiometric system such as the one shown in Figure 2.1, then it certainly requires more effort to attack every participating biometric system, than if there was only a single biometric system. If however, the fusion system can be tampered with, it can be made to give a positive result for the attacker, regardless of the individual results of the biometric systems. In other words, the fusion system becomes a single-point of failure. Therefore great care must be taken to secure the final decision points against tampering. This is essentially the same problem as with traditional authentication systems, as described in section 2.1.4.

In multibiometric systems there are other points of entry for the attacker, in particular the connections between the biometric systems and the decision/fusion system. Regardless of whether they run on the same machine or over a network, an attacker can intercept the connections and send forged data to the fusion point to make it look like the biometric systems identified an authentic user with high levels confidence. If the systems use cryptographic techniques to prevent this, their evaluation should pay close attention to traditional *man-in-the-middle* attacks as well as replay attacks.

Finally, combining multiple systems does not protect against social engineering attacks where an authentic user is manipulated into providing access to an impostor.

2.3 Access Control

We have shown various aspects of authentication and how different methods can be used to produce authentication results. But so far we have left out all discussion about why we authenticate users. The authentication results are useless unless some system relies on good authentication, such as a logging system or an access control mechanism. The most common receiver of authentication events and results are access control systems. As the name indicates, access control systems control access to physical or logical resources, such as printers, files on a computer system, and rooms of a building. In terms of computer security in particular, *"its function is to control which principals (persons, pro-*

cesses, machines, ...) have access to which resources in the system–which files they can read, which programs they can execute, how they share data with other principles, and so on" [15].

Access control systems normally use traditional authentication systems such as passwords or Kerberos [6] and once a principal has been successfully authenticated, the access control system will not question the principles identity further. In other words, access control systems rely on *binary* authentication results, i.e., either the person is who he claims he is, or he is not. This is why probabilistic systems–such as biometric systems–use thresholds to produce a binary result.

Introducing probabilities and thresholds into access control policies can however increase their flexibility. Imagine for instance an Access Control List (ACL) for a physical building where the CEO of the organization is about to enter a room. If the room he is about to enter is the cafeteria, there is hardly a need for high authentication accuracy. If on the other hand he is entering a file storage room, where the organization stores highly confidential data, the need for high accuracy is most likely much higher. In scenarios such as this one, it is beneficial for the access control system to receive probabilistic authentication events, and specify resource thresholds in its policy.

Since this thesis focuses on authentication, we will not discuss access control in further detail. It suffices to emphasize that access control systems rely heavily on the security of their respective authentication mechanisms. While the framework we present in this thesis focuses on probabilistic authentication, it needs to allow for the specification of a threshold in the framework policy, to be compatible with legacy access control systems.

2.4 The State of the Art

We discussed the technical problems of passwords in some detail in Chapter 2.1.1, concluding that there are many unresolved issues with the use of passwords. To make matters worse, an increasing number of websites are requiring their users to register an account, in order to get full access to the services. This means that a typical Internet user may be juggling as much as dozens of user accounts, all of which require a password. Since it is hard enough to remember a single password of any complexity, users tend to re-use usernames and passwords on these websites, whenever possible. While this is acceptable for some applications, i.e., those in which very limited harm will come to us if the account is compromised, it is a more serious issue if the same password is also used in more critical systems. For instance, it is certainly hazardous to use

the same password at work and on publicly available social networking sites[4], since if any successful compromise of these sites puts the company data at risk. In the case of social networking sites in particular, the risk can be even higher, since users tend to share a lot of personal information on these websites, which can include where they work. Not only will this give an attacker a password, but a likely target where the password can be used.

One solution to this problem is to use so called password vaults, which are programs that store, and sometimes generate, passwords for different applications. The passwords are encrypted using a key that is based on a single master password. This allows us to create random passwords for each account we have to register, and yet only have to remember one password, namely that of the vault. One such application for the Microsoft Windows family of operating systems is *Password Safe* [9]. Since the most common use of multiple accounts these days is for websites, another alternative is to extend web browsers with a built-in password vault. Such extensions allow users to "recall" site-specific passwords in a user-friendly way, without leaving their browser environments. An examples of such extension is the *Magic Password Generator* [7] for the Mozilla Firefox [1] browser. Password vaults allow us to have unique and strong passwords for each of our accounts, while only requiring us to remember a single password. It is of course strongly recommended to have a strong master password, since guessing it gives an attacker access to all the other accounts.

For large-scale secure authentication frameworks, some companies are offering solution suites that provide a centrally managed one-time password authentication systems which are used to secure PCs, wireless or virtual networks, specific applications, and so forth. One such system we briefly looked at[5], is the RSA SecurID [10] solution we mentioned in Chapter 2.1.2. It provides one-time password solutions, where the password is generated with either a special physical token, or one can generate it with special software which can for instance run on mobile phones and handheld PCs. Once the passcode is generated, it is only valid for 60 seconds, which prevents the attacker from gaining temporary access to the token to generate multiple passcodes and write them down. The solution is interoperable with many of today's popular network management solutions, applications, and operating systems including Microsoft Windows and Unix. Moreover, it provides an API so that it can be integrated into custom applications. This solutions is currently being used by many banks, governments, and other organizations.

Biometrics can be found in a variety of systems, ranging from physical security

[4] this is not a completely random example, social networking sites have a history of exposing user accounts

[5] that is, we browsed through documents on their website, we did not have access to an actual system to test it ourselves

systems to consumer laptops. Many laptops ship with a fingerprint scanner that can be used as a replacement for a password when logging in to the machine. Biometrics are however, not just used to protect data, but are also in other very different applications such as border-control [2, 12] and general safety applications. One such safety application is their implementation in smart-guns [60]. Biometric smart-guns are firearms that can only be fired by their rightful owner, which prevents criminals from using weapons belonging to disarmed law-enforcement officers, and prevents children from accidentally firing their parents guns. Biometrics are also being used to reduce street violence by requiring people to use a biometric system when entering night-clubs that serve alcoholic beverages [38]. Known trouble makers are flagged by the system and not allowed to enter, which seems to have contributed to a decrease in night-time violence.

2.5 Summary

Authentication systems come in many different types, all of which have some advantages and disadvantages compared to the others, in terms of security, robustness, ease-of-use and user acceptance. All these claims of security are however based on a few assumptions:

First, the authentication systems must be properly implemented, e.g., a password solution which lets an impostor into our system simply by providing a wrong username and password followed by a carriage return [15] offers little protection.

Second, the decision point, i.e., the point which delivers the result to the access control system, must be protected against attacks. If the attacker can manipulate the software process of the decision point, or tamper with its hardware, she can bypass all the security provided by the authentication systems which the decision point uses.

Third, none of the technology solutions described above help against a skilled social engineering attack. We must authenticate our legitimate users, but an authentication system cannot detect the users intent. So while authentication systems play a very important role in the security infrastructure of computer systems, they do have flaws which must be taken into account when creating a security infrastructure. Moreover, they shall generally be complemented with other means, such as policy enforcement, user education and monitoring.

There are multiple ways of combining authentication systems, and such com-

binations generally offer either better security, more reliable authentications or both. We have discussed how multi-factor authentication, i.e., the combination of knowledge-based, object-based, and ID-based authenticators, has been shown to improve security. The example of a password protected smart card clearly shows that it is harder to attack the system, compared to either guessing the password, or obtaining the card. Moreover, we have discussed how multibiometric systems have been shown to have lower error rates than the system the individual participating systems. Traditionally, the false acceptance rates and false rejection rates are balanced by the threshold value, and decreasing one will increase the other. When multiple biometric systems are combined however, it is possible to decrease one rate without worsening the other, which is a significant improvement in authentication quality.

Both multi-factor systems and multibiometric systems however, are somewhat inflexible. Multi-factor systems are normally limited to combining two or three factors, i.e., they generally only contain one instance of each factor, such as a fingerprint recognition and a smart card, rather than say, the same components combined with a second biometric and a password. Moreover, multi-factor solutions generally require that all the factors be fulfilled, i.e., it is not sufficient to have a smart card without the accompanying password, or vice versa. Similarly, multibiometric systems are limited to biometric systems, i.e. passwords and tokens do not fit into the multibiometric scheme. Further, multi-factor and multibiometric solutions are generally custom written rather than being built on a common foundation, such as an authentication framework. As a result, if we want to add a new authentication system to the mix, we are often forced to change the code of the combination system and protected system, or if the code is not available, turn to proprietary vendors for support for the new system.

Chapter 3

Co-Authentication

In the previous chapter we described some of the most common authenticators and authentication systems that are in use today. We categorized them into three factors, i.e., *something we know*, *something we have*, and *something we are* and showed that each of these categories is subject to some attacks and that combining multiple factors can increase the overall security of a system. Moreover we discussed how previous work in multibiometrics has shown that combining multiple biometric systems can increase the overall authentication accuracy by reducing error rates. That is, with a single biometric system the FAR's and FRR's are a function of the threshold, and decreasing one increases the other. However by combining multiple system we can reduce the overall error rates, e.g., reducing the FRR while FAR stays unchanged.

By reviewing the success of multibiometric systems, the question arises whether this approach cannot be generalized to other traditional authenticators, i.e., *knowledge-based* and *object-based* authenticators. Although this is done to some degree with multi-factor systems, such systems are usually limited to two or three factors, where you must fulfill every part completely, e.g., you must provide the token and the correct password and failing to provide both of them results in overall failure to authenticate. We want to extend this concept such that multiple authentication systems of different factors can be combined in a generic way, where each fulfilled part of it adds to the strength of the authentication, and that only fulfilling a subset of the participating authentication systems might

be sufficient in some cases.

The method of combining biometric systems, is often called *fusion*, and as we discussed in the previous chapter, the fusion can be performed on four levels, the feature extraction level, the rank level, the match score level, and at the decision level [51, 52]. The latter two can be generalized to all authentication systems. Fusing at the decision level can be trivially generalized, since all traditional authentication systems return their decision, i.e., whether the authentication succeeded. We propose to extend this generalization to include fusion at the match score level. While binary authentication systems typically lack match scores, they have varying degrees of strength and error rates which we can include as a probability score, or a *confidence level*. The score can either be static, e.g., all smart cards will have a confidence level of 0.72, or they can be dynamic, such as having different confidence levels for password authentication depending on the strength of the password. By employing varying confidence levels, we can generalize the fundamental principles of multibiometric verification to the general authentication problem.

We have defined Co-Authentication as the generalized fusion of authentication factors, i.e., when multiple authentication systems contribute to a unified authentication decision. The unified Co-Authentication decision can be reached either with decision level fusion or score level fusion. In the field of biometrics, score combination of multiple biometrics has been shown to increase accuracy, i.e., decrease false negatives and false positives [28]. We show however, that the benefits of information fusion can also provide added flexibility in the design of systems that include binary authenticators which can be adapted to a threshold-based scheme.

3.1 Use Cases

In the following sections we present two examples of how two existing binary authentication systems can be adapted to threshold-based systems and benefit from Co-Authentication.

3.1.1 Example: Credit card Payment Systems

Due to the rise of fraud, credit card issuers are moving away from magnetic stripe technology towards smart cards [26] and many credit cards are equipped with both an integrated chip and a magnetic stripe. While the chip is considered

to be more secure the magnetic stripe is included to give retailers more time to implement Chip & PIN technology. This indicates that the chip and the magnetic stripe have different confidence levels, yet they will be treated equally during the implementation period, after which the retailers will face the liability of frauds when the magnetic stripe is used [26].

The technology embedded in the physical card is only one of the security measures employed by the credit card companies. Another measure is to develop a profile on its customers, so that each withdrawal is compared to the profile to detect anomalies and possible fraud. For instance, if a customer frequently makes purchases ranging from $25 to $300 within a small geographic area, a $600 transaction from a location outside the normal area, is bound to raise suspicions.

By defining dynamic confidence levels for a transaction depending on how well it fits the cardholder's profile, the confidence level can be combined with the confidence level of the card technology, i.e., magnetic stripe or chip, in a Co-Authentication scheme which can decrease the risk of fraudulent transactions. This can either be a single confidence level for the overall fit of the profile, or multiple scores, each for a subset of the transaction such as location, amount etc. that are individually compared to the profile. By combining the confidence level of the chip or magnetic stripe and the confidence level for the fit to the profile, we obtain an authentication system that is more restrictive towards magnetic stripe purchases, i.e., purchases made using the magnetic stripe have to fit the profile better than purchases made using the chip. This can for instance allow small corner-stores to avoid expensive technology upgrades, at the expense of being limited to small amount transactions which fit the cardholders profile.

We will give a more detailed description on how this scheme can be implemented using our framework in Chapter 7.2.1.

3.1.2 Ranking Passwords

Password authentication is the de-facto standard when it comes to user authentication and it is a prime example of binary authentication systems with implicit weaknesses. In Chapter 2.1.1 we showed how passwords vary in terms of strength. Despite the difference in terms of how well they fare against guessing attacks, a password system treats all passwords as equals, assuming that they are accepted in the first place. In other words, systems typically do not distinguish between weak and strong passwords except for some rudimentary checks when the password is initially set, e.g., that it is longer than 4 characters and contains at least one uppercase letter and one digit. Such checks are gen-

erally insufficient to provide strong password security, since they can be passed using trivial passwords such as *'Abcd1'*. Such trivial passwords will however be treated the same way as other passwords on the same system, regardless of whether or not the other passwords are significantly harder to guess. We believe that keeping the varying degree of password strength implicit should be discouraged, since it gives a false sense of security.

Passwords can be ranked by estimating how hard it is for common password crackers to guess them. These estimates can be made when the password is set, and stored as a strength score which can be used as a static confidence level when the password is entered correctly. This allows passwords to be combined with other authentication mechanisms in a Co-Authentication scheme, where weaker passwords will make stricter demands on the other participating systems than a strong password does. For example, if such a password system is combined with a fingerprint reader, then a weak password requires a better fingerprint match score than a strong password does. We are currently working on such an estimation program and our preliminary work in this area is presented in Appendix A.

3.1.3 What is gained?

In the two examples above, traditional systems where transformed into threshold based systems. In the case of password ranking, the adaption to confidence scores allows us to make more informed decisions, and provides added flexibility when combined with other systems in a Co-Authentication infrastructure. That is, we are moving away from binary absolutes to probabilistic estimates, which allow us treat the same password differently depending on context.

The benefit of Co-Authentication is even more clear in the credit card scenario. The credit card issuers clearly indicate that they believe that the risk of fraud is higher for magnetic stripe cards than for chip and PIN cards, and yet they will treat them identically for the next few years. Co-Authentication provides them with the flexibility to treat the two technologies differently. The better a transaction fits the cardholders profile, the less likely it is to be fraudulent. It is therefore clearly a good idea, in our opinion, to enforce stricter profile matching for magnetic stripe cards, rather than treat all transactions equally.

The two scenarios above show that using non-biometric threshold-based systems in a Co-Authentication can provide flexibility that is unavailable when we use traditional binary authentication mechanisms. Moreover, in both scenarios, the adaption to a threshold based system adds no inconvenience for the user, neither in enrollment, nor in daily use.

3.2 Fusion

When designing a Co-Authentication system, we have to decide at which of the four levels we fuse data from the participating systems. If all the participating systems are biometric systems, it is possible to fuse at the feature extraction level. However this method is cumbersome and tightly coupled with the particular systems that are being fused together. In addition, it cannot be extended to include non-biometric systems at a later point. Therefore we should avoid this level of fusion unless it provides significant performance benefits, and if we are certain that non-biometric systems or incompatible biometric systems will not be added to the combination later on.

Fusing at the rank level is only an option in identification scenarios, i.e., where we are combining the input sample with the entire user database to find the best matches. In that case, each system may return a list of the top n matches, i.e., an ordered list where the best matching candidate is on top. While this approach allows for advanced algorithms to decide the correct identity, identification can also be performed with the two remaining options, i.e., fusing at the match score level or at the decision level. To do this, only the single best ranking candidate from each system is considered. If all the participating systems agree, i.e., they all have the same individual as their highest rank candidate, the match score can be computed in the same way as it is done in the verification process. If they disagree, however, we need some method of choosing the most likely candidate and compute the corresponding match score.

Fusion at the score level involves combining the match scores from the individual authentication systems to obtain a single more reliable score, for instance by producing a weighed average. This method can be generalized to all systems which can produce a match score, i.e., all threshold based authentication systems. Finally, fusing at the decision levels means that each participating system reaches an independent decision (i.e., authenticate or not authenticate), and the decisions are then combined to reach an overall decision. A common approach to decision level fusion is to combine all the individual decisions with boolean OR or AND operators. That is, if we have the decisions from three systems (S_1, S_2, S_3) which yield the decisions (d_1, d_2, d_3), the result of an OR fusion is the result of $(d_1 \vee d_2 \vee d_3)$. Similarly we call it AND-fusion when the AND operator is used for fusing the individual decisions, i.e., the decision is the result of $(d_1 \wedge d_2 \wedge d_3)$.

We have chosen to use score level fusion rather than decision level fusion for several reasons. First, decision level fusion is very inflexible. A majority voting scheme with few participating systems is a good example of a target where the attacker uses the principle of easiest penetration. For instance, if there are only

three systems in a majority voting scheme, we can expect the attacker to focus on the weakest two systems. If there are many participating systems however, the majority voting scheme can be a viable option. It does however only give binary results, and systems that rely on its services cannot distinguish between a majority vote of 51% and a vote of 100%.

AND-fusion requires all the system to agree on that the principle is authentic, whereas OR-fusion means that only one system has to believe in the principle's authenticity. In biometric systems, this corresponds to choosing to increase the false accept rate (FAR) or false reject rate (FRR) on the cost of the other, i.e., tipping the scale. Choosing OR-fusions will increase the FAR but lower the FRR, and AND-fusion will increase the FRR and lower the FAR.

More formally we state that: let $F_A(S)$ denote the false accept rate of a system S. We can then show that the FAR of an OR-fusion for a combined system will always be higher than the FAR of any individual participating system. In the case of combining two systems, S_1 and S_2 this can be formulated as:

Theorem 3.1 $F_A(S_1 \vee S_2) \geq max(F_A(S_1), F_A(S_2))$

Proof. Assume that a single system, say S_1, has a higher false acceptance rate than the OR-fusion of S_1 and S_2:

$$F_A(S_1) > F_A(S_1 \vee S_2) \tag{3.1}$$

We prove this by contradiction as follows:

By expanding the right hand side we obtain:

$$F_A(S_1) > F_A(S_1) + F_A(S_2) - (F_A(S_1) * F_A(S_2)) \tag{3.2}$$

which is equivalent to

$$0 > F_A(S_2) - (F_A(S_1) * F_A(S_2)) \tag{3.3}$$

which is further equivalent to

$$F_A(S_1) * F_A(S_2) > F_A(S_2) \tag{3.4}$$

By dividing each side by $F_A(S_2)$ we obtain

$$F_A(S_1) > 1 \tag{3.5}$$

For this to hold true the false acceptance rate of S_1 has to be higher than one, indicating that over 100% of its results are false accepts. This is clearly impossible.

The increased FRR of AND-fusion has a similar proof: let $F_R(S)$ denote the false rejection rate of a system S. We can then show that the FRR of an AND-fusion for such a system will always be higher than the FRR of any individual participating system. In the case of combining two systems, S_1 and S_2 this can be formulated as:

Theorem 3.2 $F_R(S_1 \wedge S_2) \geq max(F_R(S_1), F_R(S_2))$

Proof. Assume that a single system, say S_1, has a higher false rejection rate than the AND-fusion of S_1 and S_2:

$$F_R(S_1) > F_R(S_1 \wedge S_2) \tag{3.6}$$

We prove this by contradiction as follows:

By expanding the right hand side we obtain:

$$F_R(S_1) > F_R(S_1) + F_R(S_2) - (F_R(S_1) * F_R(S_2)) \tag{3.7}$$

which is equivalent to

$$0 > F_R(S_2) - (F_R(S_1) * F_R(S_2)) \tag{3.8}$$

which is further equivalent to

$$F_R(S_1) * F_R(S_2) > F_R(S_2) \tag{3.9}$$

By dividing each side by $F_R(S_2)$ we obtain

$$F_A(S_1) > 1 \tag{3.10}$$

Again, for this to hold true the false rejection rate of S_1 has to be higher than one, indicating that over 100% of its results are false accepts, which again, is clearly impossible.

Theorems 3.1 and 3.2 and their proofs can be extended to any number of systems according to the inclusion-exclusion principle. This suggests that the more systems we add to decision-fusion for AND or OR, the more we tip the FAR/FRR scale. The reasons that the error rates of the combined system will be higher or equal, rather than strictly equal, to the participant P_{max} with the highest error rate, is that the errors of the other participating systems are not necessarily a subset of the error rates of P_{max}. That is, P_{max} might correctly authenticate a legitimate user, whom another system fails to authenticate, and thus falsely rejects.

On the other hand, by combining scores into a single overall score, we can adjust the balance between FAR's and FRR's by adjusting the threshold for the overall score. Match score fusion also allows us to use many mathematical algorithms, in particular from the field of statistics. Which algorithms are best suited in given conditions is still an active research topic in the multimodal biometric field, and will hopefully be extended to cover the more general Co-Authentication.

3.3 The Benefits of a Generic Framework

There are two types of systems which participate in an authentication scenario, namely authentication systems (AS) and protected systems (PS). An authentication system provides authentication services such as fingerprint recognition or smart card verification. The protected systems on the other hand are typically access control systems, but can be any system which uses authentication information.

Each PS utilizes one or more AS to get the confirmed identity of the present user, and an authentication system can serve more than one PS instance. For example, a fingerprint scanner used to control a door lock might report its authentication events to the door lock, a logging system and a monitoring system used by physical guards. This means that we have a many-to-many connections between AS and PS instances, which are either run on a single machine or distributed over a network.

Large security infrastructures are likely to consist of many heterogeneous AS and PS instances, some of which may be proprietary. Moreover, they may be run on different operating systems, be written in different programming languages, and provide unique and proprietary APIs. To make Co-Authentication a realistic option for such organizations, it has to be easy to integrate these systems. Preferably, it should be done in a generic way such that each product only needs to be integrated once, e.g., where the vendor supplies the integration code for its product. Therefore we propose a generic framework which manages all the AS-PS combinations and supports re-usable modules that integrate authentication systems or protected systems with the framework. For instance, a module for Kerberos [6] can be distributed and used by multiple organizations which have Kerberos as a part of their infrastructure, i.e., there is no need for each of those organization to write the integration code.

3.4 Summary

Co-Authentication is the concept of combining multiple authentication systems into a system which makes unified decisions by combining the probabilistic scores of the participating systems. Common binary authentication systems can be adapted to this scheme, and combined with other threshold based systems. We strongly believe that implicit weaknesses are bad for security, and by making them explicit as probabilistic outcomes, we can achieve higher reliability and gain greater confidence in our authentication infrastructure. For the remainder of this thesis we will present a Co-Authentication framework, called Jury, which is a proof-of-concept demonstration of the benefits of Co-Authentication.

Chapter 4

Requirement Analysis

The main objective of this thesis is to create a proof-of-concept of a generic Co-Authentication framework, i.e., the framework is not tailored to the needs of any specific organization or a particular situation. This means that we are not constrained by any organization-specific or domain-specific requirements. Nevertheless it is helpful to list the basic functional requirements of the system and to create a set of goals which we want to achieve. Such documentation helps clarifying what it is that we are doing, and set constraints on the project scope, which in turn motivate design decisions and implementation choices. Without these constraints, it is easy to expand the project in all conceivable directions, loose track of time and miss the delivery deadline. In other words, we want to avoid common pitfalls of software projects. In this chapter we outline the design guidelines for our architecture, and the requirements we want address in our framework, which we have dubbed *Jury*.

The role of the Jury framework is to provide an easy integration of arbitrarily many threshold-based authentication systems. The main benefit of Co-Authentication is to be able to make better informed decisions based on probabilistic estimates of each participating authentication system. Jury is responsible for gathering these probabilities, combining them into a unified score, and inform protected systems about the result. Optionally, the framework can have a threshold defined in its policy, so that it can provide binary authentication results to legacy systems which do not support probabilistic authentication.

There are several methods that can be used to obtain, and combine authentication scores. The first method is when an identification mechanism provides an identity which other authentication systems verify, resulting in an overall score based on the combination of verification scores. The second method is when, given an identity, the framework requests other authentication system to identify the present user. An algorithm then sorts out the identification results and computes the most probable identity along with its score. The third and final method is when the identification mechanism provides a list of the top n most probable identities along with their scores. The score-combination algorithm can then either use a combination of the previous two approaches, or a specific algorithm that takes several ranked lists and computes the top ranking candidate along with its score.

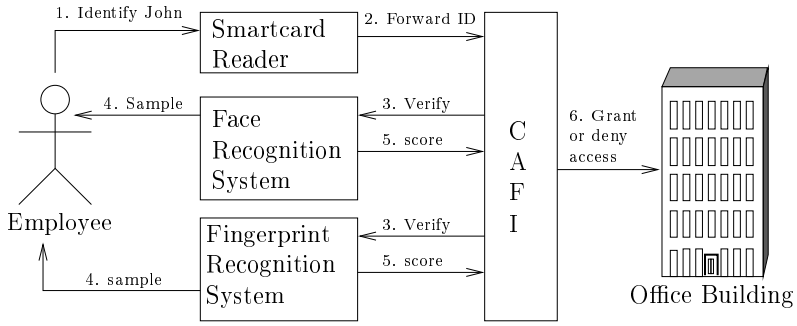

Figure 4.1: An example scenario where an employee requests access to an office building. Steps with identical sequence number can happen in any internal order. For instance, it does not matter whether the fingerprint recognition system sends its score to the Co-Authentication Framework Instance (CAFI), before or after the face recognition system does so. For brevity, the 6th step involves an access control mechanism that is not shown, i.e., the CAFI provides an authenticated identity to the AC, which grants or denies access to the door.

Figure 4.1 illustrates an example scenario. John is an employee of *Confidential Inc.* and is just about to enter the main office building of the company. When he arrives at the front door he swipes his employee smart card. The smart card reader identifies the employee (step 1) and sends the identification data to the Co-Authentication system (step 2). The Co-Authentication Framework Instance (CAFI), which is the central node of the framework, forwards this ID to a fingerprint recognition system and a face recognition system for verification (step 3). The two biometric systems then proceed to prompt for samples by scanning Johns thumb and taking his picture (step 4). They then individually compare the received samples to their templates for John and produce a match score which they forward to the CAFI system (step 5). Finally, the CAFI computes an overall score and compares it to the configured threshold for the front door.

If the combined score exceeds the threshold, John has been authenticated and as a result he is allowed entrance to the building and can now open the door. Otherwise he is not authenticated which results in him being denied entrance to the building.

4.1 Overview

A Co-Authentication system can be divided into four parts. First there are the protected systems, i.e. the systems which request services of the frameworks and subscribe to certain authentication events. Second there are a number of authentication systems, such as password protection, face recognition systems etc. which provide authentication services on request from the framework, as well as publish authentication events. Third is the Co-Authentication core system itself, and finally there are the wrappers for authentication systems and protected systems, which act as middleware between the Co-Authentication core system and the remote authentication systems or protected systems.

Figure 4.2 shows a trivial setup of a Co-Authentication system, where we have a system that we want to protect with our framework. The Co-Authentication node (CAFI), is the main component of our system and is responsible for gathering and processing data from other nodes, as well as producing a result for the protected system. And finally, we have authentication systems which authenticate users and report their scores to the framework, which in turn informs the protected system.

4.2 Terminology

For the remainder of this document we use some terminology which is specific to this thesis. Therefore we provide a short list of these terms and abbreviations here, along with their meaning.

- **Authentication System (AS):** A system which provides user authentication services to the framework, including identifying users, verifying an identity, or both. Moreover, an AS can publish authentication events to the framework. Examples of authentication systems are a face recognition system and a smart card reader.
- **Authentication Point:** A location where an arbitrary number of authentication systems authenticate a user in a Co-Authentication scheme,

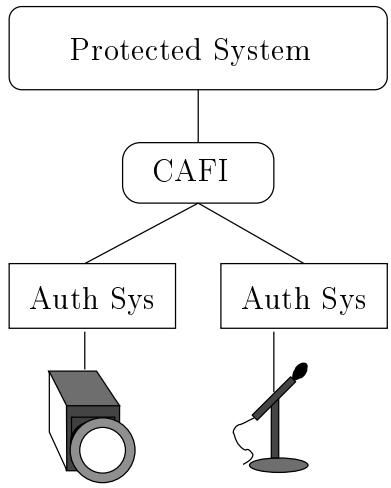

Figure 4.2: A simple Co-Authentication Network setup with one CAFI and two authentication systems, e.g., face and voice recognition systems.

e.g., in front of a physical door which is protected by a face recognition system, a fingerprint recognition system, and a smart card reader, all of which are connected via a CAFI.

- **Co-Authentication Network:** A network consisting multiple interconnected instances of AS, PS and CAFI nodes.

- **Co-Authentication Framework Instance (CAFI):** An instance of the Co-Authentication framework, i.e., the network node which runs the core framework. All AS and PS nodes communicate with the CAFI node. Moreover, the CAFI is responsible for combining scores, enforcing the Co-Authentication policy, and managing subscription relationships between AS and PS nodes.

- **Protected System (PS):** A system which uses the authentication services provided by the CAFI, e.g., an electronic lock of a physical door where the lock is connected to multiple biometric systems via a CAFI node. Moreover, a PS may be the subscriber of authentication events from certain AS nodes. All such subscriptions are handled by the CAFI.

- **Username, UserID, UID:** We will use the terms *username*, *UserID* and *UID* interchangeably, to mean a globally unique identifier for a particular user.

4.3 Design Guidelines

The following *design guidelines* function as soft requirements, i.e. they will not be specified directly in the form of functional requirements, but rather as things to keep in mind when designing the system. As with most software frameworks, we want our system to be secure, robust, flexible, scalable, reliable and as platform independent as possible, both in terms of operating systems and programming languages. Although we do not address all of these requirements in the this prototype it is important to keep them in mind since they may need to be taken into account in the framework design.

4.3.1 Scalability

Scalability is an important factor of the Co-Authentication system. We want the framework to handle everything from protecting a single computer with two authentication systems (e.g., login/password and a fingerprint scan) and a small user base, to an entire building with hundreds of users and multiple authentication points (e.g., deny access to a computer since the principle has not been seen entering the building). An authentication made with the framework should finish in a second at most, to ensure usability and avoid unnecessary user frustration. Stress testing, experimentation with large user bases, and exact time measurements are beyond the scope of this thesis. Nevertheless, scalability is to be kept in mind during the design phase of the framework, with the aim of making future scalability features easier to implement.

4.3.1.1 Horizontal Scaling

A common method for scaling, is *horizontal scaling* which is a *divide and conquer* method. Horizontal Scaling means we can scale the system in different directions, typically by adding more hardware and software systems to the existing infrastructure.

The Co-Authentication could allow for horizontal scaling by being configurable as a hierarchy of Co-Authentication Systems. Hierarchical deployment has several advantages in terms of horizontal scaling, in particular when we want to combine data from two otherwise independent authentication points which are physically far apart. For instance, if we are authenticating a user at the entrance of the office, we can consult with the CAFI at the front door entrance, to see if this user has been seen entering the building. This is an example of

history-based authentication which we will mention again in Chapter 4.4.3.

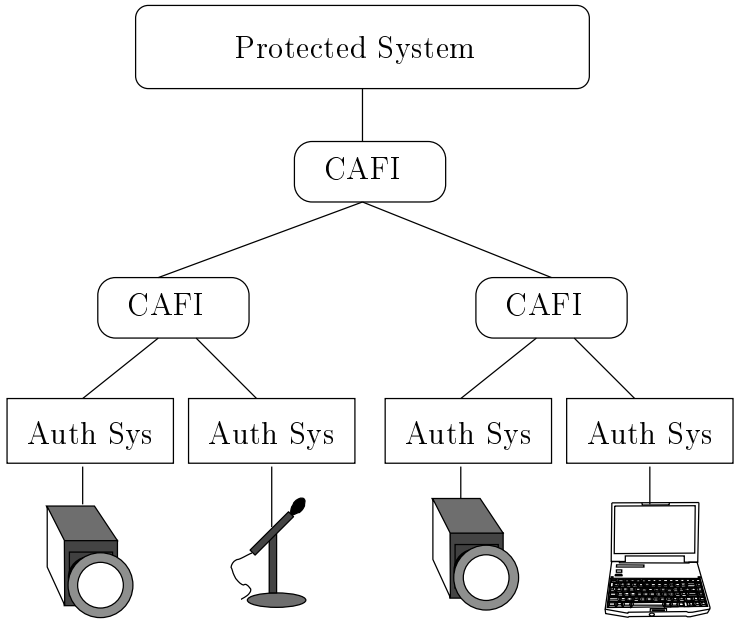

Figure 4.3: The main CAFI delegates some of its task to two sub-nodes. The only thing the main CAFI is concerned with is receiving data from the sub-nodes and making a final authentication decision. The setup of each sub-node is more or less identical to Figure 4.2

4.3.1.2 Code Optimization

The computational performance of the Co-Authentication system, needs to be good enough to handle common deployment scenarios without unacceptable delays. We will however not spend any time on performing code optimization unless a given segment of code has been proven to have significantly insufficient performance. Several reasons lie behind this decision. Tony Hoare and Donald Knuth have both pointed out that *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"* [37]. The *evil* from the quote above means that premature optimization requires spending time on micro-optimizing code, even if it has not been shown to have insufficient performance i.e., *small efficiencies*.

In addition to the extra time spent on tweaking, code micro-optimization often obscures the readability of the code which makes it harder to maintain. For

these reasons we prefer to focus on larger performance issues, such identifying bottlenecks and choosing appropriate algorithms etc., in case we find the performance to be insufficient. Moreover we want to be able to address performance issues by scaling, i.e. add resources to the system and dividing tasks between resources whenever possible. We will however keep performance in mind in the design, with the aim to identifying and avoiding potential bottlenecks.

4.3.1.3 Multiple Protected Systems

In some circumstances there may be more than one system that wish to be notified of the Co-Authentication decisions. For instance one protected system and one global logging system. An example of such configuration is shown in Figure 4.4. The framework shall be able to support this scenario, for an arbitrary number of systems.

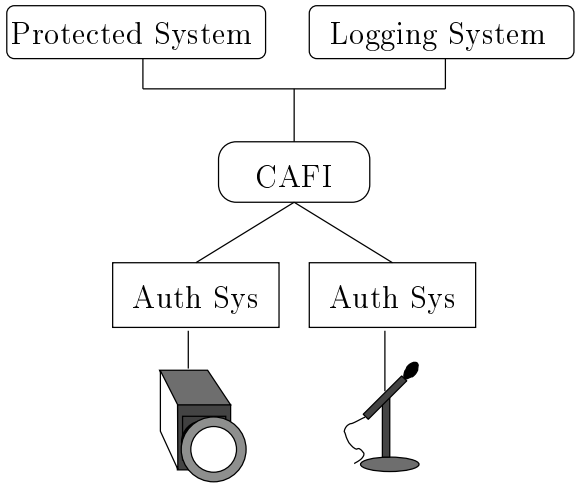

Figure 4.4: Multiple systems receiving data from the same CAFI

4.3.2 Platform Independence

In order for our framework to be applicable in as many different organizations and configurations as possible, it is important to support as many platforms as possible, within reasonable limits. For our Co-Authentication framework we are mainly concerned with two types of platform independence.

The first one is independence from operating systems. We want our framework to be easy to deploy and integrate with the existing architectures of organizations. The framework is likely to become a much more realistic and attractive option for an organization, if they know that they do not have to invest much in the surrounding hardware and software infrastructure. For instance, if an organization is mostly run on UNIX systems they are likely to want the CAFI system to run on UNIX.

Our second platform independence requirement is independence from programming languages. Dependence on a particular programming languages causes some severe limitations. First, existing authentication systems are written in a wide array of programming languages, and it is often easier to integrate them into a Co-Authentication system if they can be extended in their original language. Second, many biometric systems ship with closed and proprietary APIs that may only be compatible with specific languages. Having a language independent framework allows organizations to integrate multiple proprietary systems, regardless of if these systems have language restrictions or not. It is also highly unlikely that all the systems we want to protect are written in the same language. In other words, a language independent framework is applicable and feasible in many more situations than a language dependent solution. The other benefit of language independence is for organizations which want to write their own code for integrating their protected systems with the framework, and implement or integrate their own authentication solutions. If such an organization has several experienced C-programmers, they will likely want to write their code in C, rather than being forced to use, say, Java. Therefore, a language independent solution is more flexible, and makes it a more attractive choice for many organizations.

For these reasons independence from operating systems and programming languages are very important requirements which need to be addressed in both design and implementation.

4.3.3 Simplicity

System simplicity is an ambiguous term. For our framework it means that we will follow the KISS principle (Keep It Simple, Stupid!), to the best of our capabilities. We want the system to be easy to maintain, extend, and integrate into other solutions. To simplify the integration of the Co-Authentication framework, it needs to provide a well defined API, both with regards to the protected system and the authentication systems. We will try to achieve high maintainability by following a set of best practices in software engineering, as listed below.

We want to keep our framework as simple as possible without sacrificing flexibility or applicability. One motive for keeping it simple is that it makes it much easier to deploy and in particular to write integration code for new authentication systems. In addition, clearly defined and simple code makes the framework much more maintainable. Our main method of achieving simplicity is to have clearly defined programming interfaces for all input/output mechanisms of the system, i.e., how the framework interacts with its authentication nodes or the protected system. In particular we need to make abstractions such that the framework does not need to know any details about particular devices or authentication systems, e.g. it should not need to know whether it is communicating with a login/password system or a face recognition system.

The inner workings of the Co-Authentication framework will also need to follow a set of software engineering principles. We should strive to separate the code base into independent and replaceable modules with well defined responsibilities, i.e. good *encapsulation*. These modules should have well defined public interfaces and be as *loosely coupled* to other modules as possible. The same independence requirement applies to code units within each module, i.e., classes, where each class should have a well defined purpose and a public interface which reflects that purpose. However, it is reasonable to expect classes to be tightly coupled within a module. The public interfaces of modules and classes should provide good *information hiding*, i.e., they should present what they do but hide their method of doing so.

Finally, we will strive to follow the DRY principle (Don't Repeat Yourself) [29], by eliminating duplicate code whenever possible.

4.3.4 Flexibility

Flexibility in software is the "*extent to which you can modify a system for uses or environments other than those for which it was specifically designed*" [37]. In order to provide maximum usability, we want our system to be easily adaptable to the various requirements of different organizations. We want the framework to support any number of score combination algorithms, and we must be able to change the policy to use a new algorithm, with minimum effort. Moreover we want to be able to change various properties of the system by altering its configuration or policy, without having to alter any code. Such properties include configurations of authentication systems, decision thresholds, algorithms used and robustness configuration, such as how many, and which nodes can fail before we lock the system for further access.

4.3.5 Robustness

The system will be subject to a wide range of failures, including failing authentication systems or devices, networking problems, power outages, and deliberate malicious attacks. We must strive to have our system robust and address as many of these scenarios as possible. This includes failing gracefully and properly informing about problems whenever possible.

4.4 Requirements

In this Chapter we discuss the functional requirements of our framework implementation. In this thesis we are concerned with demonstrating the usefulness of having a Co-Authentication framework, and we address this by creating a *proof-of-concept* implementation of such a framework, which can demonstrate its core abilities.

In order for the framework to be considered an option in real organizations it needs to address certain functional requirements which are outside the scope of this thesis. In this thesis we will simply assume that the framework and its environment are secure. In particular, secure communication within the Co-Authentication Network will be assumed. Moreover, we will not perform static analysis on the code base or take other measures to secure the framework code itself. We do however fully appreciate the importance of these aspects and list them as future work.

We define three categories of requirements. In Chapter 4.4.1 we list *High Priority* requirements which we intend to satisfy in this *proof-of-concept* implementation. Specifically, these are features which serve to demonstrate the advantages of having a Co-Authentication framework. The features listed in Chapter 4.4.2 have *Medium Priority*. This means that we will address as many of them as possible within the allowed time. Finally, in Chapter 4.4.3, we briefly describe some *Low Priority* requirements which we consider as ideas for future work.

4.4.1 High Priority Requirements

In this chapter we present the functional requirements that have the highest priority, that is, the requirements we intend to fulfill in our proof-of-concept implementation. We list each requirement along with a unique identifier and a short description.

A-1 ***Authentication systems***

A-1.1 ***Add an authentication system:*** When a new authentication system is added to the Co-Authentication Network, we need to register its information (IP/host, port), its mode (see Requirement *A-1.4*), and assign it to one or more protected systems.

A-1.2 ***Update an authentication system:*** When an existing authentication system changes some of its properties (IP/host, port, mode) or is assigned to other protected systems, we need to be able to change this information easily.

A-1.3 ***Remove an authentication system:*** A remote authentication system is not an essential part of the framework, and can therefore be removed. Removing an authentication system should trigger a re-evaluation of minimal conditions, see Requirements C-2.*1* and C-2.*2*.

A-1.4 ***Running Mode:*** An authentication system shall run in one of the three following modes: *Identification*, *Verification* or *Both*. AS that run in Identification mode notify the CAFI of which user is trying to get authenticated. AS in verification mode wait until they are requested to authenticate a specified user. Finally, AS which runs both behaves as it is running both modes simultaneously (i.e., it both notifies the CAFI of users to authenticate, and accepts verification requests).

A-2 ***Communications between the CAFI and an Authentication System***

A-2.1 ***Connection Setup:*** Initialize a network connection between the CAFI and each of its Authentication Systems. The connection should remain in place at all times while the CAFI is actively running, unless explicitly disconnected (see Requirement *A-2.2*. Refer to Requirement B-1.*1* for how to deal with broken connections.

A-2.2 ***Connection Tear down:*** Gracefully terminate every connection between the CAFI and each of its Authentication Systems.

A-2.3 ***Identification:*** The CAFI shall be able to receive an identification event from one or more authentication systems running in *identification mode* (see Requirement *A-1.4*, and proceed to request other authentication systems to verify that identity.

A-2.4 ***Verification:*** The CAFI requests authentication system running in *verification mode* (see Requirement *A-1.4* to verify the presence and authenticity of a user, given his user ID. The authentication systems performs the authentication and sends the result along with a match score back to the CAFI.

A-3 ***Communications between the CAFI and a Protected System***

A-3.1 ***Connection Setup:*** Initiate a connection between the CAFI and a Protected System according to configuration. The connection should remain in place at all times while the CAFI is actively running, unless explicitly disconnected (see Requirement *A-3.2*). Refer to Requirement B-1.*2* for how to deal with broken connections.

A-3.2 ***Connection Tear down:*** Gracefully terminate all connections between the CAFI and each of its Protected Systems.

A-3.3 ***Authentication On Request:*** The Protected System requests the CAFI to authenticate a specific user. The CAFI queries its authentication systems for verification and match scores (see Requirement *A-2.4*. The CAFI then calculates an overall score (see Requirement *A-4.3* and compares it to the threshold configured for the PS (see Requirement *A-4.4*) and returns a decision and the combined match score.

A-3.4 ***Notify a Protected System about a triggered authentication:*** This requirement addresses authentications which are not the result of a request from the PS. These occur when one or more authentication system running in identification mode triggers the CAFI to authenticate a user. The CAFI performs its authentication just as when it does so on request from the PS. The CAFI then sends information about the authentication to the PS, including information about the claimed user ID, the overall match score, or the binary authentication results if the PS does not support probabilistic results.

A-4 ***Score Combination:***

A-4.1 ***Average Function:*** Calculate an overall score using the average function on a given set of match scores.

A-4.2 ***Median Function:*** Calculate an overall score using the median function on a given set of match scores.

A-4.3 ***Calculate Overall Authentication Score:*** The CAFI shall be able to calculate an overall authentication score by feeding the match scores from all the authentication systems to a score combination function (see Requirements *A-4.1*, *A-4.2* and B-9). The overall score is used to make authentication decisions (see Requirement *A-4.4*).

A-4.4 ***Compare Score to Threshold:*** The CAFI should always compare the score from the score combination function with a threshold, whenever the protected system is configured to receive authentication decisions (see Requirement *A-5.3*). If the score is equal to, or above the threshold, then the authentication is successful, but otherwise it is not.

A-5 ***Configuration and Policy:***

A-5.1 ***General Structure:*** The configuration file shall be easy to parse, human readable and non-redundant, i.e., the same information shall not be specified in multiple places.

A-5.2 ***Score Combination:*** The CAFI policy environment must specify which score combination method is used to calculate the overall match score, as a part of the authentication policy for each PS. The configuration must allow for change of the score combination method without altering any code.

A-5.3 ***Decision Threshold:*** The decision threshold for each protected system must be easily configured, and alterable without changing any code.

A-5.4 ***Authentication Systems:*** Each Authentication System has to be specified in the configuration, along with all the information that the CAFI needs for it, i.e., its IP/host, port, mode, etc. Each AS also has an identifier which is unique within the configuration. An AS which is used by more than one protected system shall only be specified once, and then referred to in the configuration for those PS.

A-5.5 ***Allowed Protected Systems:*** The CAFI has a configured list of Protected Systems that are allowed to connect to it, in order to prevent rogue systems from connecting and eavesdropping on events.

A-5.6 ***PS Scoped Policies:*** Each PS shall have an individual configuration of its authentication policy within the Co-Authentication framework.

A-5.7 ***Authentication Systems for each PS:*** Each Protected System has a policy which states which authentication systems are used for its authentication decisions.

4.4.2 Medium Priority Requirements

In this chapter we describe requirements with medium priority. These are mostly features that are nice to have but are not critical to demonstrating the use of our proof-of-concept framework. We will address as many of these requirements as time allows for.

B-1 ***Reconnect Broken Connections***

B-1.1 ***Automatic Reconnect (CAFI-AS):*** If a connection between the CAFI and an authentication system is lost, the CAFI should repeatedly attempt to reconnect to the AS, where the number of attempts

and time delay interval is defined in configuration, see Requirement B-1.*3*.

B-1.2 ***Automatic Reconnect (CAFI-PS):*** If a connection between the CAFI and a protected system is lost, the CAFI should repeatedly attempt to reconnect to the PS, where the number of attempts and time delay interval is defined in configuration, see Requirement B-1.*3*.

B-1.3 ***Reconnect Configuration:*** The Configuration file should include information on how the CAFI should react when connections break. This includes number of reconnect attempts and how long delay interval should be between attempts.

B-2 ***Node Maintenance:*** It should be possible to disconnect and reconnect a (PS or AS) node for maintenance. During the maintenance the Jury node should not attempt to reconnect (as described in Requirements B-1.*1* and B-1.*2*). Similarly, should the CAFI need maintenance it should be possible to disconnect the CAFI and reconnect it to all the authentication systems and protection systems without any effort on the behalf of any PS or AS.

B-3 ***Support for Multiple Protected Systems:*** The CAFI should support an arbitrary number of protected systems simultaneously, i.e., more than one system which will request and be notified of authentications. For instance this can be a physical door and a specialized intrusion detection system. Each PS can have a different statistical combination algorithm, different set of AS it uses, and a different weighting of the AS.

B-4 ***PS Scoped Weights for Authentication Systems:*** As a part of its authentication policy, each PS can have static weights for each of its authentication systems. These weights can then be used with the score combination functions for those PS. The motivation for assigning weights to the AS is that different authentication systems might be evaluated to be of different strengths, for instance an iris scan performed by tamper resistant hardware can be evaluated as stronger than a fingerprint recognizer with cheap consumer hardware and known defects.

B-5 ***Dynamically Loadable Algorithms:*** It should be possible to change the policy with respect to which statistical algorithms are used, without having to restart the framework.

B-6 ***Support for external algorithms:*** It should be possible to specify algorithms in the policy, which are not a part of our internal framework. That is, our framework should support external score-combination algorithms, as long as they fulfill the interface specification.

B-7 ***Default Configuration Settings:*** The CAFI configuration should allow for default value definitions. If a default value is configured, then all configuration entries which want to override the default, must explicitly state so. For instance one should be able to specify that protected systems should use the *average* statistical combination function by default. This means that all protected systems that want to use another statistical function, will have to be explicitly configured to do so. The default specification supports the following fields:

B-7.1 ***Score Combination Algorithm for PS:*** This specifies which score combination function the protected systems use by default. If this is configured, a PS has to explicitly state if it wants to use a different function.

B-7.2 ***Reconnect Settings:*** This field includes default values for reconnect attempts with disconnected nodes. These include how many times should a reconnect be attempted, how long should the time interval between attempts be, and what to do if all our attempts fail.

B-8 ***Limited Notification:*** In some circumstances it can be favorable for a protected system to receive notifications of particular types of events only. For instance a protected system may only be interested in successful authentications while another (e.g., a security log) might only want to be notified of failed authentications.

B-9 ***Score Combination Algorithms:*** Implement several advanced statistical score-combination algorithms.

B-10 ***Administrative Console:*** An text-based console, which can be used to manage the framework, e.g., change configuration and policy, disconnect nodes, etc.

4.4.3 Low Priority Requirements

In this chapter we describe low priority requirements. These are features which we have no intention of implementing in the course of this thesis project, but which we want to add to the framework at a later time. The low priority requirements are described on a high-level, partly since we do not intend to address them at this stage, and partly because some of them are large enough to be projects of their own. The list presented below may be considered as an repository of ideas for future work.

C-1 ***A Configuration Checker Tool:***

C-1.1 ***Boundary Checks:*** Check the values of all configured parameters for bounds sanity (e.g. no symbols in an alphanumeric field)

C-1.2 ***Consistency Checks:*** Check if all required fields are specified, that all field dependencies are met, and that all referenced configuration fields exist.

C-2 ***Minimal Running Conditions:*** It is possible that, at some point in time, many parts of the Co-Authentication Network have failed. This can either be caused by normal failures such as power outages, or by deliberate malicious attacks. It is therefore important to be able to define a lower boundary on when to stop trusting the Co-Authentication Network and deny all authentications. These requirements should be implemented as a part of the PS-specific authentication policies.

C-2.1 ***Minimum Number of Authentication Systems:*** In the Co-Authentication policy, we should be able to define a minimum number of AS which need to be working in order for the CAFI to make decisions.

C-2.2 ***Minimum Set of Authentication Systems:*** In some cases we need rely more on some authentication systems than others. In this case we want to be able to define a set of authentication systems that have to be running in order for the CAFI to make decisions, i.e., if one of these fails, the CAFI will deny all authentication attempts.

C-2.3 ***Combined Number and Set:*** If both a minimum number (see Requirement C-2.*1*) and a minimum set (see Requirement C-2.*2*) of AS is defined, then both conditions have to be fulfilled in order for the CAFI to make successful authentications. If the minimum number is defined to be higher than the number of elements in the minimum set, then all the AS in the set need to work, in addition to a number of arbitrary AS required to satisfy the minimum number of AS. Example: If out of the set {A,B,C,D,E,F} the minimum number is four and the minimum set is {A,B,C} then A,B,C and any one of {D,E,F} must be running, e.g., {A,B,C,E} is an acceptable combination of running authentication systems.

C-3 ***CAFI-CAFI communications:***

C-3.1 ***Connection Setup:*** Initialize a network connection between two Co-Authentication systems in a hierarchical way, i.e., that it is obvious which of the two systems is the parent and which is the child system. The connection should remain in place at all times while the parent CAFI is actively running, unless it is explicitly disconnected. Refer to Requirement C-3.*3* for how to deal with broken connections.

C-3.2 ***Connection Tear down:*** Gracefully terminate every connection between the CAFI and each of its child systems, as well as the connection to its parent CAFI (if any).

C-3.3 ***Automatic Reconnect:*** If a connection between two Co-Authentication systems is lost, the parent system should repeatedly attempt to reconnect to the child CAFI, according to configuration (see Requirement B-1.*3*.

C-3.4 ***Request Authentication:*** One CAFI should be able to request another CAFI to authenticate a user.

C-3.5 ***Variable Details of Authentication Response:*** When returning the authentication result to the parent CAFI, the child CAFI should be able to return either an overall match score, a decision, a 2-tuple (decision, overall match score) or a vector of all the independent match scores from each authentication system.

C-4 ***Identity Mapping:*** In some cases the authentication systems will come with their own user databases, which may be incompatible in such a way that a user cannot be registered in the authentication system with the same user credentials as in the protected system. For instance, the protected system may allow 16 character usernames in Unicode, while the authentication only allows for 8 character plain ASCII usernames. In these cases we may not be able to use a globally unique user identifiers and must therefore resolve to mapping. Mapping is done on a per PS basis, such that a username in a particular PS is mapped to IDs of those AS it uses.

C-4.1 ***Register an Identity Mapping:*** Register a four-tuple (ID of the PS, user ID within PS, ID of the AS, user ID within AS).

C-4.2 ***Update an Identity Mapping:*** Update any field of an identity mapping

C-4.3 ***Delete an Identity Mapping:*** Delete a single entry from the identity mapping.

C-4.4 ***Delete all Identity Mappings for an AS:*** Delete all identity mappings for a single authentication system.

C-4.5 ***Delete all Identity Mappings for a user,PS combo:*** Delete all identity mappings for a single user ID from a single protected system, i.e., all mappings with a user ID,PS-ID combination.

C-4.6 ***Look up an Identity Mapping:*** Look up a user ID for a given authentication system, given a user ID within the protected system.

C-4.7 ***Configuration for Identity Mapping:*** The following details have to be configurable without having to change any program code: database host, database port, database driver, database name, database username, database password.

C-5 ***Secure the framework:***

C-5.1 ***Secure Communications:*** Communication between various Co-Authentication nodes (PS, AS, CAFI) need to be secured in order to prevent rogue nodes from joining the network, as well as to make it harder to eavesdrop. This requires end-to-end security, since there might be multiple untrusted network nodes and appliances en route between two Co-Authentication Network nodes.

C-5.2 ***Node Authentication Scheme:*** In order to secure the Co-Authentication Network, we must have a method of authenticating the individual nodes. For instance, when we initialize the system, we need to know that the front door face recognition system is indeed the system we know and trust, and not a rogue system. This scheme could for instance use cryptographic certificates to identify each node, granted that the nodes can safely store their certificates.

C-5.3 ***Secure Framework Code base:*** A large part of system infiltrations are based on application vulnerabilities. Therefore the security of the framework source code is very important. To reduce the risk of severe vulnerabilities in our code base we need to take several steps to harden the code. These include static code analysis, rigorous input validation, and keeping the system access requirements to an absolute minimum. For instance, the CAFI probably does not need to be able to write anything except for its log files. These steps should be easier if we successfully follow our guidelines on system simplicity and low module coupling.

C-5.4 ***Secure Configuration:*** Since the configuration will specify who protects what, and what authentication systems we use, etc. it is very important that we secure the integrity of the configuration file. This can for instance be done by access control and/or encryption schemes.

C-5.5 ***Access Control for Maintenance:*** While we need to be able to disconnect and reconnect some nodes during maintenance, we must prevent unauthorized parties from doing so. It is not obvious how this can be done in the CAFI, since these measures are likely to rely heavily on the logical and physical access control of the hosting system (i.e., the machines and operating systems that run the Co-Authentication nodes). It is however necessary to give this some thought and possibly implement protection into the CAFI. At the very least, we can provide thoroughly documented guidelines on how to secure the running environment.

C-6 ***User Tracking:*** User movements are tracked within an environment which entrances are secured with an Co-Authentication network. Subsequent authentications within the environment are affected by the tracking

data, and possibly the score of the initial authentication. For instance, a user who has been successfully authenticated at the entrance to an office building is about to enter his office. If he has been successfully tracked to his office, and other users do not interfere (e.g., no unauthorized users are in vicinity of the office door), then he does not need to be authenticated again. The match score of the initial front door authentication might be used such that if it is beneath a certain threshold the user has to authenticate himself at the office door despite successful tracking.

C-7 ***Behavioral Anomaly Detection:*** This is related to *user tracking* (see Requirement C-6. The system should include a module which detects anomalies in the user behavior with respect to the environment. For instance, access to an office could be denied if the person has never been seen (or authenticated) at the entrance to the building, or if he was last seen entering the building the day before. This means that the CAFIs should keep a history of authentications, and offer a query interface. For instance, the office door CAFI should be able to query the front door CAFI, f.ex. *Have you successfully authenticated Dennis in the last 120 minutes?*.

C-8 ***Peer/Adversary Detection:*** The System should be able to detect if more than one person is at an authentication point. If two or more people are at an authentication point, the system should attempt to authenticate all of them. If one or more of them fails to authenticate, or if one or more of them cannot be identified, then all of them are denied authentication.

C-9 ***Guard Enforcement:*** This is a special case of *peer detection*. If guard enforcement is enabled (in the configuration), it is required that the CAFI successfully detects and identifies a person who is registered as a guard in order to provide access. For instance, if the CAFI is protecting a front door, no one can be authenticated unless a registered security guard is present, and as a result, no one can open the door.

C-10 ***User Relations:*** We want to be able to constraint access to a boolean condition for a given set of users. This includes *AND* and *XOR*. For example, a digital lock could enforce an XOR policy such that either Alice or Bob may be allowed to enter the room, but they may not be both present simultaneously.

C-11 ***Decision Continuity:*** Usually an authentication decision is made only once, e.g., a user can either open a file or he cannot. We want our framework to support *decision continuity*, i.e., we want to be able to trigger a re-evaluation of the authentication decision. For instance, if John has gained access to a logical file and is reading it on his monitor when Dennis enters the area, we want the environmental change to trigger the authentication of Dennis, and notify the PS of the result. The PS can then check if

Dennis is allowed to read this document. This allows the protected system to revoke access to the open document, if it so desires.

C-12 ***AS Driver Library:*** When adding a new type of authentication system to the framework we need to implement a *driver*, i.e. a piece of software that allows the authentication system to successfully communicate with the framework. This can be seen as a wrapper, which translates between Co-Authentication protocols and protocols understood by the authentication system. We want to create a driver library, a software library which includes drivers for many popular authentication systems.

C-13 ***User Documentation:*** While this is not a requirement for the framework itself, it is critical if the framework is to become a realistic option for organizations. This step involves creating thorough documentation on how to install, configure and use the system.

Chapter 5

Design

A software framework is likely to exist for a long time, so it is important to design the framework, in a way that facilitates the evolution that is inevitable in its lifetime. Further, since we want our framework to be flexible and generic, we have to pay extra care to design the framework in a way that makes future extensions as easy to incorporate as possible. Finally, we cannot expect future developments to be limited to the initial authors, and thus must make the design as clear to external contributors as possible.

For these reasons, the design has to be easy to understand and well structured into logical building blocks, so that future improvements will be easier to make. If this step is ignored, the code will be hard to understand by others, and perhaps even to the program authors themselves, after some time of inactivity. Further, the very nature of a framework is to act as a foundation which more specialized functionality is built upon. That is, the framework itself is of little use to an organization without further customizations, such as integration of their authentication systems and protected systems. Therefore it is crucial to design the framework such that it is easy to extend and integrate with other systems. In particular, the design needs to be generic, i.e., not customized for a particular scenario or an organization.

In this chapter we present our framework design and motivate the most important design decisions. We describe the framework at different levels of abstrac-

tion, starting with the big picture and gradually introduce more details. We have opted for an object oriented solution and our design is made with the Java programming language in mind.

The diagrams presented here show how different modules and layers interact. They are however, not meant to form a complete model of the system. Our implementation will have classes that are not shown in the diagrams below, and some of the classes shown here are implemented slightly differently. Similarly, in the class diagrams we only show the attributes, methods, and parameters that we believe will help clarify the purpose of that class.

5.1 Overview

The Co-Authentication infrastructure of the Jury framework is organized as a distributed system of authentication systems, protected systems and a Co-Authentication Framework Instance (CAFI). We call this distributed system the Jury Network, and it is designed as a combination of the client-server model and the publish-subscribe model. In the former model, the client process connects to the server, after which it can send requests to the server and receive responses back from it. Figure 5.1 shows how the client-server model is setup in the Jury Network. The connection between a PS and Jury is a traditional client-server implementation as described above. The connection between Jury and a remote AS is however, a bit fuzzy in the distinction between a client and a server. In the figure, we show Jury as the client, and the AS as a server, since Jury sends requests to the AS and receives responses back from it. However, in terms of connection handling, Jury acts as a server which the AS connects to. The reason for this design is that we want to make it as easy as possible to adapt authentication systems to our framework, and writing a client is considerably easier than writing a server.

Figure 5.1: The Client-Server architecture, the connections between the Jury framework and remote AS nodes are marked with a star because although they act like a client-server, the AS is responsible for initiating the connection.

The publish-subscribe model is implemented such that a PS subscribes to a set of authentication systems, but with Jury acting as a *subscription agent*. What this means is that when an AS publishes an event, the Jury framework receives it and carries out additional tasks to gather further information about the event,

before sending it to the subscribing protected systems. For instance, if an AS identifies a user, Jury receives the published message and carries out further verification of the identity by requesting other authentication systems to verify the user. When Jury has received verifications from the other authentications systems, and combined the score, it generates a new authentication event which it then publishes to the subscribing protected systems. Moreover, Jury manages the subscription relationships between PS and AS nodes, i.e., which subscribes to which, and how to handle incoming events for each subscription. We will use the term *notifications* for published messages that do not require a response. Figure 5.2 illustrates the publish-subscribe implementation at a high-level. For the remainder of this thesis, we will use the short-hand of saying that a PS subscribes to an AS, and that the AS may perform authentication services for the PS, to avoid having to repeatedly state how the CAFI acts as a subscription agent.

Figure 5.2: The Publish-Subscribe architecture: Jury acts as a subscription agent between AS and PS systems, i.e., it receives AS messages on behalf of protected systems.

Each AS or PS system has a single persistent connection with the Jury framework, i.e., a remote system does not connect to the CAFI for each request and disconnect afterwards. Moreover, the published notifications are sent over the same connections as requests and responses are. The persistent connections are illustrated in Figure 5.3.

Figure 5.3: All messages between a remote system and the CAFI are sent using the same persistent connection, regardless of whether they are request/response messages or event notifications.

We have chosen to design our framework such that most interactions and functionality are abstracted with public interfaces. Such abstractions give us flexibility by allowing us to make internal changes to method implementations without having to alter any code of the calling methods. We use this technique extensively in our design, to separate the modules from each other. When the PS-Module calls a service method in the kernel, it can only do so through the public interfaces that define the allowed interactions between the PS-Module and the kernel. If we were to completely rewrite some of these services, such

that only the signature remained intact, the PS-Module will not be affected, since it does not rely on internal implementation details of the kernel.

Besides separating modules, we use interfaces to allow multiple implementations of the same class. We frequently use this method to abstract functionality in our framework, particularly in the score combination module as we will discuss in Chapter 6. The benefit of this approach is best described with a short example: If a method needs to iterate a data collection, we will supply it with an instance of the *Iterator* interface. The method can then iterate the collection without knowing whether it is a linked list, an array or something completely different.

Our Co-Authentication framework is separated into four modules and a kernel. The Protected Systems Module (PS-Module) handles all services, networking and other issues that have to do with protected systems. Similarly the Authentication Systems Module (AS-Module) does the same for authentication systems. The Score Combination Module provides abstracted access to the score-fusion algorithms which are used to combine match scores and calculate an overall score. The AS, PS and score combination modules are all isolated from each other and can only communicate with each other via the Kernel. They do however, along with the Kernel, have read-only access to the Configuration module, which contains all system configuration and policy parameters. In addition to providing read-only access to configuration and policy parameters, the configuration module is also responsible for reading and parsing configuration and policy files, which, in the current implementation, are stored in the local file system of the CAFI. The Kernel is at the heart of the framework, and is responsible for synchronizing operations that involve more than one module.

The interactions between remote AS and PS systems are defined in the Jury policy file, which is an external file which is read by the Configuration and Policy module. It includes details on what authentication systems are used by a particular protected system, assigns PS-specific weights, or confidence levels, on each authentication system, and specifies how a PS reacts to events. Since the policy is the only place where the AS and PS interactions are defined, the actual AS and PS nodes can be completely unaware of each other. Figure 5.5 shows an example of how an authentication may be performed using four authentication systems and an average score combination algorithm. First, *AS0* identifies a user and notifies the Jury framework of this event. The framework checks which protected systems are interested in events from *AS0* and finds a single PS subscriber, simply called *PS* in the figure. The PS is configured to react to incoming identifications by verifying the claim using the other authentication systems that it subscribes to, according to its policy (in this case: *AS1*, *AS2*, and *AS3*). The Jury framework combines the scores from these three systems and concludes the event sequence by sending the result to the PS.

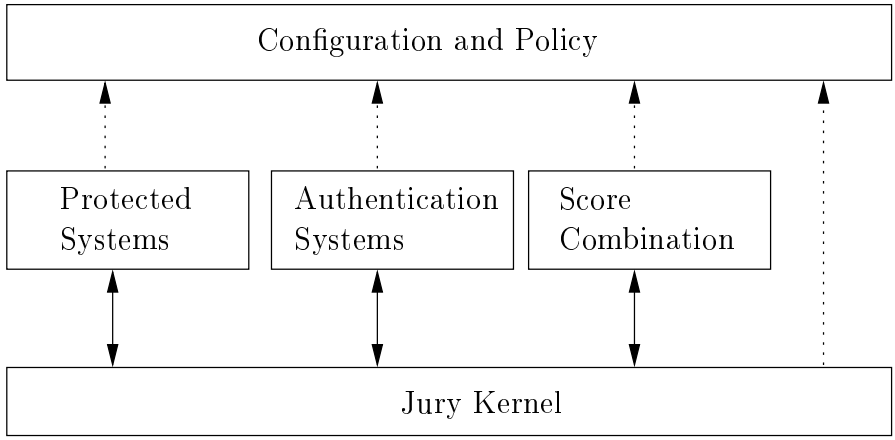

Figure 5.4: Module Overview. The four functional modules and the kernel. The solid bi-directional lines indicate direct communication, while the one-way dotted lines indicate read only communications.

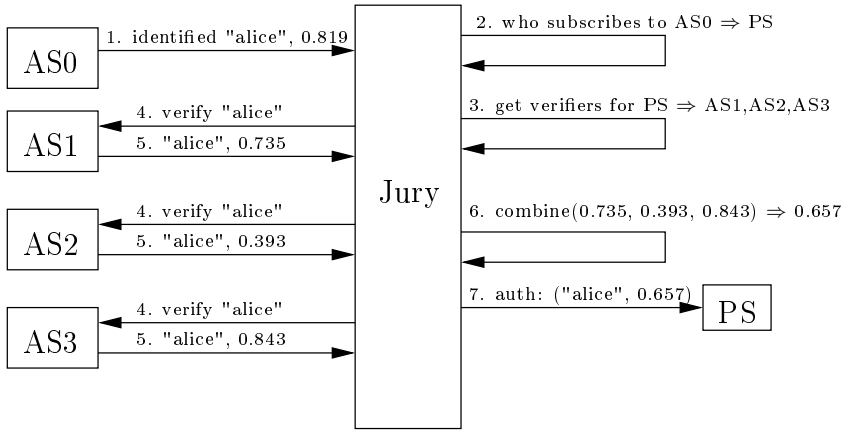

Figure 5.5: An example authentication scenario, where one AS generates an identification event which is verified by other authentication systems. Steps with identical sequence numbers can be seen as happening in parallel.

The communications between Jury and the remote AS and PS nodes takes place over a network, using a specific Jury Message Protocol (JMP). To integrate a system that does not have built-in JMP support, we create a wrapper between the remote system and the framework. We call these wrappers *Jury Interpreter Nodes* or JINs, and each JIN has to support a subset of the JMP, depending on which system it is wrapping. For instance, a JIN for a protected system must

support the part of the protocol that deals with protected systems. Figure 5.6 shows how remote systems are connected to the AS-Module and PS-Module via JINs. JINs allow organizations to gradually move towards a Co-Authentication infrastructure, which is likely to be a more attractive option for them than to replace their entire infrastructure at once.

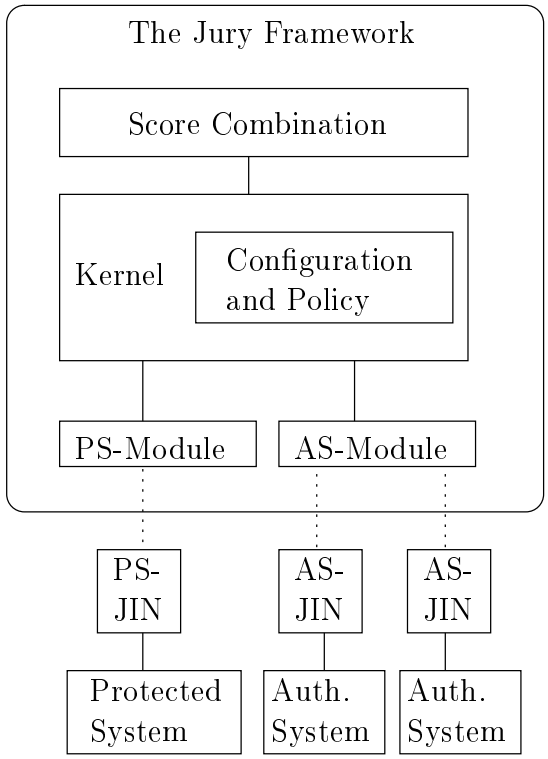

Figure 5.6: The interactions between the Jury framework and remote AS and PS nodes.

The Jury Message Protocol is composed of three types of messages. A *request* is a message that requires the receiver to reply back to the sender with a *response* message. The third type is the *notification*, which is simply a message which the recipient does not reply to. In the PS module, all requests originate in the remote PS which receives responses from the framework. The notifications of the PS module however all originate in the framework. The AS-Module reverses all of these directions. All AS-related requests are sent from the framework to a remote AS, which replies. Similarly, all notifications in the AS-Module originate in the remote AS nodes.

5.2 The Protected Systems Module

The *Protected Systems Module* is responsible for everything regarding protected systems and their relation to the rest of the Jury framework. This includes managing network connections to the various protected systems, sending and receiving protocol messages, listen for and process events from other framework modules, and acting out service requests received from the protected systems. The purpose of the Jury framework is to provide Co-Authentication services for protected systems, which is why the PS-Module is purely a service consumer and does not provide services to other modules.

The PS module is divided into two layers, namely a service layer and a network layer. The layered design is due to separation of concern, i.e., each layer bundles together related functionality, and is not to be confused with a protocol stack. The Service Layer is responsible for all interaction with the other Jury modules. When the kernel needs to deliver a notification to a remote PS, it does so via the Service layer. Similarly, when the PS-Module requests services from other parts of the framework, such as when it wants verify an identity, it goes through the Service Layer. The Network Layer handles all network communication with protected systems. This includes sending and receiving messages, relaying messages to the right protected system and synchronizing activities.

In addition we have small sub-module consisting of Message Helpers, i.e. classes that help with message handling. It consists of classes that parse incoming JMP messages from the network layer, send messages to an output stream, and data classes to represent message data that can then be used by the service layer. The classes in the message sub-module do not know where the messages come from or where they are going, nor do they know what the data they contain is used for. The Service and Network layers can communicate directly with each other, but only using primitive data types (e.g., integers and strings) and data types from the message sub-module.

Figure 5.7 gives an overview of the PS module. Chapters 5.2.1–5.2.3 will give a more detailed of the service layer, network layer and message helpers of the PS-Module.

5.2.1 The PS Service Layer

The Service Layer handles all interactions between the PS-Module and the Jury kernel. The kernel provides services that the PS-Module can access through its Service Layer. Similarly, the kernel needs to be able to deliver notifications, i.e.,

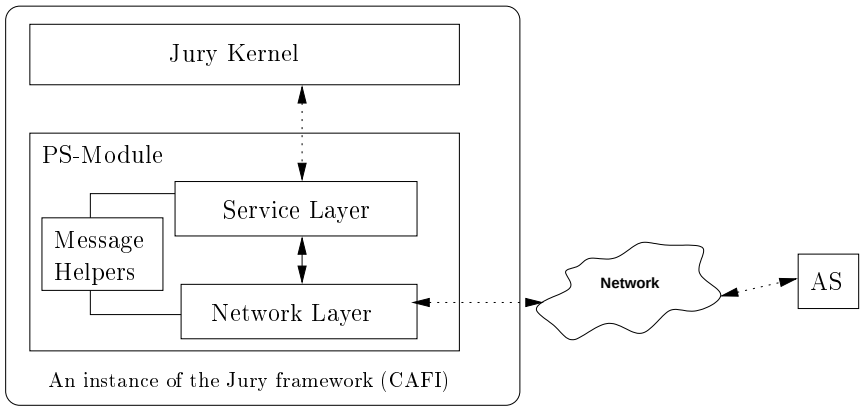

Figure 5.7: The layered PS-Module and its interactions with the Kernel and the Jury Network. The Service Layer and Network layers both use the Message Helpers.

subscription messages, to remote PS nodes, which is also done via the Service Layer. In other words, the connection between the Service Layer and the Kernel is bi-directional.

These interactions go through three public interfaces, two of which deal with commands and notifications from the framework, while the remaining one defines PS related services provided by the kernel. The interaction paths are shown in Figure 5.8, while the interface specifications are shown in Figure 5.9.

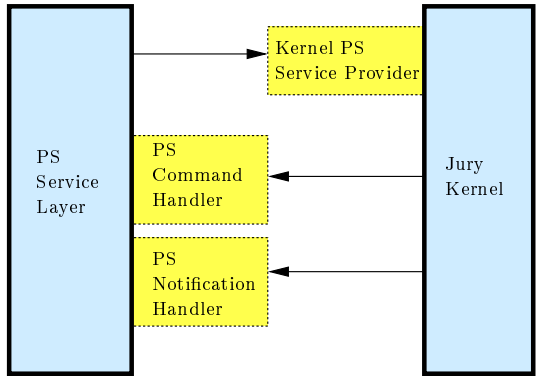

Figure 5.8: The interactions between the PS module and the Jury kernel via public interfaces.

The *PSNotificationHandler* interface specifies all notifications which the PS-

Module should be able to receive from the kernel. For instance, when an AS successfully identifies or authenticates a user, the kernel notifies the PS-Module once for each protected system that subscribes to the AS in question. The PS-Module then forwards the notification to the respective protected systems.

The *PSCommandHandler* interface defines the command methods which the kernel can call on the PS-Module. The implementation of this interface is responsible for carrying out the commands. For instance, an implementation of *disconnect* is responsible for closing the connection to the specific PS. Optionally, it can first send out a message to that PS, notifying it of the pending disconnection and the reason for it. This functionality is primarily meant to be used with the administrative console, which is described in Requirement B-10.

Finally, the *KernelPSServiceProvider* interface will be discussed along with other Kernel functionality in Chapter 5.5.

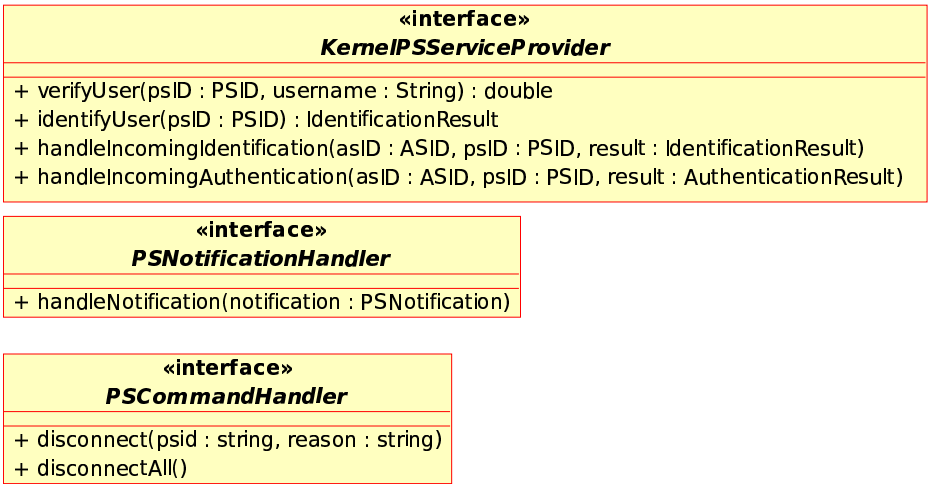

Figure 5.9: The public interfaces that define and constraint interactions between the PS module and the Jury kernel.

The implementation of the Service Layer consists of a *facade* [25] which implements the *PSNotificationHandler* and *PSCommandHandler* interfaces described above. The facade is a singleton class which creates the appropriate message data types from the *Message Helper Sub-Module* and passes them on to the *Network Layer*. The facade and classes related to it are shown in Figure 5.10. In addition we have the PSKernelRelay, which acts as a proxy whenever a PS-Module class needs access to services provided by the kernel.

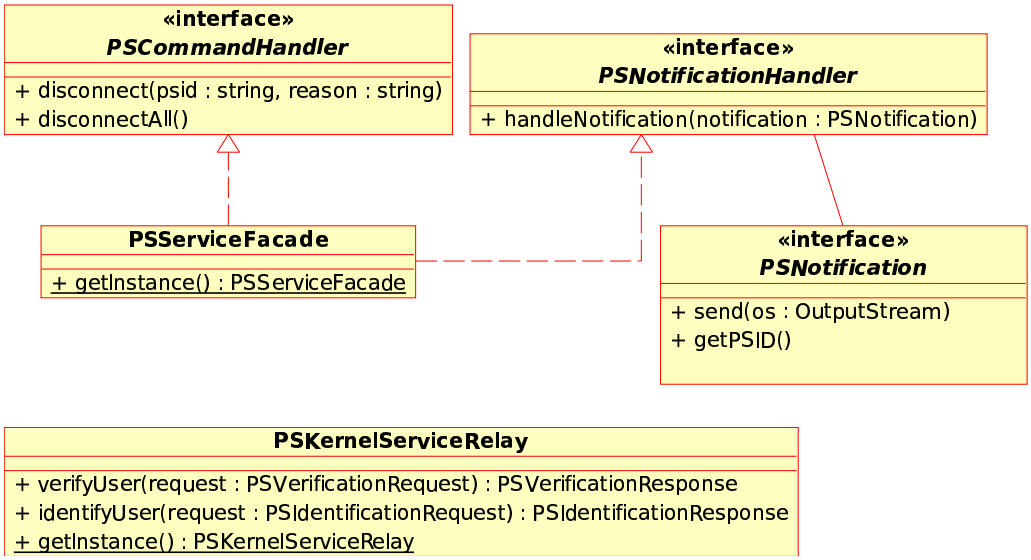

Figure 5.10: Implementation of the PS Service Layer.

5.2.2 PS Network Layer

The Network Layer is responsible for managing connections, reading from, and writing to, network connections to the protected systems, as well as to keep track of which protected system is using each connection, so that responses and notifications can be sent to the correct PS. Further, many of the network layer responsibilities require concurrency, which means that the network layer has to take great care in synchronizing events. For this reason, we can safely say that the network layer is by far the most complicated of the three. The main elements of the network layer are shown in Figure 5.11.

The network layer is essentially a multithreaded server [58]. The module listens for incoming connections on a specific port, and when a connection arrives, it will be accepted. Once a connection is accepted, the listener checks the Jury policy to see if the remote address and source port of the connection match an entry in the policy of the PS. The connection is terminated if they do not match, but otherwise the listener fetches the unique PS identifier from the policy and creates a new thread to handle that connection. The listener daemon class that accepts the connections and creates the threads is trivial, and has therefore been omitted from Figure 5.11. The most interesting classes with regards to the connections are *PSConnectionHandler* and *PSConnectionManager*.

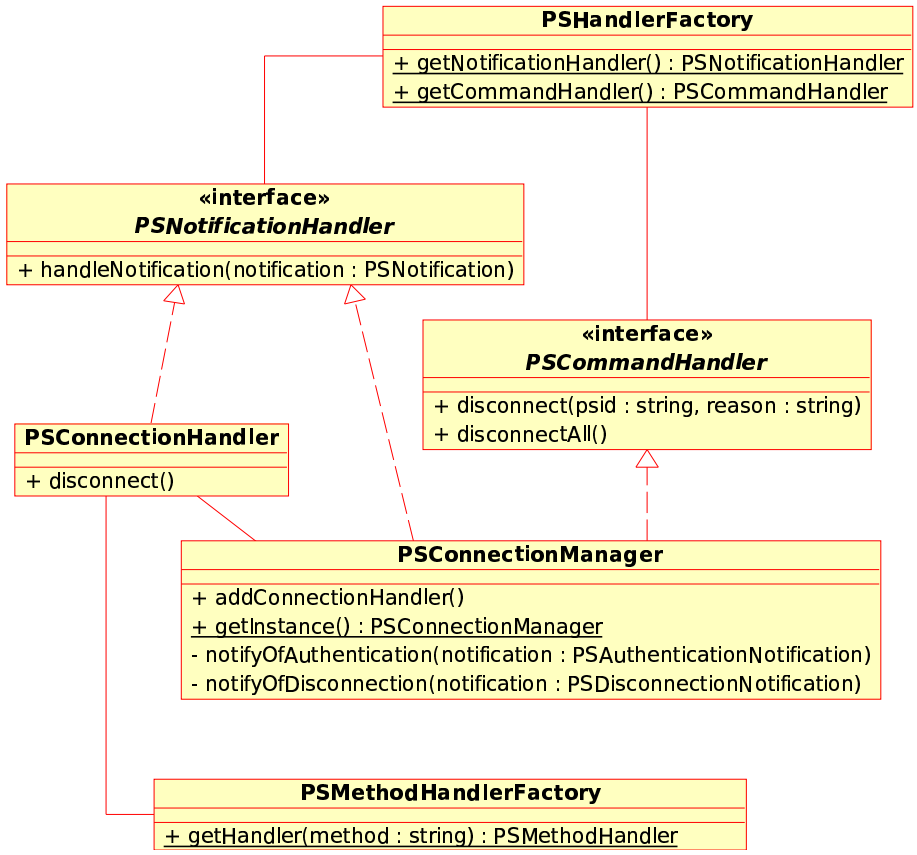

Figure 5.11: The PS Network Layer

Each instance of the *PSConnectionHandler* class is responsible for handling a single network connection. In other words, it is the thread that is started by the listener once the connection has been accepted. The connection handler will parse the headers of incoming messages, figure out what type of request it is, and forward it to the appropriate *method handler*. The method handler, implemented as an instance of *PSMethodHandler* then parses the body of the message, calls the services required to process the request via the *PSKernelServiceRelay*, and finally sends a response, containing the results of the service call, back to the protected system.

The complexity in managing these connections becomes clear once we have more than one protected systems, and is the motivation for the unique PS identifiers mentioned above. We have to be able to assign a unique identifier to each of

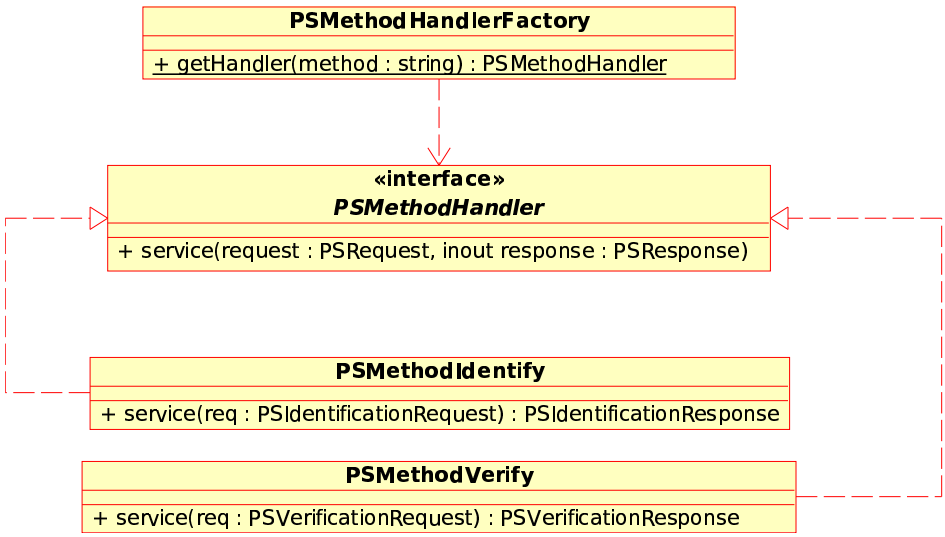

Figure 5.12: The PS Network Layer Method Handlers

these systems such that we know how to handle their requests and notifications. For instance, when a protected system needs to be informed of an authentication event, we must be able to look up which connection to use for sending the notification. This is the main responsibility of the *PSConnectionManager* singleton class, which contains handles to each of the active *PSConnectionHandler* objects, and maps them to the unique identifier which it receives from the configuration module.

5.2.3 The PS Message Helpers

The Message Helpers are a collection of data types which we call messages and classes that process them. A PS message is a set of data that we wish to send to a protected system, or information we receive from a protected system, and each helper or datatype corresponds to a message from the PS part of the Jury Message Protocol, which we will describe further in Chapter 5.7.

Messages that we receive from a PS are called requests and are described by the *PSRequest* interface, and each implementation of *PSRequest* encapsulates a specific type of request. For instance, the *PSVerificationRequest* encapsulates the PS request for the framework to authenticate a specific user, i.e., to verify his presence. When the framework has processed the request, it replies by

sending an instance of *PSResponse* back to the PS. Each response type has its own implementation of *PSResponse*, and corresponds to an implementation of *PSRequest*, e.g., *PSVerificationResponse* corresponds to *PSVerificationRequest*. The notable exception to this rule is *PSErrorResponse* which is sent when a request cannot be fulfilled, for whatever reason. Needless to say, no system is expected to request an error. The PSRequest and PSResponse interfaces along with sample implementations are shown in Figures 5.13 and 5.14. The processing of the requests is a part of the Network Layer and will be described in Chapter 5.2.2.

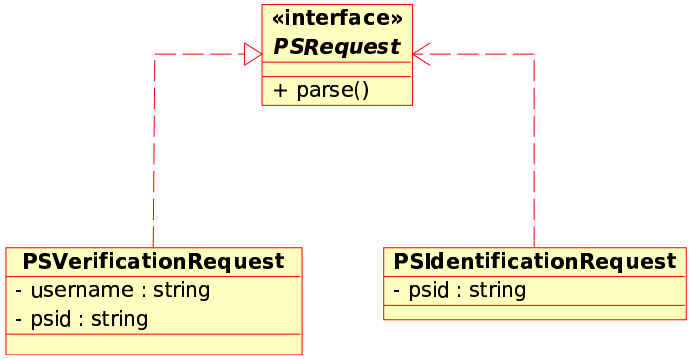

Figure 5.13: PS Requests.

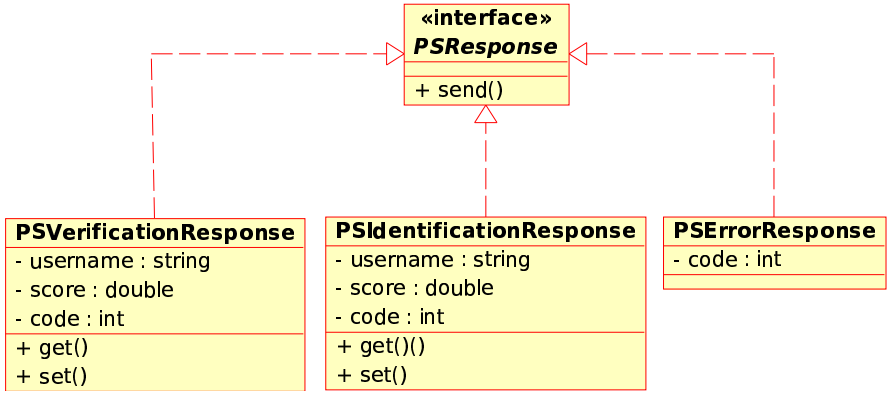

Figure 5.14: PS Responses.

On the other hand we have messages which originate in the kernel or other modules within the framework. These are called *notifications* and are described by the *PSNotification* interface. A notification sent to a PS does not trigger a response, but is used to inform the PS that some event happened within the framework, that this PS might need to know about. For instance, if a user

is identified and successfully authenticated without a PS having requested this authentication, the framework will send the relevant authentication information, in the form of a notification, to those protected systems that subscribe to the AS that triggered the event. The notification interface along with sample implementations are shown in Figure 5.15.

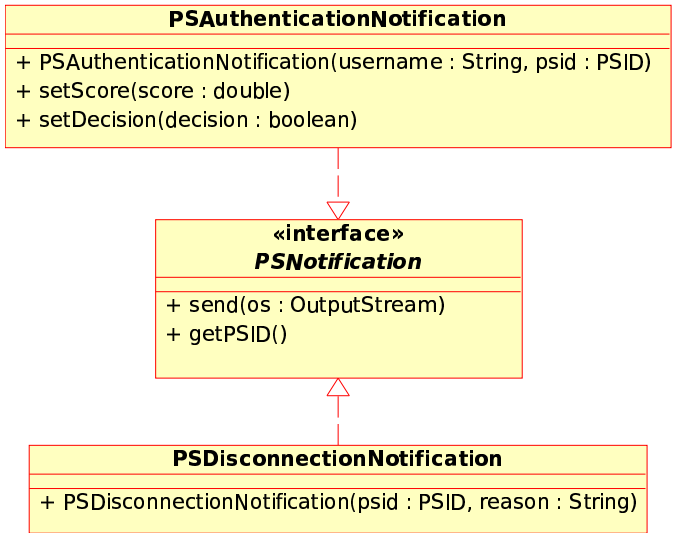

Figure 5.15: Notifications

Since the requests are constructed from data coming from the network connection, it will provide a *parse()* method which parses messages it reads from the input of the network socket. That is, a generic class will parse the protocol headers to determine the type of each incoming message, then it will forward the stream to the corresponding request implementation which will parse it and populate its attributes. As an example, a PSVerificationRequest will parse the username of the incoming request and store it in a local attribute.

Similarly responses and notifications send information to the protected system, which is why each response or notification object contains all the information required to construct a protocol message to be sent over the wire. The PSResponse and PSNotification interfaces both have a *send()* method, which are implemented such that they will construct protocol messages from their object values, and write them to an output stream. The send methods will typically be supplied with the output stream of the corresponding network socket.

5.3 The Authentication Systems Module

The *Authentication Systems Module* handles all communications with the authentication systems that are connected to the framework. The responsibilities of this module include managing network connection to the various authentication systems, sending and receiving protocol messages, listen for and process service requests from other framework modules and delivering notifications received from the authentication systems.

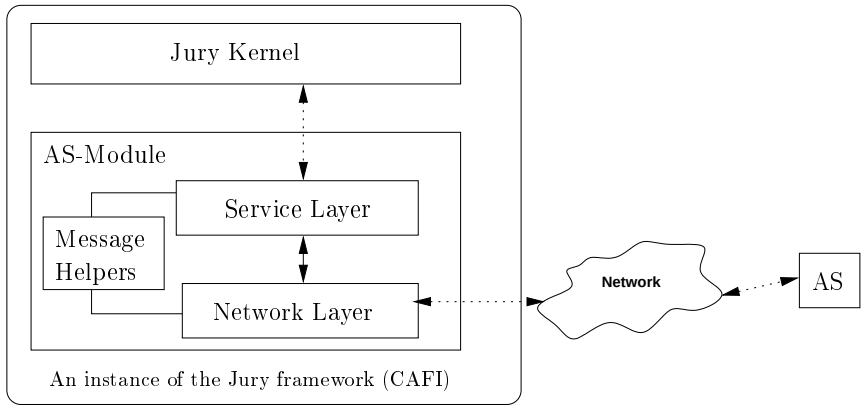

Figure 5.16: The layered AS-Module and its interactions with other parts of the system.

Just like the PS module, the AS module is divided into two layers, a *Service Layer* and a *Network Layer*, as shown in Figure 5.16. While the responsibilities of these layers are essentially identical to those of the PS module, we include them here for the sake of completeness. The Service Layer is responsible for all integration with other modules, while the Network Layer handles all network communication with remote authentication systems. This includes sending and receiving messages, relaying messages to the right authentication systems and synchronizing activities. The Network Layer also contains *Message Helpers*, similar to those of the PS-Module, which functions as the glue between the Service Layer and the Network Layer. It consists mostly of message classes that act as data types, in particular different types of requests, responses and notifications, but are also capable of parsing and sending messages. The Service and Network layers can communicate directly with each other, but only using primitive data types, e.g., integers and strings, and message helper data types.

Although the PS and AS modules are similar at a glance, they have some subtle differences. While the PS-Module is a consumer of Jury services, the AS-Module

is a service provider. It provides authentication services to the framework, and does not request any services from the framework. This has the effect that the direction of request, response and notification messages is the exact reverse of that from the PS-Module. The framework sends requests to a remote AS node, which replies with a response message. Similarly, the remote nodes send notifications of events to the framework, without expecting a reply. An example of a service provided by the AS-Module is when the framework asks an AS node to verify a specific identity.

Another notable difference between the AS and PS modules is that each AS node is registered to run in a specific mode, namely *identification*, *verification* or both. This means that some of the services offered by the AS-Module may not be supported on a specific AS node. As an example, requesting the AS-Module to verify an identity using an AS-node that is running in identification mode, will result in an error, since it does not run in verification mode.

5.3.1 The AS Service Layer

The Service Layer handles all interactions of the AS module with other Jury modules. The interactions with other modules take place over four public interfaces which encapsulate different types of interaction. Three of them deal with commands and service requests from the framework, while the remaining one handles notifications originating in the AS-Module. The interaction paths are shown in Figure 5.17.

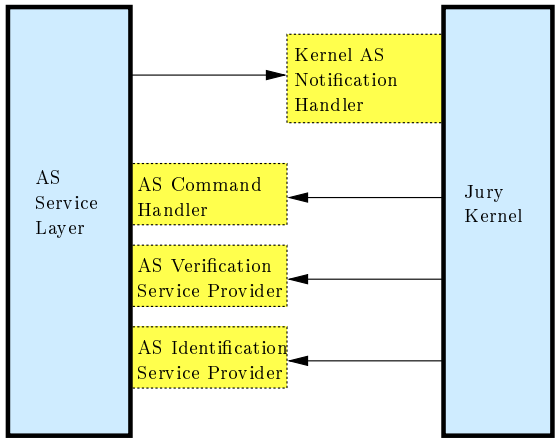

Figure 5.17: The interactions between the AS-Module and the Jury kernel over public interfaces.

The specifications of these interfaces are shown in Figure 5.18. The *ASCommandHandler* interface defines the commands that the framework can give to the AS-module, e.g., when the kernel commands the AS-module to close all active connections with remote AS nodes. The *ASVerificationServiceProvider* and *ASIdentificationServiceProvider* interfaces provide mode specific services. The *ASVerificationServiceProvider* interface is applicable only to those AS nodes which are configured to run in *verification* mode. All calls to this interface targeted at a AS node running in *identification* mode will result in an error being returned to the caller. The opposite holds true for the *ASIdentificationServiceProvider* interface, i.e., it only supports nodes running in *identification* mode and returns errors otherwise. If a remote AS node is configured to run in *both* modes, it supports all methods defined in the two interfaces and does not return errors caused by unsupported operations. Finally, the *KernelASNotificationHandler* receives notifications originating in remote AS nodes, and delivers them via the PS-Module to the event subscribers. The notification handling will be described further when we discuss Kernel functionality in Chapter 5.5 on page 82.

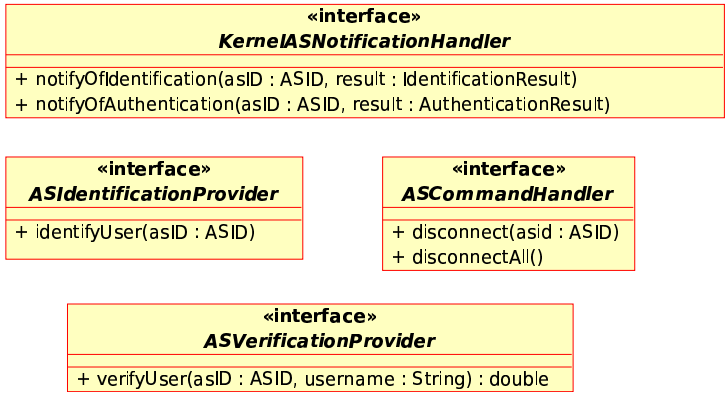

Figure 5.18: The public interfaces that define and constraint interactions between the AS module and the Jury kernel.

The Service Layer, shown in Figure 5.19, acts as a gateway to the AS-module. The kernel gains access to the services that are defined in the three interfaces described above, by having the *ASServiceFactory* give it an implementation instance of each interface. Each of the three implementation classes, i.e. those which names end with *Impl*, is responsible for creating the appropriate message helper objects from the parameters that are provided in the method call, and send them to the Network Layer. The network layer will forward the service request to the appropriate authentication systems, and return the result, which is further returned back to the caller, i.e., the kernel.

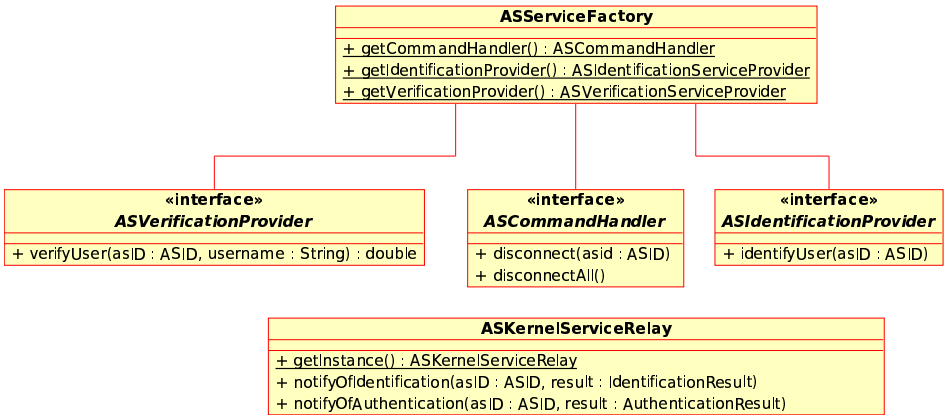

Figure 5.19: Classes that provide authentication services for the Jury Kernel, and a relay that delivers notifications from remote AS nodes, to the kernel.

For the other direction, i.e., interactions that are started by the AS-module, messages are sent via the *ASKernelServiceRelay* singleton class, which is almost identical to the *PSKernelServiceRelay* class of the PS module. It is responsible for forwarding notifications from the AS nodes, to the kernel. The methods for the notification handling are specified by the *ASNotificationHandler* interface, which is shown in Figure 5.20, along with its implementation, *ASKernelServiceRelay*.

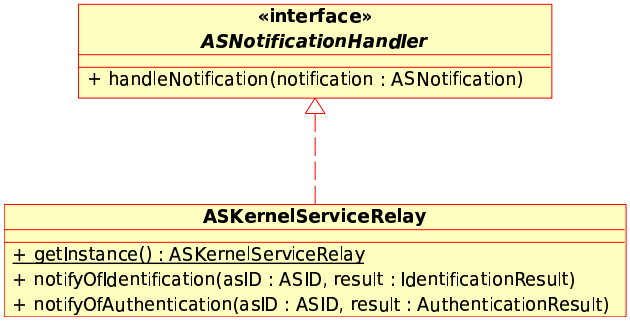

Figure 5.20: Request Processing.

5.3.2 AS Network Layer

The AS Network Layer is responsible for managing connections, as well as reading from, and writing to, network connections to the remote authentication

systems. Moreover, it keeps track of which authentication system is using each connection, so that requests can be sent out to the correct AS. Since the framework may use many authentication systems in parallel, the network layer requires a good deal of concurrency, which we will try to limit to the boundaries of the network layer. That is, the network layer synchronizes events in such a way that its interaction with the rest of the framework is not concurrent. Although this can cause the network layer to become a bottleneck, it greatly simplifies the project code and we have previously established in Chapter 4.3.1.2 on page 44 that optimization will not be performed until performance has been shown to be insufficient.

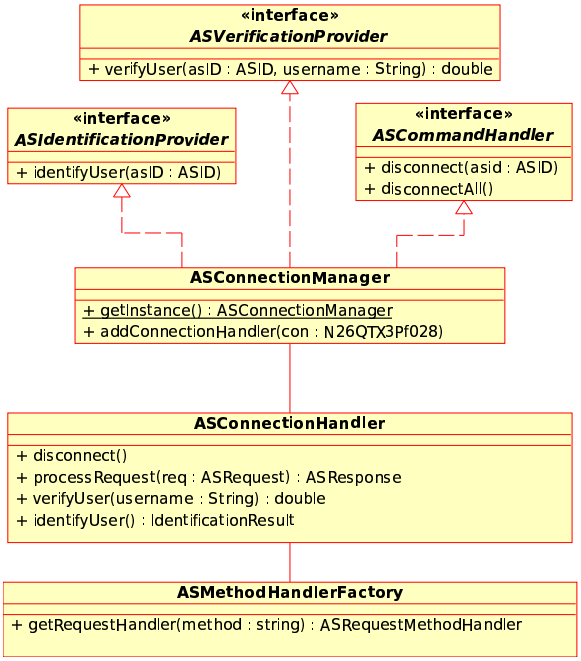

Figure 5.21: The AS Network Layer

As in the PS Network Layer described in Chapter 5.2.2, the AS Network Layer runs as a multithreaded server [58]. The *ASConnectionManager* is responsible for managing the group of active connections as well as their association with locally unique identifiers called *authentication system IDs* as assigned by the framework. These identifiers are widely used in the module, and can frequently be seen in the class diagrams as an attribute or parameter named *asID*. The *ASConnectionManager* is not directly managing network sockets, but instances of the *ASConnectionHandler* class. The *ASConnectionHandler* is responsible for managing a single connection, and carry out all operations for that connec-

tion, such as sending and receive data, and terminating the connection. The AS-module also checks incoming connection and matches it to the policy to assign a unique identifier to it, or to disconnect it if the address and source port of the connection do not match the policy.

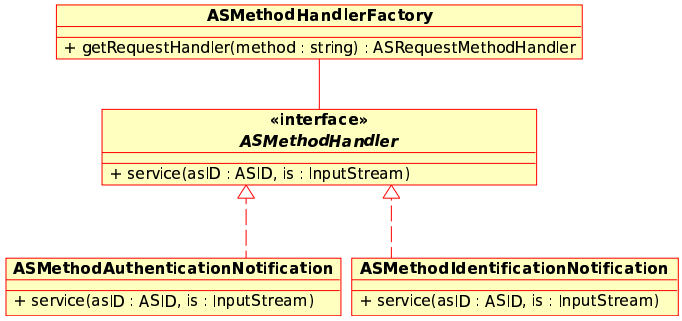

Figure 5.22: The AS Network Layer Method Handlers

Figure 5.21 shows the connection handling of the Network Layer. At the bottom we have *ASMethodHandlerFactory* which returns the appropriate processing class, depending on the message method. On top we have instances of *ASVerificationHandler*, *ASIdentificationHandler* and *ASCommandHandler*, all of which are implemented by the *ASConnectionManager* class. For each of the method calls described by the three interfaces, the *ASConnectionManager* retrieves the target ID of the authentication system, looks up which *ASConnectionHandler* is assigned to that ID, and forwards the call to that handler.

The method handler design shown in Figure 5.22, is similar to the method handlers of the PS module, except that it handles notifications and not requests. Requests are instead sent directly from the *ASConnectionHandler*, which also receives the response and delivers them via the service layer back to the kernel.

5.3.3 The AS Message Helpers

The Message Helpers consist of data types which we call messages, and classes that process them. An AS message consists of data that we wish to send to, or receive from, an authentication system. Each message class corresponds to a message type from the AS part of the Jury Message Protocol, which we will describe further in Chapter 5.7.

The AS-Module uses three types of messages. The requests are messages that are sent from the AS-Module to a remote AS node and the responses are the

answers received for these requests. The requests and response messages are encapsulated by the *ASRequest* and *ASResponse* interfaces respectively. The requests and responses for the AS module are shown in figures 5.23 and 5.24 respectively. The framework receives notifications from AS nodes running in identification mode. If the remote node is running in *both* modes, i.e., it supports both identifications and verifications, it can complete an authentication and send the result to the framework, as a notification. The notifications in the AS-module are handled by an *ASMethodHandler* object in the network layer, which receives and forwards these notifications to the framework kernel via the service layer. The Notification interface and sample implementation are shown in Figure 5.25, while the method handling is shown in context with other network processing in Figure 5.21.

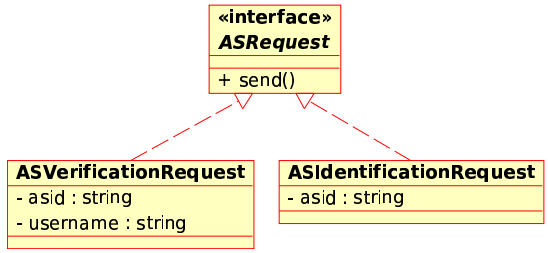

Figure 5.23: AS Requests.

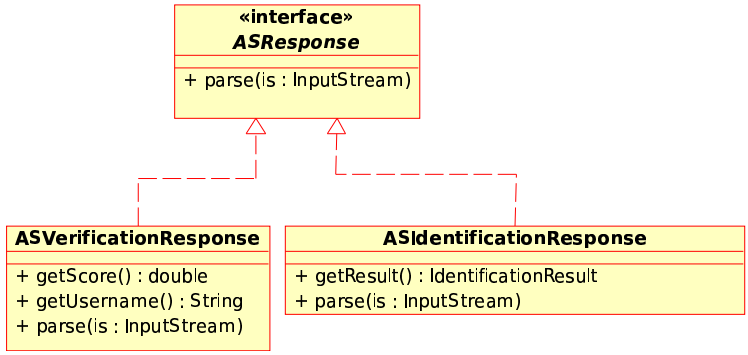

Figure 5.24: AS Responses.

Again, notifications are messages that are sent without requiring a response. In the PS module these notifications originated in the framework and were sent to the PS nodes. In the AS module this has been reversed.

Since the requests need to be sent out to their respective AS node, they provide a *send()* method, which writes message to a specified output stream in a format that complies with our network protocol. Our implementation will provide

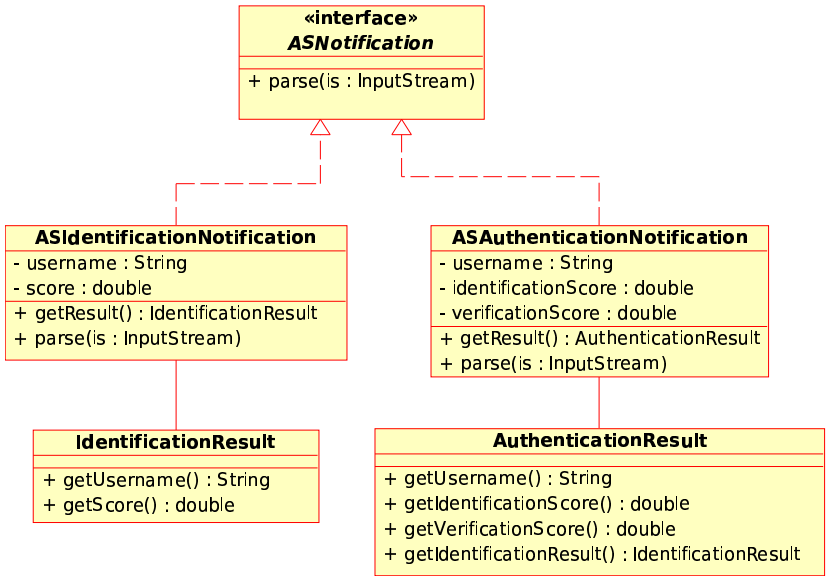

Figure 5.25: Notifications

the output stream associated with the network socket. The responses and notifications on the other hand provide a *parse()* method which reads a protocol message from the supplied input stream and constructs the respective message data type from it. These method are essentially identical to those found in the message helper sub-module of the PS module.

5.4 Score Combination Module

The *Score Combination Module* is both the simplest module of the framework and a very important one, since the core functionality of the framework is to provide more reliable authentication results by combining scores from multiple systems. The purpose of the module is to provide score combination while abstracting the details of which algorithm is being used.

The Module is organized into two layers, as shown in Figure 5.26. The service layer, as with the other service layers of our architecture, provides services to the rest of the framework. In particular, it abstracts the specific algorithms being used to calculate scores.

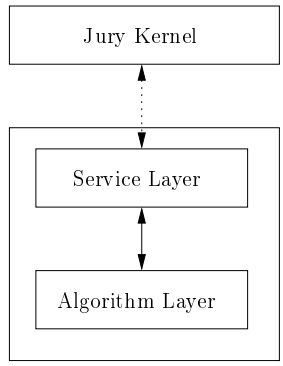

Figure 5.26: The layered Statistical-module.

The algorithm layer consists of implementations of various score combination algorithms. The separation between the service and algorithm layers abstracts the implementation-specific details away from the service, by encapsulating them in a Strategy Pattern [25]. When the kernel wants to combine a score for a PS, it requests the Service Layer to produce a result, without knowing which algorithm is being used in the process. The Service Layer will read the policy for the PS to determine which algorithm to use.

The module classes are shown in Figure 5.27. The abstraction is provided by the *ScoreCombinationAlgorithm* Strategy interface which all combination algorithms must implement. To use the algorithm, the kernel gets an implementation instance by calling the *combineScoresForPS()* method of the *ScoreCombinationFacade*, along with a collection of scores and specifying for which PS the scores are being combined for. Each PS can specify which algorithm they want to use in their policy. The facade implements a Plugin pattern [24], which reads the policy and dynamically loads the appropriate algorithm.

The most notable thing about the module structure is its simplicity. The Score Combination Module is only concerned with providing one simple service without using any services of other modules. The only foreseeable complications of the module are that more algorithms will be added. But since the service is completely unaware of which algorithms the module contains, it will maintain its simplicity for as long as the new algorithms can be abstracted with the *ScoreCombinationAlgorithm* interface.

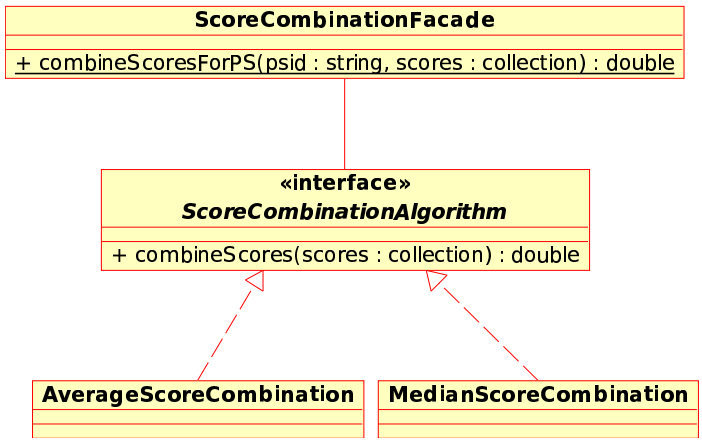

Figure 5.27: The Score Combination Module with two implemented algorithms, average and median.

5.5 The Jury Kernel

The Jury Kernel is, as the name suggests, at the core of the framework, and is responsible for orchestrating all operations which involve more than one module. Moreover it is responsible for administrative tasks, such as issuing commands to the other modules on request. For instance, if we want to shutdown the framework in a clean manner, the framework issues commands to the AS and PS modules to disconnect all connections to external systems.

The kernel implements the two interfaces specified for the PS and AS modules in Sections 5.2 and 5.3. Access to the interface implementations is given via a facade. This design is shown in Figure 5.28.

The *KernelPSServiceProvider* defines all services that are provided for the PS-Module, on request. That is, it provides methods to verify a certain user, or identify the present user. Both of which involve looking up which authentication system the PS subscribes to, request them to carry out the specific task, collecting their results, combine the scores and finally deliver a response back to the PS-Module.

The kernel is also responsible for handling incoming events from an AS. The *handleIncomingAuthentication()* and *handleIncomingIdentification()* methods of KernelPSServiceProvider receive an event from an AS, lookup up in the policy how the PS wants to handle the event, and finally, request services and combine

results from the other authentication system that the PS subscribes to.

The *KernelASNotificationHandler* interface is responsible for processing incoming events from remote AS nodes. Generally, this means looking up the subscribers of that event, and call the methods on the KernelPSServiceProvider for each of them.

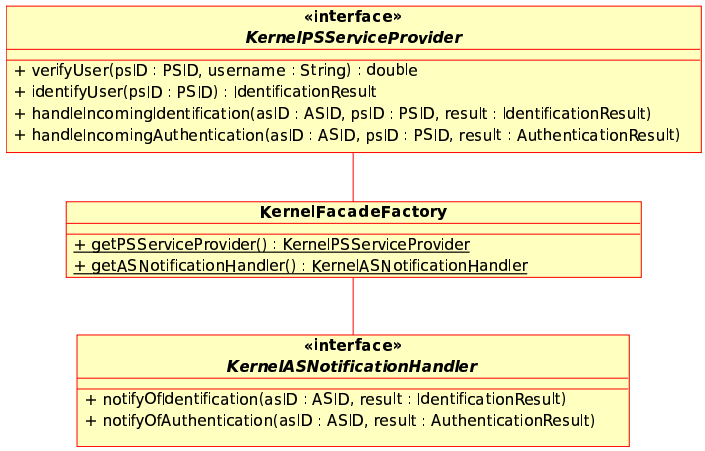

Figure 5.28: The Jury Kernel.

5.6 The Configuration and Policy Module

The *Configuration and Policy Module*, or Config-Module for short, is responsible for providing the rest of the framework with access to configuration parameters and policy specification. The configuration includes parameters such as on which ports the Network Layers of the PS-Module and the AS-Module, listen to, and where to store log files. The policy specification includes which protected systems and authentication systems are allowed to connect to the framework and what unique identifier to assign to these remote systems when they connect. Further, the policy defines the subscriptions for each PS, i.e., which authentication systems that PS uses to carry out its Co-Authentication operations. For each of the subscription entries, the policy specifies a weight which indicates a level of confidence that the PS has with regards to the AS. Finally, the policy specifies score combination parameters for each PS, such as which algorithm to use to combine scores, how to react to incoming authentication events, if it wants to receive a score, a decision or both, and in the case when a decision is

reported, a threshold to compare the Co-Authentication score to. The configuration parameters and policy specification are stored in a single file, an example of which is shown in Listing 6.4 on page 97.

The structure of the Config-Module is shown in Figure 5.29. The Data Access Layer provides access for the other modules to various configuration parameters. It is a static module which only provides read access to the configuration. By static we mean that there is only one active instance of each object, which all calling objects use. This means that if, say, the service and network layer of the AS module are reading from the same part of the configuration simultaneously, they will read from the same object. This simplifies synchronization, since only this single instance has to be updated when the configuration changes.

The *I/O Layer* is responsible for reading configuration and policy files from the disk, parsing them and populating the Data Access Layer with the corresponding parameters and values. Further, it periodically checks the files to see if they have been updated. If an update to a configuration or policy file is detected, the File I/O Layer updates the Data Access Layer accordingly.

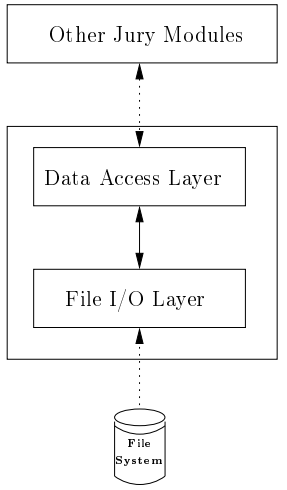

Figure 5.29: The layered configuration module and its interactions with other parts of the system.

The module structure is shown in Figure 5.30. The *ConfigurationFactory* hands out one read-only object for each of the other modules, and all configuration and policy specification takes place outside of the framework. Therefore we have designed the module in such a way that no part of the Jury system can write to the configuration or policy files, but gain read-only access to configuration attributes and values via four different *Reader* interfaces, i.e., one for each of the

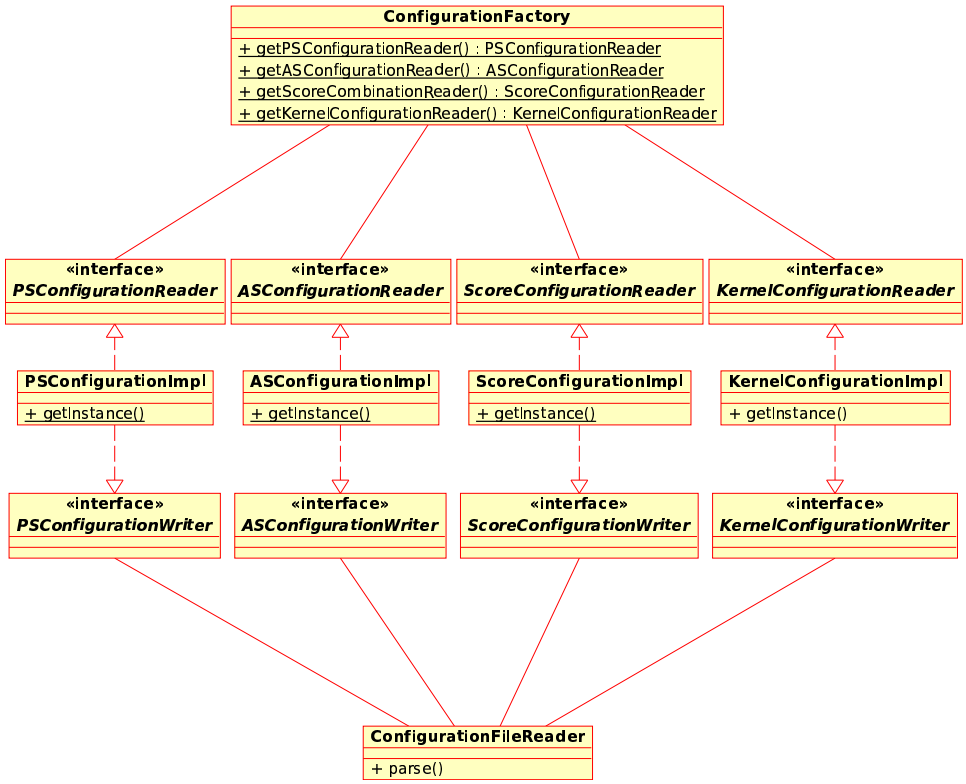

Figure 5.30: The Configuration Module. The *getInstance()* methods indicate that the implementations use a singleton pattern.

other modules. Each reader provides access to all the details that the particular module needs, e.g., the *PSConfigurationReader* provides the PS module with all the configuration and policy data it requires, which indicates that some configuration data can be accessible via more than one reader.

On the other end of the module we have the *ConfigurationFileReader* class. It is responsible for reading the configuration files from disk, parsing them and populating the readers via the four corresponding *Writer* interfaces. To achieve this, each pair of the Reader and Writer interfaces is implemented by a singleton class, which the Writer populates with data that the Reader then reads. The *ConfigurationFileReader* is also responsible for monitoring changes made to the configuration and policy files and update the readers accordingly.

In addition to preventing the framework to write to the policy file, the layered

design has an additional benefit of abstracting where the policy comes from. In the current implementation it is read from the file system of the CAFI, but it might not always be the most suitable approach. If we, at a later time, change the I/O Layer so that it reads the policy from a network resource, e.g., a URL, all other parts of the framework will be unaffected.

5.7 The Jury Message Protocol

The *Jury Message Protocol*, or JMP, is the protocol which all *Jury Network Nodes* use when they communicate with the CAFI. It defines all messages sent between the framework and the remote AS and PS nodes. For this proof-of-concept implementation, we have chosen to use a simple plaintext protocol similar to the traditional Internet protocols such as HTTP [23] and POP3 [43].

Each message of the protocol is split up into two parts, a block of headers the headers and a body. The headers are separated with a new line, and the end of the header blocked is marked with an additional line break. Similarly, each parameter of the body ends with a newline, and the message is terminated with an extra line break. Each parameter in the header block, or in the body, is specified by the name of the parameter followed by a semicolon and a space, after which the value is specified. For instance: '*method: PSVerifyUserRequest*'. Parameters within each part may appear in any order, i.e., the order within the header block is not important, and neither is the parameter order within the body.

5.7.1 The Headers

The headers are parameters that are common to all messages. In our current implementation we limit the protocol to a single header, namely the *method* header. It specifies the type of the message, which indicates which fields can be found in the body and how the framework should treat the message. The currently supported methods are described in Chapter 5.7.2.

5.7.2 The Methods

We will now describe all the message types that our framework supports. The format is as follows: The bold text, in the top-level of the list, denotes the

method name, it is followed by a short description of the message and its purpose, and finally we give a list of body parameters and describe how they are used.

- **PSVerificationRequest:** This method is used when a protected system wants the framework to verify a users presence. It is sent from the PS to the framework node.
  - **username:** The username for the user whose presence we wish to verify.
- **PSVerificationResponse:** This method is used as a reply to a *PSVerificationReqeust*. It is sent from the framework back to the PS which sent the original request.
  - **username:** The username for the user which the framework has authenticated.
  - **score:** The combined score for the authentication.
  - **decision:** Whether or not the authentication was successful.
  - **code:** The response code, as described in Chapter 5.7.3.
- **PSIdentificationRequest:** This method is used when a protected system wants the framework to identify the present user. It is sent from the PS to the framework node, and does not contain any body parameters.
- **PSVerificationResponse:** This method is used as a reply to a *PSVerificationReqeust*. It is sent from the framework back to the PS which sent the original request.
  - **username:** The username for the user which the framework has identified.
  - **score:** The combined score for the identification.
  - **decision:** Whether or not the identification was successful, i.e., that the identification score is above the threshold, as defined in the policy..
  - **code:** The response code, as described in Chapter 5.7.3.
- **PSErrorResponse:** This method is used when the framework was unable to process a request.
  - **code:** The username for the user whose presence we wish to verify.

- **PSAuthenticationNotification:** This method is used to notify a protected system about an authentication which was triggered within the framework, i.e., not as a result of a request from that PS. It is sent from the framework to a PS.
  - **username:** The username for the user which the framework has authenticated.
  - **score:** The combined score for the authentication.
  - **decision:** Whether or not the authentication was successful.
- **PSDisconnectionNotification:** This method is used to notify a PS that the framework is about to disconnect its connection, and why. It is sent from the framework to the PS which is to be disconnected.
  - **reason:** Text that describes why the connection is being terminated. This parameter may be empty.
- **ASVerificationRequest:** This method is used to request a single AS to authenticate a user. It is sent from the framework to an AS.
  - **username:** The username for the user which the framework has authenticated.
- **ASVerificationResponse:** This method is used as a reply to a *ASVerificationReqeust*. It is sent from the AS back to the framework.
  - **username:** The username for the user which the framework has authenticated.
  - **score:** The combined score for the authentication.
  - **code:** The response code, as described in Chapter 5.7.3.
- **ASIdentificationRequest:** This method is used to request a single AS to identify a present user. It is sent from the framework to an AS. No body parameters are sent with this request.
- **ASIdentificationResponse:** This method is used as a reply to a *ASIdentificationRequest*. It is sent from the AS back to the framework.
  - **username:** The username for the user which the AS has identified.
  - **score:** The match score of the identification.
  - **code:** The response code, as described in Chapter 5.7.3.
- **ASIdentificationNotification:** This method is used when an AS has identified a user and wants to notify the framework about it.

  - **username:** The username for the user which the AS has authenticated.
  - **score:** The identification score for the authentication.

- **ASAuthenticationNotification:** This method is used when an AS has identified a user and wants to notify the framework about it.
  - **username:** The username for the user which the AS has authenticated.
  - **identification-score:** The identification score for the authentication.
  - **verification-score:** The verification score of for the authentication.

5.7.3 Response Codes

The response codes are used to show if the request was successfully fulfilled, or if something odd came up. The codes are split up into blocks. Codes between 200-299 indicate successful operations, 400-499 indicate service errors and 500-599 indicate internal framework errors. The currently implemented codes and their meanings are as follows:

- **200** The request was successfully processed.
- **401** The username was not found.
- **402** Unrecognized header.
- **403** Method missing.
- **404** Unknown method.
- **405** Unknown method parameter.
- **406** Malformed message.
- **501** Service Unavailable.

5.7.4 Example messages

To further demonstrate how the JMP protocol is structured, we present a few example messages.

5.7.4.1 PSVerificationRequest

```
method: PSVerificationRequest

username: johndoe
```

5.7.4.2 PSVerificationResponse

```
method: PSVerificationResponse

username: johndoe
score: 0.56886
code: 200
```

5.7.4.3 PSDisconnectionNotification

```
method: PSDisconnectionNotification

reason: We are shutting down our servers for maintenance.
```

5.8 Patterns and Reusable Elements

There is little to gain from re-inventing the wheel, which is why we have used well established software design patterns where applicable. The aim of this section is to indicate where these patterns can be found in our code. We do not intend to fully describe the patterns here, but instead point to resources that explain them in detail, at the end of this section.

The most commonly used pattern in our framework is the *Singleton* pattern, which ensures that only one instance of an object is available at any given time. That means that during the lifetime of the singleton object, all external objects that call its methods are in fact using the same Object. Moreover, if no such instance exists, it will be automatically created when needed. Listing 5.1 shows a Java code snippet which demonstrates a singleton implementation.

```
public class SingletonExample {

  // Handle to the active instance
  private static SingletonExample instance = null;

  // Private constructor
  private SingletonExample() {

  }

  // This method is used to gain
  // access to the singleton instance
  public static SingletonExample getInstance() {

    if( instance == null ) {
        instance = new SingletonExample();
    }

    return instance;
  }
}
```

Listing 5.1: A sample Java implementation of a Singleton Class

To minimize dependency between modules, we use the Facade pattern, which provides a *"unified interface to a set of interfaces in a subsystem"* [25]. To limit redundant duplication of code, we make extensive use of the *Factory Method* pattern, which are methods that instantiate and return objects. That is, it manufactures objects, hence the name. A good example of this is the *AS-MethodHandlerFactory* class in the AS-Module, which creates method handlers based on the method. This allows us to keep all method-specific details inside the MethodHandler implementations, while the rest of the code base can treat all method handlers the same way. An example factory method from the *ASMethodHandlerFactory* class, is shown in Listing 5.2.

As we mentioned previously, the score combination module is implemented using the Strategy Pattern, where the algorithms share a method signature but differ in implementation. Our Strategy implementation is a perfect match of the more specific Plugin pattern [24]. This allows the framework to call the score combination algorithm, without having to know which particular implementation will be used. The *ASID* and *PSID* classes are examples of where we implement the simple Value Object pattern. Finally, we use some design patterns that have been built into the Java API, such as the Iterator pattern.

```
public static ASMethodHandler getHandler( String method )
  throws UnknownMethodException {

 if ( "ASIdentificationNotification".equals( method ) ) {
   return new ASMethodIdentificationNotification();
 } else if("ASAuthenticationNotification".equals(method)) {
   return new ASMethodAuthenticationNotification();
 } else {
   throw new UnknownMethodException( method );
 }
}
```

Listing 5.2: A sample factory method from the Jury framework

For a detailed description of all the patterns mentioned above, we refer to the Gang of Four [25] and Fowler [24].

Chapter 6

Implementation

In this chapter we describe parts of the Jury framework implementation that we find worthy of a more detailed discussion, along with some software engineering practices which we used to decide on how to implement these parts.

Although our work is a proof of concept, it is not to be confused with a throw-away prototype, where a piece of software is quickly hacked together with the sole aim of demonstrating what such a framework can do, after which the code is simply thrown away, never to be used again. Instead we have opted to focus our work on laying a solid foundation for future improvements and extensions. We have taken care to keep the code base clean, avoid duplicate code and build the framework in a well encapsulated, loosely coupled and modular manner, such that future work can focus on extending the framework rather than rewriting it. Further, we have extensively documented the code using standard Javadoc [4] syntax, which provides a good overview of the code base. Moreover, we have created a suite of unit tests to aid with the debugging process and to detect if new code to introduces bugs into existing functionality. These unit tests also serve as a demonstration on how many of the framework classes are to be used, and can therefore be used as a part of the documentation for new developers.

A good design goes a long way towards a well-structured modular framework, however, in the implementation phase, it is often very tempting to take a few shortcuts which blur the lines of separation and increase coupling between other-

wise independent units. One of our main objectives during the implementation phase, was to resists such temptations, and we believe that our focus on that objective resulted in a simpler and better implementation. In fact, when we implemented some parts of our design we noticed that it contained duplication of code. When this occurred we improved the design by removing the duplications, and then continued with our implementation based on the improved design.

The framework is written in the Java [3] programming language, which automatically takes care of our design guideline of independence from operating systems since it runs on a virtual machine which is available for most of the operating systems used today, including Windows, Mac OS, Linux and other UNIX variants.

6.1 Score Combination

The Score Combination Module uses Java Reflection to implement the Plugin pattern [24]. The reflection allows us to load an algorithm dynamically, depending on which PS we are performing the score combination for. Moreover, it allows for easy integration of externally developed algorithms without any modifications to the framework code, as long as they are implemented in the Java programming language.

To get an algorithm class to work as a part of the Score Combination module we have to do three things. First we need to make it implement the *ScoreCombinationAlgorithm* interface, which means that it has to implement a method shown in Listing 6.1. Second, we need to store it within the systems *Classpath*, which is a list of directories which the Java Virtual Machine will search for class implementations in. Finally, we must specify the fully qualified name of it in the policy for a PS as shown in Listing 6.2. When these conditions are fulfilled, we can load the algorithm dynamically as shown in Listing 6.3.

6.2 Configuration and Policy

The Configuration and Policies are defined in a single XML file, such as the one shown in Listing 6.4. To parse the file, the configuration and policy module uses the Apache Commons Digester [48] library, which is specifically designed to populate Java objects from XML documents. We created data objects which match the policy structure, and created Digester parsing rules to trigger the population of these objects. Finally, when the parsing is finished, the data from these

```
public interface ScoreCombinationAlgorithm {
/**
 * Combines the supplied scores.
 *
 * @param scores The scores to be combined.
 *
 * @return the combined score.
 */
  double combineScores( Collection<Double> scores );
}
```

Listing 6.1: The Score Combination Algorithm Interface

```
<!-- Remote Protected Systems -->
<protected-systems>
  <ps id="ps0" address="127.0.0.1" port="2000">
    <algorithm>
      mma.scorecombination.AverageScoreCombination
    </algorithm>
    <!-- Other policy fields removed for brevity -->
  </ps>
</protected-systems>
```

Listing 6.2: A policy specifying the score combination algorithm to be used

```
Class combinatorClass = Class.forName( strAlgorithm );
Class[] ifcs = combinatorClass.getInterfaces();
...
for ( int i = 0; i < ifcs.length; i++ ) {
  if ( ifcs[i].getCanonicalName().equals(
    ScoreCombinationAlgorithm.class.getCanonicalName() ) ) {
    // A valid algorithm
    algorithm = (ScoreCombinationAlgorithm)
        combinatorClass.newInstance();
    break;
  }
}
...
return algorithm.combineScores( scores );
```

Listing 6.3: The Reflection code – strAlgorithm is the value from the policy shown in Listing 6.2

objects is sent into the *Writers* of the configuration module. Thereafter, the other modules can access the data using their respective *Readers*, as described in Chapter 5.6 on page 83.

6.3 Exception Handling

Any software is subject to errors and exceptions, which are caused by errors in the code, unexpected input, or conditions that are hard to predict, such as hardware failures. In the Java programming language there are two types of exceptions, namely *checked* and *unchecked*, that are used to handle errors and unexpected program behavior.

Checked exceptions are those which we can react to in the code, by using combinations of *try*, *catch*, and *throw* statements. All methods which can throw a checked exception, must explicitly state so in their method signature. The benefit of this approach is that the compiler can check whether all exceptions are handled, which forces us to deal with all cases however unlikely. This is best described with an example: If a method *iCauseError()* throws an *ExampleException*, it must state so in its method signature. All methods that call *iCauseError()* must then either surround the call with a *try-catch* block, or also include it in their method signature. This is shown in Listing 6.5, where the *caller1* method deals with the error, while *caller2* passes it up the call stack. In Java, checked exceptions are always subclasses of the *Exception* class.

On the other end of the spectrum we have *unchecked* exceptions, which are errors that are not explicitly dealt with. These are often errors that are hard to foresee, and very difficult to react to, e.g., if a method receives a *null* object as a parameter, and then tries to call a method on that null object, the run-time environment will return an unchecked *NullPointerException*. Unchecked exceptions will normally move up the call stack, all the way up to the run-time environment, which will crash the program and print the exception stack trace to screen. In Java, unchecked exceptions are subclasses of either the *RuntimeException* or the *Error* class.

While the difference between checked and unchecked exceptions seems straight forward, it can present us with a dilemma. For instance, if we need to write to a file and get an exception about the disk being full, we can either throw an exception and try to handle it differently, or we can throw a run-time exception and crash the program. We have chosen to follow the exception handling guidelines from Sun Microsystems: "*If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything*

```
<?xml version='1.0' encoding='utf-8'?>
<mma-configuration>
    <defaults>
        <algorithm>
            jury.scorecombination.AverageScoreCombination
        </algorithm>
    </defaults>

    <logging>
        <dir>/var/log/jury/</dir>
    </logging>

    <!-- CAS Config -->
    <as-module>
        <listeningPort>3456</listeningPort>
    </as-module>

    <ps-module>
        <listeningPort>2345</listeningPort>
    </ps-module>

    <!-- Remote Authentication Systems -->
    <authentication-systems>
        <as id="as0" address="127.0.0.1" port="4000"
            mode="both" />
        <as id="as1" address="127.0.0.1" port="4001"
            mode="both" />
    </authentication-systems>

    <!-- Remote Protected Systems -->
    <protected-systems>
        <ps id="ps0" address="127.0.0.1" port="2000">
            <algorithm>
                mma.scorecombination.AverageScoreCombination
            </algorithm>
            <eventHandlingMode>
                combineVerifications
            </eventHandlingMode>
            <threshold>0.4367</threshold>
            <reportScore>true</reportScore>
            <reportDecision>true</reportDecision>
          <ASList>
              <as weight="1.0">as0</as>
              <as weight="1.0">as1</as>
          </ASList>
        </ps>
    </protected-systems>

</mma-configuration>
```

Listing 6.4: A sample configuration and policy file

```
public void iCauseError throws ExampleException() {
    throw new ExampleException();
}

public void caller1() {
  try {
    iCauseError();
  } catch( ExampleException e ) {
    // Error handling
  }
}

public void caller2() throws ExcampleException {
  iCauseError();
}
```

Listing 6.5: Examples of checked exceptions

to recover from the exception, make it an unchecked exception." [57].

There are some places within the Jury implementation where we use unchecked exceptions for statements that should be unreachable, but cannot be guaranteed to be so. For instance the *verifyUser* method of the *ASConnectionHandler* class in the AS-Module Network Layer, checks the mode of the AS, i.e., whether it is identification, verification or both. If the AS mode does not match any of those three values, then there is an error in the configuration. Since the framework cannot fix the configuration, we choose to crash the program rather than run it in an inconsistent state. By crashing we are also notifying the administrator of the configuration error. While it can seem a bad idea to crash the program on purpose, it is often considerably better than the alternative of leaving the program running in an inconsistent and unpredictable state.

6.4 Constants

Throughout our code, there are some constants that are used in multiple classes and in some cases even multiple modules. One of our methods for achieving maintainability and simplicity in our framework is to bundle together related constants and store them in one place, where all the other classes can easily access them. The protocol response codes are a good example of how we use this method. Every single class that constructs a response message, sets its response code as described in Chapter 5.7.3 on page 89. While it may be tempting to

set these codes using their integer value, e.g., *message.setCode(200)*, it causes related numerical constants to be scattered throughout the code base. As a result, should we want to change the code of *OK* to, say, 250, we would have to find every occurrence in the code where the message code is set to 200, and change it to 250. This is a clear violation of the DRY principle.

To solve this, we bundled related constants together, in one place. In the case of response codes, we created a class which has the sole purpose of containing response code constants, as shown in Listing 6.6. Now, all code which sets the response code to 200, will use the *ResponseCodes.OK* constant instead of using the integer directly. Consequently, if we want to change the OK code to a value of 250, we now only have to modify the *ResponseCodes* class.

```
package jury.common.protocol;

/**
 * Contains all supported response codes.
 * All response codes of messages should be
 * set using these constants.
 */
public class ResponseCodes {

    // 200 All is Good
    public static final int OK = 200;

    // 400 Service and Messsage Errors
    public static final int USER_NOT_FOUND = 401;

    public static final int UNRECOGNIZED_HEADER = 402;

    public static final int METHOD_MISSING = 403;

    public static final int UNKNOWN_METHOD = 404;

    public static final int UNKNOWN_METHOD_PARAMETER = 405;

    public static final int MALFORMED_MESSAGE = 406;

    // 500 Resource Errors
    public static final int SERVICE_UNAVAILABLE = 501;
}
```

Listing 6.6: The response code constants

6.5 Addressing Requirements

We have implemented all the high priority requirements, and a good part of the medium requirements from Chapter 4.4 on page 48. Table 6.1 shows the high priority requirements and the medium priority requirements which we implemented, along with the modules which participate in satisfying them.

Requirements	Modules involved
A-1.1 – *A-1.3*	Config
A-1.4	Config, Kernel
A-2.1	Config, AS
A-2.2	AS
A-2.3 – *A-2.4*	AS, Kernel
A-3.1	Config, PS
A-3.2	PS
A-3.3	PS, Kernel, Config, AS, Score Combination
A-3.4	AS, Kernel, Config, PS
A-4.1 – *A-4.3*	Score Combination
A-4.4	Kernel, Config
A-5.1 – *A-5.7*	Config
B-3	PS, Config, Kernel
B-4	Config, Kernel
B-5	Config, Score Combination
B-6	Config, Score Combination
B-7.*1*	Config, Score Combination

Table 6.1: High and Medium Priority Requirements and where they are addressed

Chapter 7

Evaluation

In this chapter we evaluate our work, both in terms of performance and in terms of applicability. The performance evaluation is in the form of rough estimates to see if the Jury framework performs sufficiently well to be applicable in real scenarios. We evaluate the applicability of our work by describing how existing systems can be adapted to Co-Authentication, and how some of these can be integrated into the Jury implementation. Finally, we discuss which types of attacks against the framework are most likely to occur, and suggest how to mitigate these attacks.

7.1 Performance Evaluation

There are many things that can influence the performance of a Jury network, some of which are outside our control, such as network performance, and the performance of participating authentication systems. A slow network can have a significant influence on how long it takes to gather authentication data from various systems and sending authentication information to protected systems. Similarly, if an authentication system takes a long time to authenticate a user it has an effect on the overall performance of the Jury network. For instance, if a fingerprint recognition system is asked to verify a users presence, it will have

to prompt the user for his fingerprint, wait for the user to comply, scan the fingerprint, compute a score for it, and finally send the result back to the CAFI.

We have conducted an experiment to evaluate whether the performance of the Jury framework is acceptable, i.e., that is fast enough to be applicable in real authentication scenarios. The aim of the experiment is to see how long it takes the framework to process a verification request, and our interpretations of the results should inform us of whether the overall framework performance is acceptable for the expected use of the framework. The experiment was conducted on a Dell Latitude D610 laptop with a 2.0 GHz Pentium M processor and 1 Gigabyte of memory, running Ubuntu Linux 7.04 with a typical desktop installation. Since the operating system is running many other services, our measurements may be influenced by these unrelated processes, but this is acceptable for the purpose of producing rough estimates. In particular, these processes can only make the measurements worse, so if our results are positive, we can safely ignore their interference.

To estimate the time it takes the framework to process a request from a PS, communicate with remote authentication systems, combine the results and reply to the PS, we created a simulation of a protected system that sends out requests and measures the time it takes for the framework to reply. To process the request, we use two simulated authentication systems, that receive a verification request, and immediately create a response with the username from the request and a random score, which they then send back to the CAFI. Each measurement is recorded using the *System.currentTimeMillis()* method of the Java language, which has a resolution of 1 millisecond on the system we measured on.

Figure 7.1 shows the results of running the above measurement setup with 100.000 requests. The bold lines between 0 and 3 milliseconds are not actual lines, but clusters of points since most requests are processed in less than 3 milliseconds.

Figure 7.2 shows the most commonly observed times from the same measurement, and how many requests were measured with these values. The the most common processing time by far, is 1 millisecond, with 0 milliseconds a close second. Of course, it took over 0 milliseconds to process these requests, but since the granularity of our measurements is limited to a 1 millisecond resolution, the 0 ms measurements indicate that it took less than one millisecond.

The total distribution is shown in Table 7.1 on page 104. The arithmetic mean of the measurements is 0.94 milliseconds, with a standard deviation of 0.59 milliseconds. These results indicate that the performance of the Jury framework is sufficiently good to be used in real authentication scenarios.

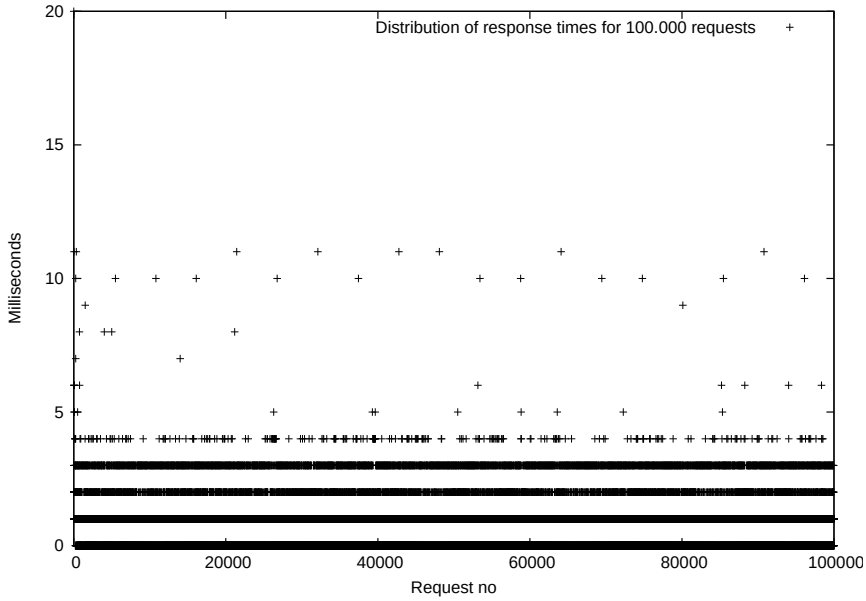

Figure 7.1: Processing times for 100.000 requests

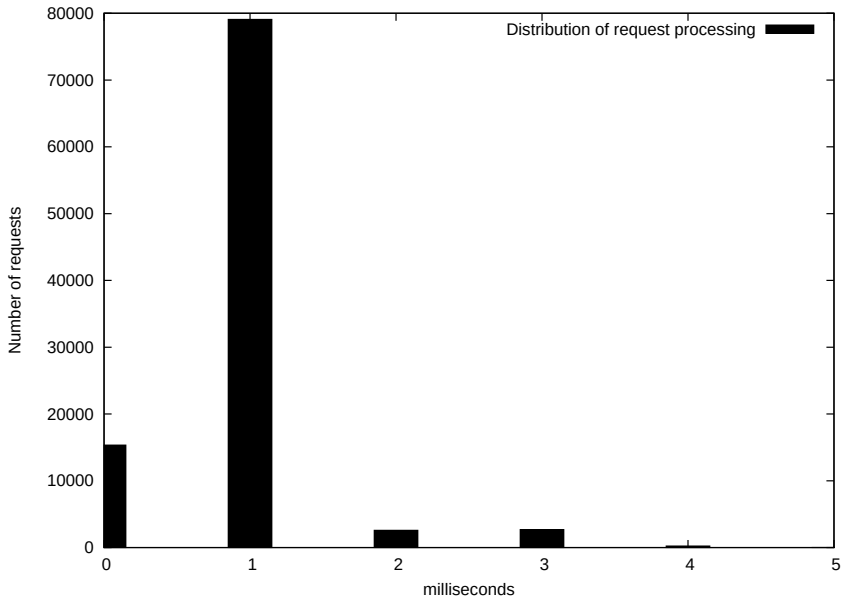

Figure 7.2: The most common processing times of the 100.000 requests

Time (ms)	No. of requests	Percentage
0	15348	15.348%
1	79082	79.082%
2	2587	2.587%
3	2702	2.702%
4	231	0.231%
5	11	0.011%
6	9	0.009%
7	2	0.002%
8	4	0.004%
9	2	0.002%
10	12	0.012%
11	8	0.008%
20	1	0.001%
45	1	0.001%

Table 7.1: Summary of total processing times for all the requests

7.2 Adapting Other Authentication Systems

Throughout the thesis we have argued that existing authentication systems can be adapted to a Co-Authentication scheme. We will now give a short description of how to integrate an anti-fraud credit card system into Jury, as well as a more general description on how to move password mechanisms from binary scheme and into to a probabilistic scheme.

7.2.1 Credit Card Payments Revisited

In Chapter 3.1.1 we introduced a scenario where Co-Authentication was used for more flexible fraud auditing, by allowing weaker technologies, i.e., a magnetic stripe card, to be used for smaller transactions which fit the customers profile. We will now describe that scenario in further detail and show how it can be implemented in Jury. Note that the probabilities, amounts, match scores and profile information presented here are only for demonstration purposes and may not reflect the numbers and profile data as used by the credit card issuers. The reason for this is that we do not have access to authentic data from credit card issuers.

The scenario is as follows. Charles is the cardholder of a credit card issued by the Example Credit Card Company (ECCC). At the points-of-sale, each transaction

made with an ECCC card is authenticated by an electronic transaction manager, which accepts or denies the transaction based on the card technology used and how well the purchase fits the cardholders profile. We consider an example where the transaction manager is processing a $600 transaction from Charles's card, originating in Austria.

The following information represents ECCC's profile of Charles. He makes purchases which typically range from $25 to $300. The rate of these purchases averages around one purchase a day, and the card is always present at the point-of-sale, i.e., Charles does not use his credit card for online or phone orders. His purchases mainly take place in Scandinavian countries, but he occasionally uses his credit card in Germany and the UK.

ECCC divides the profile validation into three separate segments, namely *amount*, *location* and *purchase rate*. The amount segment compares the amount in the transaction request to the amounts that are normally charged to the card. The location segment compares the transaction origin to the geographical location where transactions are normally made. Finally, the purchase rate checks for significant increases in the rate of purchases, which could indicate a stolen card.

We can integrate these components with Jury by creating three JINs, one for each profile segment. Each of the JINs takes transaction data as input, and compares it to the profile segment and returns a match score in the [0, 1] range, indicating how well the transaction fits the profile segment. A score of zero means that it did not fit at all where as a score of one indicates a perfect match.

They add a fourth JIN, which checks the card type and returns a confidence score. ECCC only provides two card types, namely magnetic stripe cards and smart cards which are also known as Chip & PIN cards. They have assigned static confidence values to the two types, where magnetic stripe cards have a value of 0.7 due to the high rate of fraud, but smart cards have a value of 0.95, since they are assumed to be much more resilient against fraud. Note that since these scores are static, they can be implemented directly into the JIN, i.e., the JIN does not have to be connected to any other ECCC system to perform these checks. Finally, the ECCC has decided to combine the scores using an arithmetic average function and have set the acceptance threshold to 0.8. The Jury setup is shown in Figure 7.3.

The profile checker for the amount segment scores low, or 0.65 since it is considered a small anomaly. Since Austria is a neighbor of Germany, where Charles regularly uses his card, this is considered a better fit than more remote countries, so the score is 0.7. The usage rate of the card is well within normal use, so the rate profile check scores a high 0.93. Since ECCC uses an arithmetic

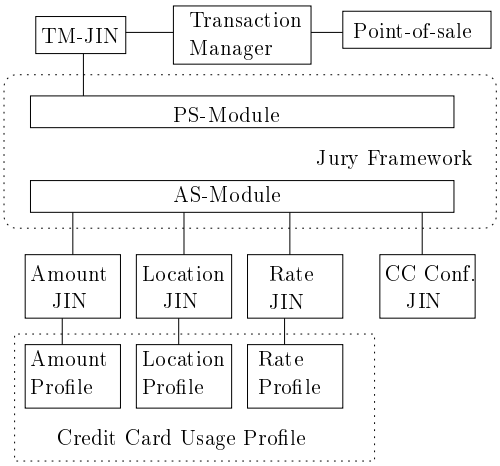

Figure 7.3: Integration the ECCC Transaction Manager and Profiles with the Jury framework

average score combination, the formula for the final score is:

$$\frac{\text{amount score} + \text{location score} + \text{rate score} + \text{confidence of card}}{4}$$

Putting in the values described above we get the following result for a magnetic stripe purchase: $(0.65+0.7+0.93+0.7)/4 = 0.745$ where as if the transaction is made using a smart card the score will be: $(0.65+0.7+0.93+0.95)/4 = 0.8075$. If the acceptance threshold is 0.8, the transaction will be accepted if it is done with a smart card, but not if it is made using the magnetic stripe.

7.2.2 Adapting Passwords to Co-Authentication

There are at least two ways of converting password protection mechanism to threshold based systems. The first one, is to assign password a static score at the time it is set, based on estimates of how hard it is for common password crackers to break it. We have mentioned this approach previously in the thesis and the work we have done so far in this area is presented in Appendix A.

The other approach is to allow passwords input by users, to contain typographic errors. We mentioned previously that one of the problems with pass-phrases is that, due to their length, people experience a higher input error rate compared to shorter password. By allowing typos, we can compute a score based on the

errors that the input contains. For instance, if the password *'For fun and Profit'* is entered as *'For fun and profit'*, it differs only by the capitalization of a single letter, and therefore should compute to a high, but not perfect, score.

This is easy to do if the passwords are stored in plaintext, but that is a very unfeasible design. Therefore we need a method that can compute similarity scores, and yet store passwords as hashes. One way to achieve this, is to change the matching mechanisms such that if the password that the user enters is incorrect, we try to guess the correct password by generating multiple variations on it, based on keyboard layout and common typographic errors. For each variation we compute a hash, compare it to the stored hash value, and if we obtain a match we assign it a score based on how much the input password and the stored password differ.

The method we have described here is in many ways similar to the methods used by password crackers in offline guessing attacks, and we plan to implement such a similarity score system as a part of our future work on password ranking and further study of common password crackers.

7.3 Attacks against Jury

The Jury framework is, like any other computer system, a possible target for malicious attacks, and in this section we will consider likely categories of such attacks. Our reason for including this discussion here is twofold. First, we acknowledge the possibilities of these attacks and that further work needs to be done with regards to hardening the framework, and second, we identify likely attack methods, so that we obtain an overview of where the most likely vulnerabilities are, and where the hardening process should focus.

7.3.1 Attacks against the central Jury framework

Throughout this thesis we have emphasized that a Co-Authentication framework can provide more reliable authentication services, by combining multiple authentication systems. This means however, that the framework instance (CAFI), which computes and delivers the overall score, becomes a single point of failure. If an attacker is able to tamper with this node, he may be able to override all authentications and provide the scores he likes. For instance, if an attacker who has gained control of a CAFI, is logging in to a system, which via the CAFI uses ranked passwords, a face recognition system and a signature

recognition system, he can bypass all of these authentication systems by making the CAFI return a high overall score. This risk however, applies to computer systems in general, i.e., if an attacker can gain control of a critical system, he can, more or less do whatever he wants. Therefore we will note provide further discussion of attacks where the attacker has full control of the CAFI.

7.3.2 Attacks against Authentication Systems and the Jury Network

If an attacker gains control of a single authentication system, or its JIN wrapper, he can send bogus messages and results from that system. The effect of such an attack depends on the overall Co-Authentication infrastructure and the Jury Policy. For instance, if the policy specifies that all incoming events should be verified by a number of other authentication systems, then the attacker is much less likely to cause real harm, given that he only controls this single system. Moreover, if the attack is discovered, the authentication system can quickly be either removed from the Jury policy, or its influence can be scaled down by decreasing its weight.

If however, the attacker is able to inject messages anywhere in the Jury network, he is able to forge messages from all the authentication systems, and can therefore manipulate any authentication scenario. For instance, if he starts by sending out an authentication event from one AS, which the CAFI reacts to by sending out verification requests to other authentication systems, the attacker can intercept those requests and send back fake responses. An even easier attack is to fake an authentication event from the framework, and send it to the PS which the attacker wants to gain access to.

These types of attacks can be mitigated by introducing device authentication, and by adding cryptographic protection into the JMP messages to reduce the risk of spoofed messages. Moreover, nonces can be added to the message protocol, to prevent reply attacks. Optionally, we can implement solutions that are independent of the framework, such as encrypting all the network traffic between the nodes of the Jury network, for instance by using IPSec [32].

7.3.3 Attacks using Rogue systems

An attacker might try to introduce a rogue authentication system and connect it to the Jury network. The framework currently includes very basic protection against these attacks, by only allowing remote systems that are specified in the

Jury Policy, and to reject all connections that do not match the IP address and source port as specified in the policy. If however, the attacker can bind to such an address and source port, his system will be treated as a legitimate authentication system. This attack can also be mitigated by device authentication and other mechanisms as described in the previous section.

7.3.4 Attacks using the Jury Message Protocol

In addition to faking messages, or replaying previous messages, an attacker might try to craft a message with a malicious payload, i.e., one that causes the protocol handler in the CAFI to execute malicious commands. While the above mentioned steps to secure the message protocol itself make this harder for an outside attacker, it is still possible to launch this type of attack from a legitimate remote system, i.e., one that is defined in the Jury policy. It is not granted that all nodes in the Jury network are managed by the same principals, and therefore an administrator of a remote AS might have insufficient access to alter the configuration or policy of Jury. He will however, have sufficient access to be able to send messages to the CAFI, with or without cryptographic protection. The main method of mitigating the risk of malicious protocol messages, is to carefully analyze and evaluate the part of the framework that deals with parsing and processing JMP messages.

Chapter 8

Conclusion

The process of authentication is not entirely black and white, but nevertheless, the output of authentication systems is generally treated in such binary terms. We believe that by explicitly quantifying the weaknesses and error rates, we can take them into account and produce more reliable results. By evaluating authentication results as different shades of grey, we can paint a more accurate picture of the authentication process.

We have proposed the concept of Co-Authentication and illustrated some of the benefits of evaluating authentications by taking the explicit uncertainties of individual authentication systems into account. Moreover, we have presented *Jury*, a generic Co-Authentication framework which can be integrated with essentially any authentication system or protected systems. The Jury framework is flexible enough to support system specific policies on how to perform authentication and how to react to incoming authentication events. Moreover, Jury allows easy integration of custom score combination algorithms, which can be plugged into the framework without having to alter the Jury code base.

We have shown that traditional binary authentication systems can be adapted to this probabilistic scheme and that by introducing intermediate wrapper nodes, this adaption can be made without modifying existing legacy systems. We believe that this fact is critical to the usability of our framework, since it allows organizations to slowly migrate to the Co-Authentication scheme, without hav-

ing to introduce major changes into their infrastructure.

8.1 Future Work

There are several topics which we wish to explore further as future work. We have previously mentioned several of these in Chapter 4.4.3, most of which require considerable work and extensions to the framework. There are however a few things which we wanted to do, but did not find time for during the course of this thesis project. In particular this work includes further evaluation and testing of the framework.

We plan on integrating several authentication systems into a Jury infrastructure to control physical access to an office, via an electronic lock. These systems should preferably include biometric systems, binary authentication systems and non-intrusive authentication systems which do fall outside the category of biometrics, e.g., proximity-based user authentication using a token [17]. This work will mostly consist of finding and installing suitable authentication systems, adapting binary authentication systems to a threshold-based scheme, and writing JIN wrappers for them. By creating such an infrastructure we can evaluate our work in more detail, in terms of performance, reliability and usability.

Further we want to experiment with different score combination algorithms and evaluate how their performances compare in different circumstances. Finally, we are interested in integrating Jury with traditional access control frameworks for further evaluation.

Appendix A

Ranking Passwords

This appendix includes our initial study on assigning ranks to passwords. These ranks can then be used as a part of a Co-Authentication. This report was written in the first weeks of our thesis work, and is included here as a work in progress.

A.1 Abstract

Passwords can be assigned a level-of-confidence based on how well they withstand attacks of common password crackers. We present an implementation of a password ranking program and discuss the value of such a tool.

A.2 Introduction and Motivation

Passwords are generally a weak authentication method. They can be shared, guessed and stolen. In many cases they are also easily broken by password crackers, which use available information such as the users login name and full name to attempt to recompute the password hash. For the purpose of authorization, most password authentication mechanisms produce a binary result, i.e.,

the claimed identity is either accepted or it is not. In contrast, other authentication mechanisms, such as biometrics, give a result along with a match score. For example: *The scanned fingerprint gave a 78% match when compared to the stored fingerprints of the claimed identity.* Biometric systems typically have a match score threshold which determines how good the match has to be in order for the sample to be accepted. The match score indicates how well the produced sample matches the templates stored in the system.

A biometric match score for a genuine sample, e.g. a sample which truly belongs to the claimed identity, is related to the sample quality. With all other things being equal, a very high quality image from a fingerprint scan is expected to return a result with a higher match score than a low quality image. This approach can be applied to passwords by substituting the sample quality with password strength. This requires a method to quantify password strength.

The strength of passwords can vary greatly. One method of assessing the strength of a password is to measure it against a good password cracker. A password cracker guesses password based on login names, real names, wordlists (dictionaries) and finally by brute-force character combinations. The time it takes a cracker to find a password given its hash and related information ranges from under one second to what can, for all practical purposes, be considered as infinity. If we can quantify the strength of a password, we can use it as a static match score for the given password.

There are various applications which can benefit from quantified password strength, e.g., an access control system can grant a user different degrees of access, based on the strength of his password. We can also combine the strength of the password with match scores obtained from other systems, such as biometrics. If we have multiple match scores for the same authentication instance, we can apply statistical methods to decide whether or not to authenticate the claimed identity.

In this report I outline my analysis on password crackers and describe how such an analysis can be used to implement a password ranking mechanism. In addition I shortly describe a ranking implementation written in Ruby [11] which given a password and a set of user data, such as username and real name, assigns a strength value based on estimates of how long it takes a password cracker to crack that password.

A.3 Analysis

A password cracker generates a sequence of guesses and computes a hash for each of those guesses. The generated hash is then compared to the hash from the password file. John the ripper (JtR) [22] is a popular password cracker that supports various different hash functions and password file formats. It is tweaked for speed and has been in active development for over 10 years. A run of JtR typically consists of one or more of its three built-in modes. *Single mode* checks for variants of information, such as the login name, full name and home directory, the weakest passwords are typically found in this mode.

The basic guess-mechanism of JtR's single-mode is as follows: For a list of words from the users login name, real name and home directory, JtR runs a series of methods on each of these words, computes the hash for the result and compares it to the hash from the password file. These methods include (but are not limited to):

- Substrings and truncations
- Case conversions: uppercase lowercase and capitalization
- Duplication: "asdf"->"asdfasdf"
- Reversal: "asdf"->"fdsa"
- Grammatical mangling: pluralization etc.
- Popular character substitutions such as a->4
- Pre- and suffixes
- Word pair rules
- Rotations: "asfd"->"dasf" etc.
- Case toggles: "asdf"->"asDf" or "AsDf" etc.
- Keyboard shiftings, f.ex. shift all letters one step to the right on the keyboard

Wordlist mode checks for variants of words from a supplied wordlist. The strength of passwords found in this mode is typically stronger then those found in single mode, but weaker than those found by incremental mode. *Incremental mode* is JtR's final and most thorough mode. It can try all possible character combinations and is not guaranteed to terminate unless configured to limit the

range of passwords [5]. The incremental mode can be described as a frequency driven brute-force. This mode uses trigraph frequencies to increase the chance of finding a password within feasible time limits.

By evaluating the strength of a password at the time it is set, we gain the advantage of having the plaintext, without having to store it permanently. Having the plaintext allows us to skip hashing and focus on comparing the plaintext to strings that a password cracker is likely to guess.

The design of the password checker is based on my analysis of the default configuration for JtR's single- and wordlist modes. While my implementation is not a one-to-one mapping of JtR's checks, the difference is only marginal.

A.4 Implementation

We have implemented a simple checker which corresponds to JtR's single- and wordlist modes. Given a username and a real name, it tries various word mangling methods and checks if they match the password. The implementation is split into three classes as well as several additions to Ruby's standard *String* implementation.

Two of the classes correspond to JtR's single-mode and wordlist mode respectively. Each of these compares the plaintext password with the default rewriting rules for the given mode. Our additions to the *String* class consist mainly of comparison methods and rewritings. One example of an added string comparison method is *equals_ignore_case?*, which as the name suggests compares two strings in a case-insensitive manner. Another example is *equals_numeric?* which tries different numeric rewritings, such as substituting all 'A's with '4's, etc.

Finally we have a separate *PasswordRanker* class which given a mode-configuration, a username, a plaintext password and the contents of the *GECOS* field (typically full name), runs a series of tests on the password, using the Single-Mode and Wordlist classes. Based on the result it calculates a rank, where $0.0 \leq rank \leq 1.0$. This rank corresponds to the match score of biometrics.

The ranking is configured as a chain of modes. Each mode is assigned a range and an offset. The range expresses the relative range of ranks the mode will produce. If the score is above zero, we add the offset to the rank and return the result. Otherwise we continue to the next iteration.An example of a configuration is shown in Table A.1.

Iteration	*Min*	*Max*	*Range*	*Offset*
Single Mode	0	0.2	0.2	0
Wordlist Mode (password.lst)	0.3	0.5	0.2	0.3
Wordlist Mode (all.lst)	0.5	0.7	0.2	0.5
Other (incremental)	1.0	1	1	0

Table A.1: This table shows a configuration where the single mode gives ranking ranging from 0-0.2, a wordlist analysis of the *password.lst* wordlist ranks from 0.3-0.5 and a wordlist analysis of *all.lst* rank between 0.5-0.7. All passwords which are not ranked by these three modes are assigned the value of 1.0. The Min and Max fields are shown for demonstration purposes only, they are not a part of the actual configuration. Note that there are no values between 0.2-0.3 or 0.7-1.0.

When a mode returns a rank which is above zero, the rank is treated as relative to that mode. That is, the single mode can return $0.0 \leq relative\_rank \leq 1.0$. The overall rank is then calculated withe following formula: $(relative\_rank * range) + offset$. As an example, if the wordlist mode for *password.lst* returns a relative score of 0.7 then we apply the formula: $(0.7 * 0.2) + 0.3 = 0.44$ which is the overall rank.

A.5 Summary and Future Work

In this report we have described a short analysis of the workings of John the Ripper, a popular password cracking tool. We have implemented a password ranking mechanism, which given a plaintext password and related user data, performs analysis on the password and returns an estimated strength based on the configuration described above. The ranking implementation is currently merely a proof of concept. In its current state it is not clear how the ranking relates to other password crackers.

While incomplete, this proof of concept is important for the topic of this thesis. We have shown that match scores can be applied to other authentication methods than biometrics. Additionally the ranking software is a good testing tool for our authentication framework. Since the authentication framework is the main focus of this thesis, we have chosen to postpone further improvements of the password ranking for the time being. We believe that the tool and its methodology can be an important contribution to the security community. However it needs to be developed further in order to be a feasible option for use in common software products.

The planned improvements are both theoretical and practical. Our analysis of password strength is far from complete. A further study of the password literature is required, along with an in-depth analysis of other password crackers. This should result in a more sound theoretical basis for the ranking of passwords.

In order for the software to be used with wordlists of any significant size, its wordlist processing speed needs to be greatly improved. Currently it takes a very long time to do a wordlist analysis on the *all.lst* wordlist, which is only 42 MBs. There are various different ways of speeding up the process, such as sorting the wordlists alphabetically, or even dividing it up into alphabetical sublists. Other methods include using a relational database for the wordlists, or even all the rewritings of the wordlists.

Our implementation does currently not account for an iterative mode, i.e. statistically guided brute-force methods. Based on more detailed analysis of password literature and other crackers, we intend to implement an iterative mode which assigns a ranking based on the estimated time it takes to crack the password in iterative mode. This rank can be based on statistical data, such as trigraph frequencies, or the entropy of the password.

Finally, in order for the software to be a feasible option for the industry, it should allow for external configuration. The configuration shown in Table A.1 is currently hard coded into the application.

Bibliography

[1] Firefox. http://www.mozilla.com/en-US/firefox/.

[2] IRIS - Iris Recognition Immigration System. http://www.ind.homeoffice.gov.uk/applying/iris/.

[3] Java technology. http://java.sun.com/.

[4] Javadoc tool. http://java.sun.com/j2se/javadoc/.

[5] John the ripper - documentation. http://www.openwall.com/john/doc/.

[6] Kerberos: The network authentication protocol. http://web.mit.edu/Kerberos/.

[7] Magic password generator. https://addons.mozilla.org/en-US/firefox/addon/874.

[8] Myth busters - episode 59: Crimes and myth-demeanors 2. http://dsc.discovery.com/fansites/mythbusters/mythbusters.html.

[9] Password safe. http://passwordsafe.sourceforge.net/.

[10] RSA SecurID. http://www.rsa.com/.

[11] Ruby programming language. http://www.openwall.com/john/doc/.

[12] US-VISIT Program. www.dhs.gov/us-visit.

[13] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (sha1), 2001. http://tools.ietf.org/html/rfc3174.

[14] Astrid Albrecht. Understanding the issues behind user acceptance. *Biometric Technology Today*, 9(1):7–8, January 2001.

[15] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[16] Jakob E. Bardram. The trouble with login: on usability and computer security in ubiquitous computing. *Personal Ubiquitous Comput.*, 9(6):357–367, 2005.

[17] Jakob E. Bardram, Rasmus E. Kjær, and Michael Østergaard Pedersen. Context-aware user authentication - supporting proximity-based login in pervasive computing. In *Ubicomp*, volume 2864 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2003.

[18] Matt Bishop and Daniel V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.

[19] Roberto Brunelli and Daniele Falavigna. Person identification using multiple cues. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(10):955–966, 1995.

[20] Kisilevsky B.S., Hains S.M.J., Lee K., Xie X., Huang H., Ye H.H., Zhang K., and Wang Z. Effects of experience on fetal voice recognition. *Psychological Science*, 14:220–224(5), May 2003.

[21] Edward J. Delp and Ping Wah Wong, editors. *Security, Steganography, and Watermarking of Multimedia Contents VI, San Jose, California, USA, January 18-22, 2004, Proceedings*, volume 5306 of *Proceedings of SPIE*. SPIE, 2004.

[22] Solar Designer. John the ripper password cracker. http://www.openwall.com/john/.

[23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. http://www.rfc.net/rfc2616.html.

[24] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, January 1995.

[26] J.C. Williams Group. The implications of chip & pin migration - a canadian retailer's perspective, January 2007.

[27] Lin Hong and Anil K. Jain. Integrating faces and fingerprints for personal identification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1295–1307, 1998.

[28] Lin Hong, Anil K. Jain, and Sharath Pankanti. Can multibiometrics improve performance. Technical Report MSU-CSE-99-39, Department of Computer Science, Michigan State University, East Lansing, Michigan, December 1999.

[29] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, October 1999.

[30] Anil K. Jain, Arun Ross, and Salil Prabhakar. An introduction to biometric recognition. *IEEE Trans. Circuits Syst. Video Techn.*, 14(1):4–20, 2004.

[31] Mark Keith, Benjamin Shao, and Paul John Steinbart. The usability of passphrases for authentication: An empirical field study. *International Journal of Human-Computer Studies*, 65(1):17–28, 2007.

[32] S. Kent and K. Seo. RFC 4301: Security architecture for the internet protocol, December 2005. http://tools.ietf.org/html/rfc4301.

[33] Josef Kittler, Mohamad Hatef, Robert P. W. Duin, and Jiri Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.

[34] Daniel V. Klein. "foiling the cracker" – A survey of, and improvements to, password security. In *Proceedings of the second USENIX Workshop on Security*, pages 5–14, Summer 1990.

[35] Tsutomu Matsumoto, Hiroyuki Matsumoto, Koji Yamada, and Satoshi Hoshino. Impact of artificial "gummy" fingers on fingerprint systems. *Proc. SPIE 4677: Optical Security and Counterfeit Deterrence Techniques IV*, 2002.

[36] Bill McCarty. Botnets: Big and bigger. *IEEE Security and Privacy*, 1(4):87–90, 2003.

[37] Steve Mcconnell. *Code Complete, Second Edition*. Microsoft Press, June 2004.

[38] Brian McKenna. Biometric scheme reduces night-time street violence in yeovil. *InfoSecurity Magazine*, October 2006. Online at http://www.infosecurity-magazine.com/news/061021_yeovil_biometrics.htm.

[39] Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[40] Robert Morris and Ken Thompson. Password security: a case history. *Communications of the ACM*, 22(11):594–597, 1979.

[41] Alec Muffett. Faq for crack v5.0a. http://www.crypticide.com/alecm/security/c50-faq.html.

[42] Justin Muncaster and Matthew Turk. Continuous multimodal authentication using dynamic bayesian networks. *Second Workshop on Multimodal User Authentication*, 2006.

[43] J. Myers and M. Rose. Post office protocol - version 3, 1996. http://www.rfc.net/rfc1939.html.

[44] Lawrence O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2019–2040, 2003.

[45] Tom Perrine and Devin Kowatch. Teracrack: Password cracking using teraflop and petabyte resources. Technical report, The San Diego Supercomputer Center, 2003.

[46] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing*. Prentice Hall Professional Technical Reference, 2002.

[47] Sigmund N. Porter. A password extension for improved human factors. *Computers and Security*, 1:54–56, January 1982.

[48] The Apache Commons Project. Commons digester. http://commons.apache.org/digester/.

[49] Bruce L. Riddle, Murray S. Miron, and Judith A. Semo. Passwords in use in a university timesharing environment. *Computers and Security*, 8(7):569–578, 1989.

[50] Ronald Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. http://tools.ietf.org/html/rfc1321.

[51] Arun Ross and Anil Jain. Information fusion in biometrics. *Pattern Recognition Letters*, 24(13):2115–2125, 2003.

[52] Arun A. Ross, Karthik Nandakumar, and Anil K. Jain. *Handbook of Multibiometrics (International Series on Biometrics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[53] Marie Sandström. Liveness detection in fingerprint recognition system. http://www.diva-portal.org/liu/abstract.xsql?dbid=2397.

[54] M. A. Sasse, S. Brostoff, and D. Weirich. Transforming the 'weakest link' – a human/computer interaction approach to usable and effective security. *BT Technology Journal*, 19(3):122–131, 2001.

[55] Bruce Schneier. Inside risks: the uses and abuses of biometrics. *Communications of the ACM*, 42(8):136, 1999.

[56] Eugene H. Spafford. The internet worm program: An analysis. Technical Report Purdue Technical Report CSD-TR-823, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2004, 1988.

[57] Sun Microsystems, Inc. Unchecked exceptions — the controversy. http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html.

[58] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[59] Umut Uludag and Anil K. Jain. Attacks on biometric systems: a case study in fingerprints. In Delp and Wong [21], pages 622–633.

[60] Raymond N. J. Veldhuis, Asker M. Bazen, Joost A. Kauffman, and Pieter H. Hartel. Biometric verification based on grip-pattern recognition. In Delp and Wong [21], pages 634–641.