Statistical Measurement of Information Leakage

Michael Yip
Supervisor: Dr. Tom Chothia

Submitted in conformity with the requirements
for the degree of MSc Computer Security
School of School Of Computer Science
University of Birmingham

Abstract

Statistical Measurement of Information Leakage

Michael Yip

This paper presents a security analysis of several security systems and for each system, two critical questions were answered, does the security system leak any side-channel information? If so, how much information is leaked. Any information unintentionally generated by the security systems which can be useful for an attacker is called side-channel information and in this paper, this is referred to as information leakage. The amount of leakage from the examined security systems was measured using the Anonymity Engine (A.E.) developed by Dr. Tom Chothia. This paper begins with an introduction to the theory of side-channel attacks and information leakage, followed by an introduction to the field of information theory which is used to measure information leakage. The capabilities and limitations of the A.E. is summarised in the first investigation as a practical demonstration of the concept of measuring information leakage from security systems. This is followed by an investigation into the workings of Freenet in an attempt to verify the widely stated theory that Freenet is susceptible to a correlation attack where an attacker can compromise the anonymity offered by making observations from the data requests. In the last investigation, the security of the British, Greek, German and Irish RFID e-passports were critically examined and analysed

Contents

Acknowledgments

I would like to sincerely thank Dr. Tom Chothia, my project supervisor for his continuous support throughout the project. His invaluable feedbacks and guidance helped me to make systematic progress and the success of the project would not have been possible without his support.

I am very grateful to the School of Computer Science for the decision to fund my MSc with the EPSRC studentship and gave me the chance to study at a world-class institution. I have thoroughly enjoyed my studies in the field of Computer Security and the experience has been very rewarding.

Dedication

To my beloved grandfather. Your inspirational character will be forever remembered.

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed ....

Chapter 1

Introduction

Security systems, such as encryption devices, accepts a certain set of secret inputs and produce a certain set of outputs based on the given inputs. In the case of an encryption device, a typical input would be the plaintext of the message which the user desire to hide and also, a secret key known only to the user or the system itself. Clearly, to break such a system, one has to know the algorithm employed by the device as well as the secret key. Attacks on encryption systems belong to a class of attacks known as cryptanalysis.

Cryptanalysis is the science of recovering the plaintext of a message without access to the secret key [7] and it is assumed that the cryptanalyst has complete details of the cryptographic algorithm. This is a reasonable assumption as many of the most secure cryptographic algorithms have already been publicised and subject to rigorous public scrutiny. Such method has proved to be a very effective way to testify the strength of a cryptographic algorithm and to increase public confidence in the effectiveness of the system. Afterall, if one cannot break the system with complete access to the details of the algorithm then the system would be even more difficult to break for those without such information. It is therefore, logical to assume that the secrecy of cryptographic systems reside solely on the secret key used.

There are four general types of cryptanalytic attacks; ciphertext-only attack, known-plaintext attack, chosen-plaintext attack and adaptive-chosen-plaintext attack. The purpose of the attacks is to deduce the secret key since it is already assumed that the attacker has complete access to the details of the cryptographic algorithm. The knowledge of the secret key should then provide the attacker with enough information to recover the plaintext of any messages encrypted by the targeted system.

The single most important element in any of the cryptanalytic attacks is the need to deduce useful information from observable events and to make use of such information in a clever and systematic way. In the case of attacking an encryption device, traditional cryptanalysis techniques rely solely on the plaintext and ciphertext of the messages encrypted by the system. However, it is now known that these are not the only observables from encryption devices, as listed in [1,8]. There are other less obvious observable outputs from such security systems, called side-channel information.

Formally, side-channel information is the information unintentionally generated by a security system as a by-product of the main process and such information can be useful for an attacker. In this thesis, this is referred to as information leakage.

The most common types of side-channel attacks make use of side-information such as timing information, power consumption rates [4], clock skews [5], network traffic load patterns [6] and fault generation [1,8].

For example, in the case of an encryption device, a certain mathematical operation in the cryptographic algorithm may consume more power or take longer to perform than others. An attacker can observe this and vary the input he provides the system to gain a better idea of how the output depends on the input. In attacks on anonymity P2P systems, an attacker can deliberately incur heavy load on to the system and to probe which nodes are affected by the load. This would enable the attacker to deduce the path the load has travelled and the identity of the nodes on the system would be exposed.

Clearly, when examining the security of a security system, one needs to examine all observable outputs produced by the system. In particular, one needs to answer: does the system leak any side-channel information? If so, how much information is leaked? How significant is the information leaked on the security of the system?

1.1 Motivation and objectives

Measuring information leakage using information theory is not a new concept. However, previous work using capacity and mutual information to measure information leakage has assumed that the exact behaviour of the system, that is, the probability of each observation under any user is known. Such probabilities are often generated from a formal model of the system. However, this is very restrictive and the definition of a formal model can be difficult as systems become increasingly large and complex.

A new method is proposed in [2,3] which measures information leakage without the need to make any kind of assumptions on the usage of the system. Instead, this method relies on the collection of observations from trial runs of a system and a probability matrix between the inputs and outputs is estimated based on the observations made. The probability matrix represents the relationship between the observed output and the secret inputs. The information leakage from the observable outputs can be measured by finding the capacity, maximised over all input distributions.

Furthermore, the specific objectives of this project are:

1. To investigate whether specific security systems leak any side-channel information
2. To carry out trial runs of the system and collect observations
3. To measure the amount of information leakage and the significance of the leakage
4. Evaluate the proposed method of measuring information leakage

1.2 Approach

As the method proposed in [2,3] requires the collection of observations from trial runs of the investigated systems, all subsequent investigations in this research project involve actual deployment of the investigated systems in order to simulate realistic interactions and behaviour. Observations were collected and were then supplied as input to the Anonymity Engine [3] for the calculation of the capacity. Since the method does not rely on any kind of assumptions about how the system is used, the results gathered from this research are applicable to the system in real settings.

1.3 Organisation of thesis

The rest of the thesis is organised as follows. In chapter 2, I present an introduction to the background concepts key to this research project. The competence of the proposed method in [2,3] was investigated and the details of the experiment is documented in chapter 3. Chapter 4 details my investigation into the workings of Freenet and the experiments carried out to measure the amount of information leakage from specific elements in its routing algorithm. Chapter 5 documents my analysis of the data collected from an analysis of RFID passports. In chapter 6, I draw a conclusion on the findings from this research and propose areas for future work.

Chapter 2

Background

Since the aim of this project is to measure the amount of information leakage from specific security systems, it is important for the reader to have a clear idea of what information leakage is and how significant a leakage can be in terms of security. Therefore in this chapter, I present an introduction on what information leakage is and how information theory can be used to measure information leakage.

2.1 Information leakage and side-channel information

Traditional cryptanalysis techniques rely solely on the ciphertext of the messages encrypted by the targeted system, and where available, the plaintext of the messages as well. The secrecy of the encrypted messages reside solely on the secret key used as it is assumed that the attacker has full details of the encryption algorithm used. The main method of attack is to make observations from the observables output such as the ciphertexts and search for any patterns. In the chosen-plaintext attack and adaptive-chosen plaintext attack, the attacker has the ability to choose which message to encrypt and observe the relevant output. This allows the attacker to observe the relationship between the inputs and the outputs.

Recent advances in cryptanalysis, as documented in [1,8], has discovered new observable outputs from cryptographic systems which further leaks information about the internal states of the system to the attackers. These are called side-channel information and side-channel attacks are attacks based on these new observables.

Side-channel attacks are of concern as they can be mounted quickly at relatively low cost. According to [4], a Simple Power Analysis (SPA) attack typically takes a few seconds while a Differential Power Analysis (DPA) attack can take several hours.

2.1.1 Types of side-channel information

Formally, side-channel information refers to the information unintentionally generated by a security system and which gives attackers better knowledge of the internal states of the system. This is also known as information leakage. The most common types of information leakage are briefly described below.

Timing information: Timing attacks can be used to break cryptosystems, such as those documented in [1,9]. The main factors which cause the distinctive differentials in execution time in cryptographic systems include performance optimisations to bypass unnecessary operations, branching and conditional statements and processor instructions such as multiplication and division. Different encryption key and input data would also have an impact on the execution time. In [9], this attack is referred to as a signal detection problem and such an attack is capable of deducing the entire secret key.

Power consumption rates: According to [4], there are three types of power consumption related attacks: Simple Power Analysis (SPA), Differential Power Analysis (DPA) and High-Order Differential Power Analysis (HO-DPA). These attacks are based on analysing the power consumption of the cryptographic unit and observe consumption rates while the unit performs the encryption operations.

Fault generation: As described in [1], in fault analysis attacks, the attackers learn about a cipher by either intentionally generating a fault in the encryption device which he possess or discover a natural fault within the algorithm itself. To spot a general fault, the attacker would force the machine to repeatedly encrypt the same message and observations on the output are made. Any observed variation in the output would immediately indicate a fault in the algorithm. An attacker can generate faults into the system by changing some variables such as voltage supply, tampering with the system clock or by applying radiation of various types. This allows the attacker to learn about the relationship between the output and the inputs, including the variables exploited by the attacker.

Traffic analysis: Traffic analysis has a long history, dating back to the Second World War [11] when it was used by the British at Bletchley Park to assess the size of the German airforce.

The technique behind the attack is simple: the attacker eavesdrops on a communication and he would search for any specific patterns in the collected traffic records. By examining the traffic data records of time and duration of a communication, a detailed shape of the communication streams can be formed. A powerful attacker such as the military powers, with the capability of eavesdropping on a large percentage of the communication network would be able to correlate the relationships between different targets and any regular patterns between them can be identified. Identities and locations of the parties involved can be established.

In computing, traffic analysis can be applied to communications across the Internet. The Internet is a communication channel through which computers communicate with each other according to a set of protocols. An important difference is that in this case, the attacker is generally less powerful than that of the military. In this case, the adversary is often the commercial entities, law enforcements or criminal organisations. Several traffic analysis attacks are described in [11]. For example, in the interactive mode of the Secure Shell Protocol (SSH), the user is required to enter a password to login. Every key stroke is sent as a packet and an eavesdropper can easily obtain the length of a password securing the private key. However, there is an even more significant impact. As the keyboard layouts are not random, an eavesdropper can collect the timings between each packet. Such timings provide very useful information to an attacker and would make a brute force attack much easier.

Other kinds of traffic analysis involves extracting and inferring information from network meta-data such as the size and timings of packets and the observable source and destination network addresses. In attacks against anonymous communications, such attack would be used to trace and expose the identities of the true orginator or destination of a message.

2.2 Measuring information leakage

Information theory was developed by Claude E. Shannon in order to find the ultimate data compression and the ultimate transmission rate of communication. Although at first sight, it seems that information theory is a subset of communication theory, it has in fact, been applied in various other fields such as Thermodynamics, Statistical Mechanics, Economics and Computer Science.

In this section, several important concepts in information theory are introduced. This is then followed by an explanation of how they can be used to measure and find information leakage.

2.2.1 Information theory

Channel

In information theory, a channel is a system which the outputs depends probabilistically on the input. By observing such a system, a probability matrix can be defined by recording the probability of each output given an input.

Formally speaking, each system has an input alphabet χ and output alphabet γ. A channel also has a probability matrix W$_{x,y}$ which gives the probability for an output y given an input x.

W$_{x,y}$ = p (y|x)

Entropy

Entropy measures the uncertainty about a random variable. It indicates how much we do not know about a random variable. In other words, it measures the average number of bits required to describe a random variable which indicates how much information is contained in a message describing the variable.

The higher the uncertainty, the more evenly distributed the probabilities of probable events for a random variable becomes hence the higher the entropy. Therefore, the higher the uncertainty, the more information is contained in a message describing the variable.

Entropy can be defined using mathematical notations as follows:

Let X be the random variable X and let χ be output alphabet (set of probable events). The entropy of X is given by:

H(X) = - $\sum$(p(x) log$_2$p(x))

p is the probability mass function and x is a bit in the message which describes the variable X. p(x) indicates the probability of x being equal to an expected value. If p(x) = 1, then it is a certainty that x will be equal to the expected value. This implies that there is no uncertainty in X and so the entropy is 0. On the other hand, if p(x) = log$_2$|X|, the probabilities for any incident of χ are evenly distributed and the entropy is equal to the number of probable events. This is the maximum entropy for X. Below is a demonstration of entropy using a Bernoulli trial of tossing a coin:

The random variable in this case refers to the event of tossing a fair coin. There are two possible outcome for the event: the coin landing on head or landing on tail. Since there are only two possible events, the variable can be fully described using 1 bit of information. The bit is 0 if the coin lands on tail and 1 if it lands on head. Since it is a fair coin, the probability for the coin to land on head is equally probable to landing on tail, which is 0.5. The outcome alphabet is denoted as χ.

If x $\in \chi$, the entropy of X can be found as follows:

H(X) = - $\sum$(p(x = 0) $\log_2$p(x = 0) + p(x = 1) $\log_2$p(x = 1))
= - ($\frac{1}{2}$ $\log_2\frac{1}{2}$ + $\frac{1}{2}$ $\log_2\frac{1}{2}$)
= - (($\frac{1}{2}$ + $\frac{1}{2}$) $\log_2$ $\frac{1}{2}$))
= - $\log_2$ $\frac{1}{2}$
= 1

Since the probability for the coin to land on head is equally probable to landing on tail, the probabilities are evenly distributed. This means that p(x) = log $|\chi|$ and the entropy of X is maximum since both events are equally probable. If a message is sent to notify the outcome of this event then the message is said to contain 1 bit of information.

Things become interesting when the tossed coin is unfair. If the probability for the coin to land on head is twice as likely as the coin to land on tail, then p(x = 0) = $\frac{1}{3}$ and p(x = 1) = $\frac{2}{3}$.

H(X) = - $\sum$(p(x = 0) $\log_2$p(x = 0) + p(x = 1) $\log_2$p(x = 1))
= - ($\frac{1}{3}$ $\log_2\frac{1}{3}$ + $\frac{2}{3}$ $\log_2\frac{2}{3}$)
= -(-0.528325643 - 0.389984618)
= 0.528325643 + 0.389984618
= 0.918310261
= 0.918

As evident from the calculations above, the entropy of tossing an unfair coin is less than that of tossing a fair coin. This is correct since we already know that something about X, that is, the probability for the coin to land on head is twice as likely as the coin landing on tail. This indicates that the uncertainty on X is not maximum. In another word, the message detailing the outcome of tossing an unfair coin contains 0.918 bits of information about X. This is less information than that of the previous message. The reason is that we already hold some knowledge about the variable and so, we learn less from the message.

In general, the entropy is maximum when the probability of head is 0.5. This has been proofed in the calculations above. The most interesting characteristic to note is that an entropy <1 can occur if the probability for head is less than or more than 0.5. This is true because in both cases, we already hold some knowledge about the variable and so uncertainty is not maximum.

Conditional entropy

Conditional entropy measures the remaining uncertainty about a random variable X given that Y is known.

Considering X and Y be the set of values for X and Y and $x \in X$ and $y \in Y$, conditional entropy of X and Y is defined mathematically as:

$$H(X|Y) = - \sum p(y) \sum p(x|y) \log_2 p(x|y)$$
$$= - \sum p(x,y) \log_2 \frac{p(x,y)}{p(y)}$$

This is best illustrated using a Venn diagram:

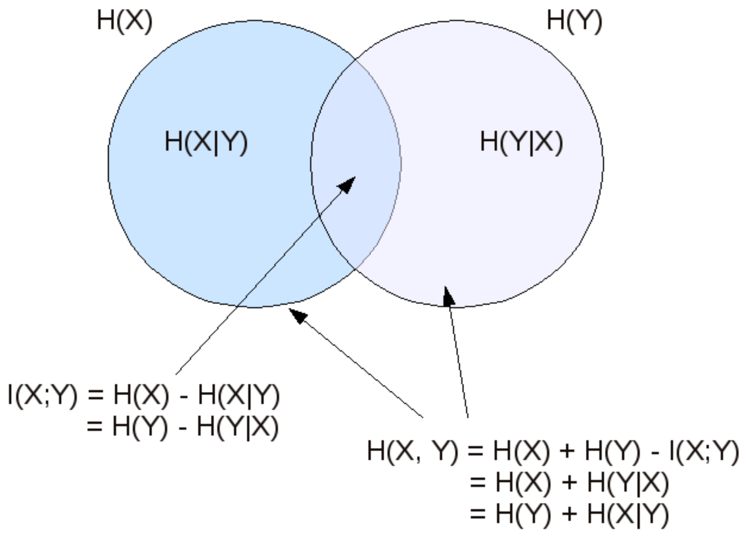


Figure 2.1: Entropy relationships

It can be observed from the diagram above that conditional entropy H(X|Y) can also be given by:

$$H(X|Y) = H(X,Y) - H(Y)$$

Mutual information

Mutual information measures the reduction in uncertainty when Y is known. In other words, it describes how much information we have learned about X by observing Y.

Mutual information is given by (observable from Figure 2.1 above):

I(X;Y) = H(X) - H(X|Y)

Since H(X|Y) = H(X) when X and Y are independent events, I(X;Y) = H(X) - H(X) = 0 when X and Y are independent. This means that when X and Y are independent events, we learn nothing about X by observing Y.

On the other hand, if X completely depends on Y then H(X,Y) = 0 and so I(X;Y) = H(X) - 0 = H(X). This means that we can learn everything about X simply by observing Y.

Capacity

Capacity refers to the maximum mutual information between random variables X and Y. This describes the maximum amount of information one can learn about X by observing Y. Capacity is given by the equation:

C = max$_{p(x)}$I (X,Y)

2.2.2 From continuous to discrete

As described in section 2.1.1, the most common types of side-channel information include timing information, power comsumption rate and temperature. All of these are continuous data which means that they all have an infinite number of possible values. This has a serious implication on the way a valid probability matrix can be formed because it is impossible to build a matrix with an infinite number of possible outputs.

Subsequently, it is necessary to somehow convert the infinite continuous data set into a finite set of discrete data. This is achieved by restricting the collected values to a fixed number of decimal places which allows the definition of a finite range of possible values from a finite number of observations. Within this range lies a finite set of possible output values and using this set, the probablistic dependency between the output and the set of input can be quantified.

2.2.3 How to measure information leakage?

Measuring information leakage using information theory has long been proposed but the proposed method in [2,3] differs in that the technique does not require any assumptions to be made about how the system is used.

Recall that a channel has an input alphabet χ, an output alphabet γ and a probability matrix W where $W_{x,y}$ = p(y|x) gives the probability of output y when x is given as input. Applying this to a system with an input distribution χ, two random variables X and Y can be defined to represent the input and output of the system respectively. The mutual information of the system can then be written as I(X,W) for I(X;Y). The capacity of the system can then be defined as the mutual information between the input and output, maximised over all input distributions.

Consider an anonymity system such as Tor as the channel. A denotes the set of anonymous events (the identity of the users) whilst O denotes the set of observable outputs (traffic patterns or latencies).The system is considered compromised when an attacker can de-anonymise any of the events in A and one way this could be achieved is for the attacker to learn about A by observing O.

Assuming that on each execution, exactly one a $\in$ A and one o $\in$ O will happen and o depends probabilistically on a. By defining two random variables α and σ to represent the anonymous events A and observable outputs O respectively, the information leakage from O can be calculated by finding the mutual information I(α;σ). If I(α;σ) does not equal to 0 then there is information leakage from σ and hence an attacker can learn about the anonymous events A of the system by observing O.

For the findings to be applicable to the system in real settings, it is most desirable when no assumptions are made about how the system is used in any of the calculations. That is, the input distribution of the system is not assumed when the calculations are made. According to [2,3], the capacity of the system can be used to measure the information leakage that is leaked by a system independently from the distribution of input events.

Before the capacity can be calculated, the probability matrix W must be found. It is fundamental in the estimation of information leakage because it describes the relationship between the anonymous events and observable events. The question then becomes how can we construct a probability matrix of a system without any prior assumptions?

As proposed in [2,3], a probability matrix $W_{x,y}$ can in fact be estimated using observations of the input - output pairing from trial runs of the system. After running n trials, a probability matrix can be estimated by defining the set of inputs, the set of outputs and the probability of each of the output when a particular input is given.

Considering a simple password system (further details will be given in the next chapter), the inputs of the system would be a set of password entries and the observable outputs are the different amount of times the system takes to accept or reject a given password. The list of observations from such a system would be in the form:

(9999,239)
(9999,240)
(9991,231)
(9991,230)
. . . so on. The probability matrix W would then be in the format:

Pwd	220	221	. . .	230	231	. . .	239	240
9999	0.0	0.0	. . .	0.0	0.0	. . .	0.5	0.5
9991	0.0	0.0	. . .	0.5	0.5	. . .	0.0	0.0

After estimating the probability matrix,the capacity based on this matrix can then be calculated and the information leakage from the outputs given the inputs can be estimated. This method can also be used to calculate the worst-case scenario.

The Anonymity Engine (A.E.)

The Anonymity Engine is developed by Dr. Chothia and his team as documented in [2]. This software automates the calculations above by taking a list of observations or a probability matrix as input. If a list of observations is given as input then the software would first estimate a probability matrix from the observations, as described above, and then the capacity is calculated. If a matrix is supplied then the capacity of the given matrix is calculated.

2.3 Chapter summary

In this chapter, a detailed introduction has been given on side-channel attacks, information theory and how information leakage is given. The work documented in [2,3] has been described above and it is my hope that the reader should have a good understanding of the background to this project by this point. The subsequent chapters in this thesis documents my investigations on several systems.

Chapter 3

Investigation 1

In this chapter, I present my first investigation carried out in this research project. There were two objectives for this investigation: firstly, to serve as a proof of concept using the Anonymity Engine to measure the information leakage from a simple password system with obvious leakages; secondly, to develop a generic method to measure information leakage from measuring execution time. Below is an introduction to the background concepts that were critical to this investigation and this is followed by a detailed description of the experiments carried out.

3.1 String matching

String matching is a fundamental branch in text processing. As described in [16], string-matching involves the search for the occurrence of a string (known as a pattern) in a given text (another string). Both strings are built over a finite set of characters from an alphabet and each character in a string is assigned an offset index from left of string to right such that s = c[0...m-1], where m denotes the number of characters c contained in the string s. This has significant impact on identity verification systems as described below.

Traditional access control applications such as those used in an operating system, store user passwords in their plaintext form. When an user attempts to login into a terminal, he would be required to provide his username and password via the login screen. Having received the two values, the login application would then attempt to match the given password to the stored password for that particular user. As computer security advances, this is now considered a very bad practice and the hash value of the passwords are stored instead.

Users who wish to login still has to supply their username and password via a login screen. After receiving the values, the login application would compute a hash of the given password and then match the generated hash string against the one stored in the database for that particular user.

As evident from the above, in either case, string-matching is an integral part of a user identity verification process. In this investigation, the impact of such algorithm on a native password system is investigated.

Before presenting the experiments, it is important for the reader to understand the challenge challenge faced in measuring program execution time as this was what most design decisions in the subsequent experiments were made upon. This is introduced in the following section.

3.2 Measuring execution time

Software applications are not executed one after the other. Rather, they are enqueued and wait in turn to be allocated with the processor. The Central Processing Unit (CPU), also known as the processor, execute the instructions in a process according to its internal timer which is regulated by a precision oscillator. In order to optimise the allocation of processing time, the processor has a process scheduler which performs the scheduling and allocation of the processor to the processes accordingly and fairly. There is also an external timer called Programmable Interval Timer (PIT), described in [18], which generates a timer interrupt (a clock tick) periodically to indicate to the scheduler that a certain time interval has elapsed. The scheduler makes use of the clock ticks to decide when to reassign the processor to another process and this process is called context-switching.

One of the most common measures of application performance is the actual execution time of the application. However, due to the lack of hardware support and an universal implementation, measuring execution time is not a straight forward task. Generally speaking, there are three different ways an application developer can obtain timing information from the machine: using the library functions , reading from a cycle counter and using the system clocks. Below is an introduction to the each of the options, further detailed in [14].

3.2.1 Process scheduling and Interval counting

External events such as keystrokes, disk operations and network activity generate interrupt signals that requires the attention of the process scheduler and reassign the processor to a different process. However, in the absence of such events, it is still desirable for the processor to be allocated effectively.

As mentioned above, the external timer, called Programmable Interval Timer (PIT), is responsible for generating timer interrupts. When a timer interrupt occurs, the scheduler decides whether to allow the processor to continue processing the currently executing process or to reassign it to a different process. The time between the occurrences of the timer interrupts is called timer interval (also known as quantum) and in order for the CPU to be allocated effectively, the timer interval should be set short enough so that the processor will be reassigned to different processes. However, it is important to note that as one or more instruction is executed per clock cycle (around 1 nanosecond) depending on the frequency of the processor and thousands of clock cycles are required to save the state of the currently executing process, it is important not to set the interval time too short. According to [14,18], the typical timer intervals range from 1 to 10 milliseconds.

There are two modes in which a processor operates: user mode and kernel mode. The CPU is operating in user mode when it is executing the instructions of an application program and in kernel mode when it is performing operating system functions on behalf of the program. From the perspective of an application program, the flow of time can be viewed as alternating between periods when the program is active which is when the processor is executing its instructions or inactive when it is waiting to be assigned the processor. The application only performs useful computation when it is assigned the processor and that its process is operating in user mode (the processor is executing the instructions of the program). Below is a diagram extracted from [14] which shows the life time of a process when it is the only active process.:

The occurrence of timer interrupts are represented by the grey triangles.

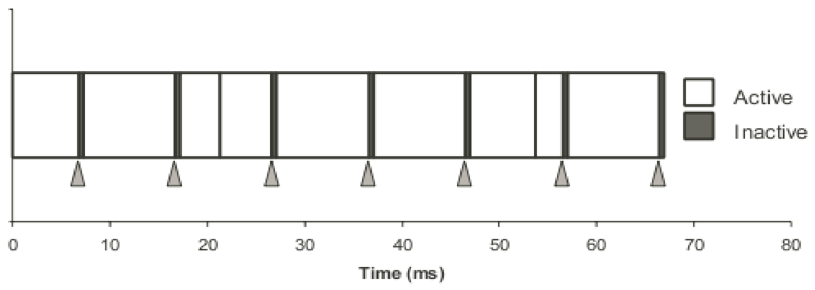


Figure 3.1: Lifetime of a process with system load = 1.

From the figure above, it can be seen that the time when the application is in an active state is not contiguous nor is it uniform. It can also been seen that the process is active for most of the time and this is due to the fact that it is the only active process on the system. The effect of adding another active process to the system load is shown below (also extracted from [14]):

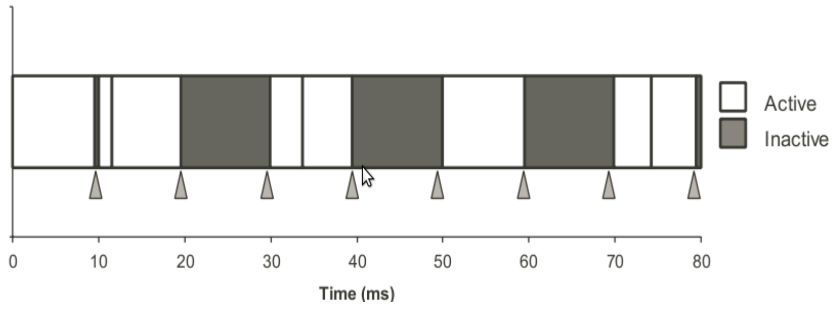


Figure 3.2: Lifetime of a process with system load = 4.

As evident from the figure, the amount of time the application is in an active state has fallen dramatically and the reason is that the process scheduler has to reassign the processor to the other active process. In general, the higher the number of active processes there are in the system hence the higher the system load, the longer the time an application program is inactive and the more frequent context switching occurs. This may seem irrelevant at first sight but their relevance would soon become apparent.

The operating system uses the PIT to record the amount of time used by each process. Since there can only be one process active at any one time and at the occurrence of a timer interval, the operating system determines which process is active and increment one of the counts for that process. Each process has two counts, user time and system time. If the process is operating in user mode, the user time is incremented and if it is operating in kernel mode, the system time is incremented.

Several programming languages offer library functions for software developers to read process times, as listed in [20]. They include:

- C library function `times()`
- `clock()` function in <time.h>
- `System.currentTimeMillis()` in Java

However, according to [14], the accuracy of time measurement using process timers is approximate at best and there are two reasons for this.

Firstly, context switching during a timer interval is not accounted for as the operating system determines the state of the currently active process only at the occurrence of a timer interrupt.

For example, a process is in user mode at the occurrence of a timer interrupt. After the operating system has incremented the relevant counter, the process is context switched into kernel mode where it becomes idle, awaiting for the processor. Since its process is in kernel mode, no useful computation is performed by the application program and hence it is not being executed. However, just before the occurrence of the next timer interrupt, the process is context switched again and allocated the processor. At the timer interrupt, the processor detects that the process is in user mode and approximates that the process has been in user mode during the entire timer interval. This means that the process times either under or over estimate the actual execution time.

Secondly, another source of inaccuracy is the fact that the timer interrupts do cause kernel activities and as evident from the figures above, the activity occurs after a timer interrupt but terminate before the next. Thus, they are not accounted for and there accumulated effect further causes inaccurate measurements.

Finally, the author of [14] indicates that the effects of this inaccuracy is most serious for applications with execution time less than 100ms and that it generally subsides for execution times longer than 100ms. Despite this finding, another option was considered and is described below.

3.2.2 Cycle counters

Another option for software developers to obtain timing information from the system is to read the value from the Time Stamp Counter (TSC). Not all processors has this timer and in this project, all systems used to perform the trials has Intel processors installed which, in its assembly instruction set, provides a special function called `rdtsc` that allows developers to read the value from this timer. Further details of TSC are listed in [18,21].

TSC is a special register and is 64-bits long so that it would wrap around only once in every 570 years. The register is incremented at every single clock cycle. For a 1GHz processor which executes 10^9 instructions per second which means that it executes one instruction per clock cycle. Therefore, for this processor, the TSC value gives the execution time elapsed in nanoseconds. However, for faster processors, the execution time will be shorter as more than one instruction are executed per clock cycle.

In general, as the number of instructions executed per clock cycle depends on the frequency of the processor and since the TSC records the number of clock cycles elapsed, the precise execution time of the application can be found using the equation, as detailed in [21]:

Execution time (seconds) = TSC value / Processor Frequency (MHz) * 10^6

In general, the TSC gives readings to nanosecond accuracy and so the accuracy is much higher than that of interval counting which gives readings to millisecond accuracy.

However, the effect of context switching still exists. An application with long execution time is likely to be context switched more during its lifetime than one with short execution time and the cumulative effect of context switching leads to a less accurate measurement as execution time increases. Context switching occurs more often if the system us heavily loaded and this adds to the inaccuracy.

The author of [14] indicates that the effect of caching can also cause variations in readings but its effect is less than the effect of context switching. He proposed that the effect of caching can be minimised by warming up the cache before taking the actual measurement.

3.2.3 System clocks

The last option for software developers is to use the library function `gettimeofday` which takes readings from the system clock and return values in seconds and microseconds.

It is indicated in [14] that there are two problems with this method. Firstly, the implementation of this function varies between different systems. Some use interval counting whilst others use cycle counters and as documented above, the precision between the two varies greatly. Secondly, the author of [14] has found that the overhead incurred in calling system calls is higher than ordinary function calls. This overhead causes a delay between the time system call is invoked and the time the measurement is taken and so the accuracy of readings from this call is questionable.

3.3 Investigation setup

This section details the experiments carried out for this investigation from the initial stages of planning and design to implementations and results.

3.3.1 Investigation objectives

The objectives of this investigation were as stated below:

1. to use the Anonymity Engine [2,3] to measure information leakages from a system with known leakages and evaluate its effectiveness

2. to develop a generic method to find information leakage by measuring the execution time of an application such as a password system. The method should be applicable to measure applications without any access to the source code since the method proposed in [2,3] requires no prior assumptions about how the system is used.

The first challenge was to find a security system with obvious leakages to be modelled in this investigation.

It was established that the `equals` method in the java.lang.String class uses a classic left-to-right string-matching algorithm. The code is shown below, as listed at [15]:

```
1 int n = count;
2 if (n == anotherString.count) {
3       char v1[] = value;
4       char v2[] = anotherString.value;
5       int i = offset;
6       int j = anotherString.offset;
7     while (n- - != 0)  // Compare EACH character, from left to right {
8       if (v1[i++] != v2[j++]){
9          return false; //Return on the FIRST character mismatch
10        }
11        return true;
12    }
```

In Java, a string of characters is represented by a String object and the String class is part of the java.lang package. The code above shows the default implementation of the `equals` method in the String class and the purpose of this method is to find out whether two String objects are the same.

The method extracts the individual characters from the two String objects into two separate arrays of characters. Line 7 shows a while-loop construct which iterates as long as there are more characters to compare. Line 8 shows that the comparison of the two strings is achieved by comparing the characters at the same offset in the two character arrays from left to right. The method returns false when the first mismatch is found.

If this method is used in a password verifying system then there is a very high chance that the execution time of the application would leak a significant of information about the passwords the users provide when they try to log in or have their identity verified. For example, if the password belonging to the user attempting to log in is 9999 and the password provided via the login screen is 1999. If the passwords are matched using the string-matching algorithm above, the given password will be rejected when the first character from the two strings are found to mismatch. However, if the given password is 9991, the string-matching algorithm would only return false when the last characters in the two strings are found to be different.

Subsequently speaking, assuming that time measurements are accurate, the time required by the system to reject the first given password 1999 should be shorter than to reject the second given password 9991 since it has to examine 3 more pairs of characters before rejecting the latter. This also implies that the time system takes to successfully verify a correctly given password 9999 would take approximately the same amount of time as rejecting the password 9991 since there are four comparisons to make in both cases.

Therefore, the string-matching algorithm used in the `equals` method should leak information about secret inputs if applied in a security system.

To investigate the above claim, a naive password system called StringPW.java was written as below:

```
public class StringPW{
    static String pass = "9999";
    public static void main(String[] args){
        pass.equals(args[0]); //Compare
    }
}
```

From the code above it can be seen that the password verifying system is of the simplest. This was intentional so that time measurement distortion from any unnecessary codes such as user notifications were eliminated. This would also enable time measurements to reflect the state of the algorithm so that different password guesses should yield significantly different time readings.

The next step was to design and implement a time measuring application.

3.3.2 Timer application design

As stated in the second objective, the method developed in this investigation should assume no access to any source code of the measured application. This is reasonable since no prior assumptions about the usage of the measured application should be made in this investigation.

Subsequently, the functional requirements of the timer application were as follows:

1. To execute an external application
2. To obtain accurate timing information
3. To produce output for the Anonymity Engine

To achieve the first requirement, it was important to choose a programming language which provides an easy method to execute external applications. I considered two programming languages for this application, Java or C. In Java, one can execute shell commands by calling the `exec` using an object from the java.lang.Runtime class. In C, one can achieve the same by calling the `systems()` system call. At this point, it seemed that there is not much of a difference between the two methods offered by the two languages.

However, in consideration to the second objective, it became apparent that C was the most appropriate language for the timer application. As described above, there are three options for software developers to obtain timing information: process timers, TSC and system clock. To obtain any kind of information leakage, the accuracy of time measurements must be of the highest possible. In consideration of the availability of hardware support in this investigation , the second option was chosen to be the most appropriate. In order to read the value from the TSC, the C programming language offers a system call called `asm` which allows a high-level application to execute assembly instructions. Below is the code used in this investigation to obtain value from the TSC, which is extracted from [14] and is used in the timer application called timer.c (included on attached CD):

```
1 void access_counter(unsigned *hi, unsigned *lo){
2    asm(”rdtsc; movl %%edx,%0; movl %%eax,%1”
3        : ”=r” (*hi), ”=r” (*lo) /* List of outputs */
4        : /* List of inputs */
5        : ”%edx”, ”%eax”); /* List of overwritten registers */
6 }
```

Line 2 shows the asm procedure call. This procedure has one required argument, which is the assembly code string, and three option arguments, separated by the character ’:’. The optional arguments are: list of outputs, list of inputs and list of overwritten registers.

This procedure provides a way for software developers to insert assembly code directly into the code sequence of the code generated by the compiler. It is compatible only with gcc compilers. Further details are given in [13].

The assembly code to be inserted in this case is: ''rdtsc; movl %%edx,%0; movl %%eax,%1 . The `rdtsc` is the instruction available in the Intel instruction set to access the TSC. The input and output operands to the `asm` procedure are enumerated as %0 to %9, in their ordering first in the output list then in the input list and in this case %0 is (*hi) and %1 is (*lo). Since rdtsc stores its highest 32 bits in the edx register and lowest 32 bits in the eax register, this assembly code moves the value from the edx and eax registers to the two output operands. Therefore, this code is used to read the 64-bit value from the TSC into two global variables cyc_hi and cyc_lo, as shown below:

```
void begin_counter(){
  /* Reset all temporary storage */
  cyc_hi = cyc_lo = 0;
  access_counter(&cyc_hi, &cyc_lo);
}
```

This function is called to obtain the TSC value just before the measured application is executed. After the application has finished its execution, another procedure is called to obtain the TSC value and to return the number of instructions executed. This procedure is shown below:

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo;
  unsigned hi, lo, borrow;
  double res;
  /* Reset all temporary storage */
  ncyc_hi = ncyc_lo = hi = lo = borrow = res = 0;

  /* Obtain number of cycles after code execution */
  access_counter(&ncyc_hi, &ncyc_lo);

  /* Obtain the difference in the lower 32 bits */
  lo = ncyc_lo - cyc_lo;
  /*
  * Check whether lo is higher than ncycz_lo:
  * if so, borrow = 1 and ncyc_lo must have borrowed from higher 32 bits
  * else borrow = 0
  */     borrow = lo ¿ ncyc_lo;
  /* Obtain difference in higher 32 bits, taking into account the borrow */
  hi = ncyc_hi - cyc_hi - borrow;
  res = (double) hi*(1¡¡30) * 4 + lo;

  if(res ¡ 0){
    fprintf(stderr, ”Error: counter returns non-positive value.
n”);
    exit(1);
  }

  return res;
30 }
```

From line 10 in the code segment above, the access_counter function is called again to obtain the TSC value after the measured application has terminated its execution. On line 13, the difference between the two lowest 32-bit TSC values is found and if the difference is larger than the value in ncyc_lo, the final counter value must borrow from the higher 32-bits. On line 21 where the difference between the two highest 32-bit TSC values is found, taking into account the borrow. On line 22, the value of the difference of the TSC value before and after the execution of the measured application is merged into one value, stored into a double (64-bit) type variable. The value stored in the variable hi is multiplied to a value of 1 left-shifted by 30 bits. It is shifted by 30 is because the width of integer is 32-bits and shifting beyond 30 would cause overflow. To accommodate the two highest significant bits, the product of the multiplication is multiplied by 4. The value in the variable lo is then added and the sum of this addition is returned as the difference in the TSC value.

3.3.3 An anatomy of the timer application

Having presented the code used to capture value from TSC, a brief anatomy of the timer application is given below. A sequence diagram is included in Appendix A. Please note that the object notation was slightly abused to represent the functions in the application instead of objects.

As with all C applications, the execution of the application is enclosed in the `main` method and execution begins by parsing the input variables in a file with extension .et, which the function `parseInput` is responsible for. The precise structure of the .et file is given in Appendix A.

After parsing the inputs, the createGuesses function is called to form a list of password guesses. The password guesses are stored in a structure as follows:

```
struct guess{
    /* A password guess */
    char pwd[NORM_BUFFER];
    struct guess *next;
    /* Next time interval for this guess */
    struct interval *outputs;
};
```

The structure guess holds three variables: pwd, which holds the password, next holds the reference for the next password and outputs is a pointer to a list of intervals for that particular password. Each password has its own list of time intervals as the output patterns varies between different passwords. A list of timing intervals is represented by the structure named interval, as shown below:

```
struct interval{
    char sBound[NORM_BUFFER];
    double bound; /* To take into account timing precision less than 1ms */
    long count;
    struct interval *next;
};
```

The variable sBound stores the timing interval in as a string whilst bound stores the interval as a double. The variable count stores the number of times the measured time is within the interval and the variable next stores the reference to the next interval. This list of intervals is used to capture the measured execution time of the measured application given each password guess and this is then used to produce a probability matrix in the format described in the section 2.2.2.

After parsing all the values from the .et file, the `parseInput` function invokes the function `generateIntervals`. This function takes the line of password guesses and split the line by each guess. For each guess, the function calls the `formCMD` function to create an execution script to execute the measured application given the password guess. After forming the script, the `findAverage` function is called and the application is executed a number of times (halveof the intended number of iterations). After this has been done for all password guesses, the highest and lowest execution time would have been found. This helps to form the upper and lowest boundaries of the timing interval. To take into account the fluctuations in time measurement, the lowest bound is extended by 10% whilst the upper bound is extended by 50%.

There are two influential variables which affects the interval gap and the number of intervals. The variable intGap specifies the difference between the width of an interval whilst the variable difference indicates the number of decimal places for the width. Tweaking these two variables carefully should help the timer application to produce the probability matrix capturing the highest amount of information leakage.

After the timing interval has been obtained, a list of password guesses is produced in the `createGuesses` function. This function takes the line of password guesses obtained from the `parseInput` function and it splits the line into each password guess. Each guess is then associated with a copy of the list of intervals.

With the initial setup complete, the `executeTiming` function is called by the `main` function to perform the actual time measurements. This function is similar to that of the `findAverage` function as both use similar code to capture timing information as shown below:

```
1   /* Warmup cache */
2   system(finalCMD);
3
4   /* Begin timing */
5   begin_counter();
6   system(finalCMD);
7   tmpc = get_counter();
```

From the code segment above, it can be seen that the system() system call is called to execute the measured application using the execution script stored in the finalCMD variable. Line 2 shows that the cache is warmed up before the measurements are taken and this was established as necessary so that the timer application can be used to accurately measure applications with short execution time (less than timer interval).

```
1   tt = ((double)tmpc/(mhz*1000000))*1000;
2   /* Round up timing */
3   time = round(tt);
4   traverseIntervals(g→ outputs, time);
```

On line 1 in the code segment above, the TSC value captured is converted into milliseconds and is rounded up to a certain number of decimal places, decided by the value stored in tt. After rounding up the time captured, the `traverseInterval` function is called to match the captured time to an interval for that particular password.

Lastly, there are four functions with names beginning with `write`. These functions are used to produce the output files.

Recall that the two objectives for this investigation were to proof the competence of A.E. in measuring information leakage and to develop a generic method to measure information leakages without access to source code. The experiments carried out to achieve these objectives are documented below.

3.4 Experiment one

3.4.1 Description

In order to proof that A.E. is capable of measuring information leakage, this experiment exploits the `equals` method in java.lang.String which has obvious vulnerabilities and should leak timing information for different password guesses.

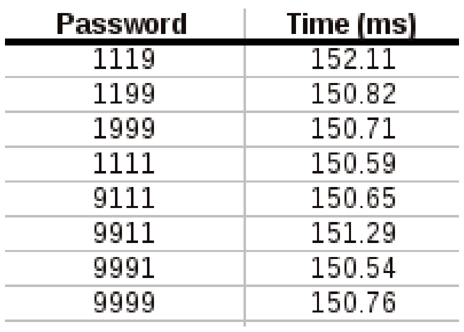
3.4.2 Experiment setup

The details of the system used are as below:

Processor: Intel Centrino
Processor speed: 600 MHz
Number of processors: 1
Memory: 500mb

In this experiment and experiment two, only eight passwords were used: 1999, 1199, 1119, 1111, 9111, 9911, 9991 and 9999. The reason was that this investigation served as a proof of concept and the source of leakage was already known. Therefore, these inputs were specially crafted to cause leakage so that the competence of the A.E. could be investigated.

3.4.3 Results



Password	Time (ms)
1119	152.11
1199	150.82
1999	150.71
1111	150.59
9111	150.65
9911	151.29
9991	150.54
9999	150.76

Figure 3.3: Results from experiment 1

The results from the A.E. for this experiment was as follows (full output listing is given in Appendix A):

8 inputs, 85 outputs and 8000 samples
estimate result = 0.055
correction = log_2(e).(noOfInputs-1)(noOfOutputs-1)/2.sampleSize = 0.053

The results are no more accurate that the correction value,
increase the sample size to decrease the correction.
Calculations are to 95% confidence.
The estimated capacity is 0.055 - 0.053 = 0.002
The 95% confidence interval for zero information leakage is :0.0839
This is consistent with the chi^2 distribution for zero leakage.
Capacity is between 0 and 0.0129

3.4.4 Experiment summary

It can be seen from the result above that only very minimal information leakage was detected. A possible reason for this low leakage was because the variation in execution time for each password guess was too small. As can be observed from the table above, the variations are different from our expectation and I believe the reason is because of the distortions from context switching and cache misses as mentioned in the previous section. Since the A.E. relies heavily on the observations made from the trial runs, if there are no significant differences in the observations, significant leakage would not be found.

I believed that the solution to this was to amplify the execution time for each password guess by introducing an iteration in StringPW.java. This was investigated in the second experiment, as described below.

3.5 Experiment two

3.5.1 Description

In this experiment, the execution times for each password guess was amplified by introducing iteration in StringPW.java. In other words, the string-matching algorithm is repeatedly executed for a certain number of times. This has a simple application and is demonstrated using an example below.

For example, assuming that the comparison of each pair of characters in the equals method takes 0.1ms and if the execution time for matching password 9911 once is 100.2ms then 9991 would require 100.3ms. The difference between the two is only 0.1lms which is less than the timer interval. With the noise in the background, the execution time for the different passwords would not be stable, as described in the previous section and such a small difference would not be easy to find.

By introducing an iteration of 100, the total execution time for 9911 would be 120ms and 130ms for 9991. The difference between the two has been amplified by a hundredfold and the difference has become 10ms, which is the approximately the same as the typical timer interval. Therefore, the higher the number of iterations, the higher the amplification and the more significant the difference in execution time.

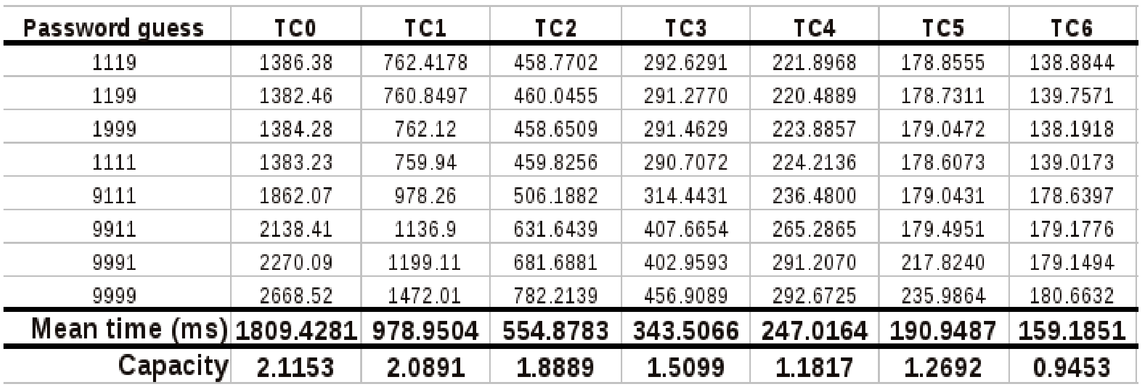
3.5.2 Experiment setup

The machine (ID: cca-ug04-0948) from the School UG04 cluster was used for this experiment as it is a system with heavy load. This was intentional as this experiment was designed specifically to overcome background noise. The details of the system are as follows:

Processor: Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz
Processor speed: 2394.000 MHz
Number of processors: 4
Memory: 4GB
Each test case (TC) was associated with a specific number of iterations in StringPW.java. In TC0, the number of iterations was 100,000,000 and each subsequent TC was associated with approximately halve the number of iterations in the previous TC. Therefore, TC1 was associated with 50,000,000 iterations and so on.

3.5.3 Results

The core results of this experiment were as follows (also see Appendix A): A graphical



Password guess	TC0	TC1	TC2	TC3	TC4	TC5	TC6
1119	1386.38	762.4178	458.7702	292.6291	221.8968	178.8555	138.8844
1199	1382.46	760.8497	460.0455	291.2770	220.4889	178.7311	139.7571
1999	1384.28	762.12	458.6509	291.4629	223.8857	179.0472	138.1918
1111	1383.23	759.94	459.8256	290.7072	224.2136	178.6073	139.0173
9111	1862.07	978.26	506.1882	314.4431	236.4800	179.0431	178.6397
9911	2138.41	1136.9	631.6439	407.6654	265.2865	179.4951	179.1776
9991	2270.09	1199.11	681.6881	402.9593	291.2070	217.8240	179.1494
9999	2668.52	1472.01	782.2139	456.9089	292.6725	235.9864	180.6632
Mean time (ms)	**1809.4281**	**978.9504**	**554.8783**	**343.5066**	**247.0164**	**190.9487**	**159.1851**
Capacity	**2.1153**	**2.0891**	**1.8889**	**1.5099**	**1.1817**	**1.2692**	**0.9453**

Figure 3.4: Results from experiment 2

representation of this can be found in section 3.5.5.

3.5.4 Experiment summary

From figure 3.4, it can be observed that there is a very distinctive difference in average execution times for each password guess, mostably "9999" and the passwords which begin with "1" in tc0 and this difference falls as the mean execution time for each test case falls. This confirms that the introduction of iteration does indeed amplify the difference in execution times and thus the signal-to-noise ratios (SNR) were increased.

It can also be observed that capacity as high as 2.1153 has been obtained. This implies that an attacker can learn approximately two out of the three bits required to describe the possible password inputs simply by observing the variations in the execution time of the password verification system, with different passwords given as secret input.

Also from figure 3.4 and figure 3.5 from section 3.5.5 , it is clear that there is a positive correlation between capacity and execution time. In general, the higher the average execution time, the higher the information leakage. This proofs that by increasing the SNR, the difference between execution times became more significant and obvious. It should also be observed from figure 3.5 that the recorded capacity began to fall dramatically for average execution time below 154ms. This was because the SNR between the distinctive differences in time and the background noise became too small.

As leakage from the string-matching algorithm was already known to exist, the findings in this experiment can only be used to proof that the A.E. is capable of measuring information leakage from trial runs of the system. However, to measure significant leakage, the observations must display distinctive patterns which would be observable from the estimated probability matrix.

3.5.5 Additional results and figures

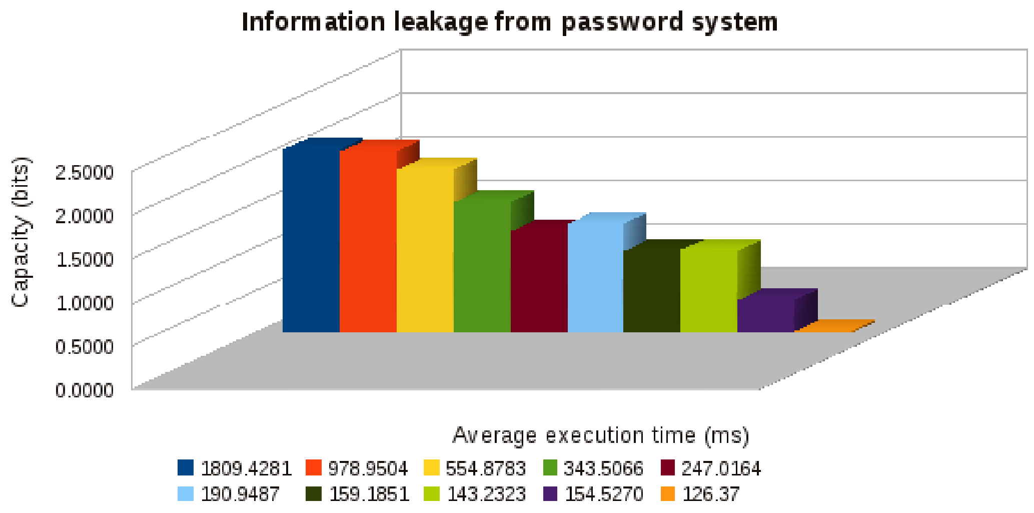


Figure 3.5: Positive correlation between capacity and average execution time

3.6 Conclusion of investigation

From this investigation, it has been shown that the Anonymity Engine (A.E.) is capable of measuring information leakage from trials runs of a target system. Furthermore, the method developed to measure program execution time is successful to a large extent. That is, as long as the background noise (system load) is kept to a minimum or there is a significantly large, positive signla-to-noise (SNR) ratio between the observable patterns and the background noise, the results from this method would display a certain amount of information leakage. It has been proofed in this investigation that using the timer application, one can indeed learn about the secret inputs by observing from the observable timing information. However, to the fact that the A.E. relies entirely on the observations from trial runs and the fact that time measurements are most often inaccurate, using the A.E. to measure information leakage from program execution time is somewhat ineffective and inappropriate.

Chapter 4

Investigation 2

In this investigation, an extensive research on the workings of Freenet has been carried out, from its high-level protocol design to its low-level implementations. The objective was to find a potential source of information leakage and to use the Anonymity Enginer (A.E.) to measure the leakage. In this chapter, I shall present an introduction of anonymous P2P and the workings of Freenet followed by a description of the experiments and findings.

4.1 Anonymous P2P

A Peer-to-Peer (P2P) network is an Internet overlay network which is often referred to as P2P. In a typical P2P network, each node acts a client as well as a server and this makes resource sharing much more efficient than using the typical client-server architecture. Nodes in a P2P network request and retrieve information from each other without assistance from any centralised component. According to [27], there are three levels of P2P:

1. **Hybrid P2P:** a central server is used to keep information about the network and control network. Nodes who wish to communicate with its peers would query the server for routing details.
2. **Pure P2P:** no central server is used. Each node in the network acts as a client as well as a server.
3. **Mixed P2P:** a mixture of hybrid and pure P2P. e.g. Gnutella [33]

Napster was a hybrid P2P where a central server was used to handle search queries and network control. This made Napster particularly vulnerable because all it required the law enforcement agents to do was to shut that server down. The author of [27] classifies this as one of the first generation P2P networks. FastTrack [35] was the P2P network used by popular P2P applications such as Kazaa and iMesh. In FastTrack, a semi-decentralised architecture is introduced where there is no centralised server but some nodes are rendered more capable than others, called supernodes. A supernode is similar to that of a normal node but it has higher capabilities in terms of computational power and network bandwidth relative to the other nodes on the network. Such nodes are used to make the network more efficient and to increase scalability. Gnutella is similar to that of FastTrack but without any supernodes. Gnutella is a completely decentralised P2P network and is used by popular application such as Limewire. To make the network more scalable, ultrapeers are used which are nodes with a high number of peers. However, in practice, the search algorithm employed by Gnutella was insufficient, making the network unscalable. Despite this drawback, Gnutella was one of the first pure P2P networks and it motivated the generation of other decentralised architectures, including Freenet. More information can be found on [33]. Both FastTrack and Gnutella are classified as second-generation P2P networks. Third generation P2P networks are fully decentralised and scalable P2P networks with the ability to keep request senders and receivers anonymous. Examples of third-generation P2P networks include Tor [10] and Freenet, which were created to promote freedom of speech in controlled environments. When examining anonymity systems, it is important to define what exactly is the anonymity offered.

The anonymity offered is not as complete as one may expect from these systems. According to [22], none of the current systems at the time of writing made any attempt to make it hard for an attacker to work out whether or not someone is running the file-sharing software. From this investigation on Freenet, it has been found that it is indeed extremely trivial for anyone running a node to find the real IP address of its peers, also known as friends. However, this is not as unsafe as it sounds. Most often, the anonymity provided in these systems refers to the fact that an attacker should not be able to find the true identity of the originator of an action such as a data request or data supply. In most cases, this is enough to allow anyone to do anything without having to worry about having their identity being linked to an action.

One of the most common method for P2P networks to provide anonymity is to assign pseudonymous addresses to the users. Communication between the nodes would then rely on their pseudonymous address rather than their actual IP address. It becomes crucial that attackers cannot link a pseudonymous address to the real IP address it is assigned to. However, there are existing attacks on unlinkability as documented in [5,6]. Freenet, as described below, employs a different strategy.

4.2 Freenet

Freenet as defined in [25,26], is a distributed anonymous information storage and retrieval system. The design goals are to provide the following:

1. anonymity for both producers and consumers of information
2. deniability for storers of information
3. resistance to attempts by third parties to deny access to information

4. efficient dynamic storage and routing of information
5. decentralisation of all network functions

Freenet operates at the application layer of the OSI reference model and is transport independent. It is stated in [24] that Freenet makes no attempt to provide anonymity for general network usage, only for Freenet file transactions.

The architecture of Freenet is simple. It is implemented as an adaptive P2P network of nodes that queries one another to store and retrieve data files which are assigned location-independent keys. Each node has a local datastore and the storage space is made available to the network, making Freenet a distributed storage system.

From Freenet 0.7, a Freenet node can operate in two modes, opennet and darknet. In opennet, a node connects to random stranger nodes on the network. On the other hand, a darknet is a pure Friend-to-friend (F2F) network and in this mode, a node connects only to friends. Each node has a location reference which is a fingerprint of the node and nodes swap their location references with each other in order to connect. In opennet, a node connects to a seednode supplied by Freenet but in darknet, the user of the node must obtain node location references through other channels.

Below is a detailed description of the workings of Freenet in which I have made references to the Freenet source code. To obtain the Freenet source code, please see Appendix B and [40].

4.2.1 Keys and searching

Each data file is assigned a unique binary key which is obtained from a hash function (SHA-256 is currently being used). The key serves as an Uniform Resource Identifier (URI) for the file and so any user can search for the file using the key. The cryptographic functions are handled by the Java classes in freenet.crypt package whilst file key generations are handled by the Java classes in the freenet.keys package.

In general, there are four types of file keys KSK, SSK, USK and CHK which are detailed in [37]. Only CHK is described below because it is the only type of key used in this investigation, please refer to [37] for more details.

Content Hash Keys (CHK) are derived by directly hashing the contents of the files so each file is assigned a unique hash (as long as SHA-256 remains unbroken). Files are encrypted by a randomly-generated encryption key so that storers of the files can deny any knowledge of the contents of the data stored on their storage space. This provides deniability as described in the previous section. Below is a sample CHK generated after a file is uploaded which can be used as an URI to the file:

CHK@SVbD9H̃M5nzf3AX4yFCBc-A4dhNUF5DPJZLL5NX5Brs,
bA7qLNJR7IXRKn6uS5PAySjIM6azPFvK∼18kSi6bbNQ,
AAEA– –8

There are three distinctive parts separated by commas:

1. **SvbD9 HM5nzf3AX4yFCBc-A4dhNUF5DPJZLL5NX5Brs,** is the actual hash of the file.
2. **bA7qLNJR7IXRKn6uS5PAySjIM6azPFvK∼18kSi6bbNQ,** is the decryption key that can be used to unlock the file.
3. **AAEA– –8** is used for cryptographic purposes.

4.2.2 Routing and Data Retrieval

Freenet employs a key-based routing algorithm similar to distributed hash tables [34] and requests are forwarded on a hop-to-hop manner. Below is a description of how data requests are routed:

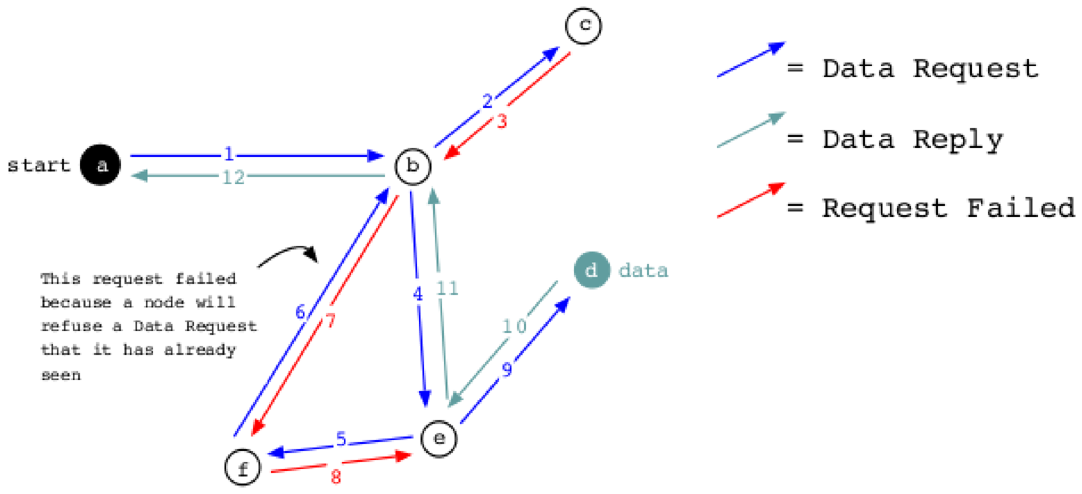

start
a
b
c
d
data
e
f
1
2
3
4
5
6
7
8
9
10
11
12
This request failed because a node will refuse a Data Request that it has already seen
= Data Request
= Data Reply
= Request Failed

Figure 4.1: Routing of data requests in Freenet. Figure extracted from [25]

In this example, Node a is the true originator of a data request and node d is the true data source. Node a sends a data request to node b which then according to key closeness (explained below), forwards the request to node c. Node c searches its own datastore for the data and returns a DATA_NOT_FOUND message to node b. Node b then forwards the request to node e which subsequently forwards to node f. Node f then forwards the request back to node b but because node b has already seen this request, it returns a failure message to node f. Since node f has no more peers, it sends a DATA_NOT_FOUND message to node e. Node e then forwards the request to its second choice node d which has the data. The data is returned from d via e, b and finally a. According to the Freenet 0.7 specification, for security reasons, returned data is only cached on the nodes which are in the return path and are more than three hops away from the request originator. The data source and the requested key is recorded on the routing table on each of the hops in the return path so that they know where to forward future requests for the key. However, it is important not to be confused with the concept of data source. From node b's perspective, the data source is node e and similarly, from node e's perspective, the data source is node d. However, node e has no way knowing whether node d is the true data source.

In general, a data request has a unique identifier (UID), hopes-to-live (HTL), depth counter as well as the requested file key. Below is an explanation of the significance of locations based on the requested file key and HTL value.

Locations

Each Freenet node has a location value, called node location, which is a numerical value between 0 and 1. This location may vary to adapt to the network topology around the node but the value should not vary by a significant amount in order to preserve keyspace specialisation. In addition, nodes swap this information with their peers. This is known as location swapping and the algorithm for assigning locations differs in opennet and darknet [38]. The purpose of location swapping is to make sure the locations are good fit for the network topology the node is connected to and to enhance routing. A node has the knowledge of the location of its peers and the locations of its peers' peers.

To request for data, a user must be in possession of the file key and each file key also has a numerical value between 0 and 1, called key location. A request message is then sent from his client to his own node. The node then searches its own datastore for the data and returns the data if found. If no data is found, the node would forward the request to the peer which has a peer whose node location is closest to the key location of the requested key. This is known as Friends-of-a-Friend (FOAF) routing and is enabled by default. The algorithm for determining key closeness was taken from the Location class in the freenet.node package and shown below:

```
private static double simpleDistance(double a, double b){
        if (a >b) return Math.min (a - b, 1.0 - a + b);
            else return Math.min (b - a, 1.0 - b + a);
        }
}
```

In the source code, **a** represents the node location whilst **b** represents the key location. As can be seen from the algorithm, locations are circular. For example, the distance between 0.99 and 0.01 is 0.02. It can also be observed that the maximum distance returned for any given pair is 0.5 e.g. one location is 0.9 and the other is 0.4.

Due to this search algorithm, it is anticipated that over time, a node would become specialised in a specific section of the keyspace and this would increase the efficiency of search and routing.

The Hops-to-live counter

Each request has a hops-to-live counter (HTL) and a depth counter. By default, the HTL counter is initialised to 18 (variable by the user) and the depth counter is initialised to a random value. The HTL counter is analogous to the Time-To-Live (TTL) counter in IP packets. The HTL counter is decremented as it is forwarded to the next hop whilst the depth counter is incremented. From my examination of the source code, I have found that a node would decrement the HTL counter and then check whether it is 0 before the request is forwarded. If the HTL becomes 0, the request would not be forwarded and a DATA_NOT_FOUND message is returned upstream back to the originator of the data request using the value of the depth counter.

However, it is obvious that the value on the HTL counter can be used by an attacker to deduce whether the request was originated by any of its neighbour nodes. Freenet tackles this problem by introducing probabilistic routing. When the HTL counter has the maximum value permitted by the node e.g. 18 or the minimum value (which is 1), there is only a probablistic chance that the HTL counter will be decermented by the node forwarding the request. Probabilistic routing has the effect that of preventing a malicious node from probing neighbour nodes for data with request that has HTL value of 1 because such a request may still be forwarded by victim node. At this point, it is useful to know that the Node class in the freenet.node package is the main class of the Freenet source code and is used to represent an user's Freenet node. The probabilistic mechanism is represented in the Node class and is described below.

Each node has a maximum permitted HTL value, 18 by default but modifiable by user. Every time a node starts up, two random numbers are generated using the Pseudo Random Number Generator (PRNG) from the java.security.SecureRandom class. Each is compared to a fixed value and the result of the comparisons determine whether the node would decrement the HTL counter of a request when it has the maximum permitted value and when it is 1. From the source code, the comparison algorithms are as follows:

```
decrementAtMax = random.nextDouble() <= DECREMENT_AT_MAX_PROB;
decrementAtMin = random.nextDouble() <= DECREMENT_AT_MIN_PROB;
```

It is observable from the Node class that the fixed value for DECREMENT_AT_MAX_PROB is 0.5 and the fixed value for DECREMENT_AT_MIN_PROB is 0.25. This means that the probability for a node to decrement at maximum HTL value is 0.5 whilst the probability for a node to decrement at minimum HTL value is 0.25.

Additionally, each of the node's peers is represented by an instance of the PeerNode class which can be found in the freenet.node package. A Node object makes use of a PeerManager object (also in the freenet.node package) which manages a list of PeerNode objects and handles location swapping. More importantly, each PeerNode object has its own probabilistic routing policy which has the same prinicple as the algorithm desribed above but different probabilities are used. The decisions are also determined when the node starts up. The code is as follows:

```
decrementHTLAtMaximum = node.random.nextFloat() <Node.DECREMENT_AT_MAX_PROB;
decrementHTLAtMinimum = node.random.nextFloat() <Node.DECREMENT_AT_MIN_PROB;
```

In the source code above, Node denotes the node to which the PeerNode object is associated with. It can be seen that the probability for a PeerNode to decrement at maximum and minimum HTL is slightly less than 0.5 and 0.25 respectively.

Since requests are forwarded in a hop-to-hop manner, if the node receives a request, it must have been forwarded from one of its peers. Subsequently, if the request is not originated by the node itself, it would be associated with a source node (a PeerNode object). Upon receiving the request, the node would search its own datastore and if no data is found, it would seek to forward the request. Before forwarding the request further, it would decrement the HTL counter and if the HTL value is at maximum or minimum value, it would be decremented according to the probabilistic routing policy of the PeerNode object rather than the policy of the node. The decrementing HTL value is handled by the following code from the Node class:

```
public short decrementHTL(PeerNode source, short htl) {
            if(source != null){
                  return source.decrementHTL(htl);
            }
            if(htl >= maxHTL) htl = maxHTL;
            if(htl <= 0) {
```

```
7           return 0;
8         }
9         if(htl == maxHTL){
10           if(decrementAtMax || disableProbabilisticHTLs) htl– –;
11              return htl;
12        }
13        if(htl == 1) {
14           if(decrementAtMin || disableProbabilisticHTLs) htl– –;
15              return htl;
16        }
17        return – –htl;
18        }
19}
```

Line 2 shows a test to see whether a source node is attached to a request. If a source exists, the HTL is decremented according to the policies in the PeerNode object. Line 3 shows that if the HTL value of the request is larger than the maximum HTL value of the node, the HTL value of the request would be changed to the maximum HTL of the node. Line 9 shows that if the HTL value equals to the maximum HTL value of the node and if `disableProbabilisticHTLs` (used to disable probabilistic routing) is false, whether the HTL value is decremented depends on `decrementAtMax` which is probabilistic. Line 14 shows that if HTL value is 1, and if `disableProbabilisticHTLs` is false, whether the HTL value is decremented depends on `decrementAtMin` which is probabilistic. Otherwise, the HTL value is decremented by 1, as shown on line 17.

4.2.3 Security of Freenet

A useful categorisation mechanism for anonymous communication is given in [36] which categorises the security of an anonymity systems on three fronts:

1. **Type of anonymity:** sender or receiver anonymity
2. **Type of adversary:** local eavesdropper, a malicious node, collaboration of malicious nodes or malicious web server
3. **Degree of anonymity:** ***beyond suspicion:*** from an attacker's perspective, the detected user appears no more likely to have originated the action than any other node; ***probable innocence:*** from an attacker's perspective, the detected user appears no more likely to have originated the action than to not to have; ***possible innocence:*** from an attacker's perspective, there is a nontrivial probability that the detected user did not originate the action

In [24], it is claimed that sender anonymity is preserved beyond suspicion against a collaboration of malicious nodes. This is reasonable since requests are forwarded in a hop-to-hop basis which means that there is no way for a node to determine whether the previous node originated or simply forwarding the request. However, sender anonymity is not preserved against a local eavesdropper. This is because a local eavesdropper can be the first node a client contacts and there is no protection on messages between the client and the first node it contacts.

Interestingly, it is co–stated in [24,30,31] that a correlation attack is possible where an attacker can deduce the true identity of the request originator by observing the HTL value and key closeness of the file key requested in a data request. Since the search algorithm in Freenet is designed so that the key closeness between the key and the node handling the request decreases with each request forwarding, a malicious node can examine the key closeness between the requested key and its node location. If there is a big difference e.g. approx. 0.5, then it can be concluded that the node is quite high up in the search route which implies that the request may have been originated by one of the nodes nearby. This suspicion can be further confirmed by examining the HTL value.

Since this research project is concerned with information leakage, this type of correlation attack serves as a perfect scenario for an investigation. In this case, the observable outputs are the HTL counter and key closeness and the information leakage from these two observables were measured as shown below.

4.3 Investigation setup

4.3.1 Investigation objectives

As described above, the main type of attack studied in this investigation is a correlation attack by examining the HTL value and key closeness on the data requests. The objectives of this investigation were as follows:

1. To measure how much information can be learned about the identity of the orginator of a request by observing the HTL value
2. To measure how much information can be learned about the identity of the orginator of a request by observing the key closeness
3. To verify whether Freenet is susceptible to correlation attacks of this kind

4.3.2 Topology design and implementation

Since a network application is being investigated, it was crucial that a realistic network topology was designed in order to simulate realistic interactions and behaviour. Below is a diagram of the topology used:

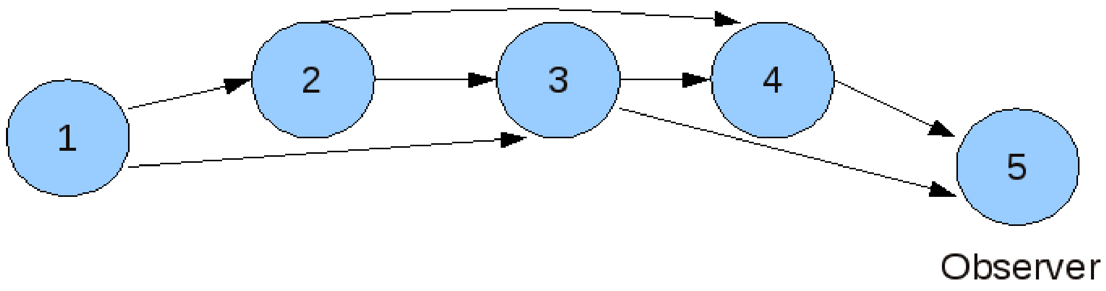


Figure 4.2: Darknet topology used in this investigation

In consideration of the Freenet protocol design, the topology was designed with the following characteristics:

Number of experiments: two experiments were carried out. In the first experiment, the maximum HTL value for all nodes was 18 but this was changed to 4 in the second experiment. The purpose was to examine the difference in probablistic routing with and without probablistic routing when HTL value is 1.

Number of nodes: a total of 5 nodes were used in the network to simulate a small Darknet.

Connectivity: since Freenet recommends that a node should have at least 3 peers, each node had at least 3 peers(except for the two end nodes), as shown in figure 4.2. This also increased the fault tolerance of the network. Note that FOAF routing (as described in section 4.2.2) was enabled in all nodes.

Type of file key: Content-hash key (CHK) was chosen to be the most suitable because no actual file upload is required in order to generate the keys.

Number of data requests: up to 2000 CHK keys were randomly assigned to each node and the nodes would send data requests requesting for data with these keys. Note that only 1000 observations were taken into account in the final results. This redundancy was necessary to take into account undelivered requests as it was found that out of the 2000 requests sent from each node, on average, around 400 of such requests were not observed by the eavesdropper node. One of the reasons for this was that due to the latency in node restarts, some requests were enqueued in the nodes' cache but because they were engaged in dealing with new requests, the queued requests were never forwarded. Note also that different sets of keys were used in the two experiments to prevent ensure requests were fresh and unique.

Interactions: an end node was chosen to represent the eavesdropper and the other four nodes represented normal users.
Each of the four nodes sent 2000 data requests and observations were captured by the eavesdropper.
Whilst a node was sending data requests, one node was randomly chosen from the

three other nodes on the network (excluding the eavesdropper) to restart after every 10 requests were sent. The purpose for node restart was to change the probabilistic routing policies of the nodes in order to increase the randomness in routing. This was necessary to take into account the fact that normal users may restart/turn off their computers and so their Freenet nodes are also restarted.

It was originally planned that all four nodes should be restarted after each data request sent but it was found that a node takes least five minutes to startup completely and node 4 took as long as twenty minutes. Since each node was required to send 200 requests, with this high latency, it became apparent that the requesting node should not be restarted. In addition, due to the small network topology, it was best to randomly choose a node to restart after a fixed interval to avoid requests from being cached which would allow enough time for a node to start up completely. It was the chosen that 10 data requests was an appropriate interval as this implies that a total of 200 node restarts would be achieved for each set of 200 requests sent. Also, this rate should be much higher than the restart rate on a real network within the time of sending 2000 data requests.

4.3.3 Naive Freenet Client

In order to obtain 8000 CHK file keys, an automated solution was needed. Fortunately, Freenet provides a well documented Freenet Client Protocol (FCP) 2.0 Specification [41] which allows developers to develop clients that can communicate with Freenet nodes on localhost port 9481. Subsequently, for this investigation, a naive Freenet client called Freenet-NaiveClient was written according to this specification. Please see Appendix B for details. The core functional requirements of this client were as follows:

1. To automate the process of obtaining CHK keys
2. To automate the process of sending data requests for given CHK keys

Before examining the workings of the client, it is necessary to describe the protocol for uploading and retrieving data. Below is the sequence of messages for retrieving data according to FCP 2.0. For more details, please see [41]:

Client → Node: ClientHello message

Node → Client: NodeHello message

Client → Node: ClientGet message. In this investigation, ReturnType was set to *direct* and Persistence was set to *connection*. Since only the HTL and key closeness of data requests were examined, it was enough for a node to send a data request without performing any real downloading. For details of the settings in this investigation, please refer to the source code of the client.

Node → Client: If data is found, a DataFound message is returned and if the ReturnType parameter in the ClientGet message is set to connection, which means the data should be returned on screen, an AllData message is returned with the data. Otherwise a GetFailed message is returned. In this investigation, only GetFailed messages were returned because no data were not uploaded to the network.

Below is the sequence of messages for uploading data according to FCP 2.0:

Client → Node: ClientHello message

Node → Client: NodeHello message

Client → Node: ClientPut message. In this investigation, it was found that an actual file upload takes approximately 3 minutes and for 8000 files, this would be inappropriate. Fortunately, it was found that there was no need to performa actual uploads because Freenet provides a method for obtaining a CHK for a given file without performing the actual upload. This is achieved by setting the GetCHKOnly parameter in a ClientPut message to true. For details of the settings in this investigation, please refer to the source code of the client.

Node → Client: a URIGenerated message is returned indicating the final URI (CHK) of the data inserted. If the upload is successful, a PutSuccessful message would be returned following the URIGenerated message. Even if GetCHKOnly is set to true, PutSuccessful would still be returned.

The FreenetNaiveClient is an implementation of the above protocol. The most important methods are as follows:

1. `sayHello():` sends a ClientHello message to the Freenet node
2. `sendGet(String URI):` sends a ClientGet request to the Freenet node
3. `sendPut(int index, String persistence, boolean global, boolean URIonly):` this method calls the `generateFile` method to generate a file and then use the generated file to send a ClientPut request to the Freenet node. The index given as a parameter ensures that each file generated is unique.
4. `putFiles():` automates the process of obtaining CHK by continuously calling `sendPut`, supplying it with an index number. This process does not stop unless it can read a file named ENOUGH.txt which is generated by the Freenet node when 8000 CHK keys have been generated.
5. `getFiles():` automates the process of sending data requests by reading the CHK keys from a designated file named **CHKlist_<node name>.log**. This method calls the `restartNode` method after every 10 requests sent and returns when all the keys have been read from the file.
6. `restartNode():` this method randomly selects a node to be restarted and then calls the `executeShell` method with the path to the relevant shell script
7. `executeShell(String scriptPath):` this method executes a given shell script that executes a Secure Shell (SSH) command to remotely restart a Freenet node running

on another machine. For this investigation, public key SSH was set up between all the machines the nodes were running on so that nodes could be remotely restarted whilst avoiding password prompts.

8. `generateFile(int index):` responsible for generating a file and filling the file with the filename.

In this investigation, two types of client were needed. A client to upload file so that CHK keys can be generated and a client to send data requests. This was achieved by calling `putFiles()` and `getFiles()` in the `main` method respectively.

4.3.4 Modification to Freenet source code

In order to understand the Freenet source code, a logger class called `_Logger` was written inside an interface called `HTL_Logger`. Some classes in the Freenet source code already extended another class and this implementation was necessary to overcome the multiple inheritance problem. The logger class has the following methods:

1. `log(String msg):` writes the specified string of text to a file named _log.log
2. `logInHTL(long uid, Key key, short htl):` logs the given UID, HTL value and key closeness of an incoming data request. This method calculates the key closeness using the following code:

   ```
   Location.distance(currentNode.getLocation(), key.toNormalizedDouble())
   ```

   The Location class is located in the freenet.node package and the method `distance` calculates the distance between the location of the current node and the requested key. The algorithm is explained in section 4.2.2. This method is inserted in the constructor of the `RequestHandler` class because this class is responsible for handling every incoming data request. `RequestHandler` is located in the freenet.node package. This method is used to record observations at the eavesdropper's node and generates a log named HTL_incoming_log.log.
3. `logOutHTL(long uid, String key, short htl):` used to log the outgoing data requests from the request originator node and is inserted in the `createDataRequest` method in the `RequestSender` class. This class is responsible for sending data requests and `createDataRequest` is responsible for creating a data request to be sent. `RequestSender` is located in the freenet.node package. A log named HTL_out_log.log is created by this method.

Another modification was made to the Freenet source code. A class called `RequestDistributor` was written specifically to capture the CHK URI generated by the Freenet node in response to a ClientPut request. The captured URI was then randomly distributed to one of the four nodes. The core method in this class is `distributeKey(FreenetURI uri)` which performs the capturing and distributing of the CHK URI. It create a file called ENOUGH.txt once 8000 keys have been distributed. This method is inserted in the `schedule` method inside the `SingleBlockInserter` class. This class is responsible for inserting a block of data following a request from a client and the class is located in the freenet.client.async package.

4.4 Experiments

4.4.1 Description

Each node sent its set of 2000 data requests in the order of the node ID as shown in figure 4.2 (excluding the node 5). After each node finished sending the data requests, the HTL_out_log.log from the request originating node and the HTL_incoming_log.log from node 5 were recorded (the HTL_incoming_log.log was renamed to HTL_<node name of request originator>@netbook.log to avoid ambiguity, netbook is the name for node 5. See Appendix B for more details). A special parser application called LogParser was written specifically to parse extract the HTL and key closeness information from the log file HTL_<node name of request originator>@netbook.log. More information is given in Appendix B.

In the first experiment, the maximum HTL value of each node was 18 which is the default value. In the second experiment, the maximum HTL value of each node was changed to 4 as the network only consists of 5 nodes. The results were obtained and analysed as shown below.

4.4.2 Results

From the first experiment, the capacity from observations on the HTL values by the eavesdropper is 0.935 bits whilst the capacity from observations on key closeness by the eavesdropper is 0.009 bits. From the second experiment, the capacity observations on the HTL value observed by the eavesdropper is 0.724 bits whilst the capacity from observations on the key closeness observed by the eavesdropper is 0.018 bits. Please refer to section 4.6 for detailed results and graphical representations.

4.4.3 Results analysis

Since there are four nodes generating data requests in the topology shown in figure 4.2, a total of two bits of information is needed to fully describe the identity of the request originator. The probability matrix generated from observations on the HTL counter values in both experiments can be seen from figures 4.3 and 4.4 in section 4.6. Figure 4.3 shows that the information leakage from the HTL counter in the first experiment is close to one bit out of a possible two and that is a big leakage because this means that an eavesdropper with some knowledge of the network topology around his node would be able to learn as much as 50% about the identity of the request originator simply by observing the HTL counters in the requests passing through his node. Gaining knowledge of the network topology is not as hard as it may sound since a node has complete knowledge of its own immediate peers and also, the number of peers they have and their locations due to FOAF routing (see section 4.2.2). Looking at Figure 4.2, it can be seen that since FOAF routing was enabled, the eavesdropper did indeed have near-complete knowledge of the network topology. For example, even though node 1 is the furthest node away, since it is a peer of node 3, the eavesdropper still knows its existence because node 3 is directly connected to the eavesdropper. In fact, the only relationship the eavesdropper cannot see is that node 1 and node 2 are interconnected.

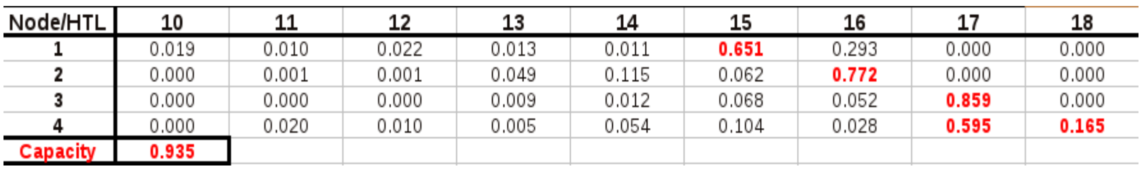
Looking more closely at figure 4.3 reveals the reason why there is such a big leakage. It can be observed that there were distinctive differences in the HTL counter values observed from the data requests passing through the eavesdropper node. For instance, only the requests from node 4 arrived at the eavesdropper's node with a HTL counter value of 18 and most of the requests from node 2 arrived with HTL counter value of 16. Such distinctive patterns were slightly flattened after the maximum HTL value were changed to 4 in the second experiment and the capacity fell by 0.2 bits approximately, as shown in figure 4.4. The only distinctive pattern from figure 4.4 was that most of the requests from node 2 arrived at the eavesdropper's node with HTL counter value of 2. One of the possible reasons for this decline was the fact that the effect of probabilistic routing when the HTL value is at minimum was introduced. With this effect, the probability for requests to arrive at the eavesdropper's node with a HTL counter value of 1 would be more evenly balance than a value of 15 from first experiment because there is a 75% chance that the nodes would not decrement the HTL value when it is at 1 but HTL values would definitely be decremented otherwise. This is evident from figures 4.3 and 4.4. Another reason contributing to the fall in leakage is that the requesting originator node was not restarted. Therefore, there was a possibility that all requesting nodes with the exception of node 4 all decremented the HTL at maximum value in experiment 1 but none decremented the HTL value at maximum in experiment 2.

Another objective for this investigation was to examine whether key closeness leak any information about the true identity of the request originator. The results are shown in figures 4.7 to 4.10. It is immediate obvious that the information leakage from both experiments were very minimal. Despite the insignificant readings, there is one noticeable difference in the results. The information leakage from key closeness in the second experiment has doubled the amount of leakage in first experiment. One of the possible reasons for this was that two different set of CHK keys were used in the experiments and so the increase was a result of the noise from the difference in the keys. Despite the rate of increase in leakage was 50%, the actual increase was a mere 0.009 bits, which is insignificant. With regards to the low leakage, there is a reason contributing to this and that is the network was not large enough. As described in section 4.2.2, over time, a node would become specialised in a specific section of the keyspace because requests are forwarded according to the distance between the key location of the requested key and the node location. This is true if the node location remains relatively constant and data belonging to a finite section of the keyspace would be formed in its datastore. However, this specialisation did not happen in this investigation due to two reasons: firstly, there was not enough requests traversing around the network and secondly, the locations of the nodes were constantly changing in order to adapt to the constantly changing network topology due to frequent node restarts.

4.5 Conclusion of investigation

From this investigation, it has been shown that assuming an eavesdropper has some knowledge of the network topology around his node, he can learn as much as one bit of information about the identity of the request originators by simply making observations from the HTL counter value in the requests. Due to the Friends-of-a-Friend (FOAF) algorithm, it is trivial for a malicious node to gain knowledge of the network topology with a radius of two hops. It has also be shown that the maximum HTL value relative to the size of the network is fairly significant as it has been shown in the second experiment that the amount of information leakage fell significantly when the effect of probabilistically decrementing the HTL counter at minimum HTL value was involved in the routing algorithm. Therefore, it is advised that the maximum HTL value is set as closely to the size of the connected network, if it is known. Lastly, there was almost no information leakage from key closeness due to the fact that the network topology was too small and was constantly changing. It is anticipated with a larger and more stable network with a longer experimental time such as a week, specialisation of key space would be visible. This implies that information leakage from key closeness and HTL counter would be much greater.

4.6 Additional results and figures



Node/HTL	10	11	12	13	14	15	16	17	18
1	0.019	0.010	0.022	0.013	0.011	**0.651**	0.293	0.000	0.000
2	0.000	0.001	0.001	0.049	0.115	0.062	**0.772**	0.000	0.000
3	0.000	0.000	0.000	0.009	0.012	0.068	0.052	**0.859**	0.000
4	0.000	0.020	0.010	0.005	0.054	0.104	0.028	**0.595**	**0.165**
Capacity	**0.935**								

Figure 4.3: Information leakage from HTL value when maximum HTL = 18

Node/HTL	1	2	3	4
1	0.123	0.373	0.140	0.364
2	0.154	0.393	0.044	0.409
3	0.315	0.000	0.314	0.371
4	0.031	**0.848**	0.001	0.120
Capacity	**0.724**			

Figure 4.4: Information leakage from HTL value when maximum HTL = 4

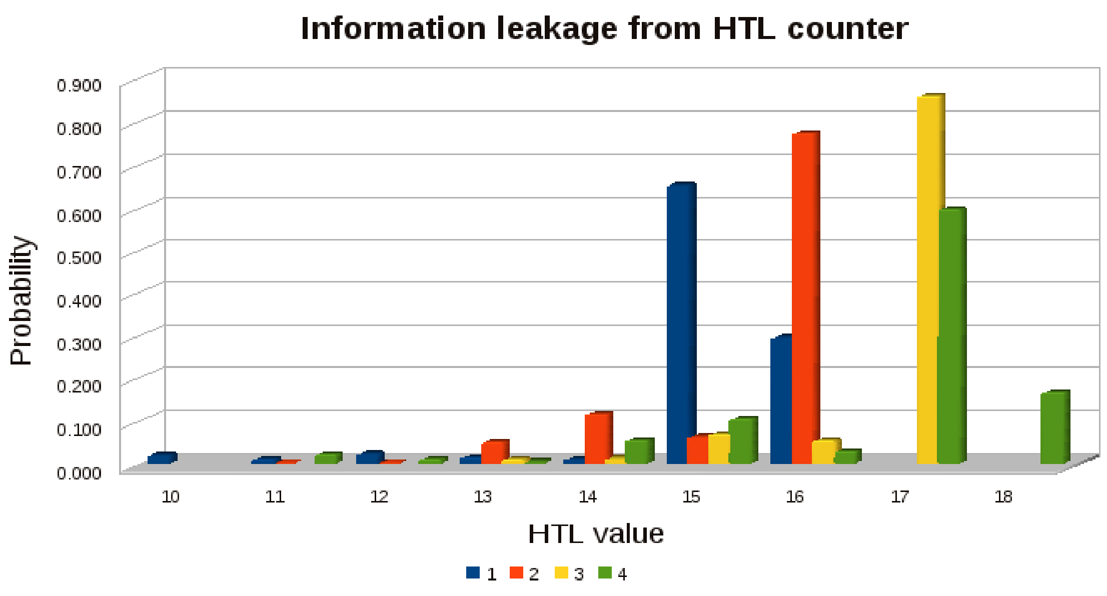


Figure 4.5: Graph showing information leakage from HTL value when maximum HTL = 18

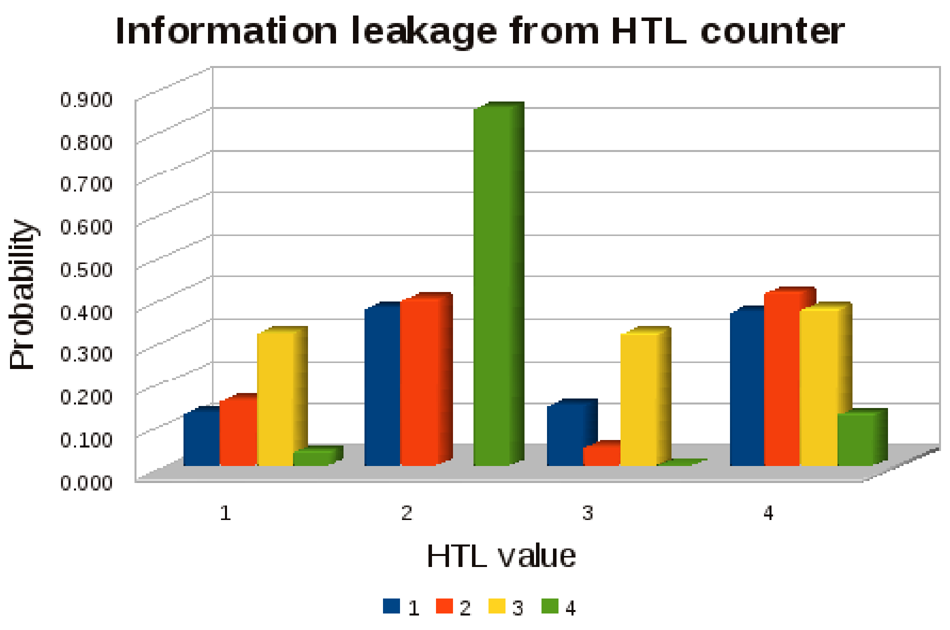


Figure 4.6: Graph showing information leakage from HTL value when maximum HTL = 4

Node/Key closeness	0	1	2	3	4
1	0.201	0.189	0.204	0.217	0.189
2	0.209	0.190	0.196	0.195	0.210
3	0.170	0.215	0.214	0.189	0.212
4	0.204	0.205	0.188	0.216	0.205
Capacity	0.0009				

Figure 4.7: Information leakage from key closeness when maximum HTL = 18

Node/Key closeness	0	1	2	3	4
1	0.190	0.169	0.220	0.204	0.217
2	0.228	0.202	0.195	0.177	0.198
3	0.223	0.208	0.188	0.183	0.198
4	0.230	0.182	0.195	0.203	0.190
Capacity	0.0018				

Figure 4.8: Information leakage from key closeness when maximum HTL = 4

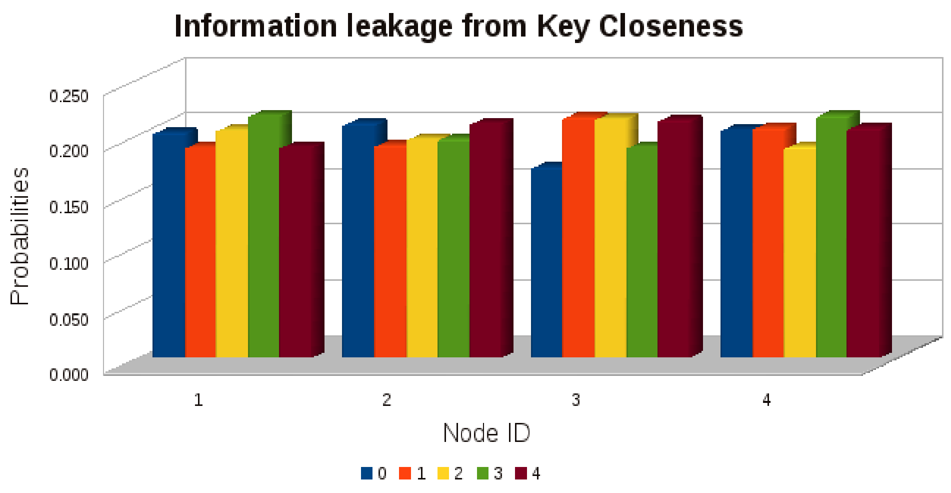


Figure 4.9: Graph showing information leakage from key closeness when maximum HTL = 18

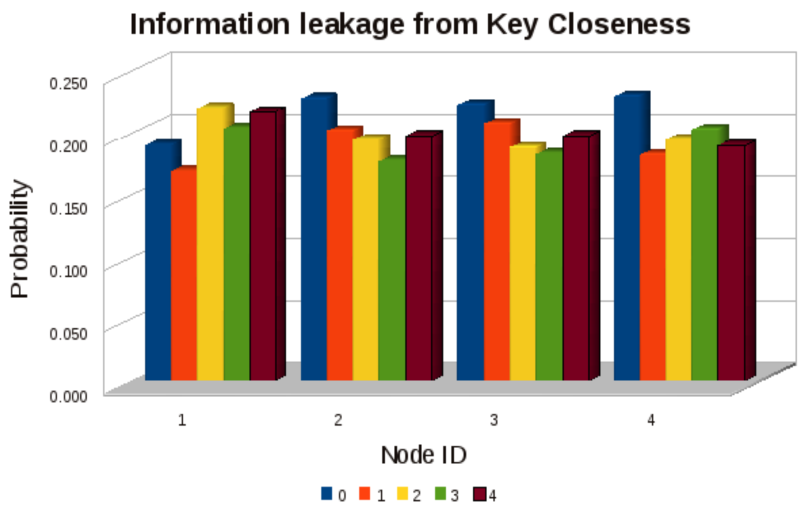


Figure 4.10: Graph showing information leakage from key closeness when maximum HTL = 4

Chapter 5

Investigation 3

In this investigation, the Anonymity Engine (A.E.) was used to measure the amount of information leakage from the timings of the message replies in a traceability attack on RFID e-passports. The attack was orginally discovered by Dr. Tom Chothia and the data used in this investigation was supplied by Vitaliy Smirnov. This investigation presents the first analysis of this data. Before I describe the findings, I shall present a brief introduction to RFID e-passports and the traceability attack.

5.1 Introduction to RFID e-passport

Since the events of September 11th, the U.S. has made major investments in authentication technologies such as Radio Frequency Identification (RFID) and biometrics. The e-passport, as they are sometimes called, is a type of passport which makes use of these two new technologies. According to [42], RFID e-passport are based on the guidelines issued by the International Civil Aviation Organisation (ICAO) which is a body run by the United Nations and is responsible for setting passport standards. It is in the ICAO 9303 document which details a call for the integration of RFID chips into passports. In general, the chip is capable of storing data and transmitting it in a wireless manner. For further details, please see [41]. Below is a description of a real attack on one of the RFID protocols for e-passports. Please note that in the subsequent sections, the term passport refers to e-passport.

5.2 Traceability attack

The Basic Access Protocol is one of a suite of protocols for RFID e-passports. It is responsible to prevent skimming attacks and it is an optional implementation. The protocol is explained below:

1. **Reader $\rightarrow$ Passport :** GET_CHALLENGE

2. **Passport** → **Reader :** NP
3. **Reader** → **Passport :** ***M*** = $\{$NR,NP, KR$\}_{Ke}$, MAC$_{Km}$($\{$NR,NP, KR$\}_{Ke}$)
4. **Passport** → **Reader :** $\{$NR,NP, KP$\}_{Ke}$, MAC$_{Km}$($\{$NR,NP, KP$\}_{Ke}$)

Once a passport is in range, the reader first scans the readable part of the passport to obtain the date of birth, date of issue and passport number and generates two keys **Ke** and **Km** based on those data. Hence, the keys **Ke** and **Km** are unique per person. After generating the keys, the reader sends a GET_CHALLENGE message to the passport which replies with a nounce NP to the reader. The nouce from the reader NR, the nounce from the passport NP and the reader's key KR, all encrypted together using the key **Ke**, is then sent by the reader together with the hash of that message. Please note that the hash is generated by a keyed hash algorithm with the key **Km**. Upon receiving the message, the passport first performs an integrity check by generating a hash of the message and check if it is the same as the hash received. It then uses its own key to decrypt the message and extract NR, NP and KR and check whether NP is the nounce it had sent to the reader in this authentication session. If verification was successful then the passport would send its own key KP to the reader. The session key is then generated by KR XOR'ed with KP.

An attacker can eavesdrop on the reader, make a copy of message **M** and then replay it to passports which are in-range. This allows the attacker to trace the person for whom's passport the message **M** was destined to, as demonstrated below.

Recall that the keys **Ke** and **Km** are generated based on a person's date of birth, the date of issue of his passport and his unique passport number. Subsequently, these keys represent an identity and become a great tool for a traceability attack.

Taking a closer look at message M above, it can be seen that the nouces are encrypted using **Ke** and the hash is generated using key **Km**, as described previously. Below shows the traceability attack:

1. **Reader** → **Random passport :** GET_CHALLENGE
2. **Random passport** → **Reader :** vNP
3. **Reader** → **Random passport:** ***Replay M*** = $\{$NR,**NP**, KR$\}_{Ke}$, MAC$_{Km}$($\{$NR,**NP**, KR$\}_{Ke}$)
4. **Random passport** → **Reader :** Authentication Failure Reply

Firstly, the malicious reader scans a random passport in range, obtain the date of birth (DoB), date of issue (DoI) and passport number (PassNum) to generate **vKe** and **vKm** and then sends a GET_CHALLENGE message to the random passport. The random passport then responds with a new nouce vNP to the reader. The reader ignores vNP and instead, replays the previously captured message M to the random passport. There are two possible outcomes after M is sent:

1. **hash check fails:** since the keys **Ke** and **Km** are generated based on the DoB, DoI and PassNum from a person's passport, they are unique to a specific passport. Therefore, if the key **Km** does not match the scanned passport, the hash check would fail. Subsequently, authentication fails.
2. **hash check succeeds:** if hash check succeeded then that means the key **Km** does match the passport. The passport would then decrypt, check the nouces and finds that the nounce is not vNP. Subsequently, authentication fails.

In both cases, the authentication fails and authentication failure replies are sent. In the first outcome, hash check fails which means that the key **Km** does not belong to the scanned passport and so, the passport does not belong to the victim. In the second outcome, the hash check succeeds. Despite the nouce check fails, the fact that the hash check succeeded means the key **Km** matches that of the passport and so the scanned passport belongs to the victim. The main challenge of the attack is how can the attacker distinguish between the first and second outcome from the subsequent authentication failure reply.

Due to the fact that RFID chips are low-powered and cryptographic functions require heavy computation, the two extra computational steps performed after a successful integrity check should cause a significant delay in the arrival time of the authentication failure reply for the second outcome relative to the first. This delay is the observable output in this investigation and the information leakage from this observable is measured below.

5.3 Measuring information leakage from e-passport

5.3.1 Data parsing

The timing data was collected by Vitaliy Smirnov and a parser (see Appendix C) was written to merge different timing data together to form lists of observations for the A.E.. In this investigation, passports from Britain, Germany, Greece and Ireland were examined. An additional experiment was carried out where the distance between the British passport and the reader was increased.

5.3.2 Results

The results were as follows:

Nationality	**Capacity**
British	0.9517
German	0.9716
Greek	0.9781
Irish	0.9921
British(distant)	0.9747

For detailed results, please refer to section 5.5 and Appendix C.

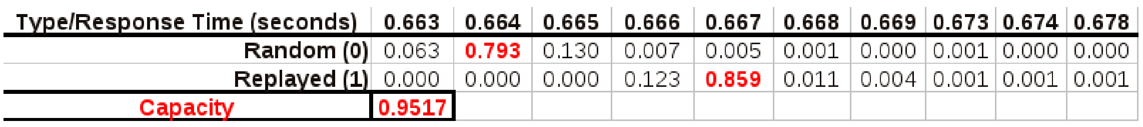
5.3.3 Results analysis

In this investigation, there are two probable outputs: random passport (mismatch) or replayed passport (trace success). Therefore, only 1 bit of information is required to fully describe the outcome. From the table in section 5.3.2, it can be observed that an attacker can learn almost the entire bit of information solely by observing the the time difference in the arrival times of the authentication failure replies. More surprisingly is the fact that this leakage is uniform for all kinds of passports examined in this investigation, see Appendix C. This means that an attacker can learn whether the passport belongs to the inteneded victom solely by observing the arrival times of the authentication failure replies. From section 5.5, figures 5.1 to 5.4 show the detailed results for British passport and the additional experiment. From figure 5.1, it can be seen that the difference in timing is very distinctive. Random passports most often return error messages in around 0.664s whilst replayed passport most often take 0.003s longer. An attacker with this knowledge would certainly be able to know with confidence whether the victim passport belongs to the victim.
An additional experiment was carried out by holding the British passport further away from the reader and the results are shown in figure 5.2. As it can be seen, the timing of authentical failed replies has increased in general but the distinctive patern remains. This means that using this method, an attacker can learn an extra 0.023 bits of information about British passports. The distinctive patterns can be seen in figures 5.3 and 5.4.

5.4 Conclusion of investigation

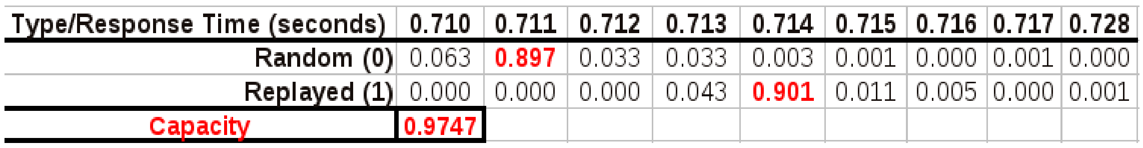
From the investigation, it has been shown that the implementation of RFID e-passports from the countries examined in this investigation has a serious vulnerability which allows an attacker to trace an individual using a replay attack. The results from this investigation shows that an attacker can distinguish whether the scanned passport belongs to a victim simply by observing the timings of the authentication failure replies from the passport.

5.5 Additional results and figures



Type/Response Time (seconds)	0.663	0.664	0.665	0.666	0.667	0.668	0.669	0.673	0.674	0.678
Random (0)	0.063	0.793	0.130	0.007	0.005	0.001	0.000	0.001	0.000	0.000
Replayed (1)	0.000	0.000	0.000	0.123	0.859	0.011	0.004	0.001	0.001	0.001
Capacity	0.9517									

Figure 5.1: Information leakage from reply arrival time



Type/Response Time (seconds)	0.710	0.711	0.712	0.713	0.714	0.715	0.716	0.717	0.728
Random (0)	0.063	0.897	0.033	0.033	0.003	0.001	0.000	0.001	0.000
Replayed (1)	0.000	0.000	0.000	0.043	0.901	0.011	0.005	0.000	0.001
Capacity	0.9747								

Figure 5.2: Information leakage from reply arrival time when passport is held away from reader

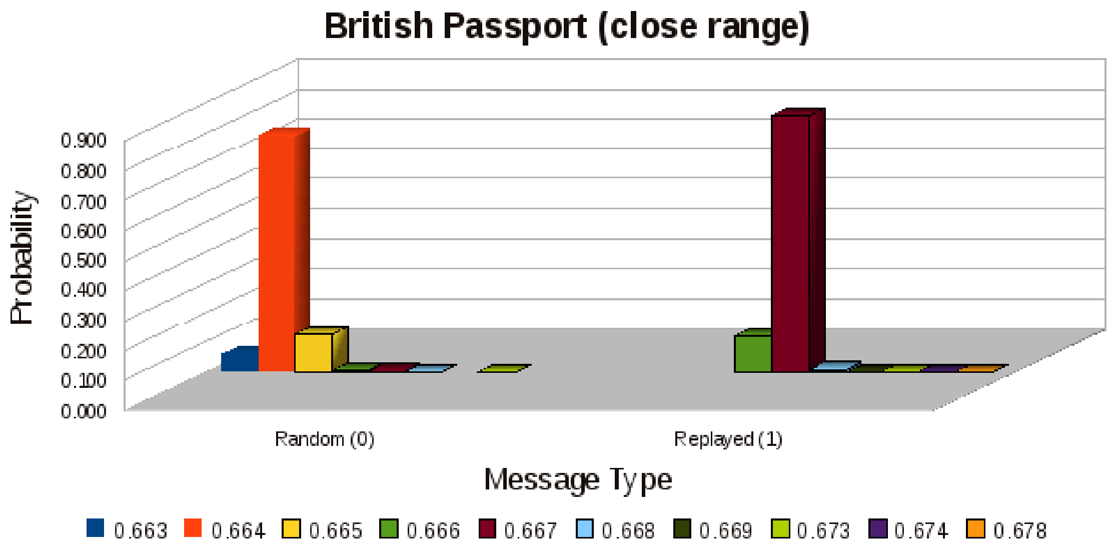


Figure 5.3: Information leakage from reply arrival time when passport is held away from reader

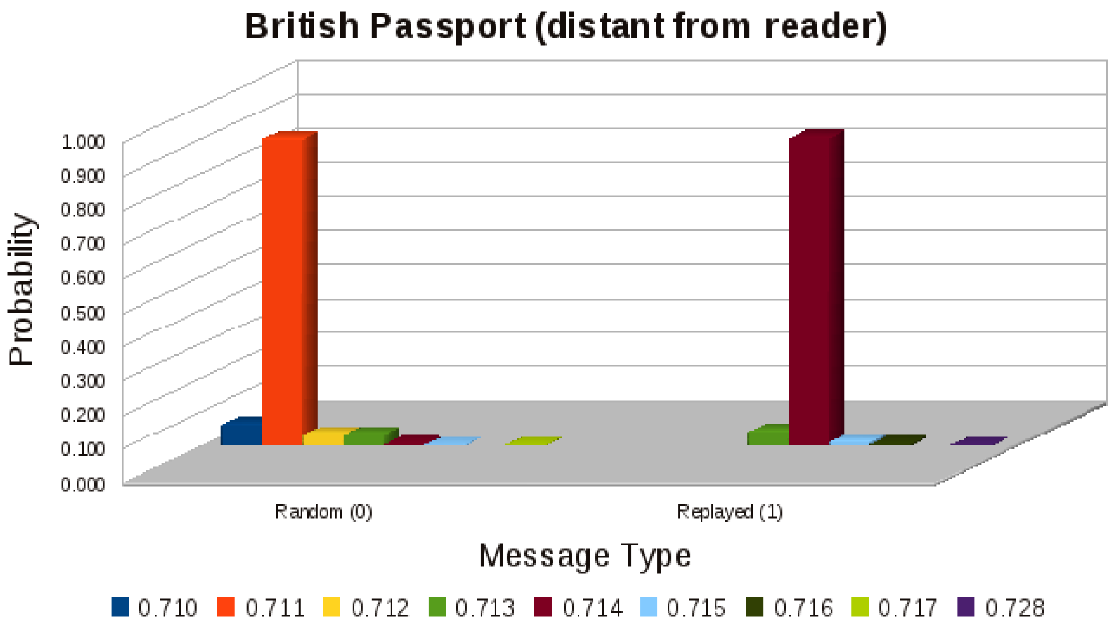


Figure 5.4: Information leakage from reply arrival time when passport is held away from reader

Chapter 6

Conclusion

In this paper, three different security systems were investigated and were found to have leaked side-channel information which could be used by an attacker to compromise the security of the systems. Since no prior assumptions were made in designing the experiments in each of these investigations, all subsequent findings are applicable to the systems in real settings.

In the first investigation, a naive password system was used as a demonstration of concept to test the efficiency and competence of the Anonymity Engine (A.E.) . Information leakage about the passwords given as input to the system was known to exist and the A.E. was expected to find this leakage. In the first experiment, due to the susceptibility of time measurements to background noise, the A.E. failed to pick up any significant amount of leakage from a realistic setting of the password system where a password is compared with the base password once. This problem was rectified in the second experiment where iterations was introduced which forced the password verifying application to match a given password repetitively for a finite number of times. This amplified the timing differences for each given password and the A.E. was able to pick up a more significant amount of information leakage, as first anticipated. Therefore, it can be concluded that the A.E. has proofed to be capable of measuring the information leakage from the observations made through a series of trial runs of the system but it is only suitable to measure information leakage from systems for which the observations made are not affected by any sort of noise. In other words, the competence of the A.E. depends on how well the observations represent the true state of the system.

In the second investigation, the workings of Freenet was investigated. It can be concluded that the HTL counter did leak a significant amount of information about the identity of the data request originators to an attacker and the amount of information leakage was reduced when the maximum HTL value was set closer to the network size in the second experiment. The reason was that the probabilistic routing for minimum HTL value was involved in the second experiment but not the first. This is particularly applicable to an attacker with some knowledge of the network topology around his node. Therefore, it is recommended that Freenet users should set their maximum HTL counter value to as close to the size of the network they are connected to as possible, if the network size is known e.g. in a Darknet. Also from the investigation, it was found that key closeness did not leak any significant amount of information to the attacker due to the size of the network used in the experiment. Given more time and a larger network size of at least 10 nodes, it is anticipated a much higher information leakage from key closeness would be found..

In the third and last investigation, RFID e-passports were examined. It can be concluded that all types of passports examined in this investigation, that is British, German, Greek and Irish passports are vulnerable to the traceability attack described in the paper. It is advised that some sort of padding is used to cover up the time difference and the amount of information leakage should be reduced. In a real attack, for the attacker to clearly distinguish whether the hash check succeeded from the timings of the authentication failure messages, it is anticipated that the attacker would first need to find out the time (T1) the passport takes to reply to any message. Therefore, given more time, the information leakage from the difference between |T1 − time| for reply from random passport and |T1 − times| for reply from a replayed passport can be measured.

As demonstrated in this investigation, the A.E. is applicable to a large number of security systems. An interesting direction is to measure the information leakage from the timings of keystrokes which can be captured from security systems such as wireless keyboards, the SSH network protocol, PIN entries at cash terminals and PIN entries for Chip and Pin payment terminals. It would be interesting to quantify the amount of leakage from these systems to confirm the threats the users are potentially facing everyday.

Bibliography

[1] Bar-El, H., "Introduction to Side Channel Attacks", Discretix Technologies Ltd. , 43 Hamelacha Street, Beit Etgarim, Poleg Industrial Zone, Netanya 42504, Israel.

[2] Chothia T., Chatzikokolakiz K., Guha A., "Calculation of Probabilistic Anonymity from Sampled Data", Technische Universiteit Eindhoven, Netherlands and University of Birmingham, UK.

[3] Chothia T., Chatzikokolakiz K., Guha A., "Statistical Measurement of Information Leakage", Technische Universiteit Eindhoven, Netherlands and University of Birmingham, UK.

[4] Kocher, P., Jaffe J., Jun B. (1998), "Introduction to Differential Power Analysis and Related Attacks", Cryptography Research, 607 Market Street, 5th Floor, San Francisco, CA 94102, USA.

[5] Murdoch S. (2006), "Hot or Not: Revealing Hidden Services by their Clock Skew", University of Cambridge - Computer Laboratory, UK.

[6] Murdoch S., Danezis G. (2005), "Low-Cost Traffic Analysis of Tor", University of Cambridge - Computer Laboratory, UK.

[7] Schneier B. (1996), *Applied Cryptography* (2nd edition), USA: Wiley

[8] Schneier B., Kesley J., Hall C., "Side Channel Cryptanalysis of Product Ciphers", Counterpane Internet Security , 3031 Tisch Way, 100 Plaza East , San Jose, CA 95128 , USA.

[9] Kocher P. (1995), "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", Cryptography Research, Inc. 2607 Market Street, 5th Floor, San Francisco, CA 94105, USA.

[10] Dingledine R., Mathewson N., Syverson P. "Tor: The Second-Generation Onion Router", The Free Haven Project

[11] Danezis G., Clayton R. (2007), "Introduction to Traffic Analysis", Microsoft Research Cambridge, Roger Needham Building, 7 J J Thomson Avenue, Cambridge, CB3 0FB, U.K.

[12] Cover T., Thomas J. (1991), "Elements of Information Theory", USA:Wiley

[13] Bryant R. (2001), "Machine-Level Representation of C Programs" in *Computer Systems: A programmer's perspective*(Beta Draft edition), pp. 89 – 200

[14] Bryant R. (2001), "Measuring Program Execution Time" in *Computer Systems: A programmer's perspective*(Beta Draft edition), pp. 449 – 483

[15] Boynton L., van Hoff A. "Source code for java.lang.String", http://www.docjar.com/html/api/java/lang/String.java.html [accessed 10 June 2009]

[16] Charras C., Lecroq T. (1997), "Exact String Matching Algorithms", http://www-igm.univ-mlv.fr/ lecroq/string/ [accessed 8 July 2009]

[17] Wikipedia (2009), "Anonymous P2P", http://en.wikipedia.org/wiki/Anonymous_P2P [accessed 20 July 2009]

[18] Bovet D., Cesati M. (2006), "Understanding the Linux Kernel" (3rd edition), USA: O'Reilly

[19] Kernighan B., Ritchie D. (1988), "The C Programming Language" (2nd edition), USA: Prentice Hall

[20] Wikipedia (2009), "System time", http://en.wikipedia.org/wiki/System_time [accessed 08 Sept 2009]

[21] (1997), "Using the RDTSC Instruction for Performance Monitoring", Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054, USA

[22] Chothia T., Chatzikokolakis K., "A Survery of Anonymous Peer-to-Peer File-Sharing", Laboratoire d'Informatique, Ecole Polytechnique, 91128 Palaiseau Cedex, France

[23] Rogers M., Bhatti S. "How to Disappear Completely: A Survey of Private Peer-to-Peer Networks", University College London, London WC1E 6BT U.K.and University of St.Andrews, Fife KY16 9SS, U.K.

[24] Clarke I., Sandberg O., Wiley B., Hong T. (2000), "Freenet: A Distributed Anonymous Information Storage and Retrieval System", Freenet Project Inc.

[25] Clarke I. (1999), "A Distributed Decentralised Information Storage and Retrieval System.", Division of Informatics, University of Edinburgh.

[26] Clarke I., Miller S., Hong T., Sandberg O., Wiley B. (2002), "Protecting Freedom of Information Online with Freenet", IEEE Internet Computing (Jan-Feb 2002) pp. 40 – 49

[27] Pretre, B. (2005), "Attacks on Peer-to-Peer Networks" (Semester Thesis), Distributed Computing Group, Dept. of Computer Science, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

[28] Sandberg, O. (2005), "Distributed Routing in Small-World Networks", available at http://freenetproject.org/papers/swroute.pdf [accessed 1 Aug 2009]

[29] Clarke I. (2003), "Freenet's Next Generation Routing Protocol", available at http://freenetproject.org/ngrouting.html [accessed 1 Aug 2009]

[30] Dhamija R. (2000), "A Security Analysis of Freenet", UC Berkeley School of Information

[31] FreenetWiki, Freenet 0.7 Security, http://wiki.freenetproject.org/FreenetZeroPointSevenSecurity [accessed 25 Aug 2009]

[32] Wikipedia (2009), "Freenet", http://en.wikipedia.org/wiki/Freenet [accessed 20 July 2009]

[33] Wikipedia (2009), "Gnutella", http://en.wikipedia.org/wiki/Gnutella [accessed 10 July 2009]

[34] Wikipedia (2009), "Distributed hash table", http://en.wikipedia.org/wiki/Distributed_hash_table [accessed 01 Aug 2009]

[35] Wikipedia (2009), "FastTrack", http://en.wikipedia.org/wiki/FastTrack [accessed 10 July 2009]

[36] Reiter M., Rubin A, "Crowds: Anonymity for Web Transactions", available at http://avirubin.com/crowds.pdf

[37] Freenet Wiki, "Using Freenet", http://wiki.freenetproject.org/FreenetZeroPointSeven [accessed 10 Sept 2009]

[38] Freenet Wiki, "Location Swapping", http://wiki.freenetproject.org/LocationSwapping [accessed 10 Sept 2009]

[39] Freenet Wiki, "Location", http://wiki.freenetproject.org/location [accessed 10 Sept 2009]

[40] Freenet Wiki, "Guide to the Freenet Source Code", http://wiki.freenetproject.org/FreenetSourceGuide [accessed 10 Sept 2009]

[41] Freenet Wiki, "Freenet Client Protocol 2.0 Specification", http://wiki.freenetproject.org/FreenetFCPSpec2Point0 [accessed 10 Sept 2009]

[42] Juels A., Molnar D., Wagner D. (2005), "Security and Privacy Issues in E-passports", *Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks table of contents*, Pages: 74 – 88.

Appendix A

Appendix for Investigation 1

A.1 Source code and execution

Please refer to the README.txt file in /code/investigation1/ directory on the CD. A sequence diagram of timer.c can also be located in the same directory.

A.2 Experiment results

Due to the size of the output listings, I feel that it is inappropriate to include the results in the Appendix. However, all results are included in the attached CD.

Results from experiment 1 can be located: /results/investigation1/experiment1/tc0/.

There are four files within this test case directory:

1. **AE_out.txt:** this is the output from the Anonymity Engine
2. **StringPW_avg.txt:** contains the timing information obtained for each password guess entry
3. **StringPW_matrix.txt:** a input-output probability matrix generated by timer.c
4. **StringPW_ob.txt:** a list of observations obtained from each trial run which is then supplied to A.E. and AE_out.txt is produced

Results from experiment 2 can be located at:
/results/investigation1/experiment2/observations_matrices/tc<test case index>.

Each test case directory contains the results from a particular test case and a different number of iterations in StringPW.java was used in each test case. There are four files in each of these test case directories which have the same meanings as listed above.

The table of overall AE_out results can be found /results/investigation1/experiment2/password.ods

Appendix B

Appendix for Investigation 2

B.1 Source code and execution

Please refer to the README.txt file in /code/investigation2/ directory on the CD.

B.2 Experiment results

Due to the volume of the outputs, they are included on the CD but not listed in the Appendix.
The results from the two experiments are separately stored in the directories described below which can be located in the /results/investigation2/ directory on the CD.

Since there were two experiments:

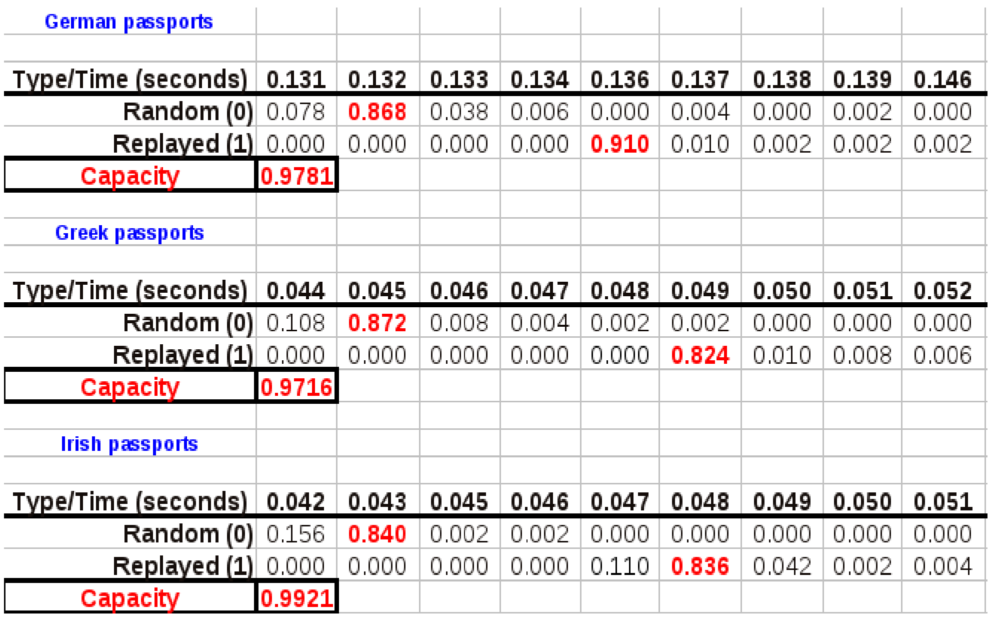
1. **/freenet_results_exp1:** - contains results from experiment 1
2. **/freenet_results_exp2:** - contains results from experiment 2

Please refer to the README.txt file in either of these directories for more information.

Appendix C

Appendix for Investigation 3

C.1 Information leakage from other e-passports



German passports

Type/Time (seconds)	0.131	0.132	0.133	0.134	0.136	0.137	0.138	0.139	0.146
Random (0)	0.078	**0.868**	0.038	0.006	0.000	0.004	0.000	0.002	0.000
Replayed (1)	0.000	0.000	0.000	0.000	**0.910**	0.010	0.002	0.002	0.002
Capacity	0.9781								

Greek passports

Type/Time (seconds)	0.044	0.045	0.046	0.047	0.048	0.049	0.050	0.051	0.052
Random (0)	0.108	**0.872**	0.008	0.004	0.002	0.002	0.000	0.000	0.000
Replayed (1)	0.000	0.000	0.000	0.000	0.000	**0.824**	0.010	0.008	0.006
Capacity	0.9716								

Irish passports

Type/Time (seconds)	0.042	0.043	0.045	0.046	0.047	0.048	0.049	0.050	0.051
Random (0)	0.156	**0.840**	0.002	0.002	0.000	0.000	0.000	0.000	0.000
Replayed (1)	0.000	0.000	0.000	0.000	0.110	**0.836**	0.042	0.002	0.004
Capacity	0.9921								

Figure C.1: A.E. output for other types of e-passport

The above table is not a complete listing of the results. All the results from this experiment can be found in the /results/investigation3 directory on the CD. It is recommended to refer to the README.txt file in the directory.

The source code of the parser can be found in the /code/investion3/ directory on the CD. Please refer to the README.txt file in the directory.