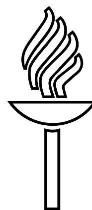
Matias Berg

FORGED FINGERPRINTS AND PGP ARCHITECTURE



UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS
2018

ABSTRACT

Berg, Matias
Forged fingerprints and PGP architecture
Jyväskylä: University of Jyväskylä, 2018, 48 p.
Computer Science, Thesis
Supervisor: Hämäläinen, Timo and Kiviharju, Mikko

The idea of this thesis is to find the most efficient way of generating new PGP fingerprints. This is done in order to try and create a fingerprint that is similar to a fingerprint of the target. Fingerprints from 2048-bit RSA keys are generated in four different ways to determine which one is the most time and space efficient. Multi-prime RSA seems to be the fastest and most space efficient when the number of primes that make-up RSA is chosen carefully.

Keywords: PGP, fingerprint, man-in-the-middle attack, RSA

TIIVISTELMÄ

Berg, Matias
Forged fingerprints and PGP architecture
Jyväskylä: Jyväskylän yliopisto, 2018, 48 p.
Tietojenkäsittelytiede, pro gradu-tutkielma
Ohjaajat: Hämäläinen, Timo and Kiviharju, Mikko

Tämän pro gradu-tutkielman tavoitteena on selvittää, että mikä tapa tuottaa PGP sormenjälkiä on tehokkain. Tämä tehdään siitä syystä, että halutaan löytää mahdollisimman tehokas tapa luoda samankaltainen sormenjälki kuin kohteen julkisen avaimen sormenjälki. Sormenjäljet lasketaan 2048-bittisistä RSA avaimesta neljällä eri tavalla, jotta selviäisi mikä niistä on aika- ja tallennustila-tehokkain. RSA useammalla kuin kahdella alkuluvulla toteutettuna näyttää olevan nopein ja vievän vähiten tallennustilaa olettaen, että alkulukujen määrä valitaan tarkoin.

Asiasanat: PGP, sormenjälki, välimieshyökkäys, RSA

FIGURES

FIGURE 1 DIRECT TRUST....10
FIGURE 2 HIERARCHICAL TRUST MODEL 10
FIGURE 3 PGP PUBLIC KEY BLOCK CREATED USING GNUPG 12
FIGURE 4 PGP PRIVATE KEY CREATED USING GNUPG 13
FIGURE 5 GNUPG OPTIONS WHILE GENERATING A KEY....24
FIGURE 6 KEY GENERATION WINDOW OF PGP DESKTOP 25

TABLES

TABLE 1: THE NUMBER OF RANDOM OR PRIME NUMBERS REQUIRED TO FILL UP THE FINGERPRINT SPACE. 35
TABLE 2: THE NUMBER OF PRIME NUMBERS REQUIRED TO FILL UP THE FINGERPRINT SPACE WITH MULTI-PRIME RSA. 36
TABLE 3: STORAGE SPACE REQUIRED TO ACHIEVE A CERTAIN FINGERPRINT SPACE IN KILOBYTES. 37
TABLE 4: TIME TAKEN TO CREATE A PRIME NUMBER OF CERTAIN BIT-SIZE....38
TABLE 5: TIME REQUIRED GENERATING ENOUGH PRIME NUMBERS TO FILL THE REQUIRED FINGERPRINT SPACE IN SECONDS...39
TABLE 6: TIME REQUIREMENT (DAYS) OF EACH APPROACH TO ACHIEVE DESIRED FINGERPRINT SPACE COVERAGE....40
TABLE 7: TIME TAKEN ON ONE ATTEMPT TO GET A MATCH OF AT LEAST 8 CHARACTERS....40

ABBREVIATIONS

CA	Certificate Authority
DSA	Digital Signature Algorithm
GnuPG	GNU Privacy Guard
IDEA	International Data Encryption Algorithm
MPI	Multiprecision Integer
PGP	Pretty Good Privacy
RSA	Rivest-Shamir-Adleman
SHA1	Secure Hash Algorithm 1

TABLE OF CONTENTS

ABSTRACT 2

TIIVISTELMÄ 3

FIGURES 4

TABLES 4

ABBREVIATIONS 4

TABLE OF CONTENTS 5

1 INTRODUCTION 7
1.1 Research Problem 8

2 THEORY 9
2.1 Trust Models for Public Key Systems 9
2.1.1 Direct trust 9
2.1.2 Hierarchical trust 10
2.1.3 Web of Trust 11
2.2 PGP Architecture 11
2.2.1 How PGP works? 12
2.2.2 Algorithms 14
2.2.3 PGP Key Structure 17
2.2.4 Security of PGP 19
2.2.5 Advantages and Disadvantages of PGP 20
2.3 Key Management 21
2.3.1 Key Generation 21
2.3.2 Key Exchange 22
2.3.3 PGP Principles for Verifying Keys 22
2.3.4 Storage of Keys 23
2.3.5 Replacement of Keys 23
2.4 Different Software Implementations of PGP 24
2.4.1 GNU Privacy Guard (GnuPG) 24
2.4.2 PGP Desktop 24

3 RESEARCH METHODOLOGY 26
3.1 Research Approach 26
3.1.1 Fingerprint creation in PGP 26
3.1.2 Practical implementation 27
3.1.3 Social Engineering 28
3.1.4 Primality testing algorithms 29
3.1.5 Multi-prime RSA 31

3.2 Data collection methods and techniques ... 31
3.3 Data Analysis ... 33

4 FINDINGS ... 35
4.1 Number of prime numbers to be generated ... 35
4.2 The space requirement ... 36
4.3 The time requirement ... 37
4.3.1 Time requirement for generating enough prime numbers ... 37
4.3.2 Other time requirements ... 39
4.3.3 Actual time taken ... 40

5 DISCUSSION AND CONCLUSION ... 41
5.1 Discussion of the main findings ... 41
5.1.1 Approach 1 ... 41
5.1.2 Approach 2 ... 42
5.1.3 Approach 3 ... 42
5.1.4 Approach 4 ... 42
5.2 Evaluation ... 43
5.3 Conclusions ... 44
5.4 Future work ... 44

REFERENCES ... 45

1 INTRODUCTION

"PGP is the closest you're likely to get to military-grade encryption" (Schneier 1996)

Bruce Schneier compared PGP to military-grade encryption back in 1996, which it basically was according to the US government who in the early 90's classified PGP and other encryption products using over 512-bit keys as non-exportable weapons (Schneier, 1996). Even nowadays PGP is a popular method for encryption especially relating to emails and it gained even more popularity with the Edward Snowden case in 2013.

The object of this thesis is to examine the PGP system and its vulnerability to man-in-the-middle attacks using keys generated from fingerprints. Quite often ownership of a key is verified by receiving the fingerprint by using another media and then comparing the fingerprint to the fingerprint generated from the key. Now the idea in this thesis is to intercept the first communication of the actual public key, calculate its fingerprint and generate another (impersonation) key, which has a similar if not identical fingerprint to pass the verification phase in some later use of the impersonation key.

This thesis will start by explaining the fundamentals of public-key encryption such as trust models and key management. The inner workings of PGP will be discussed including different types of algorithms used and the security side of the protocol. The chapter on theory is concluded by a section on practical implementations of PGP.

The third chapter will contain the practical implementations of the said attack and detailed explanation of the different attempts that will be examined. The fourth chapter includes the results of the experiments outlined in chapter 3. The fifth chapter will contain discussion of the results and the conclusions of the thesis.

1.1 Research Problem

The research question is: what is the effect of forged fingerprints on security of PGP architecture? This leads up to the following sub-question: what is the most efficient way to create a public key that has the required fingerprint?

This problem will be tackled using constructive research approach. Constructive research approach includes selecting a relevant problem, studying the field of study in question, designing one or more solution to the problem and testing these solutions (Lehtiranta, Junnonen, Kärnä & Pekuri, 2015). This approach was chosen to compare the different ways of calculating a new public key and its fingerprint and to determine which is the most efficient one.

2 THEORY

In this chapter we will discuss various aspects related to public key cryptography and PGP in more detail. Firstly we need to explain what public key cryptography is. Public key cryptography uses more than one non-identical keys. Usually there are two keys: a public key and private key. Public key is publicly available to everyone and entities who want to send a message to a person use this key to encrypt this message. Once the person who owns the private key receives this message, they can use the private key to decrypt it.

In this chapter we will start off with discussing trust models of public key systems. Then we will discuss and explore the PGP architecture in more detail, including how it works, what algorithms are used and evaluation of its security and usability. Third subchapter of this chapter is dedicated to key management. The chapter will close with a subchapter on the software implementations of PGP.

2.1 Trust Models for Public Key Systems

Oxford English Dictionary defines trust as a "firm belief in the reliability, truth, or ability of someone or something."(Oxford University, 2016) Trust is important in public key systems, but not as trust on the strength of the encryption system. The trust in this case is trust on whether a public key actually belongs to the entity claiming ownership. In this chapter three types of trust models will be discussed: direct trust, hierarchical trust and web of trust. (Network Associates, 1999)

2.1.1 Direct trust

Direct trust is the simplest of the trust models. In this trust model the user trusts all the public key person relationships, for example, due to the fact that

they have personally got the key from them. A diagram of this type of trust shown below in Figure 1. All the cryptosystems use this kind of trust in some format. (Network Associates, 1999)

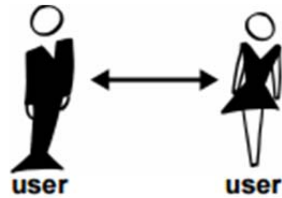


FIGURE 1 Direct trust (Network Associates, 1999)

2.1.2 Hierarchical trust

A hierarchical trust model consists of three types of roles: root certificate authority, certificate authority and user (Vacca, 2004). Root certificate authority, verifies the certificate authorities before they issue a certificate to the requestor. Certificate Authority (CA) is an authority that issues and verifies digital certificates (Rouse, 2007). Users are the entities that use this system to be able to trust that the public keys belong to the entity specified in these certificates. Figure 2 shows the layout of this type of trust model.

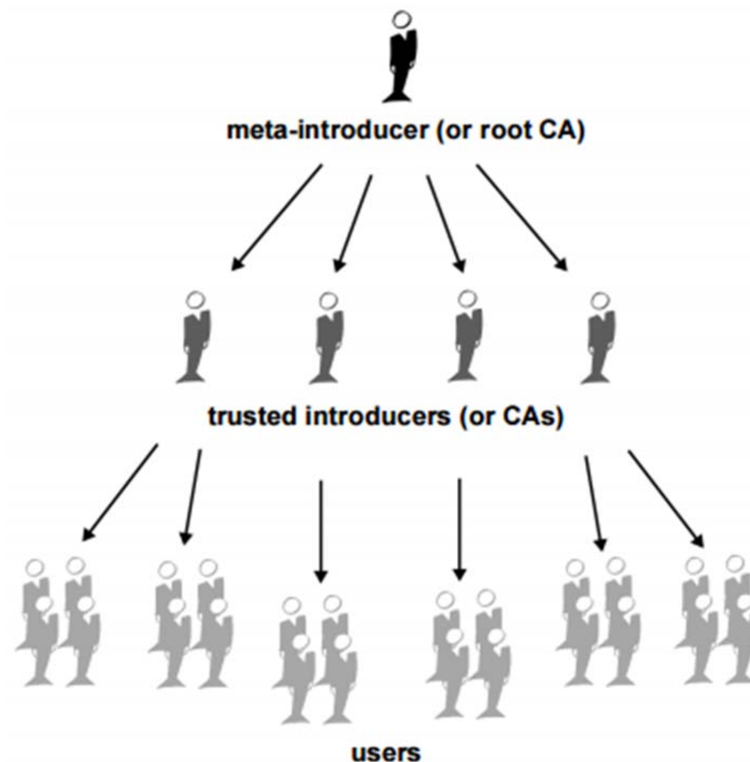


FIGURE 2 Hierarchical Trust Model (Network Associates, 1999)

To trust a person's certificate one can trace the route back to the trusted root CA (Network Associates, 1999). This expands the network of trusted personnel as there may not be a possibility to establish direct trust with everyone. These certificates generally include the owner's public key, the name of the owner and an expiration date. The CA is a so-called trusted third party, meaning that both of the communicating parties trust it and that it has verified the link between the public key and the owner.

2.1.3 Web of Trust

This is the trust model used in PGP. This model is a combination of direct and hierarchical trust models, but there is no central authority giving certificates. A user can establish trust on keys by getting them directly from another user or establish the trust by using the signatures the key has gained. When a user trusts a certificate they can sign it with their own signature and if someone trusts them then they should in theory be able to trust the certificate the user has signed (Chadwick, Young & Cicovic, 1997; Feisthammel, 1998). However, in practice there is an issue that it is hard to apply pressure to users to only sign keys that they trust, where as in a hierarchical model the CA would quickly loose reputation as a trustworthy authority if they didn't verify the keys properly.

2.2 PGP Architecture

PGP (Pretty Good Privacy) was created by Phil Zimmermann in 1991 (Lucas, 2006). He combined some of the common encryption methods at the time to create a tool usable by anyone with a computer. During the time of PGP's release the US government considered encryption to be a threat to national security and it was not allowed to export encryption software without a permit from the state (Lucas, 2006). Zimmermann put PGP together with its source code online for anyone to download. This caused the law enforcement of the US to launch an investigation on Zimmermann on the basis of violating federal arms-export laws (Sussman, 1995). The investigation lasted three years, but in the end it was dropped and no charges were filed against Zimmermann.

2.2.1 How PGP works?

To send an email using PGP one has to create a public and a private key. An example of these is shown below:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mI0EV5mq4AEEANtk3f/L/SGKrA49TpDsk6QaJbKeTenhOUo84yQEpu9JxuS1nuJk
LQkW3PE1SGSvncawbXUwp2sltJ1FmZCsU9+keXeawPkIUFMPYrc0vK7/jVnlaXVg
TMFtCIt0xCVHkDN0A/hZus1NpNokDq3HiK3SclVPOvNCQ0hATXzkQ7IPABEBAAG0
HUpvaG4gRG9lIDxqb2huLmRvZUBnbWFpbC5jb20+iLkEEwEIACMFAleZquACGwMH
CwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRDlIqpIiMQJK5srBACZjzieJ7FU
f9Q55UE7nJQdMMaJQISwiFEFK8C711KJmrgutGZRkmQX6HXdih2yW/SlNhEmkmoc
EyHeJpw5kwMSFeBH4sDBijI+3FooY6dAn85XAXeMrCIPuOHFXM0HGdy9MN1LtjFQ
BSFCbd3wJXgX8LOKT2+U5xggnRIHNhCAvLiNBFeZquABBADX1q/hs1TrKKERWq00
Uz+C5MJGHECZXlbCn/njYGQj+B5vJY1T9Qv/cOdbqBVvQsDJVJoytp9AqdXoM9yP
EW5NHlvy11g8Rp5pKCoqCKC4pRNRnseeToAtZ7KKGqiZXSrgDZoX61N27PkhztUx
RUnFn+25S1sixIl+N6YO1kzjKwARAQABiJ8EGAEIAAkFAleZquACGwwACgkQ5SKq
SIjECStqSgP/ZanLWT/opFxOS2xr50faQGfpCEd5hw0pdpEJBvwDG3C87jg4c17+
Jue504Lt62h2hM6s1JPiwsm55487uR3MfdtjHH+JWUfIgwIoA9/aYWUzsuf2QAOQ
PyeDGr6FF4MiP7T3Fzz4JMufBX/SnxfAQYq/RB8zVcVkfGRRveDV6jQ=
=w+yy
-----END PGP PUBLIC KEY BLOCK-----
```


FIGURE 3 PGP Public key block created using GnuPG

```
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: GnuPG v2

lQH+BFeZquABBADbZN3/y/0hiqwOPU6Q7JOkGiWynk3p4TlKPOMkBKbvScbktZ7i
ZC0JFtzxNUhkr53GsG11MKdrJbSdRZmQrFPfpHl3msD5CFBTD2K3NLyu/41Z5Wl1
YEzBbQiLdMQlR5AzdAP4WbrNTaTaJA6tx4it0nJVTzrzQkNIQE185EOyDwARAQAB
/gMDArMHL0oSCFdmxeFJZ5zuE6uyx9lyJtumgRU7mrLKU0qhnpVwQbSeKTQtVjr7
f50Puc3SWsuj3uocMArNClhB7uVnGg4BmeK9XkCOYL48uYajk8C5khjtZ/hkdb1S
6ALIhPuEjTg1LyO8fBmbYA7uQgI32lbL4ZlX/6zXah5eXH7zDpcdb43qqrq77AVe
YxVJqdfGUj/sf0hmRODj/VSSRDm46P6CXaaMZTbC4G6Entm4uOeGDFwqpl/nOq8t
zuaQkhEf+JDbgtwrTR9yTPoaqEgeUZGV+lfF0nyAMfOqYVCg+HMGCfNmo2Jph+Xy
f8C1n2rDaWoSx19DuDElaxSrXF/cVfkVHjXf5FP0f95/p/elkG4dlmoKRwEV4ct1
jocM0x7Iafd0o8m4SZac9EVQ+ZhpUVpCKT/HDYXDcQenEaIQgTuZXaF9tlT252Gq
FMmz2CnaQPPJ26mY2VqOjJPWzDzDlryR5B0gXZdgUo3WtB1Kb2huIERvZSA8am9o
bi5kb2VAZ21haWwuY29tPoi5BBMBCAAjBQJXmargAhsDBwsJCAcDAgEGFQgCCQoL
BBYCAwECHgECF4AACgkQ5SKqSIjECSubKwQAmY84niexVH/UOeVBO5yUHTDGiUCE
sIhRBSvAu9dSiZq4LrRmUZJkF+h13Yodslv0pTYRJpJqHBMh3iacOZMDEhXgR+LA
wYoyPtxaKGOnQJ/OVwF3jKwiD7jhxVzNBxncvTDdS7YxUAUhQm3d8CV4F/Czik9v
lOcYIJ0SBzYQgLydAf4EV5mq4AEEANfWr+GzVOsooRFarTRTP4LkwkYcQJleVsKf
+eNgZCP4Hm8ljVP1C/9w51uoFW9CwMlUmjK2n0Cp1egz3I8Rbk0eW/LXWDxGnmko
KioIoLilE1Gex55OgC1nsooaqJldKuANmhfrU3bs+SHO1TFFScWf7blLWyLEiX43
pg7WTOMrABEBAAH+AwMCswcvShIIV2bFYPA3yVg21SYBTxgTA3ZJxfodLaMaYNIa
LUbkDgNbOsxi5CsFggEIAD+DJ1/xZRAKzjrdMuD9kWjNVohfT1vYgVonuYTi4H5S
aneoaks9ocPevmb6eoTeUKliLPqbDTAmhr7wCCfvm5hqXVaUvLoQHV3m+qfFyE8D
c5xQ7lTzRZKf59KnePVrWgtruWA1NeWJ2yhdS+v87GHEgZl5VTnAkIk9oRZ1/x8K
PianfLrlbK7V3z6v3v7/Z1kDTiX7KNjQ8EtFqFfqkXZp62Pj4gn+jdbhZ2LaVJ08
nJa+TsoRQN+iJy3lrUyRQLNylWpebdibu6Llubipp1E2uw3xfE7oA8v3EZlc4met
UFL+MJmk1GXYDhj3gInzXjkWJxKxTxyvvEUgdQMtAuOVqV59imy3g+ktEnQYLMDM
0iffRHkggVGbAu4uywS9fk427cb93JkfUXGOC9AxTopSOjaSaNMTIYmJ1CeInwQY
AQgACQUCV5mq4AIbDAAKCRDlIqpIiMQJK2pKA/9lqctZP+ikXE5LbGvnR9pAZ+kI
R3mHDSl2kQkG/AMbcLzuODhzXv4m57nTgu3raHaEzqzUk+LCybnnjzu5Hcx922Mc
f4lZR8iDAigD39phZTOy5/ZAA5A/J4MavoUXgyI/tPcXPPgky58Ff9KfF8BBir9E
HzNVxWR8ZFG94NXqNA==
=d5aO
-----END PGP PRIVATE KEY BLOCK-----
```

FIGURE 4 PGP Private Key created using GnuPG

Now that the keys are generated using the algorithms explained later in this chapter a message can be written and encrypted. PGP first compresses the plaintext (Network Associates, 1999). This is done to save time and disk space. Technically this will also strengthen security since plain text attacks are more difficult if the text that is being encrypted is compressed and thus has less patterns. However, this affects only ciphertext-only attacks and most modern schemes are already designed against more advanced attacks. Like in many encryption systems a session key is generated after this and this is used to encrypt the message. This session key is then encrypted with the public key of the re-

ceiver (Network Associates, 1999). This key is send to the recipient alongside the encrypted message. Decryption is the reverse of this operation.

2.2.2 Algorithms

There are multiple algorithms that can be used in PGP. The selection depends on the software implementation that is being used.

Asymmetric algorithms

For public-key cryptography, GnuPG supports RSA (Rivest-Shamir-Adleman), DSA (Digital Signature Algorithm) and ElGamal. The most common public-key cryptography algorithm by GnuPG users is 2048-bit RSA (The GnuPG Project, 2016).

RSA(Rivest, Shamir & Adleman, 1983a) is an algorithm that works both for encryption and digital signatures (Schneier, 1996). The difficulty of breaking RSA encryption is based on factoring extremely large composite integers (The GnuPG Project, 2016). We show below conceptually, how to generate private and public keys:

1. Take two large prime numbers p and q.
2. Compute their product: $n = pq$.
3. Compute φ (the Euler's totient) of n: $\varphi(n) = (p-1)(q-1)$.
4. Choose a value for e, such that $1 < e < \varphi(n)$. And so that e and $\varphi(n)$ are relatively prime.
5. Compute a value for d, such that $ed = 1 \mod \varphi(n)$
6. The public key is n and e.
7. The private key is d. (Ireland, 2016; Schneier, 1996)

This is how the keys are generated as a textbook example. In real life there are usually standards such as PKCS #1 which impose additional requirements on the numbers chosen etc. to defend against certain types of attacks (RSA Laboratories, 2012).

To encrypt a message m one basically calculates the value of encrypted message c using the following: $c = m^e \mod n$. Likewise to decrypt the message c to get the message m one does the following: $m = c^d \mod n$ (Ireland, 2016). If $m > n$ then the solution is to split the message into blocks smaller then n and then combine the individual encryptions and decryptions as appropriate. In real life there will be padding added to the message and the message may be split into blocks to give it some randomness and make it harder to decipher.

The patent for RSA encryption mentions that creating the keys n and d is also possible with more than two primes (Rivest, Shamir & Adleman, 1983b). This approach was later titled as Multi-prime RSA. The advantage of using

more than two primes is primarily faster decryption using the Chinese Remainder Theorem and also faster key generation as generating and verifying smaller prime numbers take considerably less time. There is some research on how many primes can be used without affecting the security of the system, but that is not a huge concern in this project since we are trying to perform an attack on the system, but we do want to avoid using simple primes so that we have at least some secrecy.

ElGamal (ElGamal, 1985) can also be used both for encryption and digital signatures just like RSA, however the ElGamal signature algorithm is no longer supported by GnuPG (The GnuPG Project, 2016). The difficulty factor in this algorithm comes from calculating discrete logarithms in a finite field. (Schneier, 1996) The algorithm to generate the public and private keys is shown below:

1. Choose a prime number p.
2. Choose two random numbers q and x which are less than p.
3. Compute $y=q^x \bmod p$.
4. The public key is y, q and p.
5. The private key is x. (Schneier, 1996)

As with the RSA algorithm above, this is a text book example and if this was to be implemented in real life there would be additional requirements for p, q and x. To encrypt a message, the sender would choose a random integer k and compute $r = q^k \bmod p$ and $t = y^k M \bmod p$. Here M is the numerical representation of the message being sent. Now k is discarded and r and t are send to the recipient. To decrypt the message: $tr^{-x} = M$.

DSA is a standardized algorithm that can only be used for digital signatures. It is a variant of ElGamal and defined in the Digital Signature Standard (*Digital signature standard (DSS),* 2013) created by the US government. Both ElGamal and DSA use calculating discrete logarithms as its difficulty factor (Schneier, 1996).

Symmetric algorithms

The actual message in PGP is encrypted using a symmetric algorithm such as IDEA (International Data Encryption Algorithm; (Lai and Massey, 1991), TripleDES (Triple Data Encryption Algorithm) (National Institute of Standards and Technology, 1999), CAST-128 (Adams, 1997), Blowfish (Schneier, 1993)or AES (National Institute of Standards and Technology, 2001).

IDEA is a symmetric-key block cipher (Schneier, 1996). It was designed by James Massey and Xuejia Lai (Lai & Massey, 1991). IDEA consists of three functions: XOR, addition modulo 2^{16} and multiplication modulo $2^{16}+1$. These functions are mixed and applied to the blocks that the message is being split to (Schneier, 1996).

TripleDES is also a symmetric-key block cipher and it applies DES (Data Encryption Standard) three times to each data block, hence the name TripleDES. DES was developed by the US National Bureau of Standards based on work done by IBM (Schneier, 1996). DES uses 56-bit keys to encrypt and is no longer deemed secure on its own (Schneier, 1996). TripleDES uses three 56-bit keys K_1, K_2 and K_3 to encrypt so that K_1 encrypts, K_2 decrypts and K_3 encrypts. However, its more practical to use just two keys so that $K_1=K_3$. This is only slightly less secure then using three separate keys but still more secure than just using two instances of DES because of the meet-in-the-middle attack. The reverse operations are done to decrypt the message (Barker & Barker, 2011).

CAST-128 has a maximum key-size of 128-bits and a 64-bit block size. It was designed by Carlisle Adams and Stafford Tavares (Schneier, 1996). It is a DES-like Feistel Network cryptosystem and it appears to be resistant to several cryptanalysis attacks (Adams, 1997).

Blowfish is an algorithm designed by Bruce Schneier. It has a 64-bit block size and variable-length key size which improves the security compared to other encryption algorithms (Schneier, 1996) (Schneier, 1993).

AES is a standard that uses an algorithm called Rijndael developed by Joan Daemen and Vincent Rijmen (NIST, 2001). A big difference to other symmetric algorithms mentioned here in that it has a block size of 128-bits. It has a maximum key-length of 256 bits (Erdelsky, 2002).

Hash algorithms

GPG supports multiple hash algorithms: MD5, SHA1, RIPEMD160 and some SHA-2 algorithms with varying hash lengths. Hash algorithm or function is a one-way hash function that takes an input of variable length (pre-image) and produces a fixed length output (hash) (Schneier, 1996). There are multiple uses for hashing algorithms with regard to PGP, such as signatures and fingerprints. When a user signs a message with their private key, the software that implements PGP calculates a hash of the message and signs it with the user's private key. Now the receiver, when receiving the message, can calculate the hash as well and then verify the signature with the sender's public key. The use of hash functions in fingerprint creation is explained in more detail at the end of this section when the SHA1 hash algorithm is discussed.

Attacks against the security of hash algorithms can be divided into three main categories: preimage-attack, second preimage-attack and collision-attack (Rogaway & Shrimpton 2004). Preimage-attack means that there is a given hash and the attacker attempts to find a message that has this same hash. Second preimage-attack means that there is a given a message and the attacler attempts to find another distinct message so that these messages share the same hash.

Collision-attack is a situation where the attacker attempts to find any two messages so that they share the same hash (Rogaway & Shrimpton, 2004).

With regard to this thesis the most relevant hash algorithm is SHA1 since it is by default used in GPG for fingerprint creation. SHA1 is designed by NIST and NSA in 1995 (Schneier, 1996). The input is padded and split into 512-bit blocks to produce a 160-bit hash (Manuel, 2011). The fingerprint of a PGP key is calculated by taking a SHA1 hash of the public key packet.

2.2.3 PGP Key Structure

PGP public key, when exported out of a program like GPG, consists of multiple packets. There are two general types of keys that can be found here: primary keys and subkeys. Primary keys are mainly used for signing the subkeys which perform the encryption and decryption. Both the public and private keys have primary keys and subkeys. There is only one primary public key and private key. Each type of key can have multiple sub-keys but they must be tied to the primary key by being signed by the primary key.

The PGP public key packets consist of a varied length header and a body. The header generally consists of a one-octet packet tag and the rest of the header is packet length. The one-octet packet tag starts with one as the left-most bit (bit 7) and the bits storing the packet tag are bits 5-2 with bits 1-0 storing the length-type. The meaning of the length-type bits are shown below:

00 – Packet's length is represented by one octet
01 – Packet's length is represented by two octets
10 – Packet's length is represented by four octets
11 – Indeterminate length.

Generally, the first packet is the Public-Key Packet and it consists of values related to the actual primary key. The packet tag is 6. These fields and their lengths are shown below for a version 4 packet:

- Version number (one octet)
- Creation timestamp (four octets)
- Public-key algorithm (one octet)
- Series of multiprecision integers(MPIs) that contain the key material. For RSA these fields are modulus n and exponent e. The fields are preceded by the bit-length of that field.

The public-key packet is followed by a User ID Packet. The body of this packet consists of only one field in UTF-8 text that should have the user's name and email address. These are stored following the RFC 2822 format where the

name is first followed by the email surrounded by less-than and greater-than signs (Shaw et al., 2007).

The Signature Packet (tag 2) generally follows the aforementioned packets. This is a signature that certifies that the public key belongs to the user id mentioned in the user id packet (Shaw et al., 2007). A version 4 signature packet body consists of the following fields:

- Version number (1 octet)
- Signature type (1 octet)
- Public-key algorithm (1 octet)
- Hash algorithm (1 octet)
- Length of all the hashed subpackets (2 octets)
- Hashed subpackets (0 or more subpackets)
- Length of all the unhashed subpackets (2 octets)
- Unhashed subpackets (0 or more subpackets)
- Left 16 bits of the signed hash value (2 octets)
- One or more algorithm specific MPIs comprising the signature. For RSA this is the value of m^d mod n.

The subpackets mentioned can consist of data containing signature creation time, key expiration time, preferred algorithms (compression, hash and symmetric), issuer of the signature etc. The format of these subpackets is similar to the format of the packets consisting of a header and a body. However in subpackets in the header the length comes first followed by the subpacket type.

Public key, user id and signature packets are usually followed by the packets related to the subkey. The Public-Subkey packet (tag 14) consists of the same fields as the public key packet mentioned first in this section but it denotes a subkey. Each subkey is followed by a signature packet that certifies that the key belongs to the master key.

There are also other packet types that can occur inside a public key but they are not included in the key with the default settings in GPG. These include packets such as user attribute packet and compressed data packet.

The private key also consists of packets. The packets that are the same with the public key are “user id” and “signature” packets. New packets are also introduced, called the “Secret-Key” Packet (Packet tag 5) and “Secret-Subkey” Packet (Packet tag 7). These are very similar in structure so only the “Secret-Key” Packet will be explored in more detail. The secret-key packet contains all the information from the public-key packet as well as:

- Secret-key encryption indicator (1 octet)

- Algorithm-specific fields. For RSA these would be: secret exponent *d*, prime values p and q and u, the inverse of p, mod q. All of these values are MPI.

2.2.4 Security of PGP

In 1994 Bruce Schneier called PGP the closest thing you are likely to get to military grade encryption (Schneier, 1996). The field of cryptography has changed massively since then but even Edward Snowden trusted the security of PGP in 2013 when communicating with reporters about papers he wanted to leak. The security side of PGP seems to be quite strong but there are some weaknesses as well which will be discussed in this chapter.

PGP depends on several algorithms which have been cryptographically tested (Slegers, 2001). These algorithms are used to provide a secure channel of communication for the user. One of the most common algorithms used is RSA which, with long enough keys, is unbreakable with the current technology. However, in practice the some of the keying elements are often quite short in length and default values are used in parameters which weaken the security. Also the increased power of computation and new techniques being discovered make shorter keys easier to crack. PGP supports multiple algorithms and is built in such a way that new ones can be easily implemented in case weaknesses are discovered in some of them.

There is quite a bit of research done on Short-ID Collison Attacks related to PGP. Short-ID is the last 8 digits of a key's fingerprint and there have been some cases where people have been able to get a key with same short-ID as the key they are trying to attack against (Heller, 2016). In this example the attacker was able to create a key-pair with a fingerprint that had the same last 8 hexadecimal characters as the fingerprint for Linus Torvalds. The attack is carried out by increasing the key's public exponent and then hashing the key. The last 8 hexadecimal characters of the hash are compared to the wanted hash and if there is no match the exponent is increased yet again until a match is found (Swanson, 2017). This attack was first demonstrated in 1996 by Raph Levien when he managed to create a PGP key that has the same Short-ID as the key of Phil Zimmerman, the creator of PGP (Levien, 1996).

A chosen-ciphertext attack against PGP and GnuPG has also been successfully carried out (Jallad, Katz, Lee & Schneier, 2002). Although they had to make some alterations such as limiting the amount of compression. They used the recipient's email application as an oracle query and were able to entirely decrypt a message that was not compressed before encryption.

Wilson and Ateniese tried to improve PGP by using the techniques used in Bitcoin and it's blockchain (Wilson & Ateniese, 2015). In their article, they discuss several weaknesses and limitations with the trust system used in PGP. One of the main weaknesses is that the honor system is subjective and that the user can only completely trust the entities that have given the user their public key personally. According to them it's really difficult to get a new key certified since there is no incentive for others to endorse keys.

2.2.5 Advantages and Disadvantages of PGP

Compared to not encrypting email then one of the major advantages of PGP is the fact that with the current technology if a person encrypts their email using this system and their private key is not compromised they can be almost certain that no one can read the email apart from their intended participant. Also, the fact that a user can personally choose which keys they trust and who do they trust enough to trust a key that they signed is an important advantage. PGP can also be used for signing an email so a user can be sure that the sender is who they say they are which is important in these days of anonymity in the internet.

There are also some disadvantages of using PGP. These include issues such as compatibility (this is an issue relative to unencrypted communication), complexity and the fact that there are no known backdoors in the sense that if a user loses their private key there is no practical way to be able to read the messages encrypted using their public key. Both the receiver and the sender must have compatible versions of the software to be able to read the messages send by the other user. This isn't a big problem in most circumstances but should be taken into account. Another problem related to "compatibility" is that if the user has their private key stored in a local machine then reading PGP protected emails elsewhere or on their mobile device will not be possible. The private key could of course be copied to multiple devices but with each copy the security of the system weakens.

PGP can seem quite complex to a new user. In 1999 Whitten and Tygar conducted a study on the usability of PGP 5.0 to new users with low previous experience in encryption. The results showed that PGP 5.0 is not suitable to make encryption easier to use for the public and users understanding of the underlying principles of public key cryptography is limited (Whitten & Tygar, 1999). A study conducted by Bringham Young University (Ruoti, Andersen, Zappala & Seamons, 2015) found out that especially browser-based PGP secure email tool is also hard for new users to grasp. In the study, only one pair was able to successfully complete the tasks in the time allocated. Also, the trust system is unique and might take some time getting used to.

The idea that there are no known backdoors in the latest version of PGP is important for an encryption system but if a user is to lose or permanently delete their private key then all the encrypted messages will stay encrypted and there is no option to recover a lost password. This is why it's important to make a backup of the private key to a secure location.

2.3 Key Management

"Compromise of the system details should not inconvenience the correspondents" (Kerckhoffs, 1883)

The quotation at the start was written originally in French by Dutch cryptographer Auguste Kerckhoff in 1883, but still applies today. The security of the messaging channel should not suffer if the algorithm is known to a third party, thus keys are the most important part of an encryption algorithm. For example, a weak pair of keys can provide a weakness in otherwise strong encryption. "Key management is the hardest part of cryptography" (Schneier, 1996). Key management includes e.g. generation, exchange, verification, storage and replacement of keys.

2.3.1 Key Generation

Key generation is vital in operating secure encryption systems. If the way the keys are generated is cryptographically weak then the whole system is compromised. Keyspace and its size are an important factor in key generation, since the larger the keyspace the harder it is to perform a brute-force attack against it given that each possible member of the keyspace is equally likely.

If we take a 4-byte (32-bit) key consisting of only lowercase letters (a-z) then the possible number of combinations is 26^4 which is about $4.6 * 10^5$. However, if all the 8-bit ASCII characters are used for a 4-byte key then the number of combinations is 256^4 which is about $4.3 * 10^9$. Just by increasing the size of the charset used for the key from 26 to 256 the size of the keyspace got 10,000 times larger.

Randomization is one way of creating cryptographically good keys since if the source is truly random and the distribution is properly equalized then all the possible members of the keyspace are equally likely to be chosen as the key. However if the users have to remember a raw key then a long randomly generated key consisting of the entire ASCII key space is not practical.

One solution to the human memory problem is using passphrases which are used to create a key using a technique called key-crunching (Schneier, 1996).

In this process (which is also used in PGP architecture) a one-way hash function is used to transform a string entered by the user to a pseudo-random key which is called key derivation. The passphrase has to be random enough so it cannot be found in existing corpuses since these are easy enough to attack.

2.3.2 Key Exchange

The process of key exchange in a symmetric encryption is a challenging task as the correspondents need to try and agree on a key without sending the key in plaintext over an insecure connection. One widely known key exchange protocol is the Diffie-Hellman key-exchange, which was first published in 1975 (Diffie & Hellman, 1976). This key-exchange algorithm works by two people (let's call them Alice and Bob) agreeing on two prime numbers **g** and **p** publicly. Now Alice picks a secret number **a** and calculates the value of **A** = g^a mod p and sends A to Bob. Bob does the same thing with a secret number that he has chosen called **b** and calculates the value of **B =** g^b mod p and sends B to Alice. Now Alice can use the B she received from Bob to calculate the secret key by using B^a mod p and likewise Bob can use the A from Alice to calculate the same value by using A^b mod p. ElGamal encryption is based on the Diffie-Hellman key-exchange algorithm and is one of the optional encryption algorithms that can be used in PGP.

Diffie-Hellman key-exchange is useful for symmetric encryption with just one key as it allows two parties to decide on one key without having to send it over the web in clear text. However, public key cryptography offers more versatile applications as well, since compared to symmetric cryptography the public key doesn't have to be hidden and can be published at any place where the other parties can see it. Quite common location to publish one's key is a key server where other users can find an entity's public key by their email or key id. The problem with key exchange in regard to public key cryptography is the notion of trust and this will be discussed in more detail in section 2.3.3 when verification of keys is discussed.

2.3.3 PGP Principles for Verifying Keys

A fundamental question in public key cryptography is, how to verify that one entity's key belongs to him/her and not to someone else who claims to be this person. There are several approaches to this from certificate authorities to web of trust.

Certificate authorities are trusted third parties that issue digital certificates certifying that the key really belongs to the entity it is claimed to belong to. If one trusts the certificate authority, then they should be able to trust the certificates that are issue by the certificate authority.

A popular way in PGP architecture is to verify keys by asking for the fingerprint or part of the fingerprint using some other communication channel and then comparing it to the one calculated from the public key that is publicly available. An example of this would be receiving the public key by email and getting the fingerprint as a text message.

Trust models are discussed in more detail in the first chapter, but as opposed to certificate authorities there is no higher up entity that issues certificates but instead if a person trusts a key pair then they can sign it. These signatures build up and the key pairs become more and more trustworthy.

2.3.4 Storage of Keys

Keys must be stored securely since even if the user has a 4096-bit key, it's useless if the attacker can locate the private key. Keys can be stored in encrypted format so that the user has to enter a passphrase to be able to decrypt the key and use it to decrypt a message. Problems arise when a user has multiple devices where they want to store the key in order to be able to use it.

2.3.5 Replacement of Keys

Keys should be replaced fairly frequently as no key should be used forever however strong it is (Schneier, 1996). The longer the key is used, the more time attackers have to try breaking it and if they manage to break it the loss is higher since a lot of data was encrypted using this key. This is why a majority of the implementations of PGP include a parameter for key validation period, which is the amount of time before the key expires and a new one has to be created.

Once a key (or key-pair) is replaced, the old keys should be destroyed as old messages can still be decoded with these keys (Schneier, 1996). The keys should be destroyed carefully as even if they appear to be deleted on a computer they might not have been overwritten so it might still be possible to carve them from the hard drive.

Replacement of keys has to be announced clearly to everyone since once a user has destroyed their old keys there is no way to be able to open those messages. The new keys will have to be verified but it might be ideal to sign the new keys with the old key to prove that they really belong to the user.

2.4 Different Software Implementations of PGP

2.4.1 GNU Privacy Guard (GnuPG)

GnuPG is an open-source full OpenPGP implementation that "is a commandline tool without any graphical stuff." (The GnuPG Project, 2016) It's generally run from the command prompt but can and has been implemented in other software such as email clients. There is a windows version of this software called Gpg4win. (The GnuPG Project, 2016). It supports all the major encryption algorithms discussed earlier in this chapter as well as the most common hashing algorithms such as MD5 and SHA1. It is also available in many different languages.

For a user to generate a key-pair in GnuPG one would enter the command: `gpg --gen-key.` The user is then given a list of algorithms to use for creating the key, the length of the key and when the key should expire as shown in Figure 5. After the user has chosen the desired options details are asked about the owner of the key such as name and email address so the software can create a user ID. The user is then asked to enter a passphrase (this is used to give additional protection to the private key).

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

FIGURE 5 GnuPG options while generating a key

User can then view the keys added to their keyring including their own private keys or export their public key so they can send it to the people they want to be in correspondence with. The most important command related to this thesis is the one used to generate and display a fingerprint: `gpg –fingerprint [name]`

2.4.2 PGP Desktop

PGP Desktop is a software suite released by Symantec. PGP Desktop is a commercial software and includes encryption of hard drives as well as email

encryption (Symantec, 2011). PGP Desktop consists of several different features such as PGP Messaging, PGP Keys and PGP Viewer (Symantec, 2011).

PGP Messaging is a feature of the PGP Desktop that allows a user to encrypt and decrypt email messages in multiple email clients. It also works with some instant messaging clients. PGP Keys is a feature that lets a user control their own keys and the public keys of the people they are in contact with. PGP Viewer is used to decrypt, verify and display emails and other messages that did not come to the user in an email client supported by PGP Messaging (Symantec, 2011).

PGP Desktop seems to be easy to use as the program will guide a user through the creating of a key pair, publishing it in a key directory and enabling PGP Messaging in their email client. For Key type the program supports either Diffie-Hellman/DSS or RSA algorithms and a key-size of 1024 to 4096 bits (Symantec, 2011).

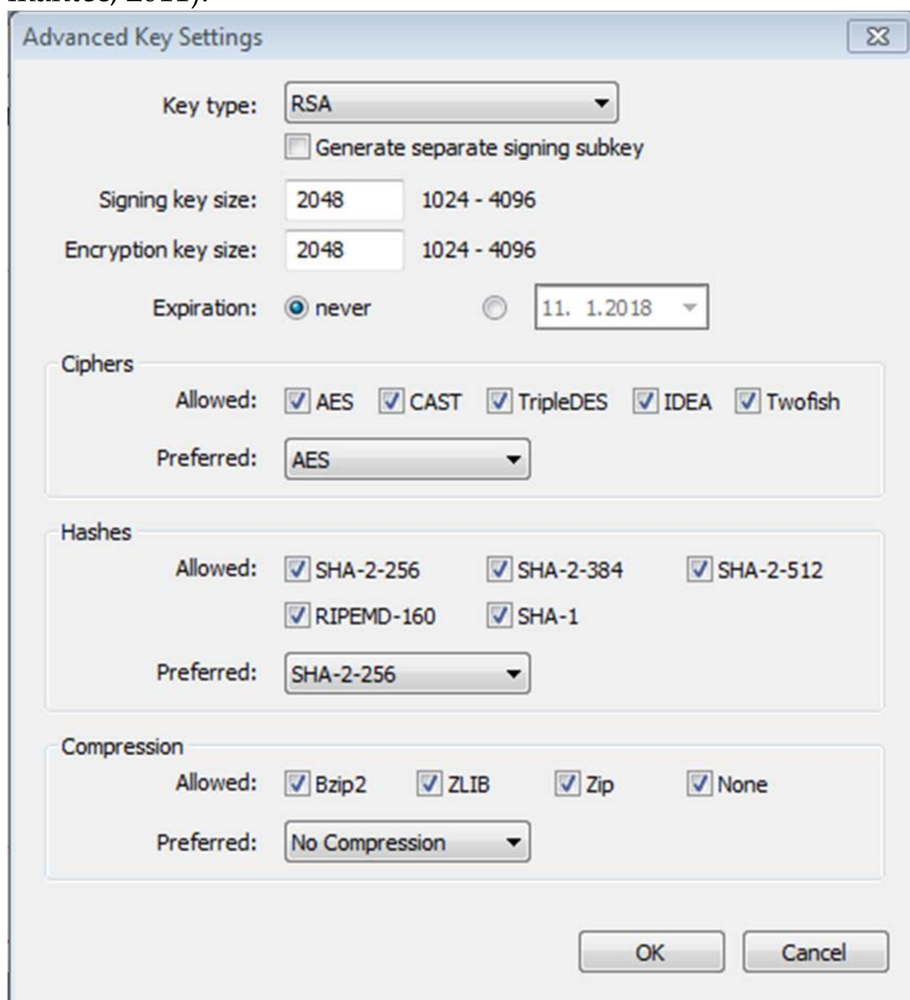


FIGURE 6 Key generation window of PGP Desktop

3 RESEARCH METHODOLOGY

This chapter will contain details on how the actual research will be carried out. The tests will all be carried out against a key that is produced with the default settings on GnuPG. These settings at the time of writing include using RSA for both encryption and signing and having the key length as 2048 bits. In this chapter we will go over the practical aspects of creating a fingerprint from a PGP key, different primality testing algorithms and multi-prime RSA. We will then explore the different approaches in more detail and explore how data will be collected and analysed. In this chapter any mention of the word character refers to a hexadecimal character.

3.1 Research Approach

3.1.1 Fingerprint creation in PGP

The fingerprint of a PGP key by the OpenPGP standard is computed by calculating the SHA1 hash of certain aspects of the public key(Callas et al., 2007). According to the OpenPGP Message Format the data that is used to calculate the fingerprint consists of:

0x99 (1 octet)
2 octets for packet length of the public key
1 octet for version number
4 octets for timestamp of key creation
1 octet for algorithm

Followed by algorithm-specific fields, which for RSA public keys are: multiprecision integer (MPI) of RSA public modulus n and MPI of RSA public encryption exponent e (Shaw et al., 2007). These two fields have their length preceding them where both lengths are 2 octets in length. So with a 2048-bit n and a 17-bit e (usually 65,537) this adds up to approximately 272 bytes or 0x110 in hexadecimal notation.

The calculation of the fingerprint is fairly simple and the algorithm is demonstrated below. In a public key packet that is exported from gpg using *gpg --output public_key.asc --export key_holder_id* the necessary information for calculating the fingerprint is at the start of the file.

```
calculate_fingerprint (public_key_packet)
  len = int(public_key_packet[1:3])
  pub_key = public_key_packet[0:len+3]
  return SHA1(pub_key) # length = 20 bytes
```

3.1.2 Practical implementation

Since the calculation of the fingerprint ends with calculating the SHA1 hash, there is no known practical way to end the calculation prematurely if we can see that the fingerprint is not going to be what we would want it to be. Because of this we need an alternative to change the input to be hashed in the most efficient way possible.

Since the generation of the fingerprint is dependent on the timestamp, we could change the three least-significant octets of it. That way the key doesn't change as often but the fingerprint changes every time. Only the three least-significant octets would be changed because this translated into days would allow for a derivation of about 195 days which would be a perfectly acceptable time for a key to exist before communication. Initially we planned on only editing the two least significant octets but this would only allow a time variation of 18 hours. The recipient is unlikely to know the exact creation date of the key so considering the three least significant octets for 195 day variance seemed like a choice that wouldn't raise any suspicion from the target (unlike a key that was created in the 1970s or in the future).

Changing these three octets allows us to get 16,777,215 fingerprints for the same values of n and e which is equal to 2^{24}. The key creation process using gpg requires the user to interact with it, as it asks for details of the key and wants the user to perform other actions so it can get enough randomness to produce suitable prime numbers. Therefore if there is no match after these tries we generate a new n by multiplying two random prime numbers p and q. We could have chosen the approach of incrementing the exponent like done in the past (Swanson 2017), but we chose not to since the value of the exponent is by default 65537 and any variation from this would raise questions from an experienced user of PGP. Furthermore, if we wouldn't want to increase the length of the key packet (again not to arise suspicion) then there would be a limit to the number of exponents possible and we would eventually still have to calculate new prime numbers.

One thing to bear in mind is that the creation date of the key is shown to the user when it has been imported, so we do not want to make a key that has this creation time in the future. For this reason we would take the current time and calculate 2^{24} seconds back from this. This way the key would be created in the past and it would be at most about half a year old.

The idea is to have a script that takes an existing public key packet, leaves the first, fourth and fifth octet as they are, calculates a new value for n and e and calculates their length to be included in octets two and three. Then start changing octets at index 5 to 7 and see if there is a match in the fingerprint. If all possible 16,777,215 values have been tried for indexes 6 and 7 then we calculate a new value for n.

Algorithm is as follows:

```
calculate_fake_public_key(target_public_key)
       tfp = fingerprint(target_public_key)
       start_time = current_time – 0x1000000
       while(true):
                new_key = RSA.genkey(2048)
                new_public_key = new_key.n() + new_key.e()
                for i in 0x000000 to 0Xffffff:
                         new_public_key[6:8] = i + start_time
                         if fp(new_public_key) == tfp:
                                  return new_key
```

Since the other packets mentioned in section 2.2.3 are not required for the fingerprint calculation these would not be created until a desired similarity is achieved between the target and the fingerprint of the attacker.

3.1.3 Social Engineering

Concerning SHA-1 there is only one known case of collision (Stevens et al., 2017) and there are no known second-preimage-attacks, meaning that it is hard to find a partner for a random point that has the same hash (Rogaway & Shrimpton, 2004). That's why we need to factor in social engineering in that when people verify a key they do not necessarily check every character of the hexadecimal representation. They might only check the first and last x characters. There are also examples of users not verifying the hash at all when connecting to an SSH server which shows a similar length hash as a PGP would (Gutmann, 2011).

Now the ideal approach to take advantage of this would be to produce a script that allows the user to enter the number of characters they want to match

from the beginning, end or both and then it's up to the computing power available to the user to compute enough fingerprints to find a match. A German hacker group called The Hacker's Choice created a program which does this to an extent (Plasmoid, 2003). In the program the user enters a weight for each digit of the hash and the program attempts to generate a similar hash based on those. It is based on the idea that, as mentioned previously, users tend to only compare only a sequence at the start and at the end of the fingerprint to verify it.

3.1.4 Primality testing algorithms

There are multiple algorithms to test primality of a number. In this project we will only focus on the probabilistic algorithms since deterministic primality testing algorithms are more computationally intensive (Menezes, Oorschot & Vanstone, 2001). In this section three algorithms will be discussed and their performance analyzed in more detail. These tests are Fermat, Solovay–Strassen and Miller-Rabin.

Fermat's primality test is a probabilistic test that applied Fermat's little theorem to find out whether a number is probably a prime or not. The algorithm of this test is as follows (Menezes et al., 2001):

```
Fermat(n,t):
    INPUT: n an odd integer >=3 and t >= 1
    for i from 1 to t:
        a = random integer between 2 and n-2
        r = a^(n-1) mod n
        if r != 1: return "composite"
    return "prime"
```

In Fermat's primality test the result "prime" may not be true if the value of t is small since not all the possible divisors will be tested for but with large enough t these false positives are rare.

The Solovay-Strassen (Solovay & Strassen, 1977) primality test was made popular by being used in the early versions of RSA (Menezes et al., 2001). It follows the basic format of the Fermat's primality test but makes some alterations for efficiency and accuracy. The algorithm is as follows (Menezes et al., 2001):

```
Solovay-Strassen(n,t):
    INPUT: n an odd integer >=3 and t >= 1
    for i from 1 to t:
        a = random integer between 2 and n-2
        r = a^((n-1)/2) mod n
        if r != 1 and r != n-1: return "composite"
        s = (a/n)
        if r != s (mod n): return "composite"
    return "prime"
```

As with Fermat's primality test the result prime may not always be a prime but the likelihood is higher than that of Fermat's primality test. This test utilizes Euler's criterion during the process. This criterion states that given an odd prime n then $a^{(n-1)/2} = (a/n) \pmod{n}$ for all a which satisfy gcd(a,n)=1 (Menezes et al., 2001).
The Miller-Rabin test (Rabin, 1980) is the most used in practice of the three tests mentioned here. (Menezes et al., 2001). It differs from the previous tests by the equation it uses as the foundation for the test. The formula for this test is as follows (Menezes et al., 2001):

```
Miller-Rabin(n,t):
        INPUT: n an odd integer >=3 and t >= 1
        Compute n-1 = 2^s r so that r is odd.
        for i from 1 to t:
            a = random integer between 2 and n-2
            y = a^r mod n
            if y != 1 and y =! n-1:
                j = 1
                while j <= s-1 and y != n-1:
                    y = y^2 mod n
                    if y = 1: return "composite"
                    j = j + 1
                if y != n-1: return "composite"
        return "prime"
```

When discussing the performance of these primality tests the most important factor is how many exponentiations are there especially with the variable n. This is important since n is a long integer and calculations of this type using that number will require a significant amount of computing power.

Fermat's primality test includes one exponent calculation that uses the variable n per iteration ($a^{n-1} \bmod n$). Solovay-Strassen requires at least one exponent calculation per iteration ($a^r \bmod n$), although n is not the exponent here r is based on the value of n and is increased by one on every iteration. Miller-Rabin on the other hand computes an operation requiring n at the start before the for-loop. During the for-loop n is used a few times for modulation but unlike in the first two tests it is not used as an exponent. Miller-Rabin does use the value r which is based on the value n as an exponent but according to the algorithm it is clearly smaller than n.

The Miller-Rabin is the most efficient of these algorithms at computing whether a number is a prime or not. The Miller-Rabin test also has an error rate that is smaller or equal to that of the other two tests. The upper bound for error probability for Solovay-Strassen is $(1/2)^t$ and for Miller-Rabin it is $(1/4)^t$ (Menezes et al., 2001). Therefore, the lower bound of the success rate as a percentage if t is 3 is 87.5% for Solovay-Strassen and 98.4% for Miller-Rabin. For

these reasons in this project Miller-Rabin will be used as the primality test of choice.

3.1.5 Multi-prime RSA

Using more than two primes to calculate the modulus n has a significant advantage over the traditional method in that the prime numbers generated can be much smaller. This allows for faster computation and it makes splitting the task of creating the primes for a key between cores easier. These primes do not need to be unique and there is not really any way of working these out from the value of n without solving the factorization problem.

There is no theoretical limit to the number of primes that n can be composed of, but the smaller they are then the factorization becomes easier. For this experiment we will consider the number of primes from 2 to 16. That means that if we take a 2048-bit value for n the maximum size is 1024-bit (the normal case where there are two primes p and q) and the minimum size is 128 bits (128*16=2048).

Using this method we will have to take into account, in addition to the space required and speed of prime generation, the time it takes to multiply two numbers together. This will be important since this operation will be performed 15 times in the case where n is a product of 16 primes.

Another problem with the multi-prime RSA is that when two n-bit numbers are multiplied together the product is not necessary 2n bits long, but it could be smaller by one bit. This is not generally a problem when n is a product of only p and q since in this case the product could be at worst 2047-bit number. The problem arises when there are more primes and the possible product gets further away from the desired length. To solve this we would have to calculate the bit-length of the product of k-1 primes and then choose a large enough prime to get the result closer to the 2048-bit value.

3.2 Data collection methods and techniques

There are multiple approaches for targeted fingerprint generation that will be compared in this project. These approaches will be timed inside the script by using Python's built-in functions for measuring time. The following approaches will be compared in creating a fingerprint from 2048-bit key:

1. The primality of p is not tested at all until a suitable fingerprint is found.

2. The primality of p is tested each time a new modulus is required (before any fingerprints are calculated).
3. A list of pre-generated primes is used, and p and q are chosen from them, each time a new modulus is required, and the fingerprints are calculated including moduli formed by each possible pair (p_i, q_k) from this set.
4. Use more than two prime numbers to generate n.

In each scenario we will first calculate 2^{24} fingerprints by changing the timestamp and only after this we would generate a new value for the modulus n by one of the means mentioned above. Initial tests have shown that the time it takes to create 2^{24} fingerprints from two new prime numbers takes approximately 56.4 seconds. Generation of a 1024-bit prime number takes on average 2 seconds. These values were achieved by running the calculations 100 times and taking the average. The tests where run on a Xubuntu 16.04 running on a VirtualBox with 8GB of base memory allocated for the machine.

Let's perform a few calculations to see what the potential outcome of this experiment would be. Let t(P_k) be the amount of time taken to create a k-bit prime number. Let t(P_{1024}) = 2 seconds, the time needed to create a 1024-bit prime number. Let t(C) = 56.4 seconds, the time taken to create 2^{24} fingerprints based on pre-calculated keys. N is the number of random numbers masquerading as primes created during the process of finding a fingerprint to meet the requirements. After initial testing the bit-size of the key packet that is being used to calculate the fingerprint from does not seem to have a significant effect on t(C).

Now in approach 1 we would save the t(P_{1024}) and each cycle would only take t(C) amount of time, however we would need to verify that the values chosen were actually prime and if they weren't then those calculations would be wasted. The calculation would only take $t(C) * N$ amount of time. On average when we deal with numbers that are 1024-bits in size (approximately 309 digits in base 10) we find that according to the prime number theorem (Hoffman 1998) a number is prime with probability 1/(1024*ln(2)), or equivalently, about 1 in 710 is a prime. If we ignore even numbers in the search, the probability of a false positive is then about 1-(1/355), and the expected amount of trials needed to get a true hit about 355/2 = 174,5 times longer than with pre-checked primes.

Number of generated fingerprints: $2^{24} * N$

Time taken to generate these fingerprints: $t(C) * N$

N: Number of generated random numbers

In approach 2 we test the primality first and then create the fingerprints and thus each cycle takes $t(P_{1024}) + t(C)$ amount of time. So the time taken would be $\left(t(P_{1024}) + t(C)\right) * N$ where N is the number of prime numbers created. Here the number of fingerprints created would be $N * 2^{24}$. This could be

improved by storing the prime numbers and pairing the newly generated number with all the others in which case the performance would increase.

Number of generated fingerprints: $2^{24} * N$

Time taken to generate these fingerprints: $(t(P_{1024)} + t(C)) * N$

N: The number of prime numbers generated

In approach 3 the cycle would only take t(C) amount of time as the prime numbers have been pre-generated but we would still need to take that time into account so the realistic time would be $t(P_{1024}) * N + t(C) * \frac{N(N-1)}{2}$. Here the number of fingerprints created would be $\big(N * (N-1)\big) * 2^{23}$. This approach should work better when N is larger compared to approach 2.

Number of generated fingerprints: $\big(N * (N-1)\big) * 2^{23}$

Time taken to generate these fingerprints: $t(P_{1024}) * N + t(C) * \frac{N(N-1)}{2}$

N: The number of prime numbers generated

In approach 4 the cycle would take $t(C)$ amount of time just like in approach 3. However, calculating the pre-generated primes should take less time since the primes will be smaller, although we would need to compute more of them. The calculation would also depend on the number of prime numbers (k) used to calculate the value for n. The time taken would be $t\left(P_{\frac{1024}{k}}\right) * N + T(C) * \frac{N!}{k!(N-k)!}$ and this would generate $\frac{N!}{k!(N-k)!} * 2^{24}$ fingerprints. Now this seems to be the fastest of the approaches but working out a suitable size for k will be an interesting problem to discuss.

Number of generated fingerprints: $\frac{N!}{k!(N-k)!} * 2^{24}$

Time taken to generate these fingerprints: $t\left(P_{\frac{1024}{k}}\right) * N + T(C) * \frac{N!}{k!(N-k)!}$

N: The number of prime numbers generated of size 1024/k

K: The number of prime numbers needed to make modulus n as a product.

3.3 Data Analysis

The best way to compare the effectiveness of the four approaches is to see the time and space requirements needed to cover a so-called fingerprint space. For this experiment we will see five different sized fingerprint spaces represented as powers of 2: 32, 48, 64, 80 and 96. These were chosen as the probability for finding a match for specific 32 bits or 8 characters in hexadecimal is $2^{-32}$and thus if we calculate 2^{32}fingerprints it is probable that there is a match. Eight characters is an important figure since the key-id is made up of the last eight characters of the fingerprint. The other values represent 12 (2^{48}), 16 (2^{64}), 20 (2^{80}) and 24 (2^{96}) characters of the fingerprint. These were chosen in increments of four since the fingerprint is usually displayed in sets of four characters.

Multi-prime RSA (Approach 4) seems to be the most promising according to the calculations. Since there are some variables that can be altered this will be studied in more detail. The number of primes that make up the modulus n is the main variable and the values from 3 to 16 will be considered.

4 FINDINGS

Findings that required timing for computation were achieved by using a virtual machine running a Xubuntu 16.04 as an operating system with 8GB of working memory allocated. This setup was running on VirtualBox 5.2.4. This was done on a virtual machine to achieve comparability between the tests as the test results seemed to change between physical machine boot ups. This chapter contains the findings of the experiment in terms of number of prime numbers generated, space required, and time required in order to fill certain preset fingerprint-spaces.

4.1 Number of prime numbers to be generated

Table 1 shows the number of prime numbers that need to be generated to fill up the desired fingerprint space. In approach 1 these are of-course just regular random numbers and the fact that on average only 1/700 (based on prime number theory) are only primes. The results for approaches 3 and 4 can be found on Table 2, which contains the results when modulus n is the product of a varying number of prime numbers. The row heading k is the number of prime numbers of the same size.

Table 1: The number of random or prime numbers required to fill up the fingerprint space.

	Size of fingerprint space (2^)				
Approach	32	48	64	80	96
1	44 672	2.93E+09	1.92E+14	1.26E+19	8.24E+23
2	256	16 777 216	1.10E+12	7.21E+16	4.72E+21
3	*See line for k=2 on Table 2*				
4	*See lines for k=3-16 on Table 2*				

Table 2: The number of prime numbers required to fill up the fingerprint space with multi-prime RSA.

	Size of fingerprint space (2^)				
k	32	48	64	80	96
2	24	5 794	1 482 911	3.8E+08	9.718E+10
3	13	467	18 756	756 155	30 486 226
4	11	144	2 268	36 265	580 222
5	11	75	669	6 131	56 323
6	11	51	307	1 934	12 266
7	11	40	181	869	4 223
8	12	34	124	486	1 931
9	13	31	95	314	1 066
10	13	29	78	225	671
11	14	28	67	173	464
12	15	27	59	140	345
13	16	27	54	119	270
14	17	27	51	104	221
15	18	27	49	93	186
16	19	28	47	85	162

4.2 The space requirement

Since approaches 1 and 2 do not need the values for n that they have calculated after all the 2^{24}fingerprints have been calculated, they do not really need any significant storage. Therefore, Table 3 shows the storage requirement for approaches 3(when k=2) and 4 (when k>2). Table 3 contains the size required for storing the prime numbers in kilobytes comparing the value for k (number of prime numbers required for calculating modulus n) and the desired fingerprint space represented as a power of 2.

Table 3: Storage space required to achieve a certain fingerprint space in kilobytes.

	Size of fingerprint space (2^)				
k	32	48	64	80	96
2	3.00	724.25	185 363	47 453 133	1.215E+10
3	1.08	38.88	1 561.47	62 951.38	2 538 038
4	0.69	9.00	141.75	2 266.56	36 263.88
5	0.55	3.74	33.40	306.10	2 812.02
6	0.46	2.12	12.78	80.50	510.58
7	0.39	1.43	6.45	30.98	150.53
8	0.38	1.06	3.88	15.19	60.34
9	0.36	0.86	2.63	8.70	29.54
10	0.32	0.72	1.94	5.60	16.71
11	0.32	0.64	1.52	3.93	10.54
12	0.31	0.56	1.22	2.91	7.16
13	0.31	0.52	1.03	2.28	5.17
14	0.30	0.48	0.91	1.85	3.94
15	0.30	0.45	0.81	1.54	3.09
16	0.30	0.44	0.73	1.33	2.53

4.3 The time requirement

The time required to generate fingerprints depends on many individual parts for which the results will be shown in this sub-section. For the time requirement we do not take into account the time spend on parsing the target key and preparing the key for the attack. The times that change are the time required to generate the prime numbers and the time taken to calculate the required fingerprints and check if they match the criteria set by the attacker.

4.3.1 Time requirement for generating enough prime numbers

The time required to calculate enough prime numbers is an important distinction when it comes to calculating the overall time taken. Table 4 shows the time taken to create a prime number of a certain bit-size on the test machine. The results were achieved by running the generation script that uses Miller-Rabin as a primality test for 100 times each and taking the mean value of these results. These values were chosen as they were the closest integer to 2048 being divided by numbers 2 to 16.

Table 4: Time taken to create a prime number of certain bit-size

bit-size	time (s)
128	0.0036
137	0.0037
146	0.0040
158	0.0051
171	0.0072
186	0.0093
205	0.0109
228	0.0151
256	0.0183
293	0.0249
341	0.0584
410	0.0869
512	0.1993
683	0.4328
1024	1.9756

Table 5 shows the overall time taken to generate enough prime numbers to calculate enough fingerprints to fill up the required fingerprint space. This table was created by using the mean creation times from Table 4 and multiplying these by the number of primes required which are shown in Table 2. Approach 1 doesn't really need to generate prime numbers and only needs to verify the once it finds a match for so the time spent on this is negligible. For approach 2 the time requirement can be easily calculated by multiplying the number of primes required from table 1 with the time taken for creating a 1024-bit prime from Table 4. The time required to generate primes for approaches 3 (k=2) and 4 (k>2) are shown in Table 5.

Table 5: Time required generating enough prime numbers to fill the required fingerprint space in seconds.

	Size of fingerprint space (2^)				
k	32	48	64	80	96
2	47.41	11 446.62	2 929 638	7.50E+08	1.92E+11
3	5.63	202.13	8 118.28	327 291.40	13 195 549
4	2.19	28.71	452.12	7 229.26	115 664.50
5	0.96	6.52	58.13	532.70	4 893.70
6	0.64	2.98	17.92	112.87	715.84
7	0.27	0.99	4.50	21.61	105.02
8	0.22	0.62	2.27	8.90	35.38
9	0.20	0.47	1.43	4.73	16.07
10	0.14	0.32	0.85	2.45	7.29
11	0.13	0.26	0.62	1.61	4.33
12	0.11	0.19	0.42	1.01	2.48
13	0.08	0.14	0.28	0.61	1.38
14	0.07	0.11	0.20	0.41	0.87
15	0.07	0.10	0.18	0.35	0.69
16	0.07	0.10	0.17	0.31	0.59

4.3.2 Other time requirements

As mentioned in chapter 3 the time taken to calculate 2^{24}fingerprints and check whether they match the specified requirement takes approximately 56.4 seconds. This was found by calculating 2^{24}fingerprints 100 times and timing the runs and calculating the average.

These are the main time costs. The other cost factors, such as parsing the target key and creating the attack key do not take as long and they only need to be done once at the start or end of the run. Even calculating the other values apart from e and n are not required on every run if we store the primes we used to calculate them after the fingerprint has been found.

Table 6 shows the time requirement in days of each approach to achieve a certain fingerprint space. For each column the time taken to calculate enough hashes to fill the fingerprint space without the prime generation is calculated. This value is used for calculating the overall time for each approach . The time was originally calculated in seconds and then divided by 86400 to get it into days. For approach 1 this value is multiplied by 174.5 because of the prime number theorem. For Approaches 2 to 4 the value is calculated by adding the

time taken to create a prime multiplied by N from Table 1 to the time taken to calculate the hashes.

Table 6: Time requirement (days) of each approach to achieve desired fingerprint space coverage.

	Size of fingerprint space (2^)				
Approach	32	48	64	80	96
1	29.16	1 911 088	1.25E+11	8.21E+15	5.38E+20
2	0.17	11 335.42	7.43E+08	4.87E+13	3.19E+18
3	0.17	10 951.93	7.18E+08	4.70E+13	3.08E+18
4	0.17	10 951.80	7.18E+08	4.70E+13	3.08E+18

4.3.3 Actual time taken

Since Approach 1 takes almost a month to create a fingerprint space of 2^{24}in this section we will use the test machine and calculate a fingerprint with 8 characters the same as the target using approaches 2, 3 and 4. The time taken will be measured. Table 7 shows these results in minutes and seconds.

Table 7: Time taken on one attempt to get a match of at least 8 characters.

Approach	time(mm:ss)
2	221:22
3	360:06
4	189:58

5 DISCUSSION AND CONCLUSION

The purpose of this thesis was to find the most efficient way for creating fingerprints in PGP architecture in order to try and find a partial collision between fingerprints. The research question also asks what the effect of this forged fingerprint approach on the security of PGP. This chapter analyses the findings from the previous chapter and tries to find answers to the questions proposed in the introduction chapter of this thesis.

5.1 Discussion of the main findings

5.1.1 Approach 1

It is quite clearly shown in Table 1 that approach 1, which means taking a random number and checking its primality only after a suitable fingerprint has been found proves to be impractical. The prime number theorem suggests that numbers that are 1024-bit in length only are a prime number approximately in 1 out of 700 cases, so on average this would have to be 174.5 times more efficient than the other approaches for this to be the most efficient choice.

In terms of memory usage this approach is very efficient since the previous numbers are not stored, however the problems arrive with time efficiency as shown in Table 6. The calculated time taken to fill a fingerprint space of 2^{32}fingerprints would take approximately just over 29 days. This only increases at a much faster rate than the other approaches when the desired fingerprint space gets larger.

5.1.2 Approach 2

Approach 2 seems like the naive choice to solve this problem by generating a new prime number once all the fingerprints have been calculated for the previous key pair by changing the time stamp. In the practical test it did not prove to be that inefficient mainly due to the fact that prime number generation only took about 2 seconds a round so especially with a smaller target (8 characters of fingerprint to match) it didn't play a major role.

The number of prime numbers starts off steadily with only 256 needed to create 2^{32}fingerprints but it increases exponentially relative to the desired fingerprint space This one like approach 1 does not need really any storage space for previously calculated primes. The time requirement is linear to the number of primes used as each prime of 1024-bits takes approximately 2 seconds to create. Due to this the time requirement on Table 6 can be seen to be a lot lower than for approach 1.

5.1.3 Approach 3

Approach 3 is a modification of approach 2 in that the primes that were generated are not wasted but are stored for future use. The time savings increase when the program has to be run for longer as the list of primes would grow and there would be more rounds when a new prime does not have to be created. Due to this the number of prime numbers needed to create a fingerprint space of 2^{32}fingerprints is only a tenth of what was needed for approach 2.

The time saving comes with a cost of requiring more storage space but for example for the before mentioned example the storage space required is only 3 kilobytes. Of course, for larger fingerprint spaces this would increase quite rapidly with 2^{80}fingerprint-space target requiring about 45 gigabytes of storage space.

This approach took the longest in the practical run of approaches 2-4 to create an 8-character match between keys. This was probably due to the random number generation being unlucky at the time since the hash outcome is not predictable. The time taken was still relatively close to the prediction of 0.17 days compared to the actual time taken of 0.25 days.

5.1.4 Approach 4

Approach 4 is the most interesting out of the options discussed in this thesis since there are the most variables in this approach. Approach 4 is based on the idea of Multi-Prime RSA in which there are more than two primes whose

product is the modulus n. The ideal number of primes that should be used is not a trivial question because, although there are increases in performance, both in speed and space, there is a problem with the complexity of dealing with 16 different factors and calculating the signature and decryption key for them.

If we put these "weaknesses" aside for a while and discuss them in the next subchapter, we can focus on the most ideal value for k. If we are trying to achieve a 2^{32} fingerprint space then the smallest number of primes needed would occur when k is between 4 and 7. If k gets any larger then more primes would be needed to make up enough combinations to satisfy the requirement. For the bigger fingerprint spaces the ideal number is 16 prime numbers.

In regard to space to store the prime numbers the ideal candidate for smallest space is of course 16. Even in the 2^{32} fingerprint space column the size is the smallest since the prime numbers are only 128-bits in size. Any k value more than 12 will require less than 10 kilobytes to store all the primes required to generate even 2^{96} fingerprints. Timewise the same holds true as above. Any k value more than 12 will be able to produce all the required prime numbers in less than 3 seconds for all the tested fingerprint spaces.

In table 6 the values seemed to be really close to approach 3 since the time spent on creating prime numbers is quite small compared to the time calculating the hashes. This method appears to be the fastest but only by minutes. In approach 4 numbers need to be generated less often than in approach 3 and they also take less time, but the time saved is quite marginal. In table 7 (the practical test) approach 4 was the fastest like it should be in real life as well, but this was only based on one run.

5.2 Evaluation

There were some major difficulties with working out how to produce keys that would satisfy GPG's import function as it required the keys to be signed. We managed to figure out a solution for the regular two-prime RSA, but when it came to multi-prime RSA, quite a lot of time was spent on trying to figure it out and on the end this was given up as the tests could be carried out without it (this problem needed a solution only after a suitable value for n and timestamp had been found).

Another big problem was getting a value for modulus n to be 2048 bits. This isn't a huge problem when the two numbers that make up the modulus are 1024 bits since the product is usually only one bit off, but when there are 16 primes to make up the product things get more interesting as each new number could shift the product by one bit.

5.3 Conclusions

To conclude, it seems that the most efficient way to produce fingerprints within our assumptions is using multi-prime RSA (approach 4). However since the time needed to create a prime was at its worst only 2 seconds and since one prime pair can be used to calculate almost 17 million fingerprints the actual approach for an 8-character match in the fingerprint seems not to matter too much. If we had stuck to the original plan of only modifying the two least significant bits of the timestamp then the speed of generating prime numbers would have been a lot faster.

The effect of this on the security of PGP is hard to estimate. Short key-id clashes have been found in the past and the use of them to identify a key has been reduced. The work in this thesis could be used to create fingerprints which could fool someone to accept a forged key by a man-in-the-middle if they do not read the fingerprint carefully. Although with the testing machine used in the tests this would take hours to even produce a match with the first and last four characters to match. The attacker would have to know the target quite well and know how they usually validate a key in order to be able to utilize this method of attack successfully.

5.4 Future work

There are various areas of interest that rise from the research performed for this thesis. One of the major ones is the idea of multi-prime RSA. One area was shortly discussed in the evaluation section and is the idea of how to optimally create keys that are of correct length when k is large. This should be studied since the product of two n-bit primes can be a bit shorter in bit-length then 2n bits and when there are more than two (a.k.a. multi-prime) primes multiplied the product length can vary quite a lot.

Another section of interest is calculating the decryption key for multi-prime RSA. Quite a lot of the results only deal with three primes so calculating the resulting decryption key is trivial, but what if there are 16 primes? Also the method of using more than three primes for RSA seems to be unheard of and its effect on security should be examined.

REFERENCES

Adams, C. (1997). Constructing symmetric ciphers using the CAST design procedure. *Designs, Codes and Cryptography* [online] 12 (3), 283-316. doi: 1008229029587.

Barker, W. C. and Barker, E. (2011). *Recommendation for the triple data encryption algorithm (TDEA) block cipher* [online] (Rev. July 2011, version 1.2. ed.). Gaithersburg, MD: U.S. Dept. of Commerce, National Institute of Standards and Technology. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-67r1.pdf .

Chadwick, D. W., Young, A. J. and Cicovic, N. K. (1997). Merging and extending the PGP and PEM trust models-the ICE-TEL trust model. *IEEE Network* [online] 11 (3), 16-24. doi: 10.1109/65.587045.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory* [online] 22 (6), 644-654. doi: 10.1109/TIT.1976.1055638.

Digital signature standard (DSS) (2013). National Institute of Standards and Technology.

ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transaction on Information Theory* [online] 31 (4), 469-472.

Rijendal encryption algorithm. http://www.efgh.com/software/rijndael.htm.

PGP: Explanation of the web of trust of PGP. https://www.rubin.ch/pgp/weboftrust.en.html.

Gutmann, P. M. (2011). Do users verify SSH keys? *Login* [online] 36 (4). http://search.credoreference.com/content/entry/marquisam/gutmann_peter_m/0.

Linux creator highlight flaw with short IDs. http://searchsecurity.techtarget.com/news/450302716/PGP-collision-attack-on-Linux-creator-highlights-flaws-with-short-IDs.

Hoffman, P., 1956. (1998). *The man who loved only numbers : The story of paul erdos and the search for mathematical truth* [online]. United States:. http://catalog.hathitrust.org/Record/003974350 .

RSA theory. http://www.di-mgt.com.au/rsa_theory.html.

Jallad, K., Katz, J., Lee, J. J. and Schneier, B. (2002). Implementation of chosen-ciphertext attacks against PGP and GnuPG. *International Conference on Information Security* [online] , 90-101.

Kerckhoffs, A. (. (1883). La cryptographie militaire. *Journal des Sciences Militaires, 9*, 5-38.

Lai, X. and Massey, J. (1991). A proposal for a new block encryption standard. *Advances in cryptology – EUROCRYPT '90*. Berlin, Heidelberg: Springer Berlin Heidelberg, 389-404. doi: 10.1007/3-540-46877-3_35.

Lehtiranta, L., Junnonen, J., Kärnä, S. and Pekuri, L. (2015). The constructive research approach: Problem solving for complex projects. *Designs, Methods and Practices for Research of Project Management* [online] , 95-106.

Levien, R. (1996). "Dead beef" attack against PGP's key management [online]. https://groups.google.com/forum/message/raw?msg=sci.crypt/JSSM6NbfweQ/A572NzAPmxMJ.

Lucas, M. W. (2006). *PGP & GPG : Email for the practical paranoid* [online] (1st ed. ed.). San Francisco, Calif: No Starch Press. http://replace-me/ebraryid=10120726 .

Manuel, S. (2011). Classification and generation of disturbance vectors for collision attacks against SHA-1. *Designs, Codes and Cryptography* [online] 59 (1), 247-263. doi: 10.1007/s10623-010-9458-9.

Menezes, A. J., Oorschot, P. C. v. and Vanstone, S. A. (2001). *Handbook of applied cryptography* (5. rev. print. with updates ed.). Boca Raton, Fla. [u.a.]: CRC Press.

National Institute of Standards and Technology. (1999). *Data encryption standard (DES)*.

National Institute of Standards and Technology. (2001). *Advanced encryption standard (AES)*.

An introduction to cryptography. ftp://ftp.pgpi.org/pub/pgp/6.5/docs/english/IntroToCrypto.pdf.

OxfordDictionaries.com. https://en.oxforddictionaries.com/definition/trust.

Plasmoid. (2003). *Fuzzy fingerprints attacking vulnerabilities in the human brain*.

R. Solovay and V. Strassen. (1977). A fast monte-carlo test for primality. *SIAM Journal on Computing* [online] 6 (1), 84-85. doi: 10.1137/0206006.

Rabin, M. O. (1980). Probabilistic algorithm for testing primality. *Journal of Number Theory* [online] 12 (1), 128-138. doi: 10.1016/0022-314X(80)90084-0.

Rivest, R., Shamir, A. and Adleman, L. M. (1983b). *Cryptographic communications system and method* [online] Google Patents. https://www.google.com/patents/US4405829 .

Rivest, R., Shamir, A. and Adleman, L. (1983a). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 26, 96-99. doi: 10.1145/357980.358017.

Rogaway, P. and Shrimpton, T. (2004). Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. *International Workshop on Fast Software Encryption* [online] . http://web.cs.ucdavis.edu/~rogaway/papers/relates.pdf.

What is certificate authority (CA)? http://searchsecurity.techtarget.com/definition/certificate-authority.

PKCS #1 v2.2: RSA cryptography standard. https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf.

Ruoti, S., Andersen, J., Zappala, D. and Seamons, K. (2015). Why johnny still, still can't encrypt: Evaluating the usability of a modern PGP client. *CoRR* [online] . http://arxiv.org/abs/1510.08555.

Schneier, B. (1993). Description of a new variable-length key, 64-bit blok cipher (blowfish). *International Workshop on Fast Software Encryption* [online] , 191-204.

Schneier, B. (1996). *Applied cryptography* (2. ed. ed.). New York, NY [u.a.]: Wiley.

OpenPGP message format. https://tools.ietf.org/html/rfc4880.

The comp.security.pgp FAQ. http://www.cam.ac.uk.pgp.net/pgpnet/pgp-faq/faq.html.

The first collision for full SHA-1. https://shattered.io/static/shattered.pdf.

Sussman, V. (1995). Lost in kafka territory. *U.S. News & World Report*, 118, 32.

GitHub - lachesis/scallion: GPU-based onion hash generator. https://github.com/lachesis/scallion.

PGP desktop 10.2 for windows user's guides. http://www.symantec.com/docs/DOC4558.

The GNU privacy guard. https://www.gnupg.org/.

Vacca, J. R. (2004). *Public key infrastructure: Building trusted applications and web services*. Boca Raton: Auerbach Publications.

Whitten, A. and Tygar, J. D. (1999). Why johnny can't encrypt: A usability evaluation of PGP 5.0. *Proceedings of the 8th USENIX Security Symposium* [online] , 169-183.

Wilson, D. and Ateniese, G.*From pretty good to great: Enhancing PGP using bitcoin and the blockchain.* International Conference on Network and System Security, , 368-375.