Aalto University
School of Science
Degree Programme in Engineering Physics and Mathematics

Ferdinand Blomqvist

Kleptography – Overview and a new proof of concept

Master's Thesis
Espoo, August 11, 2015

Supervisor: Professor Camilla Hollanti
Advisor: Professor Kaisa Nyberg

Aalto University
School of Science
Degree Programme in Engineering Physics and Mathematics

ABSTRACT OF
MASTER'S THESIS

Author: Ferdinand Blomqvist

Title: Kleptography – Overview and a new proof of concept

Date: August 11, 2015 **Pages:** xiii + 78

Major: Mathematics **Code:** F3006

Supervisor: Professor Camilla Hollanti

Advisor: Professor Kaisa Nyberg

Kleptography is the study of stealing information securely and subliminally. A successful kleptographic attack is undetectable, and, therefore, kleptographic attacks are only useful against so called black-box implementations of cryptographic primitives and protocols. The dangers of black-box cryptography have been publicly known for almost two decades. However, black-box cryptography is still widespread. The recent revelations of the United States National Security Agency's efforts to sabotage cryptographic standards and products have showed that kleptographic attacks constitute a real threat, and that countermeasures are needed.

In this thesis we study known kleptographic primitives, and examine how they can be used to compromise the security of well known secure communication protocols. We also take an in depth look at the Dual_EC_DRBG, which is a kleptographic pseudorandom number generator designed by the NSA. Finally, we present a new proof of concept – a kleptographic pseudorandom number generator that can be used as a secure subliminal channel. The difference between our generator and the Dual_EC_DRBG is that the Dual_EC_DRBG leaks its own state while our generator can be used to leak any information.

The strength of our generator is that it can be used to compromise the security of any protocol where random numbers are transmitted in plaintext. In addition, the use of our generator cannot be detected by the users of the protocol, except under very special circumstances. Compared to kleptographic attacks based on other kleptographic primitives, attacks based on our generator are usually inefficient. However, there are many protocols that cannot be compromised with the use of traditional kleptographic techniques. An example of a system that cannot be compromised with these techniques is the Universal Mobile Telecommunications System. With our generator it is, nevertheless, possible.

We have showed that all the protocols we have examined are susceptible to kleptographic attacks. The only way to definitely stop kleptographic attacks is to ensure that all implementations of cryptographic products are open, *i.e.*, implementations where the user can verify that the implementation conforms to the specifications. Therefore, we hope that this thesis will help to raise awareness about the dangers of black-box cryptography, and lead to an increased demand for open cryptographic solutions.

Keywords: kleptography, cryptography, black-box cryptography, random number generation, Dual EC, subliminal channels, secure communication

Language: English

Aalto-universitetet
Högskolan för teknikvetenskaper
Utbildningsprogrammet för teknisk fysik och matematik

SAMMANDRAG AV
DIPLOMARBETET

Författare:	Ferdinand Blomqvist		
Arbetets namn:	Kleptografi – Överblick samt ett nytt koncepttest		
Datum:	Den 11 augusti 2015	**Sidantal:**	xiii + 78
Huvudämne:	Matematik	**Kod:**	F3006
Övervakare:	Professor Camilla Hollanti		
Handledare:	Professor Kaisa Nyberg		

Kleptografi är läran om hur man kan stjäla information på ett säkert och diskret sätt. Kännetecknet för en lyckad kleptografisk attack är att den kan utföras obemärkt. Därför kan kleptografiska attacker enbart utnyttjas till fullo emot implementationer av kryptografiska primitiv som kan anses vara svarta lådor. Riskerna med svartlådskryptografi har redan länge varit kända, men tyvärr är implementationer som kan anses utgöra svarta lådor ännu vanliga. Nyligen har det avslöjats att Amerikas förenta staters National Security Agency har lagt ner stora resurser på att sabotera kryptografiska standarder och produkter. Dessa avslöjanden visar att kleptografiska attacker utgör ett stort hot och att effektiva motåtgärder behövs.

I detta diplomarbete studerar vi kända kleptografiska primitiv och undersöker hur dessa kan användas för att knäcka väl kända kryptografiska kommunikationsprotokoll. Vi undersöker även Dual_EC_DRBG som är en kleptografisk slumptalsgenerator designad av NSA. Till sist presenterar vi vår egen kleptografiska slumptalsgenerator som kan användas som en säker och gömd kommunikationskanal. Skillnaden mellan Dual_EC_DRBG och vår generator är den att Dual_EC_DRBG enbart läcker sitt interna tillstånd, medan vår generator kan användas för att läcka valfri information.

Vår kleptografiska generators styrka är att den kan användas för att knäcka säkerheten av alla kommunikationsprotokoll som exponerar slumptal. Dessutom kan användningen av vår generator inte upptäckas av kommunikationsprotokollets användare, förutom under väldigt speciella omständigheter. I jämförelse med traditionella kleptografiska attacker är det ofta svårare och inte lika effektivt att uppnå samma mål med vår generator. Dock finns det flera kommunikationsprotokoll vars säkerhet inte går att knäcka med traditionella kleptografiska metoder. Ett exempel på ett sådant kommunikationsprotokoll är säkerhetsprotokollet i 3G nätverket. Med vår generator är det däremot enkelt att knäcka 3G nätverkets säkerhetsprotokoll.

Vi har visat att alla kommunikationsprotokoll vi har studerat är mottagliga för kleptografiska attacker. Det enda sättet att definitivt gardera sig mot kleptografiska attacker är att använda kryptografiska produkter vars riktighet går att verifiera. På grund av detta hoppas vi att detta diplomarbete kan öka kännedomen om riskerna med svartlådskryptografi och som en ge upphov till en ökad efterfrågan av kryptografiska produkter vars riktighet går att verifiera.

Nyckelord:	kleptografi, kryptografi, svartlådskryptografi, slumptalsgenerering, Dual EC, säker kommunikation
Språk:	Engelska

Acknowledgements

I would like to thank my advisor, Professor Kaisa Nyberg for her support and useful suggestions during my work on this thesis. I would also like to thank my friend Unto Kuuranne for fruitful discussions and suggestions on potential "targets" worth looking into.

I would like to thank my supervisor, Professor Camilla Hollanti. Moreover, I would like to thank my girlfriend Maria for being patient and understanding during the time this thesis was written.

Finally, I would like to thank the Aalto Science-IT project for providing the computational resources needed to complete this thesis.

Espoo, August 11, 2015

Ferdinand Blomqvist

Acronyms and Abbreviations

3GPP	Third Generation Partnership Project
AES	Advanced Encryption Standard
API	application programming interface
AuC	authentication center
AUTH	authentication token
CNG	Cryptography Next Generation
CSPRNG	cryptographically secure pseudorandom number generator
DH	Diffie-Hellman
DHKE	Diffie-Hellman key exchange
DHP	Diffie-Hellman problem
DLK	discrete log kleptogram
DLP	discrete logarithm problem
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
Dual_EC_DRBG	Dual Elliptic Curve Deterministic Random Bit Generator
ECC	ellitic curve cryptography
ECDH	elliptic curve Diffie-Hellman
ECDLP	elliptic curve discrete logarithm problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECMQV	elliptic curve Menezes-Qu-Vanstone
FIPS	Federal Information Processing Standard
HE	home environment
ID	identifier
IIS	Internet Information Services
IMSI	international mobile subscriber identity
IV	initialization vector

KPRNG	kleptographic pseudorandom number generator
KS	Kolmogorov-Smirnov
OAEP	optimal asymmetric encryption padding
OFB	output feedback
P-TMSI	packet TMSI
PRNG	pseudorandom number generator
RAND	random number
RNG	random number generator
RSA	Rivest-Shamir-Adleman
SETUP	secretly embedded trapdoor with universal protection
SGSN	serving GPRS support node
SIM	subscriber identity module
SN	serving network
SQN	sequence number
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TMSI	temporary mobile subscriber identity
TPM	Trusted Platform Module
UMTS	Universal Mobile Telecommunications System
USIM	universal subscriber identity module
VLR	visitor location register
XOR	exclusive or

Contents

Chapter 1

Introduction

On September 5, 2013, The Guardian [BBG13], the New York Times [PLS13] and ProPublica [LPS13] reported about the existence of the SIGINT Enabling Project run by the United States National Security Agency (NSA) [nsa]. The mission of the program is to "actively [engage] the U.S. and foreign IT industries to covertly influence and/or overtly leverage their commercial products' designs to make them 'exploitable'." The program has an annual budget larger than $250 million, and has, according to the sources, been quite successful.

Naturally, we are interested in how the NSA has managed to achieve this. According to Edward Snowden "[p]roperly implemented strong crypto systems are one of the few things you can rely on" [LPS13]. According to The Guardian *et al.*, the NSA often bypasses the encryption all together, or as ProPublica put it: "Because strong encryption can be so effective, classified N.S.A. documents make clear, the agency's success depends on working with Internet companies – by getting their voluntary collaboration, forcing their cooperation with court orders or surreptitiously stealing their encryption keys or altering their software or hardware." [LPS13]

In other words, the NSA has been inserting backdoors into cryptographic protocols and collaborated with manufacturers of cryptographic products to insert backdoors into their products. They have also used other means to accomplish their goals, but we will focus on the possible backdoors.

When inserting a backdoor into any device or protocol the risk is that someone else will find and exploit the backdoor. This risk can be thwarted by using kleptographic, *i.e.*, *asymmetric* backdoors, which can only be used by an adversary who has access to the encryption keys protecting the backdoors. In this thesis we will take a look at known kleptographic attacks against well known encryption and key exchange protocols. Then we will show how these attacks can be used to compromise the security of protocols widely used to secure communication over insecure channels. Finally we will present a new proof of concept – a kleptographic random number generator that can be used to leak arbitrary messages – and show how it can be used to compromise the security of different protocols. Before this, we, however, need to review some of the basic concepts of cryptography and kleptography.

1.1 Basic cryptographic concepts

In this section we will only cover some really basic concepts that are relevant for the understanding of the material that will be presented later. For a more in depth treatment we refer to [Sti05] or any other introductory text on cryptography.

There are two types of encryption methods: symmetric and asymmetric. With a symmetric method the same key can be used for both encryption and decryption, while asymmetric or *public key* encryption methods require two keys; one for encryption and one for decryption. The key used for encryption is called the public key, while the key used for decryption is called the *private key*. As the names suggest, the public key can be published while the private key must be kept secret.

Symmetric encryption methods are usually significantly faster, but they require that each party knows the encryption key. This can be an obstacle for two parties wanting to communicate securely over an insecure line, as they have to find some way to agree on a secret key.

This problem can be solved by public key cryptography. Alice only needs Bob's public key to be able to send him an encrypted message, and, as knowledge of the public key does not compromise the encrypted message, Bob can send his public key to Alice over the insecure channel. However, there is a problem with this: how can Alice know that the public key she has received is Bob's? The solution is to exchange public keys in an authenticated manner, or alternatively rely on a trusted authority that confirms that the key Alice has received is Bob's.

The advantage provided by public key encryption methods comes with a price; asymmetric encryption methods are usually several orders of magnitude slower then their symmetric counterparts. As a consequence, the usual way to establish secure communication over an insecure channel is: Alice and Bob use some public key encryption method or a dedicated key exchange protocol to agree on a shared secret that is then used, either directly or indirectly, as the key for a symmetric encryption method that is used to secure the communication. As we will see in Chapter 3, both the Transport Layer Security (TLS)/Secure Sockets Layer (SSL) and Secure Shell (SSH) protocol use this approach. Note that, for this to be secure, Alice and Bob need to have authenticated copies of each other's public keys.

1.2 Kleptography

Kleptography is the study of stealing information securely and subliminally. It is a subfield of cryptology and cryptovirology, and can be viewed as a natural extension of the study of subliminal channels, which was pioneered by Gus Simmons. The concept of kleptography was introduced by Adam Young and Moti Yung in 1996 and the term itself was coined the following year.

A kleptographic attack is an attack where the attacker employs an *asymmetric backdoor*. An asymmetric backdoor is a cryptographic backdoor that is protected with asymmetric encryption. Due to this, no one except the attacker can use the backdoor, even if its existence and specifications become public knowledge.

Kleptographic attacks are mainly useful in conjunction with so called *black-box* implementations of cryptographic primitives. With black-box implementation we mean any implementation whose inner workings cannot be verified. In other words the device might be doing things it is not supposed to do, and the user has no way of knowing whether this is the case or not. Examples of implementations that qualify as black-boxes are tamper resistant hardware and closed source implementations.

Of course it is possible to launch kleptographic attacks against implementations that cannot be considered black-boxes. However, such attacks are relatively easy to detect. Essentially, a secure kleptographic attack is one that is computationally undetectable, and thus attacks against open implementations cannot be considered secure. This security notion will be made more precise in the next section.

1.2.1 SETUP and other definitions

The notion of a secretly embedded trapdoor with universal protection (SETUP) was introduced by Young and Yung in 1996 in [YY96]. It was further refined in [YY97a] and the definitions given below conform with the refined versions. The notion is a way to precisely specify what we mean by a secure kleptographic attack.

Definition 1.1. Assume that C is a black-box cryptosystem with a publicly known specification. A (regular) SETUP mechanism is an algorithm modification made to C to get C' such that:

1. The input of C' agrees with the public specifications of the input of C.
2. C' computes efficiently using the attacker's public encryption function E (and possibly other functions as well), contained within C'.
3. The attacker's private decryption function D is not contained within C' and is known only to the attacker.
4. The output of C' agrees with the public specifications of C. At the same time, it contains published bits (of the user's secret key) which are easily derivable by the attacker (the output can be generated during key-generation or during system operation like message sending).
5. Furthermore the output of of C and C' are polynomially indistinguishable (as in [GM84]) to everyone except the attacker.
6. Given the specifics of the setup algorithm and its presence in the implementation (*e.g.*, by reverse-engineering of hardware tamper-proof device), users (except the attacker) cannot determine future or past keys.

Definition 1.2. A weak SETUP is a regular SETUP except that the output of C and C' are polynomially indistinguishable to everyone except the attacker and the owner of the device who is in control (knowledge) of his or her own private key.

Definition 1.3. A strong SETUP is a regular SETUP, but in addition we assume that the users are able to hold and fully reverse-engineer the device

after its past usage and before its future usage. They are able to analyze the actual implementation of C' and deploy the device. However, the users still cannot steal previously generated/future generated keys, and if the SETUP is not always applied to future keys, then SETUP-free keys and SETUP keys remain polynomially indistinguishable.

These definitions are not directly compatible with all the scenarios discussed in this thesis. For instance, we will not always be leaking secret keys. However, the notions can be canonically extended/modified to make sense in all these cases, and we will use the modified versions without mentioning it further.

Other useful notions that will be used later are:

Definition 1.4. A (m, n)-leakage scheme is a SETUP mechanism that leaks m keys/secret messages over n keys/messages that are output by the cryptographic device ($m \leq n$).

Definition 1.5. A *kleptogram* is an encryption of a value (hidden value) that is displayed within the bits of an encryption/signature of a plaintext value (outer value).

1.3 Random Number Generation

There are many situations in cryptography that require random numbers or bits. Examples of such situations are key generation and generation of random parameters needed in different primitives. The distinction between random numbers and bits is not important as they are essentially the same. Therefore we choose to always speak about random number generation.

Physically generating random numbers, *e.g.* by tossing coins or collecting hardware entropy, is very time consuming. As cryptographic applications usually require large amounts of random numbers, this is a problem. The solution is to use so called *pseudorandom number generators (PRNGs)*. A PRNG is a deterministic algorithm that takes an initial *seed* and extends it into a long string of numbers that appear random. In cryptographic applications the seed is usually extracted from a physical source of randomness.

PRNGs are also needed in many other applications, *e.g.*, in Monte Carlo simulations. However, the requirements imposed on the PRNGs vary considerably between different applications. For instance, in Monte Carlo simulations it is enough that the generated numbers satisfy certain statistical properties, while cryptographic applications require randomness and unpredictability. More precisely, given a long string of output it should be infeasible to predict the next *bit* with probability higher that 0.5, even if the algorithm in use is known. However, if the secret seed is known, then all output can be predicted.

PRNGs suitable for use in cryptographic applications are usually called cryptographically secure pseudorandom number generators (CSPRNGs). In this thesis we only consider CSPRNGs, so for ease of notation we speak about PRNGs, and random number generators (RNGs) when we mean true random number generators.

1.4 Prerequisites and notation

The reader is expected to be familiar with basic group theory, modular arithmetic and the group structure of points on elliptic curves.

The basics of group theory can be found in any introductory text on abstract algebra and/or group theory, and most introductory texts on discrete mathematics. [Pin12] and [BJN94] are examples of the former. An example of the latter is [Big99], which also covers modular arithmetic in detail.

A good introduction to elliptic curves – which is also suitable for cryptographers and computer scientists – is [Was12]. Other good texts are [ST13] and [Sil09]. These are, however, written with mathematicians in mind, and, especially the later, requires a stronger foundation in mathematics. A really short introduction to elliptic curves can also be found in [Sti05].

We use p to denote primes. Usually, but not always, q also denotes a prime. Whether this is the case should be clear from the context. The finite field with n elements is denoted by $\mathbb{F}_n$. Recall that finite fields of a given size are unique up to representation. Furthermore, there does not exist finite field of any given size; n must be a prime power.

We denote the set of integers modulo n by $\mathbb{Z}_n$, and the corresponding multiplicative group by $\mathbb{Z}_n^*$. We usually use E to denote an elliptic curve, and, given a field $\mathbb{K}$, $E(\mathbb{K})$ denotes the additive group of points on the elliptic curve defined over $\mathbb{K}$. In this thesis, $\mathbb{K}$ will always be a finite field.

Given a group G, we use $\#G$ to denote the number of elements in G. For $a \in G$, $\text{ord}(a)$ denotes the order of a in G. We use $\|$ to denote concatenation of bitstrings. If a and b are not bitstrings, then $a\|b$ denotes the concatenation of their bitstring representations.

Chapter 2

The Discrete Log Kleptogram

The discrete log kleptogram (DLK) and applications of it were presented in [YY97a] and [YY97b]. In this chapter we present these results. We also show how the Rivest-Shamir-Adleman (RSA) public key encryption method can be SETUPed. The SETUP in Section 2.6, the usage notes in Section 2.2.4, and the details in Section 2.4.5 are our own work. Otherwise, all the attacks are due to Adam Young and Moti Yung.

2.1 Prerequisites

Before we can present the DLK we need to review the Diffie-Hellman key exchange (DHKE) and the ElGamal public key encryption scheme.

2.1.1 Diffie-Hellman Key Exchange

The DHKE is a way for two parties to establish a shared secret over an insecure channel. It is one of the cornerstones of public key cryptography. Originally it was used over multiplicative groups of finite fields of prime order but, as it can also be used with other groups, we present it in a more general setting.

Let G be a cyclic group of order q with generator g. All of these are public knowledge. To establish a shared secret Alice and Bob do the following:

- Alice chooses a secret integer $1 < a < q$ and computes g^a, which she then sends to Bob.
- Bob chooses a secret integer $1 < b < q$ and computes g^b, which he then sends to Alice.
- Now Alice can compute $(g^b)^a$ and Bob can compute $(g^a)^b$. Hence they share the secret g^{ab}.

It is obvious that the security of the secret is dependent on the hardness of the discrete logarithm problem (DLP), *i.e.*, given $g^x = h$, find x. The hardness depends on the group and therefore only certain types of groups should be used.

We do not, however, dwell on this detail here, since this is dealt with in detail in Section 2.1.3.

2.1.2 ElGamal Public Key Encryption

The ElGamal public key cryptosystem was presented in [ElG85]. It is based on the DHKE, and was originally also restricted to multiplicative groups of finite fields of prime order. However, we choose to work in a more general setting. The ElGamal Public Key Encryption scheme is as follows:

Suppose that Alice wants to send a message to Bob. Bob creates a public key by choosing a cyclic group G of order q with generator g. He also chooses a secret integer $1 < b < q$, which will be his private key, and publishes his public key (G, g, g^b). Once Alice has obtained Bob's public key she sends her message $m \in G$ to Bob by:

- Choosing a secret integer $1 < a < q$ and computing $m_1 = g^a$.
- Computing $m_2 = m(g^b)^a = mg^{ab}$.
- Now she sends the pair (m_1, m_2) to Bob.

When Bob receives Alice's message he decrypts it by calculating

$$m_2 m_1^{-b} = mg^{ab}(g^a)^{-b} = mg^{ab-ab} = m.$$

The observant reader notices that Bob's public key g^b and the one-time value m_1 serve as inputs to a DHKE, and then the shared secret is used to mask the message. Note that the secret integer a should not be reused. Reusing a will not break the encryption, but knowledge of one message encrypted with a will compromise all other messages encrypted using a, assuming they are encrypted with the same key. We will conclude this section by showing that ElGamal encryption is secure if and only if the Diffie-Hellman problem (DHP) is secure.

Assume that we are able to to solve the DHP, *i.e.* given g^a and g^b we are able to obtain g^{ab}. Upon seeing the ElGamal encryption (g^a, mg^{ab}) and the public key g^b we can obtain g^{ab}. Now it is easy to calculate the inverse of g^{ab} and hence we can recover m. Now assume that we are able to break ElGamal encryption, *i.e.*, given g^a, g^b and $m_2 = mg^{ab}$ we can recover m. Clearly, the only way of obtaining m is to multiply m_2 with $\left(g^{ab}\right)^{-1}$, and hence any one that is able to recover m can calculate g^{-ab}, and therefore g^{ab}.

2.1.2.1 Practical considerations

The ElGamal public encryption scheme is usually modified when used with elliptic curves. The reason for this is simple: it is hard to map an arbitrary message m to an elliptic curve point in an invertible way, since the x-coordinates of all the points on the curve do not cover the base field. We will now describe the modified version.

Let E be an elliptic curve over $\mathbb{F}_p$ with generator G. Bob's public key is (E, G, bG). The first part of the ElGamal encryption is computed as usual, *i.e.*, $M_1 = aG$. The second part is set to $m_2 = m \cdot \hat{x}(a(bG))$, where $\hat{x}(P)$ denotes the x-coordinate of P and the multiplication is performed in $\mathbb{F}_p$.

Let $P = (x, y) \in E(\mathbb{F}_p)$. We know that $-P = (x, -y)$, and hence $\hat{x}(P) = \hat{x}(-P)$. Recall that any viable x-coordinate identifies a point up to its sign. Given $\hat{x}(M_1)$, Bob can compute bM_1 up to sign and obtain $\hat{x}(abG) = \hat{x}(\pm bM_1)$. Thus it is enough for Alice to send the x-coordinate of M_1 to Bob.

2.1.3 Security considerations

The security of the DHKE, and therefore also the ElGamal encryption scheme, depends on the hardness of the so called DHP. Formally it is stated as: Given a group G of order q with generator g and also g^x and g^y, where x, y are randomly chosen integers between 1 and q, find g^{xy}. Currently the most efficient means to solve the DHP is to solve the DLP.

It is obvious that the hardness of the DLP depends on the group G. Suitable and frequently used groups include:

- Multiplicative groups of finite fields of prime order, *i.e.*, $\mathbb{F}_p$.
- Multiplicative groups of other finite fields, *i.e.*, $\mathbb{F}_{p^n}$.
- Groups of elliptic curves considered over finite fields.

These groups are only secure if they are large enough and the order of the group must be divisible by a large prime to thwart known attacks. Note that due to the different structures of the different groups, some require a much larger size then others in order to be considered secure. For further details and specific algorithms to solve the DLP we refer to [Sti05] and [Was12]. However, the information can probably be found in most introductory texts on cryptography.

2.2 The DLK and SETUPing Diffie-Hellman

2.2.1 The discrete log kleptogram

Let us assume the following scenario: We have a device that is only allowed to display the value g^c, where g is the generator of a group G of order q and $1 < c < q$. Then it is possible to leak a value c_2 over a message $m_1 = g^{c_1}$ such that the next message $m_2 = g^{c_2}$ is compromised. Let (G, g, g^k) be the attacker's public ElGamal key, w a fixed odd integer and a, b integer constants. Furthermore, let $H : G \to L$, where $L = \{2, \ldots, q-1\}$, be a pseudorandom cryptographic one-way function. The following describes the operation of a contaminated device.

1. For the first usage, choose $1 < c_1 < q$ uniformly at random.
2. Calculate and output $m_1 = g^{c_1}$.
3. Store c_1 in non-volatile memory for the next usage.
4. For the second usage choose $t \in \{0, 1\}$ uniformly at random.
5. Calculate $z = g^{c_1 - wt}(g^k)^{-ac_1 - b}$.
6. Calculate $c_2 = H(z)$ and output $m_2 = g^{c_2}$.

In order to recover c_2 the attacker only needs to passively tap the communication and obtain m_1 and m_2. Then he/she recovers c_2 by:

1. $r = m_1^a g^b = g^{ac_1+b}$.
2. $z_1 = m_1 r^{-k} = g^{c_1} g^{-k(ac_1+b)}$.
3. If $g^{H(z_1)} = m_2$ then $c_2 = H(z_1)$. Otherwise $c_2 = H(z_1 g^{-w})$.

This is a $(1, 2)$-leakage system and it is clear that only the attacker can recover c_2 as no one else knows k. The reader probably wonders about the role of w, a, b and why H should be a pseudorandom cryptographic one-way function. We, however, defer the treatment of this to the Section 2.2.2.

This SETUP mechanism – with certain assumptions it is in fact a strong SETUP – is interesting in a few ways. First it does not use any existing subliminal channel, but instead creates a new channel which is then exploited. Secondly, we do not simply choose the value of c_2 and then public key encrypt it. Instead it is designed so that g^{c_1} is the ElGamal encryption of the secret z we want to leak. ElGamal encryption doubles the size of the message, but we are only able to send one group element. However, we can circumvent the problem by designing z such that the second part of the ElGamal encryption of z is equal to the first. This gives us the equation

$$g^{c_1} = z \left(g^k\right)^{c_1}.$$

Solving for z gives

$$z = g^{c_1} \left(g^k\right)^{-c_1},$$

and this is exactly the same as the z given earlier, if we omit the constants. The constants

$$g^{-wt} \quad \text{and} \quad \left(g^k\right)^{-b}$$

are straightforward to add, and a can be added by replacing the c_1 in $(g^k)^{-c_1}$ with $c_1 a$. It is worth noting that this is possible only because the device is free to choose its own random parameters.

2.2.2 Security of the DLK

For this mechanism to be a SETUP, it must be intractable for everybody except the attacker to recover c_2 even if the device is reverse engineered. If this is the case, then we call the DLK secure. Further, it is also required that no one except the attacker can detect the presence of the mechanism. Let us consider the first issue.

Lemma 2.1. The DLK is secure if and only if the DHP is secure.

Proof. Assume that we are able to solve the DHP, *i.e.*, given g^u and g^v we can calculate g^{uv}. From g^{c_1} and g^k we can calculate $g^{c_1 k}$ and therefore

$$z_1 = g^{c_1} \left(g^{c_1 k}\right)^{-a} \left(g^k\right)^{-b} = g^{c_1} g^{-k(ac_1+b)}.$$

Hence we have solved the DLK.

Now assume that we are able to solve the DLK, *i.e.*, from g_1^c and g^k we are able to obtain $z_1 = g^{c_1} g^{-k(ac_1+b)}$. We want to show that given g^u and g^v we

can obtain g^{uv}. Solving the DLK gives $f = g^u g^{-v(au+b)}$. Multiplying by $g^{-u}g^{vb}$ we get

$$f_2 = fg^{-u}g^{vb} = g^{-auv} = (g^{uv})^{-a}\,.$$

We can easily find the inverse of a modulo q (assuming it exists) and thus we can obtain g^{uv}. □

This shows that a user of the device that does not know the random exponents chosen by the device cannot determine z, and hence not c_2, even if the specifics of the SETUP mechanism are known.

Turning our attention to the second issue we obtain the following results.

Lemma 2.2. Assume that $g_1 = g^{1-ka}$ is a generator of G and that c_1 is uniformly distributed. Then z is uniformly distributed in G.

Proof. Recall that $z = g^{c_1-wt}(g^k)^{-(ac_1+b)} = g^{c_1(1-ka)}g^{-kb-wt}$. Thus $z = g_1^{c_1}g^{-kb-wt}$. Since g_1 is a generator, there exists u such that $g_1^u = g^{-kb-wt}$. Hence $z = g_1^{u+c_1}$, which proves the claim. □

Corollary 2.3. If H is a pseudorandom function, then c_2 is uniformly distributed in L. Furthermore, g^{c_2} is uniformly distributed in G.

In Lemma 2.2 we assumed that g^{1-ka} is a generator of G, which begs the question: is the assumption sound? The standard groups used for elliptic curve Diffie-Hellman (ECDH) have prime order, and hence all elements, except the identity element, generate the whole group. When using multiplicative groups of finite fields there are essentially two options: groups of prime order and groups of order $p-1$ for a safe prime p. Assume that g has order $p-1 = 2q$, for some prime q, *i.e.* p is a safe prime. It is easy to check that if m is odd and does not divide q, then g^m has order $p-1$ and thus generates the whole group. Thus we have shown that, in these settings, k and a can always be chosen such that g^{1-ka} generates G.

Combining all these results we obtain:

Theorem 2.4. The Discrete Log kleptogram is a strong SETUP.

We will conclude this section by explaining why H should be a one-way function, and why w, a, b are used. The constants a and b serve as an extra precaution when the DLK is used to SETUP ElGamal signatures. We will return to this in Section 2.3.2. Turning our attention to H and w, consider the following scenario: the user has a contaminated black-box device which makes the choices of the secret exponents available to the user, and the Diffie-Hellman (DH) group used has a generator of order $p-1$ for some safe prime p. In addition, the user is aware of how the SETUP mechanism works, but does not know the attacker's public key. Now assume that H is invertible and that $t = 0$ all the time. Without loss of generality, assume that $a = 1$ and $b = 0$. The user can probabilistically detect the presence of the SETUP as follows. The user generates several pairs of DH values and obtains the corresponding exponents. Let c_1 and c_2 be such a pair. Since H is invertible, the user can calculate z. If the attacker's private key k is even, then $g^{c_1}z^{-1} = g^{kc_1}$ must be a quadratic residue modulo p. On the other hand, if k is odd, then $g^{c_1}z^{-1}$ is a quadratic residue whenever c_1 is even, and a quadratic non-residue otherwise. By looking

for quadratic residues or non-residues, the user can probabilistically detect the presence of the SETUP mechanism.

There are two ways to stop this. If H is non-invertible, then the user cannot obtain z, and therefore not use the probabilistic technique described above. The use of w in conjunction with a random t serves the same purpose. According to the original paper, [YY97a], w is used as a precaution in case H turns out to be invertible. However, the probabilistic detection only works in the above situation. Changing G to the maximal prime ordered subgroup of $\mathbb{Z}_p^*$, for some safe prime p, renders the method useless, since all elements in G are quadratic residues modulo p. In addition, the technique is of no use when elliptic curve groups are used. Due to this, we consider the use of both w and an non-invertible H unnecessary, and would simply omit w altogether.

2.2.3 Strong SETUP in Diffie-Hellman

We have shown that the DLK is a strong SETUP for the DHKE. However, the kleptogram can easily be extended from a $(1, 2)$-leakage system to a $(l, l + 1)$-leakage system. This is done as follows: Assume that the device has outputed $m_n = g^{c_n}$ and has c_n in memory. Then c_{n+1} is calculated by:

1. Choose $t \in \{0, 1\}$ uniformly at random.
2. Calculate $z = g^{c_n - wt}(g^k)^{-ac_n - b}$.
3. Calculate $c_{n+1} = H(z)$ and output $m_{n+1} = g^{c_{n+1}}$.

In other words, the device works like the normal DLK device except that it repeats the three last steps l times. c_{n+1} is recovered in the same way as before, just substitute c_1 and c_2 with c_n respective c_{n+1} in the recovery steps.

2.2.4 Practical considerations

Recall that we presented the DHKE, the ElGamal encryption scheme and the DLK in a general group setting, and hence we can deduce that the DLK should work equally well with the finite field and elliptic curve versions of theses protocols. However, one technical detail must be considered: Is it possible to find a pseudorandom one-way function H from any group to $\mathbb{Z}_q$?

For finite fields this is feasible. Simply take any good quality cryptographic hash function and restrict its output to $\mathbb{Z}_q$, where q is the order of the group generated by the generator g. This is not as easy as it sounds, but it is easy to get close enough. In practice, most cryptographic hash functions have range $\mathbb{Z}_2^n$, for some suitable n. With suitable choices of p it is possible to find g such that $q \approx 2^n - 1$ for some n. In these cases, the output will be close to uniformly distributed in $\mathbb{Z}_q$. For efficiency, many implementations of DHKEs restrict the size of the private exponents to n-bits. This removes the requirement that the output of H should be uniformly distributed in $\mathbb{Z}_q$, and it is enough to be uniformly distributed in $\mathbb{Z}_2^n$. Any good quality hash function should be able to satisfy this requirement, and hence a suitable H can easily be found.

It is also possible to construct a suitable function for elliptic curves, and we propose the following construction. Let E be an elliptic curve defined over $\mathbb{F}_p$, where p is an n-bit prime. For simplicity we assume that n is even. Let

$h : E(\mathbb{F}_p) \to \mathbb{Z}_2^n$ be a transformation that takes the x-coordinate of the given point and splits it into two parts, the least and most significant $n/2$ bits, which are then permuted randomly. In other words, for each input there are two possible outcomes: the unchanged x-coordinate of the input, or then the least significant $n/2$ bits have been swapped with the most significant $n/2$ bits. Now let H_1 be any cryptographic hash function with range $\mathbb{Z}_2^n$ and define

$$H : E(\mathbb{F}_p) \to \mathbb{Z}_2^n, \quad H(P) = H_1 h(P).$$

The results in Chapter 5 imply that if the input points are uniformly distributed in $E(\mathbb{F}_p)$, then the output of h will be close to uniformly distributed in $\mathbb{Z}_2^n$. Furthermore, given this input, the output of H_1 should also be uniformly distributed. This does not mean that the output is uniformly distributed in $\mathbb{Z}_q$, where $q = \#E(\mathbb{F}_p)$. However, the elliptic curves used in cryptography often have $p \approx q$, and hence the output will have a distribution that is close to uniform in $\mathbb{Z}_q$. Therefore, H is suitable for use in the DLK. Note that H is not a deterministic function, and hence the attacker must try all possible outputs after obtaining z. However, as there are only two possible outputs for each input, this is not a problem.

2.3 SETUPs in ElGamal systems

The reason why the DLK was given that specific name is that it can be applied to many encryption/signature schemes that are based on the (assumed) difficulty of the DLP in certain groups. In this section we describe how to SETUP both the ElGamal encryption and ElGamal signature scheme, while the next section is dedicated to describing how the Digital Signature Algorithm (DSA) can be abused.

2.3.1 Strong SETUP in the ElGamal Encryption Scheme

One obvious way to SETUP ElGamal encryption is to use the DLK when generating keys. This is, however, not a very efficient SETUP. If the user's keys are generated by a personal device it would require the user to generate several keys in order for the attack to be successful. On the other hand, if the keys are generated by some central authority, and the attacker has access to all the generated keys, and, in addition, knows in which order they were generated, then the attacker could obtain l of $l + 1$ private keys. SETUPing the keys is also possible with the ElGamal signature scheme and the DSA. However, the efficiency problem persists.

A more efficient method is to compromise encrypted messages. Recall that the first part of an ElGamal encrypted message is g^k, and hence it is easy to use the DLK to create a SETUP. It is a $(1, 2)$-leakage scheme but can be extended to a $(l, l+1)$-leakage scheme in the same way that the SETUP for the DHKE. It should be noted that this SETUP mechanism leaks messages and not private keys. As already mentioned, breaking ElGamal encryption is equivalent to breaking the DHP and by the same arguments as in the previous section we can conclude that this is a strong SETUP.

Public key cryptography is usually avoided when large amounts of data should be encrypted. This is a consequence of the fact that public key cryp-

tography is much slower than symmetric key cryptography and usually requires greater bandwidth. Therefore hybrid systems, *i.e.*, systems where the encryption key used for the symmetric cipher is encrypted with a public key encryption method, are often used. However, hybrid systems based on ElGamal are also susceptible to SETUP attacks. By employing the DLK and setting the symmetric key to c_2 we have effectively created a $(1, 1)$-leakage system.

2.3.2 Regular SETUP in the ElGamal Signature Scheme

Before we present the SETUP mechanism we review the ElGamal signature scheme. Let p be prime, g a generator of $G = \mathbb{Z}_p^*$ and (G, g, g^x) Alice's public ElGamal key. The signature of the message m (represented as an integer modulo $p-1$) is (r, s), where

$$r = g^k \pmod{p} \quad \text{and} \quad s = k^{-1}(m - xr) \pmod{p-1}.$$

The signature is verified by checking that

$$g^m = (g^x)^r r^s \pmod{p}$$

holds. The same nonce, *i.e.*, k should never be used twice, as this will compromise Alice's private key x.

It is also possible to use this signature scheme with other groups. In these cases s should be calculated modulo the order of the generator, and r must be mapped to an integer. There are no special requirements for the mapping, except that its image should be large and only a small number of inputs should produce the same output [Was12].

Since $r = g^k \pmod{p}$, we can use the DLK to implement a SETUP attack that leaks the signer's private key in two signatures. The first signature exposes $g^{k_1} \pmod{p}$, which leaks k_2. Once the attacker obtains m_2, r_2 and s_2 he/she can easily calculate the user's private key x. Naively one might assume that this is a strong SETUP, but, as the signer, who is in possession of the private key, can recover the choices of k, it is not. However, it turns out that, under the assumption that H is a pseudorandom function with secret seed, it is a regular SETUP. Here the secret constants a and b serve as extra protection in case H turns out to be invertible, as they prevent the user from calculating z. The reason for the SETUP not qualifying as strong if the simple fact that if the device is reverse engineered, then H, a, b and the secret seed will be exposed. This allows the user to calculate k_2 and detect the SETUP mechanism.

2.4 SETUPing DSA

The DSA was proposed for use in the Digital Signature Standard (DSS) in 1991 an was adopted as Federal Information Processing Standard (FIPS) 186 in 1993. Nowadays it is, together with its elliptic curve variant, one of the most widely used signature schemes.

2.4.1 The Digital Signature Algorithm

Before we present some of the different ways to abuse the DSA, we quickly review the algorithm. The specifications can be found in FIPS 186 [KG13]. Keys are generated as follows:

1. Choose a cryptographic hash function H.
2. Choose an N-bit prime q and an L-bit prime p such that $p-1$ is a multiple of q.
3. Choose $g \in \mathbb{Z}_p^*$ such that $\text{ord}(g) = q$.
4. Choose $x \in \mathbb{Z}_q^*$ randomly and calculate $y = g^x \pmod{p}$.
5. The public key is (p, q, g, y) while x is the private key.

The parameters p, q and g may be shared by different users. To sign a message m one does the following:

1. Choose a random $k \in \mathbb{Z}_q^*$.
2. Calculate $r = \left(g^k \mod p\right) \mod q$
3. In the very unlikely event that $r = 0$, start again.
4. Calculate $s = k^{-1}(H(m) + xr) \pmod{q}$.
5. In the very unlikely event that $s = 0$, start again.
6. The pair (r, s) is the signature.

Given a message m and a signature (r, s) the validity of the signature can be checked with the following algorithm.

1. If $r \notin \mathbb{Z}_q^*$ or $s \notin \mathbb{Z}_q^*$, then the signature is rejected.
2. Calculate $u_1 = H(m)s^{-1} \pmod{q}$ and $u_2 = rs^{-1} \pmod{q}$.
3. Calculate $v = (g^{u_1} y^{u_2} \mod p) \mod q$.
4. If $v = r$, then the signature is valid. Otherwise the signature should be rejected.

This works since

$$\begin{aligned} g^{u_1} y^{u_2} &= g^{H(m)s^{-1}} y^{rs^{-1}} = g^{(H(m)+xr)s^{-1}} \\ &= g^{(H(m)+xr)k(H(m)+xr)^{-1}} \equiv g^k \pmod{p}. \end{aligned}$$

The last congruence follows from the fact that g has order q.

The DSA can easily be modified to work with elliptic curves and is then called the Elliptic Curve Digital Signature Algorithm (ECDSA). Let E be an elliptic curve defined over $\mathbb{F}_p$ such that $\#E(\mathbb{F}_p) = eq$, for some small integer e and large prime q. Furthermore, let G be a point of order q and $Y = xG$ the public key. The signing operation is similar to the standard algorithm, just replace r with the x-coordinate of $R = kG$.

2.4.2 How to SETUP the DSA

As noted in Section 2.3.1, one obvious, but very inefficient, way to SETUP the DSA is to use the DLK when generating keys. Fortunately, for the attacker, there are also other viable SETUPs. We notice that $g^k \pmod{p}$ is recovered during the signature verification process. This means that we can leak the second choice of k securely with the DLK. As the knowledge of k compromises the user's private key x, we have created a SETUP mechanism that leaks the user's private key over two signatures.

2.4.3 Information leakage through the DSA

In this section, we show how one can send an N-bit message hidden in two DSA signatures. We will do this in the "Prisoners' Problem" setting, *i.e.*, Alice and Bob are both in prison and their only means of communication is by plain text messages through the warden. They are however allowed to authenticate their messages. In this setting, it is possible for Alica and Bob to communicate with subliminal messages hidden inside DSA signatures. Assume that Alice has shared her private DSA key with Bob. Now she can send a subliminal message $m < q$ to Bob by setting $k = m$. Without the knowledge of Alice's private key k the message cannot be obtained, and hence the warden cannot read the message. For further information about the scenario and the subliminal channel the reader can consult [Sim84] and [Sim94] respectively.

To be able to use this channel Alice and Bob have to know each other's private keys. In addition, the subliminal channel has two notable weaknesses. The first is that sending the same message twice will compromise the signer's private key and therefore the message. To prevent this, randomness has to be introduced, which in turn lowers the effective bandwidth. The second weakness is that the subliminal channel is susceptible to guessed plaintext attacks: the warden can try to guess m and easily check if the guesses are correct by computing g^k himself. Furthermore, a correct guess compromises the sender's private key. By employing the DLK we can overcome these weaknesses.

Alice can send Bob an N-bit secret message m ($m < q$) as follows:

- She chooses two messages M_1 and M_2 to be signed. It does not matter if the messages are real or bogus.
- Let w be the smallest value such that $H(M_1)^w \pmod p$ generates $\mathbb{Z}_p^*$ and let
$$g_1 = (H(M_1)^w)^{(p-1)/q} \pmod p.$$
Then g_1 has order q in $\mathbb{Z}_p^*$.
- Let $k_1 = (m(g_1^x \mod p)) \mod q$ and calculate k_2 from k_1 using the DLK. Here x is Alice's private key.
- Now she computes the signatures (r_1, s_1) and (r_2, s_2) as usual, but uses k_1 and k_2 as the secret exponents.

When Bob has received the two messages and their signatures, he first recovers Alice's private key x using the regular DSA SETUP. After this he calculates

$$k_1 = s_1^{-1}\left(H(M_1) + xr_1\right) \pmod q$$

and g_1 in the same way as Alice. Finally he recovers m as

$$m = k_1\left((g_1^x \mod p) \mod q\right)^{-1} \mod q.$$

In the above scheme Alice's private key x, and thus $g_1^x \pmod p$, is the shared secret that is used to hide the subliminal message. Note that the only purpose of the second message is to leak Alice's private key securely to Bob. After Bob has recovered Alice's private key it is thus possible for Alice to send one N-bit message per signature.

The value k_1 depends on the signed message M_1 and hence Alice can send the same subliminal message several times without compromising the security of the channel. Furthermore the scheme is not susceptible to guessed plaintext attacks, since, the warden would need to guess both the message and Alice's private key in order to succeed.

2.4.4 Using the DSA as a public key cryptosystem

In addition to the already mentioned attacks, it is possible to create a public key cryptosystem by abusing the DSA. We recall that the value $g^k \pmod p$ is recovered during the signature verification process, and hence it is possible to hide an DHKE inside a signature. This is done as follows. Let y be Bob's public key. Alice chooses k randomly and calculates $z = y^k \pmod p$. Now she encrypts her message m with a symmetric cipher using z, or something derived from z, as the key and signs the ciphertext with k as the secret exponent. Finally she sends Bob the ciphertext and the corresponding signature.

From the signature Bob is able to recover $g^k \pmod p$, and by raising it to his private key he obtains z. Hence he is able to decrypt the ciphertext and obtain m. This shows that we are able to use DSA signatures as key exchange messages. We say that we are *abusing* the DSA since, in response to criticism that the DSA does not provide secret key distribution, [SB92] stated: "The DSA does not provide for secret key distribution because DSA is not intended for secret key distribution".

2.4.5 The ECDSA analogs

The SETUP in Section 2.4.2 and the scheme presented in Section 2.4.4 can trivially be used with the ECDSA. In contrast, the information leakage scheme presented in Section 2.4.3 cannot be used as such with the ECDSA.

Let m, M_1, M_2, x be as in Section 2.4.3 and let G be the generator of the elliptic curve group that we are using with the ECDSA. We need to map $H(M_1)$ to a point $P \in \langle G \rangle$ with order $q = \text{ord}(G)$. If we choose H such that H has range $\mathbb{Z}_q^*$, then we can set $G_1 = H(M_1) \cdot G$, and $k_1 = m \cdot \hat{x}(xG_1) \pmod q$, where $\hat{x}(P)$ returns the x-coordinate of the elliptic curve point P. With these modifications to the scheme described in Section 2.4.3, it is possible to leak arbitrary information in ECDSA signatures.

2.5 The DLK and other protocols

Young and Yung show that the Schnorr Digital Signature scheme has a regular SETUP [YY97b]. They also note, but do not show, that the Menezes-Vanstone PKCS has a regular SETUP. These protocols are rarely used, and therefore we choose to omit the details.

2.6 Using the DHKEs as a hybrid system

Let us assume the following scenario. Let A be an arbitrary secure communication protocol that uses the DHKE for key exchange and transmits random numbers in plaintext during the handshake. In this setting it is possible to use

the random numbers as a subliminal channel in an undetectable fashion. Let G be a DH group with a generator g, and let $y = g^x$ be the attacker's public DH value. Let m be the message we want to leak, and for simplicity assume that the length of m in bits is equal to the size of the random number. To securely leak m, do the following. Choose k randomly and compute the public DH value g^k. Now compute $z = y^k$ and encrypt m with a symmetric cipher, using z, or something derived from z, as the key. Set the random number to the encryption of m.

Upon watching the handshake, the attacker recovers the g^k and the encryption of m. Using his/her private key x, the attacker computes $z = (g^k)^x$ and decrypts the encrypted message to recover m.

Let us now consider the security of this scheme. To obtain z from g^k, x is required. Hence, no one except the attacker can recover z. This also holds in the case the device is reverse-engineered. The reverse-engineer will recover y, but as the choices of k are not revealed, z cannot be obtained. Therefore, no one except the attacker can recover m. Furthermore, the encryption of m cannot be distinguished from random bits if a high quality symmetric cipher is used. Similarly, any device implementing this scheme cannot be distinguished from a legitimate implementation, since there is nothing special to be seen in the DHKEs. Hence, we have showed that this scheme is a strong SETUP.

An observant reader has probably noticed that this scheme is almost equivalent to the one where the DSA is used as a public key cryptosystem (Section 2.4.4). Indeed, if A includes DSA signatures instead of DHKEs, then the same objective can be reached by using the DSA scheme. However, there is one notable difference: the user knows his/her own DSA private key and can hence recover the choices of k. After reverse-engineering the device, the user can thus recover z, and therefore m. This will possibly also allow the user to distinguish between a contaminated and legitimate device; the user assumes that the device is contaminated and recovers the subliminal messages. If the messages are something else then seeds or other random looking messages, the user will be able to draw the conclusion that the messages that make sense come from the contaminated device.

The observant reader has probably noticed that this scheme could be considered a hybrid system based on ElGamal. Such systems were briefly considered in Section 2.3.1, and the conclusion was that they are easy to SETUP. However, we consider the situation here to be different to such an extent that a separate treatment is justified. We do not SETUP an existing hybrid system. Instead, we turn the random numbers that are transmitted as part of the protocol A into a subliminal channel by utilizing a hybrid system.

2.7 Other kleptograms

2.7.1 RSA public key encryption

RSA is one of the first practical public key cryptosystems. It was first publicly described by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. The English mathematician Clifford Cocks had developed an equivalent system independently in 1973, but it was not declassified until 1997.

Let $n = pq$, where p and q are primes, be the public modulus. Both the

plaintexts and ciphertexts are elements of $\mathbb{Z}_n$. The keyspace is

$$\mathcal{K} = \{(n, p, q, a, b) \mid ab \equiv 1 \pmod{\varphi(n)}\},$$

where φ is Euler's totient function. For $K = (n, p, q, a, b)$ define the encryption function

$$e_K(x) = x^b \mod n$$

and decryption function

$$d_K(y) = y^a \mod n.$$

Here (n, b) is the public key and (p, q, a) is the private key. Technically, any one of p, q and a suffices for the private key, as knowledge of one makes it possible to deduce the others. If either one of p or q is known, then n can trivially be factored, after which it is straightforward to acquire a. If a is known, then n can be factored in polynomial time by means of a randomized algorithm [Sti05]. It is also possible to speed up the decryption by precomputing values derived from p, q and a. For these reasons, different sources may give different definitions of the public key.

Before we continue, we show that the encryption and decryption function are inverses. To accomplish this we need two theorems, which we will present. However, we omit the proofs as they are standard and not relevant for our purposes. The proofs can be found in [Ste11], [Sti05] and most introductory texts on number theory.

Theorem 2.5 (Euler's Theorem)**.** If $\gcd(x, n) = 1$, then

$$x^{\varphi(n)} \equiv 1 \pmod{n}.$$

Theorem 2.6 (Chinese Reminder Theorem)**.** Let $a_1, \ldots, a_r \in \mathbb{Z}$, and suppose that $m_1, \ldots, m_r$ are pairwise coprime positive integers. Then the system of congruences

$$x \equiv a_i \pmod{m_i}, \quad 1 \leq i \leq r,$$

has a unique solution modulo $m = m_1 \cdots m_r$.

Let $\gcd(x, n) = 1$. We have $ab = \varphi(n) + 1$ for some $s \in \mathbb{Z}_+$, and hence, by Euler's Theorem,

$$(x^b)^a = (x^a)^b = x^{ab} = \left(x^{\varphi(n)}\right)^s x \equiv 1^s \cdot x \equiv x \pmod{n}.$$

If $\gcd(x, n) \neq 1$, then either $x \equiv 0 \pmod{p}$ or $x \equiv 0 \pmod{q}$. Without loss of generality, we can assume that $x \equiv 0 \pmod{p}$. Now $x^{ab} \equiv 0 \equiv x \pmod{p}$, and

$$x^{ab} = x^{s\varphi(n)+1} = \left(x^{s(p-1)}\right)^{q-1} x \equiv x \pmod{q},$$

where the last step follows from Euler's Theorem. By the Chinese Reminder Theorem, this system of congruences has a unique solution modulo n. It is obvious that the solution must be x, since $x \in \mathbb{Z}_n$ by assumption. This shows that the encryption and decryption functions are inverses.

Breaking RSA encryption, *i.e.* given a public key (n, b) and a ciphertext c, find m such that $m^b \equiv c \pmod{n}$, is known as the RSA Problem. It is clear that if an adversary can factor n, then he/she can trivially compute

$$\varphi(n) = (p-1)(q-1)$$

and then $a = b^{-1} \pmod{\varphi(n)}$ with the extended Euclidean algorithm. This allows the adversary to break the encryption. However, it is not known whether breaking RSA is as hard as factoring n.

Currently the most promising way of solving the RSA problem is to factor the modulus n. Hence the parameters must be chosen so that n cannot be factored by an adversary. In practice this means that n must be large enough, and the primes p and q should be of equal size. There are also other considerations, but as further details are irrelevant for our purposes, we omit them. Another security consideration is that it is insecure to use RSA without padding the messages that are to be encrypted. The interested reader can find plenty of material, *e.g.* [Sti05], covering these topic in the literature.

2.7.2 SETUPing RSA

There is only one viable option for SETUPing RSA public key encryption, namely leaking information about the private key in the public key. We will present two different methods. The first one is a proof of concept and not really useful in practice, while the second one is practical.

A simple way to SETUP RSA keys is presented in [YY96]. It is more of an example and not the main result of the paper. We choose to include it as an enlightening example. Let (N, B) and A be the attacker's public respective private RSA key. The following steps are needed to generate a new kleptographic RSA key.

1. Generate two suitable primes p and q.
2. Set $n = pq$ and $b = p^B \pmod{N}$.
3. If $\gcd(b, \varphi(n)) \neq 1$, then choose a new p and go to step 2.
4. Compute a the usual way.

To recover the private key a, the attacker simply looks up the public key, decrypts b with his/her own private key, and finally computes a as the multiplicative inverse of b the usual way. The last step is possible as the knowledge of p allows the attacker to easily factor n. Note that while the coprimality requirement in step 3 is not stated explicitly in the description of the RSA cryptosystem, it is implicitly imposed, as a cannot have a multiplicative inverse modulo $\varphi(n)$ otherwise.

The problem with the above method is that, in many of the implementations of RSA, the accepted range of values for b is limited. The reason for limiting b is efficiency; a b with short bit length and low hamming weight gives the most efficient encryption. Thus, RSA keys generated this way are almost useless in practice.

The other kleptographic key generation method we present is again due to Moti Yung and Adam Young [Yun04]. Let N, A, B be as previously with the

additional assumption that N is k-bit. Let $H : \mathbb{Z}_2^k \to \mathbb{Z}_2^k$ be a cryptographic one-way function. To generate a new $2k$-bit key one does the following:

1. Choose the public exponent b or go to step 2 if it is given.
2. Choose a k-bit value s randomly.
3. Compute $p = H(s)$. Start over if the most significant bit of p is not one, p is composite or $\gcd(p-1, b) \neq 1$.
4. Choose a k-bit value t randomly.
5. Compute $c = s^B \pmod{N}$, *i.e.*, RSA encrypt s with (N, B).
6. Solve for (q, r) in $(c \parallel t) = pq + r$.
7. Start over if q is composite or $\gcd(q-1, b) \neq 1$.
8. Output the public key $(n = pq, b)$ and the private key.

To recover the private key the attacker does the following:

1. Looks up (n, b) and sets u to the k uppermost bits of n.
2. Sets $c_1 = u$ and $c_2 = u + 1$.
3. Decrypts c_1 and c_2 to obtain s_1 respective s_2.
4. Either $p_1 = H(s_1)$ or $p_2 = H(s_2)$ will divide n.

Why does this work? Clearly $n = (c \parallel t) - r$, and $r < p$. Hence the only change the subtraction can do to the k uppermost bits of $(c \parallel t)$ is to "steal" a carry bit. Therefore either c_1 or c_2 will be equal to c. By decrypting and then hashing c_1 and c_2, we obtain two candidates for p, and the one that divides n must be the correct one. Note that one must be in possession of the private key A in order to be able to decrypt c_1 and c_2. Thus, the attacker is the only one that can perform these steps.

This scheme is only secure for the attacker if k is large enough and therefore not suitable for generating RSA keys with small modulus. If we want to generate 1024-bit keys, then, the attacker's own RSA key must be 512-bits, which is definitely insecure. Thus this method should not be used to generate keys shorter than 2048-bits if the security of the backdoor is to be preserved. This is the absolute minimum; nowadays many sources consider 1024-bit keys insecure. It is of course feasible to generate RSA keys of arbitrary length in this way, but if the attacker's key is small enough, then the backdoor can be exploited by any one that learns about its existence.

2.8 Conclusion

In this chapter we have seen that all of the most used public key cryptosystems, key exchange methods and signature methods are susceptible to kleptographic attacks. Furthermore, all the cryptographic primitives, except RSA, can be SETUPed using the same basic cryptographic primitive, namely the DLK.

While it is interesting to study how basic cryptographic primitives can be SETUPed, we are more interested in how these kleptographic methods can be used to compromise the security of the secure communication protocols that are used to secure traffic on the Internet. Due to the quirks of many of these protocols, it is not possible to directly make conclusions like: this protocol uses the DHKE and can hence be SETUPed efficiently using the DLK. In Chapter 3 we take an in depth look at how usable these SETUPs are in practice.

Chapter 3

Using the kleptograms

In this chapter, we show how the SETUP mechanisms presented in Chapter 2 can be used to compromise the security of the SSH and TLS/SSL protocols. Some of the attacks are due to Filip Zagórski *et al.* – see [GKZ06] – and others are our own work.

3.1 Secure Shell

SSH is a cryptographic network protocol for securing data communication. It works in a client-server architecture, and establishes a secure channel (over an insecure network) between an SSH-client and an SSH-server. One of the most common usages of SSH is remote login to shell accounts on Unix-like operating systems. It can, however, be used to secure any communication by tunneling through the secure channel established between the SSH-server and client.

3.1.1 The protocol

There exists two major versions of the SSH protocol, SSH-1 and SSH-2. The older version is now obsolete so we only consider version 2. SSH consists of three major components: The Transport Layer Protocol, the User Authentication Protocol and the Connection Protocol. These are documented in [YL06c], [YL06b] and [YL06d] respectively. In addition, the overall architecture of SSH is documented in [YL06a].

The Transport Layer Protocol provides server authentication, confidentiality and integrity. The User Authentication Protocol runs on top of the Transport Layer Protocol and authenticates the client to the server. The Connection Protocol runs over the User Authentication Protocol and multiplexes the encrypted connection into logical channels. The Connection Protocol is not relevant for our upcoming discussion and hence we only describe the Transport Layer Protocol and User Authentication Protocol.

3.1.1.1 The Transport Layer Protocol

The Transport Layer provides a secure channel over an insecure network. More specifically, it provides server authentication, key exchange, encryption and in-

tegrity. Furthermore it derives a unique session identifier (ID) that can be used by the higher lever protocols.

The server authentication is performed during the key exchange, and the result of the key exchange is used to derive all keys used for encryption and integrity protection. Furthermore, the unique session ID is obtained as a byproduct of the key exchange. Performing a DHKE in one of two predefined groups is the only key exchange method in the original standard. However, other alternatives have been added afterwards. [FPS06] expands the choice to DHKE in $\mathbb{Z}_p^*$, while a method based on RSA is documented in [Har06]. Furthermore, the integration of elliptic curve based key exchange methods is specified in [SG09].

To establish a SSH session the client first establishes a Transmission Control Protocol (TCP) connection to the server.[1] After this, both parties exchange identification strings. The exact format of this string is not relevant, but it contains information about the protocol version and software in use. The next step is to initialize the key exchange. To do so both parties exchange `SSH_MSG_KEXINIT` messages. The information in these messages is used to agree upon the key exchange method and all other cryptographic primitives used to secure the insecure channel. At this point, the actual key exchange can take place. The details of the key exchange are covered in Section 3.1.1.1.1. When the key exchange is complete, both parties exchange the `SSH_MSG_NEWKEYS` message, which is used to indicate that the key exchange was successful, and after this everything will be encrypted using the agreed upon keys and algorithms. This process is shown in Figure 3.1.

3.1.1.1.1 Key Exchange As mentioned, there are many different key exchange methods, which, however, are quite similar. We will describe the standard key exchange specified in [YL06c]. The standard DHKE is combined with a signature with the host key to provide server authentication. Let p be a large *safe prime*, g a generator for a subgroup G of $\mathbb{Z}_p^*$, and let q be the order of G. Let v_c and v_s be the identification strings of the client and the server respective. Let i_c and i_s be the `SSH_MSG_KEXINIT` messages of the client and the server respective. Furthermore, let k_s be the server's public host key, and let H be the hash function that was agreed upon with the `SSH_MSG_KEXINIT` messages. The following steps are used to exchange a key:

1. The client generates a random number x such that $1 < x < q$ and computes $e = g^x$. Then the clients sends e to the server.

2. The server generates a random number y such that $1 < y < q$ and computes $f = g^y$, $k = e^y = g^{xy}$, $h = H(v_c \parallel v_s \parallel i_c \parallel i_s \parallel k_s \parallel e \parallel f \parallel k)$ and a signature s on h. The server sends $(k_s \parallel f \parallel s)$ to the client.

3. The client verifies that k_s is the server's host key. This can, for instance, be accomplished by using certificates or a local database. The client is also allowed to ignore this step. This is, however, not recommended as this makes the protocol susceptible to active attacks (*e.g.* Man-in-the-Middle attacks). After this, the client computes $k = f^x = g^{yx}$, $h = H(v_c \parallel v_s \parallel i_c \parallel i_s \parallel k_s \parallel e \parallel f \parallel k)$ and verifies the signature s on h.

[1] The SSH protocol works over any clean, binary-transparent transport that protects against transmission errors. However, in practice TCP is almost always used.

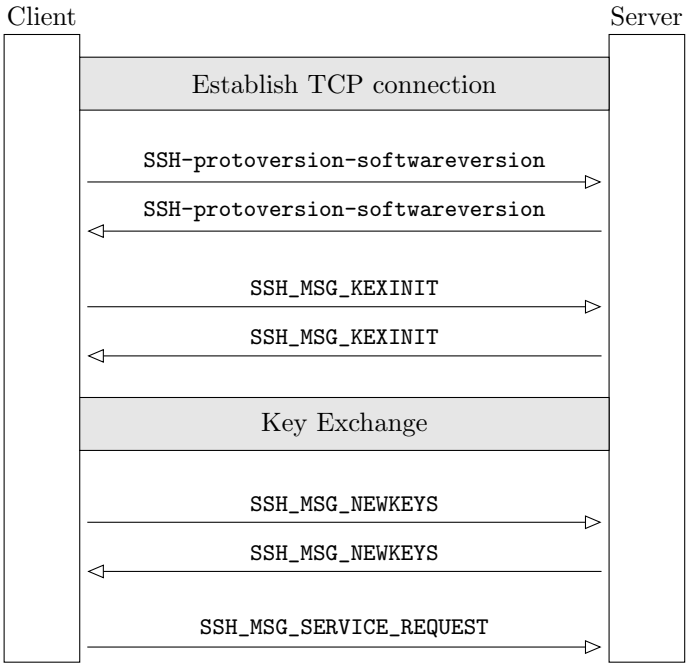

Figure 3.1: Transport Layer

The key exchange will fail if $e \notin \mathbb{Z}_p^*$ or $f \notin \mathbb{Z}_p^*$. Since this cannot happen if e and f are computed correctly, this check can been seen as an extra precaution. The value h is called the exchange hash, and is used to authenticate the key exchange. The exchange hash is to be kept secret.

The signing algorithm used was agreed upon with the `SSH_MSG_KEXINIT` messages and is one of the following: DSA, RSA or ECDSA. Note that, even though the signing algorithms hash the input during the signing operation, the signature must be applied on h and not the original data, *i.e.*,

$$(v_c \parallel v_s \parallel i_c \parallel i_s \parallel k_s \parallel e \parallel f \parallel k).$$

The overall structure of the key exchange is exactly the same when using the elliptic curve key exchanges defined in [SG09]. The only difference is that instead of performing a DHKE, an ECDH key exchange or an elliptic curve Menezes-Qu-Vanstone (ECMQV) key agreement is performed. The validity of the received public parameters is also checked in this case, although the algorithms that are used differ from those described earlier. When using elliptic curve based key exchange methods, ECDSA must be used for signing.

The message flow is similar when using DHKEs over $\mathbb{Z}_p^*$, with p some arbitrary safe prime, as specified in [FPS06]. However, the client and server must first agree on a suitable group. This is done as follows: the client proposes acceptable and preferred sizes of p, after which server replies with suitable parameters p and g. The key exchange then continues as in the normal case, with the small modification that all the extra parameters sent over the insecure channel are added to the data that is to be hashed.

3.1.1.1.2 Computing the session keys All the session keys are derived from k, h and the *session ID*. The session ID is equal to the first exchange hash, *i.e.*, h. This means that the session ID is not changed if the keys are re-exchanged at some point. Let id denote the session ID. The keys are computed as:

- Initial initialization vector (IV) client to server: $H(k \parallel h \parallel \text{“}A\text{”} \parallel id)$;
- Initial IV server to client: $H(k \parallel h \parallel \text{“}B\text{”} \parallel id)$;
- Encryption key client to server: $H(k \parallel h \parallel \text{“}C\text{”} \parallel id)$;
- Encryption key server to client: $H(k \parallel h \parallel \text{“}D\text{”} \parallel id)$;
- Integrity key client to server: $H(k \parallel h \parallel \text{“}E\text{”} \parallel id)$;
- Integrity key server to client: $H(k \parallel h \parallel \text{“}F\text{”} \parallel id)$.

The key material is taken from the beginning of the hashes. If the output of H is shorter than the required key, then the key is extended by hashing the concatenation of k and h and the entire key so far, and appending the result to the key. This process is repeated until enough key material is available.

3.1.1.1.3 Key Re-Exchange The specification recommends that a key re-exchange is performed after one gigabyte of data has been transfered or after an hour of connection time, whichever occurs sooner. The re-exchange is initialized with a `SSH_MSG_KEXINIT` message and continues as a regular key exchange after that (using whatever key exchange method that was agreed upon this time), with the exception that the session ID is not changed. Note that the re-exchange is performed using whatever encryption was in effect when it was initialized. The new keys are applied once the `SSH_MSG_NEWKEYS` messages have been exchanged.

3.1.1.2 The User Authentication Protocol

The User Authentication Protocol provides a suite of mechanisms to authenticate the client to the server. These suites use the unique session ID and/or depend on the security provided by the Transport Layer Protocol.

The three authentication methods available are: `publickey`, `password` and `hostbased`. Each one of them can be used separately, but they can also be combined. While the only authentication method required by the SSH-2.0 standard is `publickey`, practically all available SSH servers and clients implement all three methods.

With the authentication method `publickey`, the possession of the private key corresponding to an accepted public key serves as authentication. The idea is that the clients sends a `SSH_MSG_USERAUTH_REQUEST` including the *username* of the user, the public key and a signature, created using the private key, over the content of the message. Upon receiving this message the server first checks that the public key is authorized for authentication of this user. If this is the case, then the server verifies the signature. The authentication is only successful if both of these requirements are met.

In contrast, the authentication method `password` is very simple: The client simply sends the username and password in plaintext to the server, which then

verifies the authenticity of these. The security of this method relies on the security provided by the encryption used in the Transport Layer Protocol. The use of this method is prohibited, if, for some reason, the Transport Layer Protocol does not use encryption (this is possible, but certainly not recommended).

The authentication method `hostbased` authenticates the user by checking the host and user that runs the client. The host is essentially authenticated by the `publickey` method (the authentication messages are not the same, but the principle is), requiring the possession of the host's private key. In other words, the client sends a signature created with the private host key to the server, which can then verify this signature. Once the identity of the host has been confirmed, authorization is performed based on the user names on the server and the client, and the client host name. Thus the server trusts the client host when it says that it has already authenticated the user. The rationale behind this authentication method is that is it convenient and secure in the sense that an attacker cannot abuse this authentication method without access to the client host.

3.1.2 Compromising SSH

Now we have a rudimentary understanding of the SSH protocol, which allows us to see how it can be compromised using the kleptographic techniques presented in Chapter 2. We recall that it is possible to SETUP DHKE, RSA, ECDH key exchange, ECMQV key agreement, DSA and ECDSA. In other words, we are able to setup every key exchange and signing method available in the SSH protocol. We will look at how to compromise the security of SSH by SETUPing each of these algorithms, and also review the advantages and disadvantages associated to SETUPing each algorithm. Throughout this section, we take on the role of the attacker and assume that we are able to deploy our SETUP on a SSH server of our choice. Furthermore, we assume that we are able to monitor all traffic to and from the server. We start by reviewing the attacks by Gołębiewski *et al.* and the continue with our own attacks.

3.1.2.1 The attacks by Gołębiewski *et al.*

The possibility of using the DLK to compromise the DHKEs of the SSH protocol was pointed out in [GKZ06]. However, the attack they proposed is different from ours. They modify the behaviour of the client so that it disconnects after sending $m_1 = g^{c_1}$ to the server. After that it starts the connection again and sends $m_2 = g^{c_2}$ in the second key exchange. Here c_2 is calculated from c_1 as described in Chapter 2. To be able do decrypt the SSH session, the attacker only needs to passively observe the communication.

They also noted that there are several other possibilities for kleptographic attacks:

- The `SSH_MSG_KEXINIT` contains 16 bytes of random cookies. The purpose of this random data is to prevent so called *replay attacks*. It does, however, open up the possibility of leaking sensitive data like private keys, seeds *etc.*. How to do this in an undetectable fashion is not obvious. One option is to use the techniques described in Section 3.1.2.6. Other approaches will be discussed in Chapter 6.

- The protocol defines a `SSH_MSG_IGNORE` message which every implementation must understand and ignore. The message may contain arbitrary data and can be sent at any time after the identification strings have been sent. This can obviously be abused. However, there exists at least one problem: unless the messages are sent before the connection is encrypted, they are of little use to an attacker – assuming the attacker is not using any other kleptographic techniques that allow him/her to decrypt the communication. Thus the attacker is confined to send these messages during the handshake/key exchange. These messages are noticeable to anyone that passively observes the traffic, and will therefore cast suspicion on the party that is sending them.

- Every SSH message must contain between 4 and 255 bytes of padding, which must be random. Similarly to the `SSH_MSG_IGNORE` messages, this is hard to abuse once the connection is encrypted. It does, however, allow an attacker to leak sensitive data during the handshake/key exchange. More precisely, depending on the key exchange method, the server and client send three or more, messages each during this initial unencrypted phase. Hence there is lot of potential to leak sensitive information in the random padding. As with using the random cookies to leak information, the question that remains is: how can one leak information securely within data that must appear random?

- It is possible to encode a few bytes of information in the list of algorithms contained in the `SSH_MSG_KEXINIT` message. The authors claim that this is possible due to the permutations of the algorithm names. No further details are given. While this seems feasible, it is not very useful, at least when compared to the other methods presented earlier.

Note that while all of these methods were mentioned in the original paper, it contained no discussion on how to exploit them in an undetectable fashion. If this is not possible, then the attacks are not, in a certain sense, real kleptographic attacks.

3.1.2.2 SETUPing the DHKEs

Let us first consider the case when the DHKE is constantly performed using the same fixed group. The most obvious approach for creating a SETUP would be to apply the DLK in the $(m, m+1)$-leakage scheme directly to the DHKEs. However, the possibility of key re-exchange forces us to make a small modification. To see this, consider the case when the first key exchange is used to establish a session that is re-keyed before a new session is opened. In this case, we obtain m_1, but are unable to obtain m_2, since it is protected by the encrypted SSH tunnel. Since the kleptogram is designed in such a way that, to obtain the n-th private exponent, one needs to obtain both m_{n-1} and m_n, we need to be able to observe two successive key exchanges. Therefore we modify the scheme such that it can be considered a $(m, m+1)$-leakage scheme with respect to the number of SSH sessions. Our SETUP is as follows:

- The private exponents for all the key exchanges for the first SSH session are chosen randomly. The first key exchange of the session is unencrypted, and thus we use it to leak information that will compromise the following

sessions. The private exponent of the first key exchange, c_1, is stored in memory.

- When a second SSH session is started, the exponent of the first key exchange, c_2, is calculated as in the DLK. This compromises the key exchange.
- The exponents for all of the key exchanges in the second and $m-1$ subsequent sessions are computed using the $(l, l+1)$-leakage scheme.
- Start over when session number $m+2$ is started.

It is obvious that these leaks allow us to decrypt all but the first of $m+1$ SSH sessions.

In the general case when a number of groups are used for the DHKEs it is not possible to use this approach directly. Nevertheless, we can use the same approach for each of the groups known to the server separately. All it requires is a bit more memory and keeping track of all the added variables.

3.1.2.3 SETUPing the ECDH and ECMQV key exchanges

The ECDH key exchanges can be SETUPed in the same way as the normal DHKE, effectively creating a scheme that leaks m of $m+1$ SSH sessions. As noted in Chapter 2, the ECMQV key agreement can also be SETUPed. However, the ECMQV key agreement is hardly used, at least in this context, and hence this is mostly a curiosity that is of little practical use.

3.1.2.4 SETUPing the RSA key exchanges and signatures

Recall that it is possible to generate RSA keys in a way that hides information about the private key in the public key. Whenever RSA is used for key exchange, the keys should be temporary ones. Hence the keys must be generated by the server. This makes it possible to generate the keys as described in Chapter 2, and hence the key exchanges will be compromised.

The private RSA host key that is used to authenticate the server is static. If the key is generated during the installation process, then it is possible to generate it with the same process that can be used to compromise the keys used for the key exchange. In this case the attacker only needs to obtain the server's public key to be able to launch a successful *Man-in-the-Middle* attack.

3.1.2.5 Abusing the signatures

Recall that the only accepted signing methods, in addition to RSA, are DSA and ECDSA. As shown in Chapter 2, it is possible leak the server's private DSA key over two DSA signatures. In principle this means that we can recover the server's private key after observing two SSH sessions where DSA was used as the signature algorithm. Similarly, it is possible to leak an ECDSA private key over two ECDSA signatures. Therefore it is possible to efficiently leak all the private keys of the server. An attacker that has access to the server's private keys can impersonate the server, and thus launch a successful *Man-in-the-Middle* attack.

Another option is to leak sensitive information through the subliminal channel in DSA presented in Section 2.4.3. In addition to the possibility of sending

arbitrary messages, this SETUP compromises the servers public key. This channel can be used to leak the seed, or part of it, used to derive the private values used in the key exchanges. This enables passive attacks as well as active attacks.

3.1.2.6 Using the random numbers as a subliminal channel

The 16-byte random numbers in the `SSH_MSG_KEXINIT` messages can be used as a secure subliminal channel by utilizing the schemes described in Sections 2.4.4 and 2.6. These can be used to leak any sensitive information in the 16 byte random number. The idea is to encrypt the message with a symmetric cipher and set the random bytes to the ciphertext. As mentioned, these SETUPs is undetectable, since the ciphertexts produced by any high quality block or stream cipher cannot be distinguished from the output of a high quality PRNG. The most efficient way to use abuse this SETUP is probably to leak the seed or state of the underlying PRNG or some other sensitive information that compromises the key exchanges.

Clearly, this SETUP can be exploited by a passive attacker. However, the subliminal channel can only be used when DH is used for the key exchange or the (EC)DSA is used for signatures. Let us first consider the case when DSA is used. The group G used in the DSA is fixed with respect to the public and private key. Therefore, the G is always know at the time the 16 byte random number is to be generated. The only parameter that is not known, and needed to generate z, is k. However, k can be selected before the message is to be encrypted, and stored in memory until it is needed for the signing operation. Hence, the implementation of the SETUP can be designed so that each signature compromises the corresponding handshake.

The same applies when DHKEs are used, with one exception: when arbitrary DH groups are used, then the DH group G will not be known when the 16 byte random number is to be generated. Thus we have a synchronisation problem. Whenever one of the standardized DH groups are used, this is not a problem, and the implementation of the SETUP can be designed so that each DHKE compromises the corresponding handshake. With the arbitrary DH groups this is not possible. Nevertheless, the following random can be SETUPed by the previous handshakes. Another possibility is to leak the state of the PRNG that is used to generate all secret values. Thereafter, there is no need to SETUP more handshakes (until a reseed), as knowing the seed will compromise all the upcoming key exchanges.

Note that, if the device is reverse-engineered, then the secret keys (z) shared with DSA will be compromised, while the secret keys shared with the DHKEs are unaffected. For further details, see Section 2.6.

3.1.2.7 Consequences

The security of the SSH protocol relies on the server authentication and key exchanges. Any adversary that is able to recover the keys agreed upon during the key exchanges is able to decrypt the corresponding traffic. This only requires passive monitoring of the traffic to and from the server. In addition, it is possible to store all traffic to and from the compromised servers for later decryption as needed.

Any adversary that has access to the private keys of the server can, given enough resources, impersonate the server and thus launch a successful *Man-in-the-Middle* attack. This allows the attacker to read any traffic; not just the traffic from the compromised key exchanges. This, however, requires an active attack.

Whenever the encryption of the SSH tunnel is compromised, the credentials of any user that uses the `password` authentication method is compromised. The other authentication methods are not compromised.

3.2 TLS/SSL

TLS and its predecessor SSL are cryptographic protocols that are used to secure communication over computer networks. Similarly to SSH, it works in a client-server architecture. The protocols are used to secure the majority of all secure network connections.

3.2.1 The protocol

In this section we will review the parts of the TLS protocol that are relevant for our discussion, *i.e.*, the handshake and the derivation of the session specific master key, which is used to derive all the encryption keys needed for that session. The current standard is TLS version 1.2 [DR08].

3.2.1.1 The TLS handshake protocol

The role of the handshake is to agree on the cryptographic parameters that will be used to establish a secure connection, and optionally authenticate the server and the client. The full TLS handshake involves the following steps[2]:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption;
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret;
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves;
- Generate a master secret from the premaster secret and exchanged random values;
- Provide security parameters to the record layer;
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

The handshake begins when the client sends a `ClientHello` message to the server, which has to respond with a `ServerHello` message, or else the connection will fail. These messages establish the following attributes: Session Id, Protocol

[2]Excerpt from [DR08].

Version, Cipher Suite and Compression Method. In addition two 28-bit random numbers, `ClientHello.random` and `ServerHello.random`, are generated and exchanged.

After this, the server sends its Certificate, a `ServerKeyExchange` message, a `CertificateRequest` message, and finally the `ServerHelloDone` to indicate that hello-message part of the handshake is complete. All these messages except the last one are optional or situation-dependent and are not always sent. Whenever the server is to be authenticated, it must send its certificate to the client. The `ServerKeyExchange` message is sent when required, *e.g.*, if the server has no certificate or if the certificate is for signing only. The `CertificateRequest` can be sent if the server is authenticated, and it means that the server requests the client to authenticate itself by sending its certificate.

The client responds by sending its certificate (if required), a `ClientKeyExchange` message, and a digitally signed `CertificateVerify` message in case a certificate has been sent and the certificate supports signing. The purpose of the `CertificateVerify` message is to verify to the server that the client is in possession of the private key associated to the certificate. The client finishes its part of the handshake by sending a `ChangeCipherSpec` and the `Finished` message. The server completes the handshake by sending its own `ChangeCipherSpec` and `Finished` messages. The `ChangeCipherSpec` message tells the other part that the sending part has switched to the agreed upon cipher and that everything after this will be encrypted with that cipher. The `Finished` messages are encrypted under the parameters chosen for that session. At this point, the handshake is complete and the transfer of application data can begin. The complete message flow is shown in Figure 3.2.

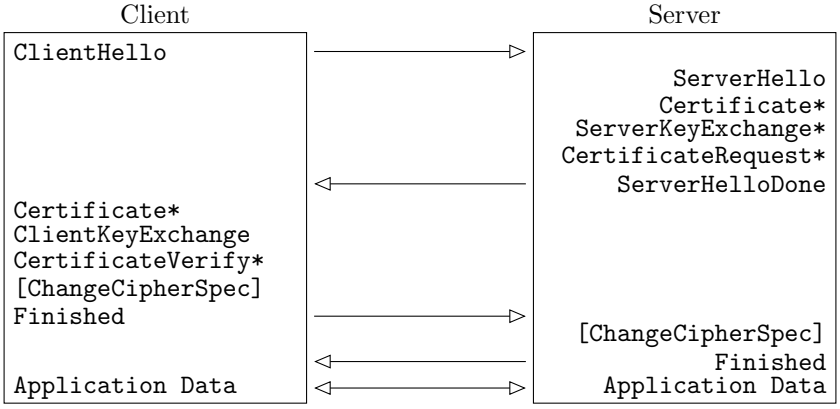

Figure 3.2: Message flow in a full TLS handshake.

The TLS protocol also gives the option to resume sessions. This reuses the state of the session – including all cryptographic parameters. The reuse of parameters saves one round trip to the server, and hence it is more efficient to resume sessions than to renegotiate a new session every time. When resuming a session, both the server and client proceed directly to the `ChangeCipherSpec` messages after the initial hello messages. The message flow when resuming a session is shown in Figure 3.3.

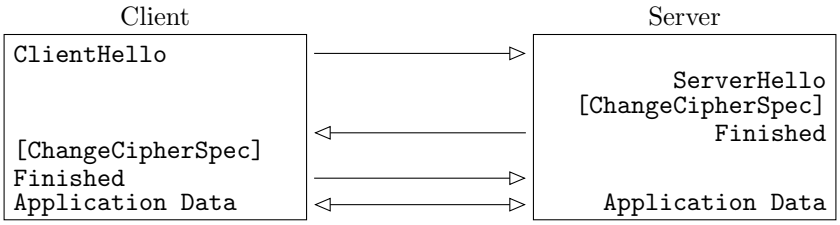

Figure 3.3: Message flow when resuming a TLS session.

3.2.1.2 The key exchange

There are essentially two possible key exchange methods: RSA public key encryption and DHKE. The key agreed upon during the key exchange is called the *premaster secret*. When using RSA, the basic idea is that the client generates the key, encrypts it with the servers's public key and sends it to the server. Details about this can be found in Section 3.2.1.3.1.

When using DHKE there are three options: fixed DHKE, *ephemeral* DHKE and anonymous DHKE. Fixed DHKE means that the public (and thus private) values used are fixed and embedded in the certificates of the server and client. This means that the premaster secret is the same every time. In contrast, ephemeral DHKE means that the private exponents used in the DHKE are randomly chosen for each key exchange. The server is responsible for selecting the group and its generator and sends them, along with the public value, to the client. This means that the server is free to choose a different group each time. However, finding a suitable group, along with a generator, is a very time-consuming process, and therefore servers can not generate new ones on the fly. Thus, servers are confined to use groups given to them. Most servers use a single group for each security level they support even though they, at least in principle, could choose randomly from a arbitrarily sized list of suitable groups. Anonymous DHKE is simply an ephemeral DHKE without any authentication. It is susceptible to *Man-in-the-Middle* attacks and is usually not recommended.

3.2.1.3 Computing the master secret

All the parameters needed to begin connection protection are agreed upon during the handshake, except the master secret. Nevertheless, the master secret can be derived from the premaster secret using the other parameters that were agreed upon. The algorithm used to convert the premaster secret into the master secret is always the same and independent of the key exchange method. The exact algorithm used is irrelevant for our purpose, and it is enough to know that the conversion function should be a *pseudorandom function* that takes the premaster secret and the random numbers generated by client and server as inputs.

Recall that the premaster secret is agreed upon with a key exchange method, and hence, assuming the key exchange is secure, an attacker will have no knowledge about it. Therefore there is no feasible way for an attacker to obtain the master key of a TLS/SSL connection.

3.2.1.3.1 RSA When RSA is used for server authentication and key exchange, a 48-bit premaster secret is generated by the client. The premaster secret is then encrypted with the server's public RSA key and sent to the server. Now both the client and server has access to the premaster secret, and can therefore convert it to the master secret.

3.2.1.3.2 DHKE When DHKE is used as the key exchange method, then the shared secret Z that is agreed upon with the DHKE is used as the premaster secret. Leading bits of Z that are zero are discarded before Z is converted to the master secret.

3.2.1.4 Protocol extensions

There are many extensions to the TLS protocol, version 1.2. The one that is relevant for our purposes is [Res08], which adds the possibility of using elliptic curve cryptography to the protocol. More precisely, it adds ECDH as a key exchange method and the ECDSA for signing. Thus the supported signature methods are the same as in the SSH protocol, namely RSA, DSA and ECDSA.

3.2.2 Compromising TLS

As is the case with SSH protocol, there are essentially two methods to compromise the security of the TLS protocol. The first one is to defeat the encryption, while the second approach is to impersonate the server in a *Man-in-the-Middle* attack. We will again begin the overview of the possible attacks by looking at the attacks by Gołębiewski *et al.*. Thereafter we will present our own attacks.

3.2.2.1 The attacks by Gołębiewski *et al.*

The idea presented in [GKZ06] is to use the `ClientHello.random` to leak a secret seed that is used to seed the generator that is used to generate the secrets used in the key exchanges. They have chosen to attack the client, but parts of the same attack could also be performed on the server.

Let G be a group that is suitable for DHKE, and g its generator. Furthermore, let $s = H(y, i)$, where H is a hash function, $y = g^x$ is the attackers public key and i a sequence number. The idea of the attack is:

1. The client generates a random k and computes $c = g^k$. Then c is leaked to the attacker through the `ClientHello.random` messages. Use as many messages as needed.

2. The client sets i to some predefined value and computes s.

3. The next step depends on whether RSA or DH is used for the key exchange. If RSA is used, then the client is responsible for choosing the premaster secret, so it can be derived deterministically from s. If DH is used, then the secret exponent used by the client is derived deterministically from s.

The original paper leaves a few things open: the exact usage over many sessions and how to change i. The two most obvious ways to exploit this over many sessions are:

- Leak c over the needed amount of sessions. For each session after this: increment i, recompute s, and derive the required secret value from s. In principle this can be continued for as long as the attacker wants.
- Start by leaking c. Then omit i and just set $s = H(y)$. Seed some predefined PRNG with s and generate all future secret values with this PRNG.

Both of these options allow the attacker to decrypt all sessions after the initial ones that are required to leak c. Another option is to regularly choose a new k and leak the corresponding c.

As was the case with the attacks on the SSH protocol, the original paper did not consider how undetectable this attack is. If c is leaked very seldom, then it is hard to distinguish the output of a SETUPed client and a regular one. If c is leaked often (not the same c), then it is theoretically possible to distinguish the SETUPed client from a normal one. To see this, consider the case where G is an elliptic curve. The normal representation of points on an elliptic curve is the bitstring describing its x-coordinate and, in cases where it matters, one bit representing the sign of the y-coordinate. The x-coordinates of all points on an elliptic curve do not cover the base field in which the coordinates are defined. In most cryptographic applications, the viable x-coordinates cover about half of the base field. Therefore any client who regularly outputs x-coordinates of elliptic curve points instead of random bits can be distinguished from one that outputs random bits.

The situation is similar when the DHKE is performed with modular arithmetic, *i.e.* G is a subgroup of $\mathbb{Z}_p^*$, since the prime ordered subgroup generated by g has order at most $(p-1)/2$. Given an arbitrary nontrivial subgroup G of $\mathbb{Z}_p^*$, there does not, at least to our knowledge, exist any efficient algorithm to determine if a given element $a \in \mathbb{Z}_p^*$ belongs to G (assuming p is large enough to make exhaustive search infeasible). Nevertheless, if p is a safe prime and G is the maximal prime ordered subgroup of $\mathbb{Z}_p^*$, then all elements in G are quadratic residues modulo p. Since it is possible to efficiently check whether any integer is a quadratic residue modulo p, frequent leakage of c will be noticeable. The value c can be represented with significantly fewer bytes when elliptic curves are used, and therefore usage of elliptic curves is probably preferable over classical DH groups. The exception is DH groups $G = \mathbb{Z}_p^*$, as their use cannot be discovered by looking for quadratic residues. The randomness of DHKE public values and how they can be distinguished from random bits, assuming the group used is known, will be discussed further in Chapter 5.

We end this section by noting that only the DHKEs can be compromised if the attack is implemented on the server side. This is a consequence of the fact that the client is responsible for choosing the premaster secret whenever RSA is used for the key exchange.

3.2.2.2 SETUPing the key exchanges

The ephemeral DHKEs and ECDH key exchanges can be SETUPed exactly like in the SSH protocol. This scheme enables the attacker to decrypt m of $m+1$ TLS sessions. Anonymous DHKEs key exchanges can also be compromised using the same technique. This is, however, nothing more than a curiosity, since anonymous DHKEs should not be used due to their lack of protection

against *Man-in-the-Middle* attacks. The fixed DHKEs can not be SETUPed efficiently with the DLK. The public values used are contained in certificates, and in theory it is possible to SETUP the generation of these. However, the exploitation of this SETUP is probably not so easy or practical.

The remaining key exchange method is RSA public key encryption. The keys used for these are fixed and a part of the server's certificate. Like the keys for the fixed DHKEs, the public RSA keys are contained in the server's certificates. The keys must be SETUPed when they are generated, which is most probably done in conjunction with the rest of the certificate. Hence, an attacker would probably have to SETUP the generation of certificates. Recall that, in this SETUP, information about the private key is hidden in the public key, which makes it easy to exploit this SETUP.

3.2.2.3 SETUPing the signatures

Similarly to the SSH protocol, the client authenticates the server by verifying the signature supplied by the server. In addition, the signing algorithms employed are the same. Hence it is possible to leak the server's private keys using exactly the same SETUPs as with the SSH protocol.

3.2.2.4 Using the random numbers as a subliminal channel

The attacks on the SSH protocol described in Section 3.1.2.6 also works with the TLS protocol. In fact, they work even better with the TLS protocol; the random numbers are larger, allowing greater bandwidth, and there is no problem with the synchronisation. Recall that the server is responsible for selecting the DH group. Furthermore, the server can always be implemented in such a way that the group is selected before the `ServerHello.random` is generated. Hence, it is possible to broadcast a 28-byte encrypted message, along with information that the attacker can use to decrypt it, during each handshake. It is clear that this attack can be implemented so that each handshake that uses a DHKE can compromised. Note that it does not work with fixed DHKE.

This attack can only be used to compromise handshakes relying on the DHKE, even if the DSA is used as a public key cryptosystem, since the client chooses the premaster secret whenever RSA is used as the key exchange method. Nevertheless, if the client is SETUPed similarly, then it is also possible to compromise handshakes relying on RSA for sharing the premaster secret.

3.3 The feasibility of the attacks

Let us first consider the attacks on the SSH protocol. The attacks we have described can be used to attack selected servers and clients, but they are more geared towards mass surveillance. This is also the reason we have chosen to attack the servers and not the clients; if one server has SSH software with the SETUPs we described, then it compromises the communication with all the clients, while a SETUPed client only compromises itself. There are essentially three scenarios for launching these attacks:

- Modify existing software.
- Create and distribute new software.

- Create malware that infects and modifies SSH server software.

Let us consider each of these in turn. Modifying existing software is easy in principle. The hard part is to get people to use the modified software. The most commonly used SSH server software in 2008 was OpenSSH.[3] Since OpenSSH is open source software, it is probably very difficult to get the project to accept kleptographic code. There does exist closed source implementations of the SSH protocol, but their use is marginal. So even though an intelligence organization would have the resources to somehow include kleptographic code into these implementations, it would only allow it to read a marginally small part of all SSH communications.

The problem with creating new software is the same as with existing closed source implementations, *i.e.* getting people to use them. As there exists a free high quality implementation, the public is probably reluctant to adopt a closed source implementation they have to pay for. Distributing a closed source implementation for free is also not an option, as this will raise suspicion.

Using malware to modify SSH server software is certainly possible. However, how easy it is to do it without being detected is the real question. We will not speculate about this, since is not our area of expertise However, it is fair to assume that an attacker that is able to pull off such an attack could as well steal the private keys of the server. This should be easier to do without being detected, which makes the idea of modifying the SSH server software obsolete.

The conclusion is that mounting these kleptographic attacks on the SSH protocol on a large scale is probably unfeasible. This is, however, not the case with the TLS protocol. For TLS software, the situation is similar to the aforementioned situation, which makes it a hard target for mass surveillance. Nevertheless, there is hope for the attackers, namely, so called SSL accelerators. A SSL accelerator is a hardware device that performs the TLS handshake, or in certain cases the whole protocol. These devices are used to unload servers. As dedicated hardware, these SSL accelerators can be considered as black boxes. The user of such a box can verify that the device's output conforms to the specifications, but there is no way to verify, at least for average users, that it does not do anything extra or something kleptographic. In other words, these SSL accelerators open up a huge possibility for an attacker with wast resources. It is certainly possible to get manufacturers of these devices to add kleptographic features to their products. These devices are used in most professional server environments, and hence this could compromise most of the network traffic protected by TLS. A potential problem for an attacker is that many of these devices are FIPS certified. Thus they are required to conform to additional specifications that are not imposed on them by the TLS protocol. Although this certainly complicates things for an attacker, it seems feasible that an attacker with resources to persuade manufacturers of SSL accelerators to include kleptographic features can find ways to overcome this inconvenience.

3.4 Conclusion

We have seen that the security of the SSH and TLS protocols can be effectively compromised with the kleptographic primitives presented in Chapter 2. Some

[3] `http://www.openssh.com/usage/`

of the attacks can be exploited by a passive attacker, while effective exploitation of others require *Man-in-the-Middle* attacks. The most efficient and secure way to SETUP the TLS protocol can be fully exploited by a passive attacker and compromises all sessions that rely on the DHKE. The same SETUP can be used to compromise the SSH protocol. Due to quirks of the protocol, it is not as straightforward. It is, however, possible.

In general, the TLS protocol is more vulnerable to kleptographic attacks than the SSH protocol. In addition, black-box implementations of the TLS protocol are used on a large scale, while black-box implementations of the SSH protocol are almost nonexistent. This is unfortunate, as the TLS protocol is the more widely used one.

Chapter 4

Dual_EC_DRBG

The Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG) was one of the four PRNGs standardized by the National Institute of Standards and Technology (NIST) [nis] in NIST SP 800-90A [BK+12]. It is also included in the ISO/IEC 18031 standard. Despite public criticism, it was included in the NIST standard and was not removed until April 2014. The removal was a result of the revelation that the NSA, who designed the PRNG, had, allegedly, inserted a backdoor into it.

4.1 How the generator works

Let E be an elliptic curve over a finite field, and let $P, Q \in E(\mathbb{F}_p)$. Let $\hat{x} : E(\mathbb{F}_p) \to \mathbb{F}_p$, $\hat{x}(P) = \hat{x}((x, y)) = x$. The generator is wrapped in a generate function that takes as input a state, denoted by s, the requested number of bits, and additional input, denoted by a. Disregarding any possible reseeding, the generate function works as follows: first, the additional input is XORed with the state. More precisely, $s_0 = s \oplus a$. Then enough output is generated to satisfy the request. The generation of output can be described by the equations:

$$\begin{aligned} s_i &= \hat{x}(s_{i-1}P), \\ r_i &= \hat{x}(s_i Q), \\ o_i &= h(r_i), \end{aligned}$$

where $h : \mathbb{F}_p \to \mathbb{Z}_2^n$ is a truncation function and the output blocks are denoted by o_i. The truncation function h works as follows: first, it converts the given field element to the bit representation of the corresponding unsigned integer. Then the k most significant bits are truncated to obtain the result. The value k depends on the elliptic curve and can in principle be chosen freely. However, the recommended values are 16, 16 and 17 for the respective curves P-256, P-384 and P-521, and any implementation that wishes to get a FIPS 140-2 validation must use these values. The last step of the generate function is to update the state one additional time, *i.e.*, $s = \hat{x}(s_j P)$, where j is the largest value of i used when generating the output. Thereafter the generate function outputs the requested bytes, *i.e.*, the concatenation of all the outputs o_i, and the newly updated state.

There are two versions of the Dual_EC_DRBG, one from 2006 and an updated one from 2007. What we have described so far is the 2007 version. The 2006 version omits the final state update in the generate function.

4.2 The flaws

In [Gjo06] Kristian Gjøsteen showed that the first part of the generator is essentially cryptographically sound, while the extraction of random bits from the generated elliptic curve point sequence creates a biased output. In addition, he constructed a simple bit predictor with advantage 0.0011. The existence of a bit predictor with such an advantage is unacceptable for any good quality PRNG.

4.3 The alleged backdoor

Let us consider the alleged backdoor. Assume that we know r_i for some i. Then we know s_iQ up to sign, since the sign simply changes the sign of the y-coordinate, and the x-coordinate identifies a point up to sign. If, in addition, we know e such that $P = eQ$, then we can calculate

$$s_{i+1} = \hat{x}(s_iP) = \hat{x}(s_i(eQ)) = \hat{x}(e(s_iQ)),$$

Since $\hat{x}(R) = \hat{x}(-R)$ and $-mR = m(-R)$ for all $R \in E(\mathbb{F}_p)$, we obtain the same result even if we use $-s_iQ$ instead of s_iQ. Under these assumptions we can hence predict all future output. The question that arises is whether our assumptions are reasonable.

Let us first consider the first assumption, *i.e.*, can we obtain r from the output of the generator? About half of the possibilities for x are viable x-coordinates of some point and it does not, as mentioned, matter if we choose R or $-R$. Assuming that the curve used is P-256, only 16 bits of the x-coordinate are truncated, and there is only about 2^{15} possibilities for $R = s_iQ$. Thus, it is possible to try every r by brute force and then verify the output that follows by comparing it to future output of the generator.

The second assumption is the more pressing one; the security of ellitic curve cryptography (ECC) in general, and the Dual_EC_DRBG in particular, depends on the hardness of the elliptic curve discrete logarithm problem (ECDLP), *i.e*, given an elliptic curve E, and $P, Q \in E$, find e such that $Q = eP$. Here we assume that P is the generator of E, or equivalently we can work in the subgroup generated by P, in which case Q must belong to this subgroup. If the points P and Q are chosen in an appropriate way, then, assuming that the ECDLP is hard, it is unfeasible to solve the ECDLP for P and Q. Hence, one can only use the backdoor if one knows the relationship between P and Q. In other words, the backdoor is an asymmetric backdoor with the secret key e.

Herein lies the problem; the constants P and Q given in [BK+12] are supplied by the NSA without any explanation of their origin. Furthermore, an implementation must use these constants to comply with the FIPS standard and be eligible for FIPS validation. This means that the majority of all implementations will use the potentially backdoored constants – even though one can generate new safe constants in a transparent way.

4.4 The conspiracy

This potential backdoor was independently recognized by members of the ISO standardization committee in 2004, who published their findings quietly in a patent application [BV07], and researchers in 2007 [SF07]. Despite this, and all the other apparent flaws of the Dual_EC_DRBG – it is inefficient and the output is biased – the generator was still included in NIST SP 800-90A. In 2013 whistleblower Edward Snowden leaked information about the NSA's BULLRUN program, which implies that the alleged backdoor is real. After this revelation, the standardization process and other related matters have been under scrutiny. Furthermore, there are three proposed, but non standardized, extensions of TLS, that makes the TLS protocol more vulnerable to the backdoor. All of these extensions have the side effect that more randomness is broadcasted in plaintext. Two of these were proposed by the United States Department of Defense.

Despite the known weaknesses Dual_EC_DRBG is implemented in many cryptographic libraries. It is, however, usually not enabled by default. The exception is RSA Security's BSAFE library where Dual_EC_DRBG is used as the default PRNG. In 2013 Reuters reported that the NSA had paid RSA Security to use Dual_EC_DRBG as the default PRNG [Men13]. We will not focus on these aspects. Nevertheless, the interested reader can find information about all of this in the references and on Wikipedia.

4.5 Exploiting the backdoor

In principle, it is possible to exploit the backdoor in any application that uses the Dual_EC_DRBG in such a way that enough of its output is exposed. This allows the attacker to recover the state of the generator. After this, whenever the generator is used to generate random numbers/bits that must be kept secret, the attacker will have access to these random numbers/bits. Hence, the security of any protocol where these secrets are used is compromised. In practice, the feasibility of these kinds of attacks depends on the protocols and the implementation in use. The relevant questions are, for instance: How much output is exposed? How much output is used internally by the protocol? How much output is simply discarded?

4.5.1 The practical exploitability of the Dual_EC_DRBG in TLS implementations

It has been suggested that the NSA has the capability do decrypt SSL/TLS connections that rely on the Dual_EC_DRBG for randomness. In 2014, Checkoway *et al.* published a paper [CFN+14] on the practical exploitability of the Dual_EC_DRBG in TLS implementations. The conclusion of the paper is that the attacks are indeed practical and can, with sufficient computational resources, be mounted on a large scale.

To test the practical exploitability Checkoway *et al.* chose four existing implementations of the TLS protocol with the Dual_EC_DRBG as default or as an option. These implementations are RSA BSAFE Share for C/C++ (henceforth BSAFE-C), RSA BSAFE Share for Java (henceforth BSAFE-Java), Windows

SChannel and OpenSSL. To experimentally verify the performance of their attacks, Checkoway *et al.* generated their own backdoored P and Q to replace the ones given by NIST. Getting the libraries to use the new constants required reverse-engineering of the relevant parts of the closed source libraries, *i.e.*, all except OpenSSL, and modification of the OpenSSL source code. The major findings during this were:

- A previously unknown bug in the OpenSSL library makes the library non functioning when Dual_EC_DRBG is enabled. The authors still considered it possible that someone is using OpenSSL with Dual_EC_DRBG since the bug has an obvious and easy fix [CFN+14]. Therefore they applied the fix and tested the fixed version, which they call OpenSSL-fixed. It also turned out that OpenSSL-fixed uses additional input to Dual_EC_DRBG in such a way that makes the attack significantly more expensive.
- The BSAFE implementations of TLS turned out to make exploitation of the backdoor easy. The BSAFE-C library broadcasts longer strings of consecutive random bits than seems possible at the first glance at the TLS protocol, and this drastically speeds up the attack. The Java version of the library does not broadcast these extra bits. However, the connections are fingerprinted, which makes it trivial to identify them.
- Windows SChannel's implementation does not conform with the current Dual_EC_DRBG standard as it omits an important computation. It does not, however, prevent the attack. Instead it makes it slightly faster.

The summary of the results obtained by Checkoway *et al.* is shown in Table 4.1. The version of SChannel tested is the one used in Windows 7 Service Pack 1

Library	Bytes per Session	Adin Entropy	Attack Complexity	Time (minutes)
BSAFE-C v1.1	31-60	—	$30 \cdot 2^{15}(C_v + C_f)$	0.04
BSAFE-Java v1.1	28	—	$2^{31}(C_v + 5C_f)$	63.96
SChannel I	28	—	$2^{31}(C_v + 4C_f)$	62.97
SChannel II	30	—	$2^{33}(C_v + C_f) + 2^{17}(5C_f)$	182.64
OpenSSL-fixed I	32	2^{20}	$2^{15}(C_v + 3C_f) + 2^{20}(2C_f)$	0.02
OpenSSL-fixed III	32	2^{35+k}	$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$	$2^k \cdot 83.32$

Table 4.1: Summary of the results in [CFN+14]

and Windows Server 2010 R2. OpenSSL-fixed stands for the repaired version of the OpenSSL FIPS Object Module version 2.0.5 in combination with OpenSSL 1.0.1e. Bytes per session indicates how many contiguous, useful bytes from Dual_EC_DRBG is reaveled during each session, and Adin Entropy how much entropy, in form of additional input, is added in each generate call. The Attack Complexity gives the complexity of the attack in terms of scalar multiplications of fixed (C_f) and variable (C_v) elliptic curve points. These two operations are separated, since, if the point that is to be multiplied is known beforehand, then the computation can be made significantly faster with the help of precomputed constants and lookup tables, while this optimization is not possible when multiplying arbitrary points. With the optimizations used by Checkoway *et al.*, C_f is approximately two orders of magnitude faster than C_v. The reported time is the measured worst case time for the attack on a four-node, quad socket AMD

Opteron 6276 cluster. The time for OpenSSL-fixed III was measured with $k = 0$. All tests were conducted using Dual_EC_DRBG over the NIST P-256 curve. We conclude this section by reviewing the details of each implementation and the corresponding attacks.

4.5.1.1 BSAFE-C

Version 1.1 of BSAFE-C does not support TLS version 1.2, and hence it does not support ECC for key exchanges and signing. A TLS server that uses BSAFE-C generates the following pseudorandom values during each handshake: a 32-byte session identifier, a 28-byte server random, a 20-byte ephemeral DH secret key, and a 20-byte nonce whenerver DSA is used for signing. The preferred cipher suites all use DH for the key exchanges, and as these were the ones considered, the above description is satisfactory, even though it does not cover all the possible configurations of the library.

The library does not cache unused random bytes by default. However, it has an internal wrapper around the Dual_EC_DRBG which provides its own layer of caching. This layer is implemented in such a way that, even though an attacker has no way of knowing if the first value generated by the server – the session ID – contains cached bytes, the concatenation of the session ID and the server random always contains a full output block. Furthermore, one to 30 bytes of the subsequent block are also exposed. To recover the internal state of the generator, the attacker has to consider each of the 30 possible output blocks in the concatenated 60-byte value B, since 0 to 29 of the first bytes can be drawn from the cached bytes. First the attacker generates more output from the generator by considering each of the possible states in turn. Thereafter, the correct state can be found by comparing the generated output to the rest of the bytes in B and, if needed, compute the secret DH exponent the same way the server would, compute the public DH value, and finally compare this to the server's public DH value.

From this we deduce that it takes approximately $30 \cdot 2^{15}$ scalar multiplications of a variable point ($s_i Q$) and an equal number with the fixed point Q to recover the state of the generator. Hence the total cost of the attack is $30 \cdot 2^{15}(C_v + C_f)$.

4.5.1.2 BSAFE-Java

The examined version of the BSAFE-Java library was 1.1, and instead of focusing on all the possible configurations, only connections using the `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` cipher suite were considered. In contrast to the C-version of the library, the Java-version does not cache bytes, which means that every generated output value is aligned with a block of generator output. Unfortunately for the attacker, the session ID is not a 32-byte pseudorandom value. Instead the session id is watermarked as follows: the 20 first bytes are set to the 20 first bytes of the server random, and the twelve remaining bytes are set to the string "RSA␣SSLJ␣␣␣␣". Therefore, the attacker has to rely on the 28-byte server random.

The random values generated during a TLS handshake are: a 28-byte server random, a 32-byte secret ephemeral ECDH key and a 32-byte nonce for the signing with ECDSA. A passive attacker that wants to recover the state of the generator knows that the server random represents an x-coordinate of the curve,

but the two most and least significant bytes are missing. Hence the attacker has to guess 4 bytes in total, which is equal to 2^{32} possibilities. As previously discussed, approximately half of the possible x-coordinates do not lie on the curve, and hence, the attacker needs to check approximately 2^{31} x-coordinates in the worst case. In order to verify each guess, the attacker has to generate a 32-byte ECDH secret key, compute the corresponding public key and finally compare the result to the public key sent by the server. This requires one variable point multiplication and five fixed point multiplications. Hence, the total cost of the attack is in the worst case $2^{31} \cdot (C_v + C_f)$.

4.5.1.3 SChannel

SChannel ("Secure Channel") is a security component in the Windows operating system. It provides security and confidentiality for socket-based communications and supports many different protocols. It is, however, mostly used for TLS. Similarly as with BSAFE-Java, the attacked configuration relied on ECC on the curve P-256. More precisely, the key exchange method used was ephemeral ECDH and ECDSA was used for signatures. As mentioned earlier, the version of SChannel that was used is the same as used by Internet Information Services (IIS) distributed with Windows 7 and Windows Server 2010 R2. SChannel relies of Microsoft's FIPS 140-2 validated Cryptography Next Generation (CNG) application programming interface (API), which includes an implementation of the Dual_EC_DRBG.

The way SChannel handles the TLS handshakes differs significantly from how the other libraries are doing it. Each handshake is performed by one of many worker threads, and Dual_EC_DRBG in CNG maintains a separate state in each thread. Furthermore, ephemeral ECDH keys are cached for two hours (hard coded in this configuration) and shared between threads until the timeout expires.

The output of the generator is not cached. Instead, enough fresh blocks to satisfy the request are generated, after which the remaining bits are discarded. During each TLS handshake, random bytes are generated as follows: 32 bytes for the session id, 40 bytes for the secret ECDH key, 32 bytes that are not relevant for the attacker, 28 bytes for the server random and 32 bytes for the ECDSA signature. The 40-byte random for the ECDH key, even though the key should be 32 bytes, is due to the fact that SChannel is using FIPS 186-3 B.4.1 (Key Pair Generation Using Extra Random Bits) to generate key pairs. Also note that the bytes for the server random are generated after the ECDH key. This forces an attacker to deduce the state of the generator from the session ID or previous handshakes.

The implementation of Dual_EC_DRBG in CNG differs from the current SP-800-90A specification. The reason for this appears to be a bug, which makes the implementation behave like an earlier version of Dual_EC_DRBG, namely the 2006 version. The difference is an extra state update at the end of each call to the generate function in the 2007 version. What we have described earlier is the 2007 version, so in order to get the 2006 version one simply omits the final state update. The effect of this is that, in contrast to the 2007 version, an attacker who knows the state at the end of the previous generate call cannot predict the first block of the next call even if he/she knows the additional input. The last detail of the TLS implementation in SChannel is that the session ID

contains what could be considered a probabilistic fingerprint. The session ID is obtained by taking the 32 bytes requested and modifying the first four bytes by interpreting them as an unsigned integer and then considering it modulo 20000 (a hard-coded constant). Thus the presence of zeros in the third and fourth byte of the session ID is a likely fingerprint for SChannel. In other words, it is not possible to without doubt identify SChannel implementations of the TLS protocol by looking at the session IDs. However, given enough session IDs, SChannel implementations can be identified with high probability.

In Table 4.1 we saw SChannel I and SChannel II. These correspond to two different attacks on the SChannel implementation of the TLS protocol. The first attack uses the server random to recover the state of the generator, and then compromises the ephemeral ECDH key that is used for all active sessions until it expires. The attack is complicated by the use of many threads with different states. The attack is as follows: on observing a handshake with a new ephemeral server public key, denoted h, the attacker works backwards through previous handshakes and uses the server random from these to obtain candidate Dual_EC_DRBG states. Each candidate state must be checked against the ECDSA public key to determine the state used in that handshake, and then against the session ID in h to check if the same state was used to generate the ephemeral key. When a matching state is found, it is easy to generate the private ephemeral ECDH key and then the session keys. The worst case complexity of the attack is $2^{31}(C_v + 4C_f)$.

The second attack uses the session ID in the handshake where a new ephemeral ECDH key is derived to determine the Dual_EC_DRBG state. This completely bypasses the issue created by the threading in SChannel, but the resulting attack is still more complex than the previous one since the attacker has to try all the possibilities for the first four bytes of the session ID. Therefore, the attacker can recover the state of the generator by enumerating all the possibilities for the first four bytes of the session ID and the last two bytes of the output block that generated the session ID. This gives all the possibilities for the output block, and therefore the only thing the attacker has to do is generate the private ECDH key and the corresponding public key in the same way as the server for all the possibilities, and then check these against the server's public key. Guessing the first four bytes requires approximately 2^{18} guesses, since these bytes are considered modulo 20000, and the last two bytes give 2^{16} possibilities. Thus there will be 2^{33} candidate curve points, of which approximately 2^{17} will coincide with the last two bytes of the session ID. The final step is to generate two blocks of output for the ECDH key for each of these states. Putting all together we obtain the worst case complexity of $2^{33} \cdot (C_v + C_f) + 2^{17}(5C_f)$.

4.5.1.4 OpenSSL-fixed

OpenSSL-fixed fully supports the TLS protocol version 1.2, and the default cipher suites use ephemeral ECDH for the key exchange and either RSA or ECDSA for the signatures. Due to this, only connections using ECDH were considered.

During each handshake, the following random values are generated with Dual_EC_DRBG: a 32-byte session id, a 28-byte server random, a 32-byte ephemeral ECDH key, and a 32-byte nonce whenever ECDSA is used. These values are generated by repeatedly calling the generate function, and as the imple-

mentation of Dual_EC_DRBG does not cache unused bytes, every sequence of random bytes begins with up to 30 bytes from one elliptic curve point. In contrast to the other libraries considered in this work, OpenSSL-fixed uses additional input in each call to the generate function. The additional input has the form:

$$\text{adin} = (\text{time in secs} \parallel \text{time in } \mu \text{ secs} \parallel \text{counter} \parallel \text{pid}).$$

Each of the components of the additional input string has a length of 4 bytes. The time is obtained using system calls (`gettimeofday()` on Unix-based systems), the counter is a monotonically increasing global counter that is set to 0 at library initialization, and on operating systems where process IDs are available, pid contains the process ID returned from `getpid()`.

An attacker can use the first 32 bytes to begin the basic attack, and narrow down the possible states to a few using the two last bytes of the session ID. Due to the use of additional input, the attacker is unable to deduce the output of the next call to the generate function, unless he/she knows the additional input. Luckily, for the attacker, the additional data is easier to predict than random data. For each possible adin string, the attacker runs the generate function and computes the ECDH public key and then compares the result to the servers public key.

The complexity of the attack depends primarily on two factors: the number of candidate states and the number of possibilities for the additional input. The experiments showed that the number of possible states is small, usually one or two, and practically never more than 3. The time in seconds is already transmitted during the handshake, so the attacker only has to guess the time in μ seconds, the counter, and the process ID. There are two cases to consider. In the first case, which corresponds to the OpenSSL-fixed III attack, the attacker knows that counter value is no bigger than $k \leq 32$ bits but does not know anything about the state of the generator. The attacker is able to recover the state of the generator with the basic attack, but then has to try all possibilities for the additional input. There are approximately 2^{35+k} additional input strings to try. The worst case complexity of the attack is $2^{15}(C_v+C_f)+2^{35+k}(2C_f)+2^{13}(5C_f)$. For details we refer to the original paper [CFN+14].

In the second case, the attacker has already broken a previous connection, and therefore knows both the counter and process ID. This gives at most 2^{20} additional input strings to try, and hence the total complexity becomes $2^{15}(C_v + C_f) + 2^{20}(2C_f) + 2^{13}(5C_f)$. This situation corresponds to OpenSSL-fixed I.

4.6 Conclusion

We have seen that it is indeed possible to insert a backdoor into Dual_EC_DRBG, and that the attacks exploiting the backdoor are practical. Furthermore, it is likely that the constants P and Q provided by the NSA are backdoored. The real question is how such an apparently flawed and particularly slow PRNG can be included in a standard. In hindsight, it is easy to say that the standardization process was flawed.

The revelations about the NSA's SIGINT Enabling Project, the BULLRUN program, and the alleged backdoor in Dual_EC_DRBG, show that organizations like the NSA and Government Communications Headquarters (GCHQ) [gch]

do have the resources to insert vulnerabilities in cryptographic standards and persuade manufacturers of cryptographic devices to use weak or backdoored algorithms. Aside from being bad news for the privacy of the general public, this also shows that the attack models described in Chapter 3 are plausible in practice.

Chapter 5

A kleptographic PRNG

In Chapter 4, we saw an example of a kleptographic PRNG. In this chapter, we will show how one can construct a kleptographic PRNG that does not reveal its state, but instead leaks information of the attacker's choosing. The idea for this construction originated when we tried to find a SETUP for the Universal Mobile Telecommunications System (UMTS). However, as the PRNG in question turned out to be a good quality – albeit very slow – generator, we chose to dedicate one chapter to this construction. Examples of how one can employ this PRNG to create SETUPs in different protocols are presented in Chapter 6.

5.1 The construction

The first requirement for any kleptographic construction is that the backdoor must be asymmetric. In other words, we are forced to encrypt the messages we want to leak with a public key encryption method. As mentioned, the original target was to compromise the security of the UMTS by leaking the 128-bit subscriber keys. The public key encryption of a 128-bit message will have size at least the same as the key size of the chosen encryption method. For instance, using RSA with a 1024-bit modulus will result in 1024-bit ciphertexts. For this reason, we chose to go with ECC, or more specifically 192-bit elliptic curve ElGamal.

The second requirement for this SETUP is that the output must be indistinguishable from a good quality PRNG with unknown seed. Furthermore, we are not able to apply any one-way function, such as a hash function, on the output of the public key encryption, since it would make it impossible to recover the ciphertext, and hence the plaintext. In other words, the output of the encryption must be pseudorandom, or alternatively, we must be able to apply some simple reversible transformation that makes the output appear pseudorandom.

Elliptic curve ElGamal is a probabilistic encryption method, and is hence a step in the right direction. However, we do have a problem: elliptic curve points are usually represented by their x-coordinate – and if necessary, one bit indicating the sign of the y-coordinate – and the x-coordinates of an elliptic curve are not uniformly distributed in the base field. Nevertheless, by applying a randomization transformation to the x-coordinates it is possible to make them appear random enough for practical purposes. The transformation simply splits

the x-coordinate in two equally sized parts and permutes these randomly. Due to the random element, the transformation is not reversible. However, given a transformed x-coordinate, there are at most two possibilities for the original x-coordinate, and therefore the transformation can be considered reversible. The properties of the randomized x-coordinates are discussed in Section 5.3

Now we are ready to describe the details of our construction. We begin by describing the version that we have tested and then turn our attention to a more general version. Some of the details might seem strange at this point, but the reason will become apparent later. Let E be the NIST P-192 elliptic curve, $\mathbb{F}_p$ the finite field with p elements, where p denotes the NIST P-192 prime. Let $q = \#E(\mathbb{F}_p)$, which, in this case, is prime. Let m be a 128-bit message that we want to leak. Let (E, G, xG) be our public elliptic curve ElGamal key. We can leak the message over 512-bits of output. The generator produces its output in blocks of 512 bit. To generate one block worth of output, do the following:

1. Choose n randomly from $\mathbb{Z}_2^{64}$ and let $a = n\|m$. If both m and n are large enough when considered as integers, it is possible, but very improbable, that $a \geq p$. Repeat this step if this happens.

2. ElGamal encrypt a, i.e. $a \mapsto (M_1, m_2)$.

3. Create the following bit strings:

$$\begin{aligned} c_0 &= r\|\operatorname{sign}(\hat{y}(M_1))\|\hat{x}_{z_1}(M_1), \\ c_1 &= s_0\|\hat{x}_{z_2}(M_1), \\ c_2 &= s_1\|m_{2,1}\|s_2\|m_{2,2}. \end{aligned}$$

   Here r is a 31-bit random number and s_0 and s_1 are 32-bit random numbers. z_1 is uniformly distributed in $\mathbb{Z}_2$, and $z_2 = (z_1 + 1 \mod 2)$. $\hat{x}_0(P)$ and $\hat{x}_1(P)$ returns the first 96, respectively, the last 96 bits of the x-coordinate of P,

$$\operatorname{sign}(y) = \begin{cases} 0, & \text{if } y \leq \frac{p-1}{2}, \\ 1, & \text{if } y > \frac{p-1}{2}, \end{cases}$$

   and $\hat{y}(P)$ denotes the y-coordinate of $P \in E(\mathbb{F}_p)$.

4. Output $c_0\|c_1\|c_2$.

There are several simplifications that can be made. Recall that the sign of the y-coordinate is not needed for the decryption of m_2. Thus it could be dropped. Similarly the padding is not necessary. However, the reason for the padding is simple: the size of the ElGamal encryption of a is 384-bit which can be split into three 128-bit parts. However, when adding one bit for the sign, the message cannot fit into 384-bit, and since we wanted to align the output with the 128-bit random numbers transmitted in UMTS, we chose to add 127-bits of random padding.

Let us now describe a general version of our generator. Let E be an elliptic curve defined over the finite field of size p, for some prime p, and let (E, G, xG) be our public key. Furthermore, let m be the message we want to leak. For simplicity, we assume that m is at least a couple of bytes smaller than p. To generate a block of output, do the following:

1. Consider m to be the least significant bytes of an element in $\mathbb{F}_p$ and choose the most significant bytes randomly. Denote the resulting element of $\mathbb{F}_p$ by a.
2. ElGamal encrypt a, i.e. $a \mapsto (M_1, m_2)$.
3. Create the bit string

$$c_1 = \hat{x}_{z_1}(M_1) \| \hat{x}_{z_2}(M_1),$$

where z_1 is uniformly distributed in $\mathbb{Z}_2$, and $z_2 = (z_1 + 1 \mod 2)$. $\hat{x}_0(P)$ and $\hat{x}_1(P)$ returns the first respective the last half of the x-coordinate of P.
4. Output $c_1 \| m_2$.

To make the generator faster, or just to align the output to some constraints, the output can then be padded with random bits.

To recover the secret message, the attacker pieces together m_2 and M_1 from the output and then decrypts m_2 with his private key. Since the attacker has no way of knowing which way the first and last half of the x-coordinate were permuted, and both of these permutations might give feasible x-coordinates, he/she might end up with two possible plaintexts. This is not a problem in cases where the correctness of the message can be verified, and otherwise the same message can be transmitted twice to resolve the ambiguity.

It is important to note that our construction needs a source of randomness. In addition, our generator inherits the determinism of the underlying source of randomness. Thus it might be more appropriate to consider our construction as a deterministic transformation of the random input provided, than to consider it as an independent RNG/PRNG. This is also the case with normal PRNGs. However, they only need a small random seed to produce a long string of output that appears random, while our construction needs almost as much random input as it produces.

Recall that our generator only encrypts messages probabilistically and finally does a small transformation of the initial encryption. Thus it seems reasonable that the quality of the underlying source of randomness should be reflected in the randomness of the output, and as will be seen later, our tests appear to support this conjecture.

5.2 The security of our construction

In essence, the output of the PRNG is the ElGamal encryption of the message that the attacker wants to leak. Consequently, obtaining the leaked message is equivalent to decrypting the ElGamal encrypted message without the private key. Since the attacker is the only one with access to the private key x, no one else can obtain m even if they are aware of how the PRNG works.

5.3 Undetectability

The undetectability of our our construction is vital for successful deployment, and in this section we analyze whether an adversary can distinguished the output

of our generator from the output of a high quality PRNG. The two scenarios we will focus on are

- The adversary knows all the details of our construction.
- The adversary does not know anything about our construction, except that we claim that is is a PRNG.

We will start with the first scenario.

5.3.1 A distinguisher

If all the specifics of the implementation of the generator, including the curve used, are known, then, assuming enough output is available, it is easy to distinguish between our generator and any good quality PRNG. Let r be a 512 bit string produced by any PRNG. Assuming that the output is generated by the tested version of our generator, there are, in principle, two alternatives for the x-coordinate of M_1. However, in practice, the alternatives might not be the x-coordinates of any point on the curve. Tests show that for 'truly random' output the number of viable x-coordinates is on average one. For our generator the corresponding number is 1.5. The amount of output needed to estimate the expected value accurately enough to be able to distinguishing between 1 and 1.5 is quite small, only a few blocks on average. In addition, it is easy to verify that the generator used does not use our construction: our generator will never output a block with no viable x-coordinates, while this will happen, on average, in approximately 25% of the cases with a "regular" generator. We see that these are indeed practical methods of distinguishing this generator from others. Note by that one must know which curve is used and exactly how the original output is scrambled and/or padded in order for this technique to work.

The heuristic explanation of the difference in the expected value of viable x-coordinates is simple. Let x_1 and x_2 denote the upper respective the lower bits of a given x-coordinate. The base field contains approximately as many elements as the elliptic curve has points. Since every viable x-coordinate gives two points on the curve it follows that the probability that a randomly chosen x-coordinate lies on the curve is approximately 0.5. If we assume that $x_1 \| x_2$ being on the curve is independent from $x_2 \| x_1$ being on the curve, then we arrive at the result given by the tests. Tests show that with our generator 50% of the blocks have only one viable x-coordinate on average, while all the remaining blocks give two viable coordinates. For both `AES_OFB` and `Threefish_OFB`, the corresponding proportions are: 25% of the blocks give zero viable coordinates, 50% gives one, and the remaining 25% give two. This implies that that 'reversing' the x-coordinate is indeed independent.

5.3.1.1 Recovering the elliptic curve and padding scheme

As noted earlier, this weakness can only be used to distinguish our generator from other PRNGs if the curve used and the padding scheme is known. Since the number of curves and padding schemes that can be used is very large, it is infeasible to try them all. Therefore the natural question: if all the other details of our construction are known can the padding scheme and curve used be recovered?

Let us first consider the case where the elliptic curve used is known but the padding scheme is not. In addition, assume that we know how much padding is used but not its placement. Let k be the number of bits of padding, and n be the length of one output block. Then there are $\binom{n}{k}$ possible padding schemes. Clearly, the correct padding scheme can be found by trying all possibilities, since the correct one can be probabilistically distinguished by checking the number of viable x-coordinates per output block. The correct padding scheme is the one where the number of viable x-coordinates per output block is on average 1.5, and all other padding schemes have 1.0 viable x-coordinates per output block on average. The problem with this is that, already with a relatively small amount of padding, the number of possible padding schemes is enormous. For example, with $n = 512$ and $k = 128$, there are approximately $1.69 \cdot 2^{410}$ possibilities. Therefore, the padding scheme can only be recovered if the amount of padding used is very small. In the case that the amount of padding is unknown, this technique is essentially useless, unless an upper bound on the amount of padding is known and the upper bound is small enough to make exhaustive search feasible.

Let us now consider the case where the padding scheme is know but the elliptic curve is not. Under the assumption that the construction uses some standardized elliptic curve, it is easy to recover the used curve. There are only a handful of alternatives so one can try them all. However, if the curve used is non-standard, then we are in trouble, as there is no way to try all possible curves.

We have seen that exhaustive search is not possible and hence we have to consider other options. The problem we have to solve is: *Given pairs (x_1, x_2) where at least one of $x_1 \| x_2$ and $x_2 \| x_1$ is an x-coordinate on an unknown elliptic curve, construct the elliptic curve.* We do not know if there exists an efficient algorithm to solve this problem, or even if the solution to the problem is unique. However, the problem seems hard and throughout the rest of this thesis we will assume that it is. This assumption is formally stated as Assumption 5.1.

Assumption 5.1. Given pairs (x_1, x_2) where at least one of $x_1 \| x_2$ and $x_2 \| x_1$ is an x-coordinate on an unknown elliptic curve, it is infeasible to construct the elliptic curve.

We have seen that under Assumption 5.1 it is infeasible to recover the elliptic curve and padding scheme used by our construction even if all the other details of the construction are known. Therefore the distinguisher presented in the beginning of Section 5.3.1 can only be used after at least one device implementing our generator has been reverse-engineered. In other words, the method can only be used to find so called contaminated devices after the initial revelation that there does exist contaminated devices, and the reverse-engineering of one of these devices. Thus this weakness is hard to exploit in practice.

5.3.2 Our construction as a PRNG

To asses the quality of our construction when considering it as a PRNG, we tested it with different statistical tests specifically designed for testing PRNGs. Before we go into further details about the tests and how they were conducted, we review the basics of the subtle art of testing random numbers.

5.3.2.1 Random number testing

What is a random number? Is a specific number, such as 7, random? Well, it depends. If it generated by some random process it might be. If it it is made up for the sake of some argument, like this one, then it is definitely not. Perfect RNGs produce "unlikely" sequences of random numbers at exactly the right rate. Testing a RNG is therefore quite subtle.

Random number generators are usually tested by subjecting them to different statistical tests. These tests return so called p-values, and in order to understand what this means we must first review the so called *null hypothesis*. The null hypothesis is: "This generator is a perfect random number generator, and for any choice of seed produces a infinitely long, unique sequence of numbers that have all the expected statistical properties of random numbers, to all orders" (excerpt from the Dieharder [BEB] manual). We know that, technically, the null hypothesis cannot hold, since all software PRNGs have a period and do not provide enough entropy. In addition, many hardware generators also fail, due to bias or correlation; a consequence of the fact that nature often is unpredictable, but rarely random.

It is, however, possible that the null hypothesis is practically true, *i.e.*, the generator can be good enough so that its output sequences are indistinguishable from truly random ones. Here we mean indistinguishable in the sense that there are no known tests or tools that are able to make the distinction with available resources and bounded error probability.

To test the null hypothesis, one generates a sequence of presumably random numbers. From these numbers it is possible to generate tests statistics, *i.e.*, empirically computed numbers that are considered random samples drawn from a known distribution. These samples may be covariant subject to the null hypothesis, if overlapping sequences are used to generate successive samples. If this is not the case, then the samples should not be covariant. Since the target distribution of the test statistics is known, it is possible to compute the probability of the obtained empirical result in the case that the null hypothesis is true. This probability is the p-value of the particular test run.

Normally a low p-value, typically 0.05 or less, is used to reject the null hypothesis. It is improbable to get such a result if the generator is good. For any other p-value the generator is not rejected. Thus a good random generator would be one that we have not been able to make fail yet. However, this criterion is naive and should definitely not be used when testing RNGs. A p-value greater than 0.95 is as unlikely as one less than 0.05, and hence it would make as much sense to reject a generator whose output sequences give p-values greater than 0.95.

A perfect random number generator produces all bit strings of a given length with equal probability, and hence the p-values should be uniformly distributed in the range $[0, 1]$, as different runs of the same test always considers presumably random sequences of the same length. In other words, a perfect RNG should produce p-values less than 0.05 on 5% of all test runs (on average), and similarly p-values greater than 0.95 in 5% of the test runs. Hence, a RNG that fails to produce p-values less then 0.05 5% of the time is a bad generator.

The p-values themselves can, and should, be used as test statistics. The question that arises is: how do we know if the generator yields p-values with the correct distribution? We can subject the set of p-values to a Kolmogorov-

Smirnov (KS) test against a uniform distribution. This will effectively turn the set of p-values into a single p-value that tells us the probability that, assuming our generator is perfect, we would have obtained the original set of p-values. Clearly, the p-values of the second level should also be uniformly distributed. Thus, it is possible to again subject these p-values to a KS test, and this should again yield an uniform distribution of p-values on this level of aggregation. In other words, the aggregation of KS tests is, in this setting, idempotent. Therefore, a perfect generator will yield a distribution of p-values that converges to uniform, at any level, as more p-values are added. If, however, the generator is a bad one, then the p-values are non-uniformly distributed, and as more p-values are added, the non-uniformity will become more apparent, and eventually lead to convergence of the p-value of the highest level of aggregation to zero.

As a conclusion, non-uniformity of the p-values, at any level, is a sign of trouble. However, this does not automatically mean that the tested generator is a bad one. It is possible that the problem lies elsewhere, *i.e.*, in the test itself. There are many possibilities for errors in the tests, *e.g.* programing errors, rounding errors or limitations of the numerical precision used. To detect such errors, the tests are usually tested with known high quality generators.

5.3.2.2 The Dieharder test suite

The Dieharder test suite for random number generators [BEB] is an open source tool for testing random number generators. The tests are drawn from George Marsaglia's "Diehard battery of random number tests", the NIST Statistical Test Suite [RS+10] and other sources, such as the author's invention, user contributions, other open source test suites and the literature.

Dieharder contains 31 different tests, and some of these tests accept an additional parameter that effectively turns the test into a different one. The total number of tests, with all the relevant parameter variations, is 80. All of these return p-values, so the discussion in Section 5.3.2.1 applies. By default Dieharder runs each test, with a few exceptions, with 100 p-value samples that are then aggregated to one p-value using a KS test.

To test our construction, we decided to do a more thorough testing. Therefore we ran the tests in the so called "test-to-destruction" mode. In this mode, Dieharder first runs the test with the default number of p-value samples and checks the aggregated p-value. If the p-value is within the acceptable range, 100 more p-value samples are added and the aggregate p-value is recomputed and checked. This is repeated until the aggregate p-value is out of bounds, or the number of p-value samples exceeds 100000. From the earlier discussion, we know that any bad generator will fail this test eventually as the aggregated p-value converges to zero. On the other hand, we also know that it is possible for a good generator to fail this test due to some "error" in the test implementation. Hence, the results should be compared to those of a perfect generator to rule out bad tests. Unfortunately, no perfect generator exists. Nevertheless, Dieharder includes implementations of two "gold standard" generators, which we can compare our results to. These generators are called `AES_OFB` and `Threefish_OFB`, and they are the generators obtained when running the block ciphers Advanced Encryption Standard (AES) and Threefish in output feedback (OFB) mode.

5.3.2.3 Our results

To asses the quality of our construction we implemented it and tested it thoroughly with Dieharder in the test-to-destruction mode described in Section 5.3.2.2. We used the default value for the failure threshold and fixed the message m that was leaked. Our generator needs a source of randomness and we chose to use `AES_OFB`. We also run the same tests on the reference generators `AES_OFB` and `Threefish_OFB`. We ran three complete sets of tests with each one of the reference generators and only two complete sets with our generator. We restricted the number of tests run with our generator due to the expensive computations involved. Our implementation is highly optimized and trivially almost completely parallelizable – we wanted to preserve the determinism of the generator and could not come up with an efficient way of parallelizing the generation of the random input needed without loosing the determinism. Nevertheless, in practice the computational complexity of generating the random input needed with the our own optimized version of the `AES_OFB` generator is negligible when compared to the of complexity of the elliptic curve and modular arithmetic operations needed for the ElGamal encryption. Thus, in practice, our implementation can be viewed as fully parallelizable when the number of cores available is low enough.

Dieharder only utilizes one thread. Due to this all tests were not parallelizable in practice, but luckily the computationally most demanding ones were. Using Dieharder in test-to-destruction mode requires a huge amount of output from the used generator. As a reference value, the longest running test took approximately 28 days to complete when using 8 cores, and practically all the computational power was used by our generator. Generating one Gigabyte of random output using one thread takes about 7-8 minutes. Hence a quick calculation reveals that the test required approximately 40320 GB of output from the generator.

Presenting the results complete with the numbers of p-value samples and aggregate p-values is impractical and hence we choose to only give a short summary. Anyone interested in the precise results and/or the source code for our implementation should contact the author. Our generator passed almost all of the tests. By passing the test, we mean that it reached the 100000 limit for p-value samples. The reference generators behaved similarly, failing the same tests at approximately the same number of p-value samples on average. This implies that our generator is a good PRNG, and furthermore, its output cannot be distinguished by Dieharder from output generated by the reference generators unless the speed of the generator is considered.

5.4 Efficiency

There are two types of efficiency to consider: The computational complexity of the generator and how fast we can leak messages. We will begin by considering the former.

5.4.1 Computational efficiency

Our generator is very slow. Simple tests show that it is approximately 150 times slower than a fast implementation of AES used in OFB mode. As noted ear-

lier, we used AES in OFB mode as the source of randomness for our generator. As our implementation of the generator needs 384 bits of randomness for one block of output, *i.e.*, 512 bits, it is apparent that practically all of the computational complexity is due to the elliptic curve point multiplication required by the ElGamal encryption.

On the other hand, our construction should be at least four times as fast as Dual_EC_DRBG, which has been used as the default random number generator in at least one notable cryptographic library, namely RSA Security's BSAFE. Hence, it is probably fast enough for users not to notice its presence unless they are searching for it. We end this section by noting that there is a direct trade-off between the computational complexity and the leakage bandwidth: more padding makes the generator faster but lowers the leakage bandwidth.

5.4.2 Leakage bandwidth

Let us first consider applications where the leaked message can be verified. In this setting, the effective leakage bandwidth of the version we tested is $(1, 4)$, *i.e.*, we can leak one 128-bit message over four 128-bit random numbers. By abusing the definition of leakage bandwidth we could as well say that it is $(128, 512)$, *i.e.*, we can leak 128-bit in 512-bit of output. This version of the leakage bandwidth is more convenient in this setting, and hence we will adopt it. For the general version of our construction the best case leakage bandwidth is $(k, 2k)$, where k is the bit length of the prime p that defines the base field of the curve. In principle the leakage bandwidth is a little lower, since the maximum length of the messages should be a couple bits shorter than k. In applications where the messages sent vary considerably, it should be possible to send k-bit messages without compromising the randomness of the output, but this is untested.

In situations where it is impossible for the attacker to verify the correctness of the leaked message, it is probably wise to use a modified version of the generator. This modified version always checks the number of viable x-coordinates in the output block when a new message is sent, and in the case of two possible coordinates, the message is simply resent in the next output block. This obviously lowers the leakage bandwidth, and from the discussion in Section 5.3.1 we can deduce that it is $(k, 3k)$ on average in the best case. The addition of the extra step makes the generator a bit slower, but the increase in bandwidth over the alternative of sending every message twice probably makes up for it.

5.5 Open questions

The first open question is the impact of the elliptic curve used. Must the curve satisfy any specific properties or is it possible to use any curve suitable for cryptographic applications? We have only tested one curve, but it seems like a reasonable assumption, since the NIST P-192 has no special properties compared to other standard curves, to assume that the choice of curve will not affect the properties of the generator.

Another open question is how much randomness needs to be added to the message? Let us assume that we have a fixed k-bit message m, *i.e.*, the plaintext will be exactly the same every time. Here k is the length of p in bits. Only the second part of the ElGamal encryption depends on m, and it is $\hat{x}(M_1)m$,

considered in $\mathbb{F}_p$. The problem is that we do not know exactly how the x-coordinates of the elliptic curve points are distributed. We do, however, know that multiplication by a fixed field element is a bijection, and thus there are only as many possible values for the second part of the ElGamal encryption as there are distinct x-coordinates on the curve. Hence $\hat{x}(M_1)m$ cannot be uniformly distributed in $\mathbb{F}_p$, and therefore certainly not in $\mathbb{Z}_2^k$. But what happens if we add one bit of randomness to the message, or in other words, only allow $k-1$ bit messages. In best case we would get $\#\hat{x}(E(\mathbb{F}_p)) \cdot 2$ distinct elements in $\mathbb{F}_p$, but in the worst case we are stuck with $\#\hat{x}(E(\mathbb{F}_p))$ distinct elements, which is what we started with. We do not know the probabilities associated with each case, or even if they are possible or not, but it is probably safe to assume that we end up somewhere in between. In short, we are no wiser at this point. The best method to find out how many bits are needed in each case (fixed message, varying messages, different curves) is probably empirical testing.

5.6 Practical considerations

Our construction can leak arbitrary messages. However, to use it efficiently in practice is not so straightforward. Normally, most protocols are implemented in a way that assumes that a PRNG is available and that any random numbers required can be requested from the PRNG. The PRNG can also be swapped to another one if desired. In other words, the PRNG is not integrated with the rest of the protocol. Consider a scenario where we want to use our construction in conjunction with a protocol to leak sensitive information that compromises the security of the protocol. It is obvious that in order to leak the information, our construction must have access to the information. As the PRNG is not integrated with the rest of the protocol, we cannot simply swap the PRNG in use with our construction; we also need to have to make the sensitive information available to our construction.

Another issue is that the amount of consecutive output that needs to be exposed is large. Therefore the random numbers used internally by the protocol should be taken from some other source of randomness, and the output of our construction should be reserved for the random numbers that are transmitted in plaintext. This can only be accomplished by integrating our construction tightly with the implementation of the protocol.

In contrast, the Dual_EC_DRBG can be used directly; swap the default PRNG to Dual_EC_DRBG and then, assuming enough consecutive output is exposed, the attacker can recover the internal state of the Dual_EC_DRBG in use. Thereafter, he/she can predict all future output and compromise the security of the protocol in use. The limited amount of consecutive random bytes in many protocols makes it harder to exploit the backdoor in the Dual_EC_DRBG. However, as we saw in Chapter 4, the attacks exploiting the backdoor are practical against the TLS protocol.

5.7 Alternative methods

As discussed earlier, the only requirements when kleptographically leaking arbitrary messages in a stream of bits that appear, or is, random are: the messages

must be encrypted with a public key method, and the output must, as said, appear random. Therefore the natural question is: is it possible to reach the goal with any other public key encryption method? We have a couple of constructions that should work. They are, however, untested.

5.7.1 ElGamal over finite fields

We already know that ElGamal is a probabilistic encryption method so the only question is how uniformly distributed the output is. This depends on the group used. If $\mathbb{Z}_p^*$ is used along with a generator that generates the whole group, then the two parts of the output will obviously be uniformly distributed in $\mathbb{Z}_p^*$. Assuming p is chosen such that $p \approx 2^n - 1$, where n is the number of bits in p, the output will be close to uniformly distributed in $\mathbb{Z}_2^n$. Therefore, these group are, at least with a suitable choice of p, very suitable for use with our construction.

Many standardized DH groups have a safe prime p as modulus, and a generator g that generates a cyclic subgroup $G \leq \mathbb{Z}_p^*$ of maximal prime order $q = (p-1)/2$. Recall from group theory that the size of a subgroup always divides the size of the group, and hence there cannot exist a subgroup with prime order larger than q. The first part of the ElGamal encryption of a message m is g^x. Assuming that the secret exponent x, $1 < x < q-1$, is chosen uniformly, g^x will be uniformly distributed in G. However, as G only covers half of $\mathbb{Z}_p^*$, g^x cannot be uniformly distributed in $\mathbb{Z}_p^*$, and therefore not in $\mathbb{Z}_2^n$, where n is the number of bits in p. By using the same trick we did with the elliptic curve version, *i.e.*, randomly permuting the two halves of the bit representation of g^x, we should get something that is closer to uniformly distributed in $\mathbb{Z}_2^n$.

The second part of the ElGamal encryption of m is $m_2 = m(g^y)^x$, where g^y is the recipient's public key. If $m \in G$, then so is m_2, and otherwise $m_2 \in mG = H$, which together with G partitions $\mathbb{Z}_p^*$. Let us fix m. Then, assuming x is uniformly distributed, m_2 is uniformly distributed in either G or H. On the other hand, if m is uniformly distributed between G and H, then it seems reasonable to assume that m_2 is uniformly distributed in $\mathbb{Z}_p^*$.

It is easy to check that all elements in G are quadratic residues modulo p. Whether an element of $\mathbb{Z}_2^n$ is a quadratic residue modulo p can be efficiently checked by computing the corresponding Jacobi symbol. Assume that an adversary knows the group G and the padding scheme used by our construction. Then the adversary can probabilistically distinguish our construction from any good quality PRNG by looking at the ratio of quadratic residues and non-residues. This works similarly to the distinguisher described in Section 5.3.1.

The conclusion of all the heuristics is that our proposed generator should work at least as well if we turn to ElGamal over finite fields. If the group used is the maximal prime ordered subgroup of $\mathbb{Z}_p^*$ for some safe prime p, then the detectability of the generator is similar to that in the elliptic curve case. On the other hand, if we set $G = \mathbb{Z}_p^*$, then our construction cannot be distinguished from a good quality PRNG by looking at quadratic residues modulo p. Therefore, it is probably advisable to use the multiplicative group $\mathbb{Z}_p^*$ directly instead of a prime ordered subgroup. One notable advantage of the finite field versions is that they should be about one order of magnitude faster then the elliptic curve version, assuming that no extra padding is used.

5.7.2 RSA

The RSA public key encryption method is a deterministic encryption method, *i.e.*, the same public key and message always yields the same ciphertext. So at first glance, RSA seems unfit for this purpose. However, there does exist a padding scheme called optimal asymmetric encryption padding (OAEP) that can be used in conjunction with RSA to obtain a probabilistic encryption method. The padding scheme was proposed by Bellare and Rogaway [BR95] in 1994. We will now describe this scheme.

Let k be the security parameter, k_0 and k_1 additional parameters, $n = k - k_0 - k_1$ is the fixed message length. Furthermore, let $F : \mathbb{Z}_2^k \to \mathbb{Z}_2^k$ be a so called trapdoor function, $G : \mathbb{Z}_2^{k_0} \to \mathbb{Z}_2^{n+k_1}$ be a "generate" function, and $H : \mathbb{Z}^{n+k_1} \to \mathbb{Z}_2^{k_0}$ a hash function. To encrypt the message m, choose a random $r \in \mathbb{Z}_2^{k_0}$ and compute the encryption as

$$E(m) = F(m0^{k_1} \oplus G(r) \parallel r \oplus H(m0^{k_0} \oplus G(r))).$$

The padding scheme is illustrated in Figure 5.1. Usually this scheme is used

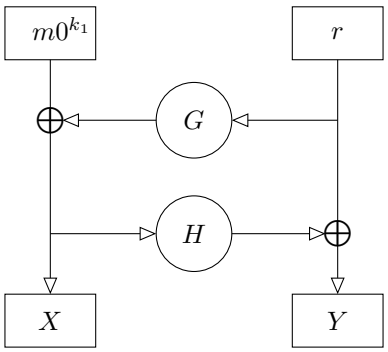

Figure 5.1: The OAEP padding scheme.

with RSA. We refer to the original paper for the exact security claims and their proofs, and concentrate on the randomness of the output.

We choose F to be RSA encryption with a modulus of k bits, and k_0 to be $k/2$ bits (in practice k is even). Choosing k_0 this way gives a probabilistic encryption method which essentially doubles the message length. Thus, the resulting method is similar to ElGamal in this sense. RSA encryption is a bijection, and since there are $2^{k/2}$ possibilities for r, there is essentially $2^{k/2}$ possible ciphertexts for any given message m. This means that approximately $2^{k/2}$ of all possible 2^k k-bit output blocks can be obtained from the generator. How these $2^{k/2}$ elements are distributed in $\mathbb{Z}_2^k$ is not known. It is possible that the distribution is such that the output appears uniform enough, or it might be very biased. Unless the distribution is very clearly biased, it is most likely very hard to find the possible bias and distinguish this generator from any good quality PRNG.

5.8 Conclusion

It is possible to construct a PRNG that can be used as a secure subliminal channel. We have showed that it cannot be distinguished from a high quality PRNG by known statistical tests, unless the specifics of the algorithm, including all the variables, are known. While we have only tested our construction for a specific set of variables, we see no reason why other configurations would not perform similarly.

In contrast to the Dual_EC_DRBG, which leaks its own state, our construction can be used to leak any information, including its state, securely and subliminally. While this makes it possible to compromise the security of any protocol where random numbers are transmitted in plaintext, the real novelty is that this opens up the possibility to compromise protocols that cannot be compromised by the techniques presented in Chapter 2. An example of such a protocol is the UMTS, and in Chapter 6 we show how to compromise the security of this protocol with our construction.

Chapter 6

Where to use our KPRNG

As mentioned in Chapter 5, our kleptographic pseudorandom number generator (KPRNG) was originally designed as a way to compromise the security of the UMTS identification and encryption protocol. Associated to each subscriber identity module (SIM) card is a 128-bit secret subscriber key, which is stored on the card and in a database (termed authentication center (AuC)) on the carrier's network. If the key is obtained by an attacker, then the attacker can impersonate the SIM card and decrypt all communication between the mobile device and the base station.

The network authenticates all mobile devices with a *challenge-and-response* protocol. As part of this protocol, a 128-bit random number generated in the AuC is sent over the network in plaintext. The original idea was to embed our KPRNG into the carrier's network and leak the subscriber keys in the transmitted random numbers.

In February 2015, it was revealed in documents leaked by whistleblower Edward Snowden, that the NSA and its British counterpart GCHQ had stolen the subscriber keys of some of the SIM cards manufactured by Gemalto, one of the world's largest SIM card manufacturers [SB15]. Compared to this approach, our method is not so convenient. We will, however, present our method of leaking the subscriber keys, since it serves as an illustrating proof of concept of how to take advantage of a KPRNG that can leak arbitrary information.

6.1 UMTS authentication and encryption

We start by giving an overview of the mutual authentication between the 3G network and a mobile device, and then present how the encryption keys are derived from the subscriber key. The UMTS Authentication and Key Agreement system is defined in the Third Generation Partnership Project (3GPP) specification TS 33.102 [3gpa]. A potentially more readable description, and a thorough explanation of everything related to UMTS security, can be found in [NN03].

There are three entities involved in the mutual authentication process, namely:

- home environment (HE);
- serving network (SN);

- a terminal, more precisely a universal subscriber identity module (USIM).

The authentication is mutual, and hence the SN verifies the identity of the USIM, and the USIM verifies that the SN has been authorized by the HE to do so. The cornerstone of the mutual authentication is the subscriber key K associated to each USIM. As mentioned earlier, K is a fixed 128-bit key that is shared between the USIM and the AuC. K is kept secret and never transfered from these locations.

The authentication process begins when the SN obtains the identity of the user. The identity is either an international mobile subscriber identity (IMSI), temporary mobile subscriber identity (TMSI) or packet TMSI (P-TMSI). Then the visitor location register (VLR) or serving GPRS support node (SGSN) sends an *authentication data request* and the identity of the user to the AuC. The AuC responds by returning an *authentication data response*, which consists of *authentication vectors* that are derived from the IMSI (or some other identity) and the subscriber key K. This process is illustrated in Figure 6.1. The generation of these authentication vectors involves several cryptographic algorithms. We do, however, defer the overview/treatment of these to Sections 6.1.1 and 6.1.2.

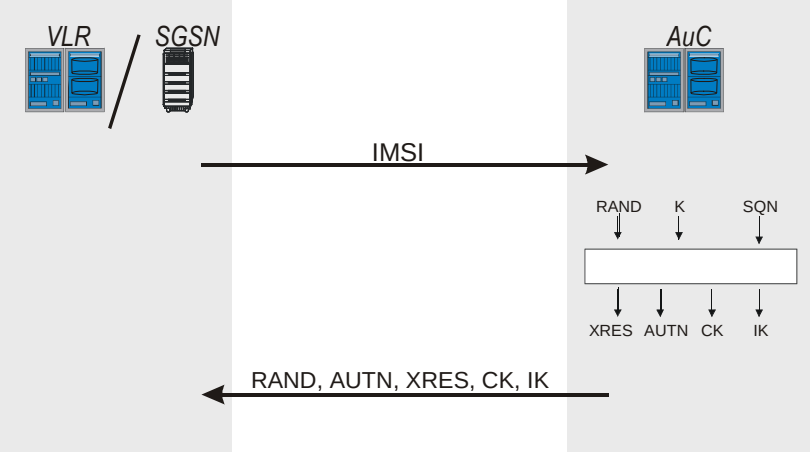

Figure 6.1: Authentication data request and authentication data response.

The SN, *i.e.*, the VLR or SGSN sends the random number (RAND) and authentication token (AUTH) from an authentication vector to the terminal, which then performs a series of computations that resemble the computations performed when generating the authentication vector. The results of this computation gives the USIM the ability to verify that the AUTH parameter was indeed generated by the AuC and that the same AUTH has not been seen by the USIM earlier. If the computation verifies that everything is as it should be,

then the computed RES parameter is sent to the SN as part of the *user authentication response*. Now the SN has the ability to compare RES to the expected response (XRES), which was supplied as a part of the authentication vector. If these parameters match, then the authentication process ends successfully. This process is shown in Figure 6.2.

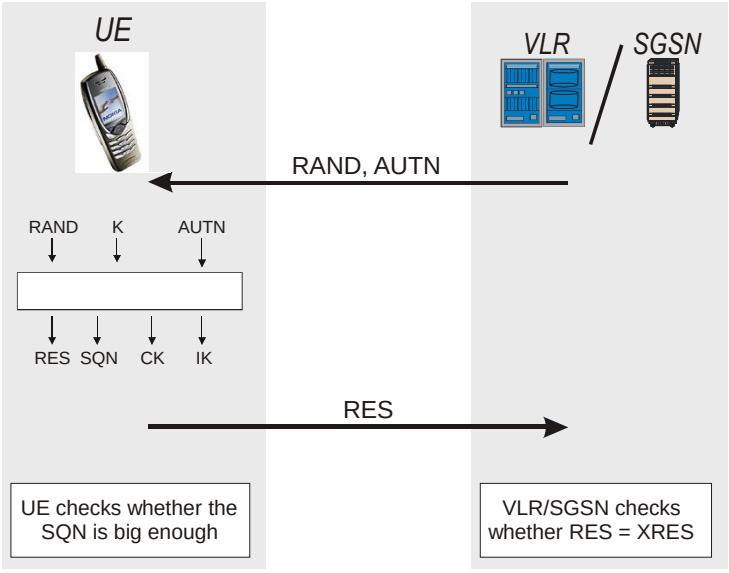

Figure 6.2: User authentication request and user authentication response.

6.1.1 Generation of authentication vectors in the AuC

The authentication vector generation begins by choosing an appropriate sequence number (SQN). Roughly speaking, it is enough to choose the SQNs in ascending order. For the precise requirements we refer to [NN03] or [3gpa]. The next step is to generate a 128-bit RAND. After this, the following values are computed

- MAC-A $= f_1(K, \text{SQN} \parallel \text{RAND} \parallel \text{AMF})$.
- XRES $= f_2(K, \text{RAND})$,
- CK $= f_3(K, \text{RAND})$,
- IK $= f_4(K, \text{RAND})$,
- AK $= f_5(K, \text{RAND})$.

Here the functions $f_i, i \in \{1, \ldots, 5\}$ are cryptographic *one-way* functions, which are considered in Section 6.1.3. The authentication token AUTH $=$ (SQN $\oplus$ AK) $\parallel$ AMF $\parallel$ MAC-A is assembled and the AuC outputs the quintuplet

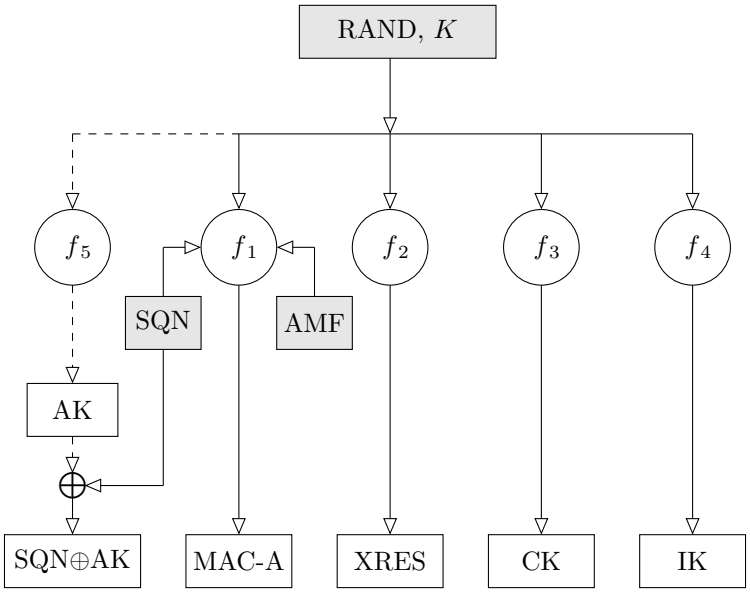

Figure 6.3: Authentication vector generation in the AuC. The input is marked with gray.

$Q = (\text{RAND}, \text{XRES}, \text{CK}, \text{IK}, \text{AUTH})$, which is usually referred to as the *authentication vector*. This process is depicted in Figure 6.3.

The concealment of SQN is optional and to omit it we simply let AUTH = SQN $\|$ AMF $\|$ MAC-A. In this case it is unnecessary to compute AK.

6.1.2 Authentication and key derivation in the USIM

Upon receiving the pair (RAND, AUTH), the USIM computes the following values

- $\text{AK} = f_5(K, \text{RAND})$,
- $\text{SQN} = (\text{SQN} \oplus \text{AK}) \oplus \text{AK}$.
- $\text{XMAC-A} = f_1(K, \text{SQN} \parallel \text{RAND} \parallel \text{AMF})$.
- $\text{RES} = f_2(K, \text{RAND})$,
- $\text{CK} = f_3(K, \text{RAND})$,
- $\text{IK} = f_4(K, \text{RAND})$.

If the SQN is not concealed, then the first two steps are omitted. This process is depicted in Figure 6.4.

6.1.3 The cryptographic primitives f_1–f_5

The cryptographic functions f_1–f_5 are only deployed in the AuC and USIMs, which means that different operators can use different functions. Hence, there

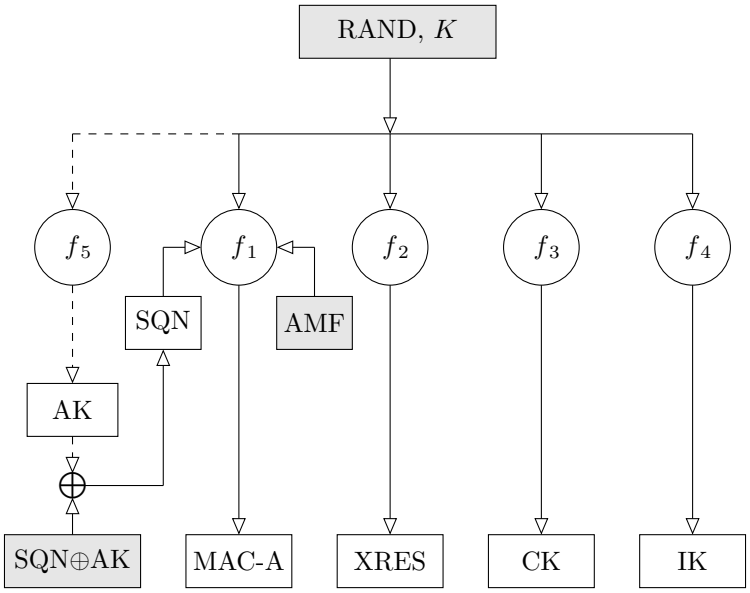

Figure 6.4: Authentication vector generation in the USIM. The input is marked with gray.

was, according to the 3G Security Group, no need to formally standardize these functions. Nevertheless, the 3G Security Group agreed to develop an example set of functions to be used as a reference or by operators who did not wish to develop their own functions. The cryptographic requirements for f_1 – f_5 are specified in the 3GPP specification TS 33.105 [3gpb] and the example functions can be found in the 3GPP specification TS 35.206 [3gpc]. In addition, all of this is thoroughly discussed in [NN03].

The output of the example functions depends, in addition to the input, on a constant OP that is used to add operator-dependent information to the design even if the chosen functions are the same. The OP is a 128-bit value that can be chosen freely. This value is never used directly in the example functions. Instead, a value derived in a non-invertible way from OP and K is used. This value is denoted by OP_C. For added security, OP should not be stored on the USIM, since it suffices to store OP_C, and OP is not compromised even if OP_C is. Furthermore, it is more efficient to store OP_C and not have to derive it from OP and K each time it is needed. Similarly to K, OP_C is never transfered anywhere.

6.1.4 Confidentiality and integrity

To ensure the confidentiality and integrity of the data transmitted between the user and the network all data is encrypted and authenticated. CK is the encryption key and IK is the integrity key. The details of the encryption and data authentication is irrelevant for our discussion and hence we omit them.

6.2 The attack

From the previous section we can deduce that, if an adversary that passively listens to the communication has access to the subscriber key K, the specifications of f_1–f_5, and any additional constants, then he/she can decrypt all communications and, in addition, impersonate both the USIM and the HE, *i.e.*, the network. Assuming that the goal is to conduct mass surveillance on the users of the 3G network, we notice that there are only a few sets of specifications for f_1–f_5 (one set for each operator), while the K and OP_C is different for every user.

6.2.1 Obtaining f_1–f_5

The functions f_1–f_5 must be implemented on each USIM and hence it is probably easiest to obtain a USIM from each targeted operator and simply reverse engineer the algorithms. For an attacker that has the resources to conduct mass surveillance this should be an easy task.

6.2.2 Obtaining the subscriber keys

In principle, it is possible to leak the subscriber keys with our KPRNG. How one can do it in practice without being detected is the real question. One possibility is to persuade the manufacturers of the AuCs to integrate our KPRNG into the device. After this, the attacker only needs to passively monitor the radio traffic on the network to (in time) obtain the subscriber keys of its choice.

6.2.3 Obtaining any additional user specific constants

Any additional user specific constants that are stored on the USIMs must also be stored or derivable at the AuC. Hence, these constants can be leaked via our KPRNG in the same way as the subscriber keys.

6.2.4 The efficiency of the attack

Compared to stealing the subscriber keys directly from the manufacturers of the USIMs, as the NSA and GCHQ allegedly did, our approach seems very slow and cumbersome. However, given enough patience and resources, this method would also yield the desired results. Furthermore, our method serves as a proof of concept of how the security of big systems can be compromised by leaking information in random numbers. While it is evident that any protocol that relies on sending random numbers over insecure channels can be compromised by using the random numbers as a subliminal channel, our construction shows that this is possible in an virtually undetectable way, if we assume that the KPRNG can be embedded in a so called *black-box*.

6.3 Other targets for our KPRNG

In principle, any communication protocol where random numbers are sent over an insecure channel as plaintext, could potentially be compromised by 'correct' usage of our KPRNG. This includes a majority of the one-time handshakes

used in communication protocols that establish secure channels over insecure networks. The reason for this is simple: the handshakes use these random numbers as a way to prevent so called *replay attacks*, *i.e.*, an attack where the attacker records earlier handshakes and then replays them in an attempt to impersonate the real user. Simultaneously, the use of random numbers in this way opens up the possibility of using them to leak sensitive information. However, this is only feasible in black box implementations, since otherwise the user can easily see that the random numbers are not random. Even if the handshake is implemented as a black box, the user might be able to detect the leakage of information through the random numbers if the output is not random enough.

This is where our construction comes in. Assume that a user has access to two devices that implement some protocol that includes a handshake involving random numbers sent in plaintext. Both devices implement the protocol correctly but one of the devices leaks information through the random numbers using our construction. Assuming that the user does not know the details of our implementation, there is no way for the user to distinguish the contaminated device from the legitimate one.

In Chapter 3 we saw that both the SSH and TLS protocol uses random numbers as part of the handshake to prevent replay attacks. Some of the attacks by Gołębiewski ([GKZ06]) exploited this. However, they did not show/discuss how to make the attacks undetectable. In contrast, our construction makes it possible to successfully mount these kinds of attacks in a way that is completely undetectable. This opens up the possibility of both active and passive attacks. Leaking the private keys of the server enables *Man-in-the-Middle* attacks, while leaking the seed of the PRNG used to generate the secrets in the key exchanges enables the attacker to passively monitor the traffic and decrypt it at will.

In Chapter 5, or more precisely Section 5.6, we saw that, compared to our construction, the Dual_EC_DRBG is more of a "plug and play solution" as it does not need to be integrated into the implemented protocol. However, the backdoor in the Dual_EC_DRBG cannot be exploited unless enough consecutive bits of its output is exposed by the protocol. The only protocol that we know of that exposes enough random bytes is the TLS protocol. Hence the backdoor in the Dual_EC_DRBG is not useful in a general setting. In contrast, our construction can be used to compromise any protocol where random numbers are transmitted in plaintext. As mentioned, it requires tight integration of our construction with the implementation of the protocol. In addition, if the bandwidth is small, leaking the sensitive information may take a while. However, if the attack is planned correctly, once enough sensitive information is leaked, the attacker has the possibility to completely compromise the security of the protocol.

6.4 Preventing these attacks

The easiest way to prevent these kinds of attacks is to not use any black box implementations of cryptographic primitives or protocols. Instead only implementations that allow verification of their correctness should be used. The dangers of black box implementations was pointed out already in 1996 by Young and Yung. For some reason, this has not made black box implementations ob-

solete. An example of this is Trusted Platform Module (TPM) chips, which are small tamper resistant chips that are used to store users encryption and signing keys. In addition, several cryptographic primitives are built into these chips. The idea behind this is that this way all cryptographic operations can be done inside the tamper resistant chip, and thus the keys never need to leave the chip. As the chips are tamper resistant, this should prevent key theft. At first glance these chips seem like a good idea. However, one of the cryptographic primitives included in TPM chips is a PRNG, which makes it possible for an attacker with wast resources to integrate a KPRNG into these chips.

The problem with black-box cryptography is that, even though it can made harder, *e.g.*, by requiring validation, a black box implementation of cryptographic protocols/primitives can always be kleptographic. Hence we recommend that they should be avoided whenever possible. Furthermore, it is the only way to ensure that one is not using a kleptographic device/product.

6.5 Conclusion

We have showed that our KPRNG can be used to compromise the security of the UMTS identification and encryption protocol. It can also be used to compromise the SSH and TLS protocols. Compared to the attacks presented in Chapter 3, using our KPRNG is not as straightforward. The KPRNG must be tightly integrated into the implementation of the protocol that is to be attacked. In addition, it takes much longer to exploit the backdoor provided by our KPRNG.

The real power of our construction lies in its versatility; it can be used to leak any information. For instance, it is possible to securely leak private keys, seeds and any other sensitive information that can be used to compromise the security of the target protocol. Another possibility is to not compromise any protocol, but simply utilize the subliminal channel for normal messages that we want to hide. This is ideal for situations where we want to conceal the fact that we are sending any messages at all.

Chapter 7

Conclusions

In this thesis we have seen that all of the most commonly used key exchange methods, public key encryption methods and signature algorithms can be SETUPed. In addition, we have seen two different types of KPRNGs; one that leaks its state and another one that can be used to leak arbitrary information.

Successfully deploying these SETUPs requires the use of so called *black-box* cryptography, *i.e.*, implementations of cryptographic primitives whose correctness cannot be verified. More precisely, it is possible to verify that the input and output of the implementation conforms to the specifications of the implemented primitive, but it is impossible to verify that the primitive is implemented according to the specifications. Another option is to use malware to modify software already in use. However, this is easier to detect.

As seen in Chapter 3 and 6, successfully SETUPing black-box products requires the co-operation of the device manufacturers. One would think that any manufacturer of cryptographic products would refuse to insert backdoors or weak algorithms into their products, and that these kind of attacks are a theoretical curiosity. However, the documents leaked by whistleblower Edward Snowden has showed that this assumption is false. Apparently the NSA has been able to persuade manufactures to insert backdoors and set an obviously, in hindsight, flawed algorithm as default.

Most of the SETUPs and attacks presented in this thesis can be prevented by not allowing the use of black-box cryptography. Most of the techniques needed for these attacks have been publicly available for almost two decades, and hence it would seem logical to see at least some distrust against cryptographic devices that can be considered black-boxes. There is, indeed, distrust against black-box cryptography. However, it is still used in many places due to lack of alternative solutions.

Black-box cryptography is not only potentially flawed, it does have some notable upsides. A classical problem in cryptography is key management. How to generate keys? Where to store the keys safely? The TPM chips discussed in Chapter 6 solve the latter problem. The user's master keys are stored on the chip, and since the chip is tamper-resistant and the keys never leave the device, it is impossible for anyone to steal the master keys from the chip. This sounds like an ideal solution, and in theory it is. However, as we have shown, it is possible to SETUP TPM chips so that they secretly leak the users keys.

In the end it all boils down to trust. Can the user trust the manufacturer?

If not, what alternatives does the user have? As the average user cannot – and should not – implement the cryptographic primitive, he/she is forced to trust some manufacturer or a third party that has certified the product. Unfortunately open source solutions have the same problem. In theory the user can verify that the product does what it is supposed to, but this requires good programming skills and enough understanding about cryptography. Thus, in practice, the user cannot verify the functionality of the open source alternative, and simply has to trust the authors or an authority that has checked the functionality. In addition, most open source products accept no liability. Hence, an average user, at least from the industry, probably rather pays for a solution from a reputable company than uses an open source product from authors that accept no liability.

We will now turn our attention to the consequences of the theorized kleptographic attacks.

7.1 Consequences

We have shown that the SSH and SSL/TLS protocols are susceptible to several different kleptographic attacks. The efficiency of the attacks vary, ranging from enabling an attacker to decrypt only parts of the traffic, to completely compromising the protocol. We have also seen that, not all, but many of the SETUPs are more efficient with the TLS protocol. This is unfortunate, as the TLS protocol is the more widely used one; it is used to secure essentially all sensitive web traffic. Moreover, the TLS protocol is also often implemented in specialized hardware devices called SSL accelerators. These devices can be classified as black-boxes, and an attacker with wast resources can probably persuade manufacturers to embed kleptographic attacks into these. In contrast, it seems unfeasible to successfully inject kleptographic code into OpenSSH; the most widely used SSH server in the world.

The most efficient attacks against these protocols use the random numbers transmitted in plaintext to broadcast sensitive information that compromises the secure connection. This information is, in true kleptographic fashion, encrypted and the encryption key is leaked securely to the attacker in the DHKEs or the DSA signatures. Hence, the attacker is the only one who can access the sensitive information that is broadcasted. The novelty of the attacks is that they are undetectable; the “message” is encrypted with a high quality block or stream cipher, which makes the ciphertext indistinguishable from random bits, and there is nothing special that can be seen in the DHKEs or the (EC)DSA signatures that are used to leak the encryption key.

These kind of attacks can be used to compromise any protocol that includes random numbers sent in plaintext and uses DHKEs or DSA signatures. Depending on the exact specifications of the protocol, the attacks may be extremely efficient or very hard to leverage. Nevertheless, they are possible, completely undetectable in black-box implementations, and can be exploited to their full extent by passive eavesdropping.

In addition to all of this, we have showed that it is possible to leak arbitrary information in random numbers kleptographically. While this might seem trivial, it is not. Broadcasting arbitrary information in bits indistinguishable from random bits is easy; simply encrypt the information with any high quality

symmetric encryption method. The problem with this is that this backdoor is not secure. The encryption key must be included in any device that is leaking information in this manner, and hence the backdoor can be used by anyone that reverse-engineers one such device and obtains the key. What we have done is different; we encrypt the information that is leaked with a public key method, and hence the information is secure even if a device is reverse-engineered, since this only compromises the public key. However, the output of the public key encryption is normally distinguishable from random bits. Nevertheless, we have empirically showed that it is possible to make the output indistinguishable – for anyone not knowing all the details of our scheme – from random bits by applying a simple "reversible" transformation to the output. Therefore, it is possible to compromise essentially any protocol that involves random numbers transmitted in plaintext, regardless of the key agreement and signature algorithms used.

So far, all attacks have taken advantage of random numbers transmitted as part of the protocol. There are, however, many attacks that do not depend on the random numbers. As seen in Chapter 2, it is possible to SETUP the DHKE, ElGamal encryption and signatures, RSA and the DSA. SETUPing RSA is done by SETUPing the key generation process, such that the attacker can obtain enough information about the private key from the corresponding public key to be able to decrypt any message. The keys of the other algorithms can also be SETUPed. This is, however, not efficient, and there are better ways to SETUP these algorithms. These SETUPs do not compromise all the key exchanges or encryptions. However, they do compromise the user's private signing keys.

7.2 Preventing these attacks

We have seen that many of the attacks rely on abusing the random numbers that are transmitted as part of some protocol, which begs the question: why not omit the random numbers? The answer is simple; no! The random numbers are used for many security reasons. For example, they are used to thwart so called *replay attacks*, *i.e.*, attacks where an eavesdropper tries to impersonate a legitimate user by replaying a past key exchange or handshake.

Another valid question is: how to stop an attacker from deploying kleptographic PRNGs? There are two scenarios: KPRNGs that appear to be normal but are in fact kleptographic, and KPRNGs that are obviously kleptographic. Dual_EC_DRBG is an example of a KPRNG from the first category. The only way to prevent such a KPRNG from being accepted as a PRNG is close scrutiny and simply making sure that it is not standardized. The second category is more problematic. Such a KPRNG can obviously not pass a standardization process, and furthermore, it is really not useful as such. A KPRNG that leaks arbitrary information must have access to that information, and hence such a generator can not be effectively utilized unless it is integrated with the rest of the implementation of the attacked protocol. In other words, such a KPRNG can only pass unnoticed in a black-box implementation. Therefore we have returned to the problems of black-box cryptography. Even if the black-box implementation must use a specific PRNG there is no way for the user to verify that this is the case. Yes, the device can include self tests and be validated, but as the device is a black-box, these obstacles can be circumvented.

Let us now turn our attention to the other attacks, *i.e.* the SETUPs of the basic cryptographic primitives. Similarly to the second category of KPRNGs, these attacks will only go unnoticed in black-box implementations. It is probably possible to make modifications to most protocols that use these primitives that will render the attacks infeasible. We have, however, not focused on this, and will leave this as an open question.

Another possibility is to avoid black-box cryptography, and hence, we are back where we started. A cryptographer can probably verify the functionality of open implementations, but the average user always has to trust someone. Unfortunately, black-box cryptography is still used in many places. However, we hope that, by raising awareness of the possibility of kleptographic attacks, we can at least prompt some discussion about these issues.

Bibliography

[3gpa] 3GPP TS 33.102 V5.2.0 (2003-06) Technical Specification; Third Generation Partnership Project; Technical Specification Group Service and System Aspects; 3G Security; Security architecture (Release 5).

[3gpb] 3GPP TS 33.105 V4.1.0 (2001-06) Technical Specification; Third Generation Partnership Project; Technical Specification Group Service and System Aspects; 3G Security; Cryptographic algorithm requirements (Release 4).

[3gpc] 3GPP TS 35.206 V5.1.0 (2003-06) Technical Specification; Third Generation Partnership Project; Technical Specification Group Service and System Aspects; 3G Security; Specification of the MILENAGE Algorithm Set; An example algorithm set for the 3GPP authentication and key generation functions f1,f1*,f2,f3,f4,f5 and f5*; Document 2; Algorithm specification (Release 5).

[BBG13] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how US and UK spy agencies defeat internet privacy and security. `http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security`, September 2013.

[BEB] Robert G Brown, Dirk Eddelbuettel, and David Bauer. Dieharder: A random number test suite. C program archive dieharder, version 3.31.1, `http://www.phy.duke.edu/~rgb/General/dieharder.php`.

[Big99] Norman L Biggs. Discrete mathematics oxford university press, 1999.

[BJN94] Phani Bhushan Bhattacharya, Surender Kumar Jain, and SR Nagpaul. *Basic abstract algebra*. Cambridge University Press, 1994.

[BK+12] Elaine Barker, John Kelsey, et al. Nist special publication 800-90a: Recommendation for random number generation using deterministic random bit generators. `http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf`, 2012.

[BR95] Mihir Bellare and Phillip Rogaway. Optimal Asymmetric Encryption – How to Encrypt with RSA. In *Advances in Cryptology—EUROCRYPT'94*, pages 92–111. Springer, 1995.

[BV07] Daniel R. L. Brown and Scott A. Vanstone. Elliptic curve random number generation. `http://freshpatents.com/`

Elliptic-curve-random-number-generation-dt20070816ptan20070189527.php, 2007.

[CFN+14] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, and Hovav Shacham. On the practical exploitability of dual ec in tls implementations. In *USENIX Security*, volume 1, 2014.

[DR08] Tim Dierks and Eric Rescorla. RFC5246, The Transport Layer Security (TLS) Protocol Version 1.2. https://tools.ietf.org/html/rfc5246, 2008.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, pages 10–18. Springer, 1985.

[FPS06] Markus Friedl, Niels Provos, and William A. Simpson. RFC4419, Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol. https://tools.ietf.org/html/rfc4419, 2006.

[gch] The website of the Government Communications Headquarters. http://www.gchq.gov.uk.

[Gjo06] Kristian Gjoesteen. Comments on dual-ec-drbg/nist sp 800-90, draft dec. 2005. *Mar*, 16:8, 2006.

[GKZ06] Zbigniew Gołębiewski, Mirosław Kutyłowski, and Filip Zagórski. Stealing secrets with SSL/TLS and SSH – Kleptographic attacks. In *Cryptology and Network Security*, pages 191–202. Springer, 2006.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[Har06] Ben Harris. RFC4432, RSA Key Exchange for the Secure Shell (SSH) Transport Layer Protocol. https://tools.ietf.org/html/rfc4432, 2006.

[KG13] Cameron F. Kerry and Patrick D. Gallagher. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013.

[LPS13] Jeff Larson, Nicole Perlroth, and Scott Shane. The NSA's secret campaign to crack, undermine Internet security. http://www.propublica.org/article/the-nsas-secret-campaign-to-crack-undermine-internet-encryption, September 2013.

[Men13] Joseph Menn. Exclusive: Secret contract tied NSA and security industry pioneer. http://www.reuters.com/article/2013/12/20/us-usa-security-rsa-idUSBRE9BJ1C220131220, December 2013.

[nis] The website of the National Institute of Standards and Technology. http://www.nist.gov/.

[NN03] Valtteri Niemi and Kaisa Nyberg. *UMTS security*. John Wiley & Sons, 2003.

[nsa] The website of the United States National Security Agency. `https://www.nsa.gov/`.

[Pin12] Charles C Pinter. *A book of abstract algebra*. Courier Corporation, 2012.

[PLS13] Nicole Perlroth, Jeff Larson, and Scott Shane. N.S.A. able to foli basic safeguards of privacy on the web. `http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html`, September 2013.

[Res08] Eric Rescorla. RFC5289, TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM). `https://tools.ietf.org/html/rfc5289`, 2008.

[RS+10] Andrew Rukhin, Juan Soto, et al. NIST Special Publication 800-22rev1a: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. `http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf`, 2010.

[SB92] Miles E. Smid and Dennis K. Branstad. Response to Comments of the NIST Proposed Digital Signature Standard. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 76–88. Springer, 1992.

[SB15] Jeremy Scahill and Josh Begley. THE GREAT SIM HEIST – How Spies Stole the Keys to the Encryption Castle. `https://firstlook.org/theintercept/2015/02/19/great-sim-heist/`, February 2015.

[SF07] Dan Shumow and Niels Ferguson. On the possibility of a back door in the nist sp800-90 dual ec prng. *CRYPTO Rump Session*, 2007.

[SG09] Douglas Stebila and Jon Green. RFC5656, Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer. `https://tools.ietf.org/html/rfc5656`, 2009.

[Sil09] Joseph H Silverman. The arithmetic of elliptic curves. *Graduate Texts in Mathematics*, 2009.

[Sim84] Gustavus J Simmons. The prisoners' problem and the subliminal channel. In *Advances in Cryptology*, pages 51–67. Springer, 1984.

[Sim94] Gustavus J Simmons. Subliminal communication is easy using the DSA. In *Advances in Cryptology—Eurocrypt'93*, pages 218–232. Springer, 1994.

[ST13] Joseph H Silverman and John T Tate. *Rational points on elliptic curves*. Springer Science & Business Media, 2013.

[Ste11] William Stein. Elementary Number Theory: Primes, Congruences, and Secrets. http://wstein.org/ent/ent.pdf, 2011.

[Sti05] Douglas R Stinson. *Cryptography: theory and practice*. CRC press, 2005.

[Was12] Lawrence C Washington. *Elliptic curves: number theory and cryptography*. CRC press, 2012.

[YL06a] Tatu Ylönen and Chris Lonvick. RFC4251, The Secure Shell (SSH) Protocol Architecture. https://tools.ietf.org/html/rfc4251, 2006.

[YL06b] Tatu Ylönen and Chris Lonvick. RFC4252, The Secure Shell (SSH) Authentication Protocol. https://tools.ietf.org/html/rfc4252, 2006.

[YL06c] Tatu Ylönen and Chris Lonvick. RFC4253, The Secure Shell (SSH) Transport Layer Protocol. https://tools.ietf.org/html/rfc4253, 2006.

[YL06d] Tatu Ylönen and Chris Lonvick. RFC4254, The Secure Shell (SSH) Connection Protocol. https://tools.ietf.org/html/rfc4254, 2006.

[Yun04] Moti Yung. Kleptography: The outsider inside your crypto devices (and its trust implications). http://dimacs.rutgers.edu/Workshops/Intellectual/slides/yung.ppt, 2004.

[YY96] Adam Young and Moti Yung. The dark side of "black-box" cryptography or: Should we trust capstone? In *Advances in Cryptology—CRYPTO'96*, pages 89–103. Springer, 1996.

[YY97a] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In *Advances in Cryptology—Eurocrypt'97*, pages 62–74. Springer, 1997.

[YY97b] Adam Young and Moti Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In *Advances in Cryptology—CRYPTO'97*, pages 264–276. Springer, 1997.