Università degli Studi di Trento

Facoltà di Scienze Matematiche Fisiche e Naturali

Dipartimento di Scienze Informatiche

Question Authority

An Inquiry into The Secure Layer

Michele Orrù

Relatore:	Prof. Massimiliano Sala
Correlatore:	Dott. Emanuele Bellini
Controrelatore:	Prof. Giulia Boato

Anno accademico 2012/2013

Università degli Studi di Trento

Facoltà di Scienze Matematiche Fisiche e Naturali
Dipartimento di Scienze Informatiche

Question Authority

An Inquiry into The Secure Layer

Tesi di: Michele Orrù

Relatori: Prof. Massimiliano Sala

Dott. Emanuele Bellini

Controrelatore: Prof. Giulia Boato

Anno accademico 2012/2013

Contents

Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give results in numerical notation, the nature of the processes must consequently be arithmetical and numerical, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation, were provisions made accordingly.

Augusta Ada, Countess of Lovelace

Preface

Even if the basic RSA key generation algorithm is fairly straightforward, it turns out that any software willing to provide such a feature does have to test the candidate key against a substantious number of threats before claiming its security.

The purpose of this project is to examine the TLS protocol, to study in depth the OpenSSL library, and to survey some of the attacks to which a bad key generation is exposed. On the footprint of [1], where most of these attacks have already been analyzed, we are going to describe the mathematical basis of each attack, reason about a possible solution in procedural programming, and finally, give some hints about a distributed version of it.

Besides the pseudocode already available in this document, the project led to the development of a real, open, C implementation consultable at `https://github.com/mmaker/bachelor`.

In addition, the application has then been deployed on the university cluster and pointed against a huge number of websites - Alexa's *top 1 million global websites*. Some of the statistical result extracted from this investigation are later examined in chapter 10.

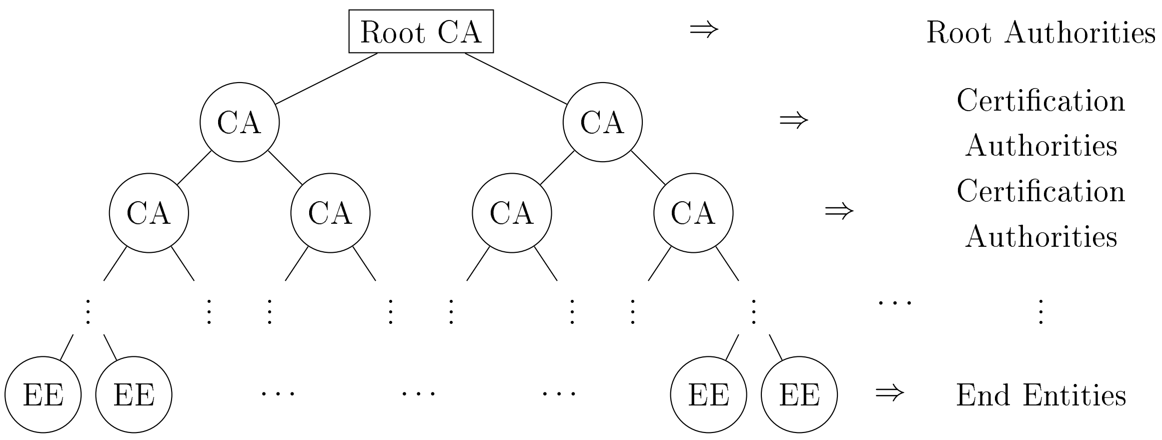
Part I

Prolegomena

The Secure Layer

Transport Layer Security, formerly known as SSL (Secure Socket Layer), aims to bring some security features over a communication channel, specifically providing **integrity** and **confidentiality** of the message, **authenticity** of the server and optionally the client. Many ancient application protocols wrapped themselves to be over TLS/SSL, with the only difference of the “s” appended to the protocol name (such as HTTPs, IMAPs). It is nowadays widely adopted all over the world, becoming the de-facto standard for end-to-end encryption.

Certification Authorities are authorities to whom it is granted the power to *authenticate* the peer. Pragmatically, they are public keys pre-installed on your computer that decide who and who not to trust by employing a digital signature. In order to overcome the proliferation of keys to be distributed, and satisfy the use-case of a mindless user willing to accomplish a secure transaction on the internet, the following, hierarchical trust model proliferated ([2], Fig.2)[1]:



There are two types of authorities: root CAs and intermediate CAs. Root Authorities are the only nodes ultimately considered trustoworthy by the end user. Their

[1] The image is merely esemplificative, there is no boundary to the structure of the tree.

private key is used to sign digital certificates, either to Certificate Authorities, to which is delegated the power of authenticating others, or End Entities, holders of a private key and their corresponding certificate whose identity has been verified.

Upon connecting, the client will check to see if the certificate presented was issued by a CA present in the trust store (root CA); otherwise it will check to see if it has been issued by a trusted CA, and so on until either a trusted CA is found or no trusted authority is found. In the latter case, the connection is aborted.

The protocol is actually a collection of many sub-protocols:

- ***handshake*** protocol, a messaging protocol that allows to *authenticate* the peers, and eventually restore a past encrypted session.
- ***record*** protocol, permitting the encapsulation of higher level protocols, like HTTP and even the next two sub-protocols. It is the fulcrum for all data transfer.
- **alert** protocol, which steps-in at any time from handshake to closure of the session in order to signal a fatal error. The connection will be closed immediately after sending an alert record.
- **changespec** protocol, to notify and negotiate with the receiver that subsequent records will be protected under the just negotiated keys and `Cipher Spec`.

We will now proceed with a brief synopsis of the first two of these protocols, due to their relevant role inside the connection, but will not proceed further, as they were the only two we actually used in our research.

2.1 The `handshake` protocol

As mentioned above, the handshake occurs whenever a machine attempts to start a TLS connection. If there is no session identifier, a new one is being built up; otherwise the client will include the session-id in the initial communication and the server will eventually skip the key agreement phase since it has happened recently[2]. A new session identifier gets built as follows. Once a communication channel over the transport layer has been established, the client sends a hello message, to which the server must respond with a server hello, or else a fatal error will occurr. The above hello messages agree the two parties on the TLS protocol version, compression and encryption methods, and establish a session identifier ([3] §7.3).

[2]"recently" is not well-defined in the standard - it is suggested an upper limit of 24-hours lifetime, but the only actual constraint is that both client and server agree on it.

Following the hello messages, the server will send its certificate, if it is to be authenticated. If the client is happy with it, a RSA or Diffie-Hellmann key exchange is initiated by the client to establish the symmetric key to be used for the ensuing session.

2.2 The `record` protocol

Once the two parties share a common secret, called *premaster secret*, they can generate a new key to be used for symmetric encryption of message, and another for message authentication.

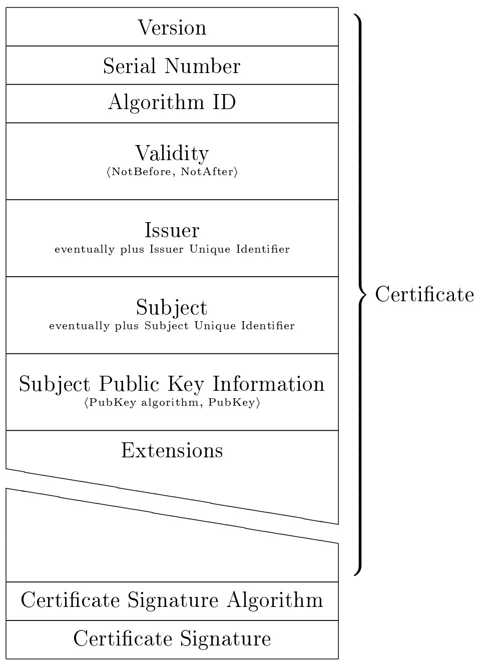
All TLS protocol messages move in records of up to 16K, containing 3 main components: MAC-data, data, and padding.

- MAC-data is no other than the Message Authentication Code over the encrypted *data* sent (SSL performs the encrypt-then-mac mode of operation). It provides **authenticity** and **integrity** of the message.
- Data is the actual message, encrypted after a possible compression.
- The Padding section contains informations about the padding algorithm adopted, and the padding size.

Failure to authentication, or decryption will result in I/O error and a close of the connection.

2.3 What is inside a certificate

SSL certificates employed the X.509 PKI standard, which specifies, among other things, the format for revocation lists, and certificate path validation algorithms.


Version
Serial Number
Algorithm ID
Validity
⟨NotBefore, NotAfter⟩
Issuer
eventually plus Issuer Unique Identifier
Subject
eventually plus Subject Unique Identifier
Subject Public Key Information
⟨PubKey algorithm, PubKey⟩
Extensions
Certificate Signature Algorithm
Certificate Signature
Certificate

It is a pretty old standard, defined in the eighties by the International Telecommunication Union. Born before HTTP, it was initially thought *in abstracto* to be extremely flexible and general[3]. And precisely for this flexibility and its adaptation to the SSL/TLS protocol without a very-well defined structure have been its major flaws: it is still difficult to write good, reliable software parsing a X.509 certificate.

2.4 Remarks among SSL/TLS versions

The first, important difference to point out here is that SSLv2 is no more considered secure. There are known attacks on the primitives adopted (MD5, for example [5]) as well as protocol flaws. SSLv2 would allow a connection to be closed via a not-authenticated TCP segment with the `FIN` flag set ([5] §2). Padding informations are sent in clear, and the payload is not compressed before encrypting, allowing a malicious attacker traffic analysis capabilities [6]. The ciphersuite is negotiated using non-authenticated informations, allowing an attacker to influence the choice of the `Cipher Spec` and weaken the security of the communication [5] §2. Most of these

[3] *"X.509 certificates can contain just anything"* [4]

vulnerabilities have been addressed by the later SSLv3, which introduced compression and protection against truncation attacks. Its standardized twin, TLS 1.0, only differs on the cipher suite and key calculation requirements, strengthen in order to increase the security of the channel [3]. Both SSLv3 and TLS 1.0 have been threatened in 2011 by an attack that could break the same origin policy, known as BEAST. It is not dramatic, and almost any browser now mitigates its spectrum of action.

Even if TLS 1.1, and TLS 1.2 are considered safe as of today, attacks such as CRIME, and lately BREACH constitute a new and valid instance of threat for HTTP compressions mechanisms. However, as their premises go beyond the scope of this document, all these attacks have not been analyzed. For further informations, see `http://breachattack.com/`.

Mathematical Principles

In this chapter we formalize the notation used in the rest of the thesis, and furthermore attempt to discuss and study the elementary functions on which the whole project has been grounded.
The $\ll$ and $\gg$ are respectively used with the meaning of left and right bitwise shift, as usual in computer science.
The *isqrt*() function will be defined in section 3.3, with the acceptation of discrete square root.
The logarithmic log function is assumed to be in base two, i.e. $\log_2$.
The // symbol is the integer division over $\mathbb{N}$, i.e. $a//b = \lfloor \frac{a}{b} \rfloor$, as usual in the python language.
$\mathbb{P} \subset \mathbb{N}$ is the set containing all prime intgers.
The binary operator $\overset{r}{\leftarrow}$, always written as $x \overset{r}{\leftarrow} S$, has the meaning of "pick a uniformly distributed random element x from the set S".
The summation in $\mathbb{F}_2$ is always expressed with the circled plus, i.e. $a \oplus b$.

Definition (Smoothness)**.** *A number n is said to be $\mathcal{B}$-smooth if and only if all its prime factors are contained in $\mathcal{B}$.*

Definition (Quadratic Residue)**.** *An integer a is said to be a* quadratic residue $\bmod n$ *if it is congruent to a perfect square* $\bmod n$*:*

$$x^2 \equiv a \pmod{n}$$

Definition (Legendre Symbol)**.** *The* Legendre Symbol*, often contracted as (a/p) is a function of two integers a and p defined as follows:*

$$(a/p) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p} \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a non-residue modulo } p \end{cases}$$

3.1 Algorithmic Complexity Notation

The notation used to describe asymptotic complexity follows the $\mathcal{O}$-notation, abused under the conventions and limits of MIT's Introduction to Algorithms [7].

Let $\mathcal{O}(g)$ be the asymptotic upper bound of g:

$$\mathcal{O}(g(n)) = \{f(n) : \exists n_0, c \in \mathbb{N} \mid 0 \leq f(n) \leq cg(n)\ \forall n > n_0\}$$

With $f(n) = \mathcal{O}(g(n))$ we actually mean $f(n) \in \mathcal{O}(g(n))$. Moreover, since the the expression "running time" has achieved a certain vogue, we shall sometimes use this term as interchangeable with "complexity", even though imprecise ([8] §1.1.4).

3.2 Euclid's Greatest Common Divisor

Being the greatest common divisor a foundamental algebraic operation in the TLS protocol, OpenSSL implemented it with the following signature:

```c
int BN_gcd(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

The computation proceeds under the well-known Euclidean algorithm, specifically the binary variant developed by Josef Stein in 1961 [9]. This variant exploits some interesting properties of $gcd(a, b)$:

(a) if a, b are even, then $gcd(a, b) = 2gcd(a/2, b/2)$;

(b) if a is even and b is odd, then $gcd(a, b) = gcd(a/2, b)$;

(c) $gcd(a, b) = gcd(a - b, b)$, as in the standard Euclid algorithm;

(d) the sum of two odd numbers is always even.

Both [9] and [7] analyze the running time of the algorithm; [7]'s proof is fairly simpler and proceeds by induction. Anyway, both show that algorithm 1 belongs to the class $\mathcal{O}(\log b)$.

Algorithm 1 OpenSSL 's GCD

```
function GCD($a, b$)
    $k \leftarrow 0$
    while $b \neq 0$ do
        if $a$ is odd then
            if $b$ is odd then                          ▷ by property (c) and (d)
                $a \leftarrow (a - b) \gg 1$
            else                                        ▷ by property (b)
                $b \leftarrow b \gg 1$
            if $a < b$ then $a, b \leftarrow b, a$
        else
            if $b$ is odd then                          ▷ by property (b)
                $a \leftarrow a \gg 1$
                if $a < b$ then $a, b \leftarrow b, a$
            else                                        ▷ by property (a)
                $k \leftarrow k + 1$
                $a, b \leftarrow a \gg 1, b \gg 1$
    return $a \ll k$
```

Unfortunately, there is yet no known parallel solution that significantly improves Euclid's GCD.

3.3 Square Root

Computing the square root is another important building block of the project, though not available in OpenSSL. Apparently, OpenSSL does only provide the discrete square root implementation using the Tonelli/Shanks algorithm, which specifically solves in x the equation $x^2 = a \pmod p$, with $p \in \mathbb{P}$:

```
BIGNUM* BN_mod_sqrt(BIGNUM* x, const BIGNUM* a, const BIGNUM* p,
                    const BN_CTX* ctx);
```

Instead, we are interested in finding the the pair $\langle x, r\rangle \in \mathbb{N}^2$ such that $x^2 + r = n$, that is, the integer part of the square root of a natural number and its rest. Hence, we did come out with our specific implementation, first using Bombelli's algorithm, and later with the one of Dijkstra. Both are going to be discussed below.

Unless otherwise specified, in the later pages we use $\sqrt{n}$ with the usual meaning "the half power of n", while with $x, r = isqrt(n)$ we mean the pair just defined.

Bombelli's Algorithm dates back to the XVI century, and approaches the problem of finding the square root by using continued fractions. Unfortunately, we weren't able to fully assert the correctness of the algorithm, since the original document [10] presents a difficult, inconvenient notation. Though, for completeness' sake, we report in table 2 the pseudocode adopted and tested for its correctness.

Algorithm 2 Square Root: Bombelli's algorithm

```
function SQRT($n$)
    $i \leftarrow 0$;   $g \leftarrow \{\}$
    while $n > 0$ do                          ▷ take pairs of digits and store them in $g$
        $g_i \leftarrow n \pmod{100}$
        $n \leftarrow n//100$
        $i \leftarrow i + 1$
    $x \leftarrow 0$;   $r \leftarrow 0$
    for $j = i - 1$ downto 0 do
        $r = 100r + g_i$                      ▷ take next pair
        for $d = 0$ to 9 do                   ▷ find gratest multiplier $d$
            $y' \leftarrow d(20x + d)$
            if $y' > r$ then break
            else $y \leftarrow y'$
        $r \leftarrow r - y$
        $x \leftarrow 10x + d - 1$            ▷ $d$ is the next digit
    return $x, r$
```

For each digit of the result, we perform a subtraction, and a limited number of multiplications. This means that the complexity of this solutions belongs to $\mathcal{O}(\log n \log n) = \mathcal{O}(\log^2 n)$.

Remark 3.3.1. Note that Bombelli actually has found a solution in x for a slightly different equation than the one we initially formulated. Specifically, he found the pair $\langle x, r\rangle$ such that $(x+r)^2 = a$, where x is the mantissa, while r is the decimal part. For our purpose this change is irrelevant: we just need to be able to distinguish perfect squares, and thus assert that r is zero.

Dijkstra's Algorithm can be found in [11], §8, p.61. There, Dijkstra presents an elightning process for the computation of the square root, making only use of binary shift and algebraic additions. Specifically, the problem attempts to find, given a natual n, the integer a that establishes:

$$a^2 \leq n \wedge (a+1)^2 > n \tag{3.1}$$

Take now the pair $\langle a = 0, b = n + 1\rangle$, and consider the inverval $[a, b[$. We would like to reduce the distance between the upper bound b and the lower bound a, while holding the guard 3.1:

$$a^2 \leq n \ \wedge \ b > n$$

The speed of convergence is determined by the choice of the distance d, which analougously to the dicotomic search problem, is optimal when $d = (b - a)//2$.

Algorithm 3 Square Root: an intuitive, naïve implementation

```
1: function SQRT($n$)
2:     $a \leftarrow 0$;    $b \leftarrow n + 1$
3:     while $a + 1 \neq b$ do
4:         $d \leftarrow (b - a)//2$
5:         if $(a + d)^2 \leq n$ then $a \leftarrow a + d$        ▷ increment left bound
6:         else if $(b - d)^2 > n$ then $b \leftarrow b - d$     ▷ decrement right bound
7:     return $a, a^2 - n$
```

Now optimization proceeds with the following change of variables:

a) $c = b - a$,

b) $p = ac$,

c) $q = c^2$,

d) $r = n - a^2$;

resulting into algorithm 4. For any further details, the reference is still [11].

Algorithm 4 Square Root: final version

```
1: function SQRT($n$)
2:     $p \leftarrow 0$;    $q \leftarrow 1$;    $r \leftarrow n$
3:     while $q \leq n$ do $q \leftarrow q \ll 2$
4:     while $q \neq 1$ do
5:         $q \leftarrow q \gg 2$
6:         $h \leftarrow p + q$
7:         $p \leftarrow q \ll 1$
8:         $h \leftarrow 2p + q$
9:         if $r \geq h$ then
10:            $p \leftarrow p + q$
11:            $r \leftarrow r - h$
12:    return $p, r$
```

A fair approximation of the magnitude of the Dijkstra algorithm can be studied by looking at the pseudocode in 3. Exactly as in the dicotomic search case, we split the interval $[a, b]$ in half on each step, and choose whether to take the leftmost or the rightmost part. This results in $log(n + 1)$ steps. During each iteration, instead, as we have seen in 4 we just apply summations and binary shifts, which are upper bounded by $\mathcal{O}(\log n/2)$. Thus, the order of magnitude belongs to $\mathcal{O}\left(\log^2 n\right)$.

Even if both algorithms presented have *asymptotically* the same complexity, we believe that adopting the one of Dijkstra has lead to a pragmatic, substantial performance improvement.

3.4 The RSA Cipher

The RSA cryptosystem, invented by Ron Rivest, Adi Shamir, and Len Adleman [12], was first published in August 1977's issue of *Scientific American.* In its basic version, this *asymmetric* cipher works as follows:

- choose a pair $\langle p, q\rangle$ of *random prime* numbers; let N be the product of the two, $N = pq$, and call it *public modulus*;
- choose a pair $\langle e, d\rangle$ of *random* numbers, both in $\mathbb{Z}^*_{\varphi(N)}$, such that one is the multiplicative inverse of the other, $ed \equiv 1 \pmod{\varphi(N)}$ and $\varphi(N)$ is Euler's totient function;

Now, call $\langle N, e\rangle$ *public key*, and $\langle N, d\rangle$ *private key*, and let the encryption function $E(m)$ be the e-th power of the message m:

$$E(m) = m^e \pmod{N} \tag{3.2}$$

while the decryption function $D(c)$ is the d-th power of the ciphertext c:

$$D(c) = c^d \equiv E(m)^d \equiv m^{ed} \equiv m \pmod{N} \tag{3.3}$$

that, due to Fermat's little theorem, is the inverse of E.

From now on, unless otherwise specified, the variable $N = pq$ will always refer to the public modulus of a generic RSA keypair, with p, q being the two primes factorizing it, such that $p > q$. Again, e, d will respectively refer to the public exponent and the private exponent.

Part II

Questions

Fermat's factorization method

Excluding the trial division, Fermat's method is the oldest known systematic method for factorizing integers. Even if its algorithmic complexity is not among the most efficient, it holds still a practical interest whenever the two primes are sufficiently close. Indeed, [13] §B.3.6 explicitly recommends that $|p-q| \geq \sqrt{N}2^{-100}$ for any key of bitlength $1024,\ 2048,\ 3072$ in order to address this kind of threat.
The basic idea is to attempt to write N as a difference of squares,

$$x^2 - N = y^2 \tag{4.1}$$

So, we start by $x = \lceil \sqrt{N} \rceil$ and check that $x^2 - N$ is a perfect square. If it isn't, we iteratively increment x and check again, until we find a pair $\langle x, y \rangle$ satisfying equation 4.1. Once found, we claim that $N = pq = (x+y)(x-y)$; it is indeed true that, if we decompose $x^2 - y^2$ as difference of squares, then it is immediately clear that $x + y \mid N \ \wedge\ x - y \mid N$, and that both are non-trivial divisors.

Complexity [14] contains a detailed proof for the complexity of this algorithm, which is $\mathcal{O}\left(\frac{(1-k)^2}{2k}\sqrt{N}\right)$ $, 0 < k < 1$. We summarize it down below here to better clarify the limits of this algorithm.

Proof. Since, once we reach the final step x_f it holds $N = pq = x_f^2 - y_f^2$, the number of steps required to reach the result is:

$$\begin{aligned} x_f - \sqrt{N} &= \frac{p+q}{2} - \sqrt{N} \\ &= \frac{p + \frac{N}{p}}{2} - \sqrt{N} \\ &= \frac{(\sqrt{N} - p)^2}{2p} \end{aligned}$$

If we finally suppose that $p = k\sqrt{N},\ 0 < k < 1$, then the number of cycles becomes $\frac{(1-k)^2}{2k}\sqrt{N}$. □

Remark 4.0.1. Note that, for the algorithm to be effective, the two primes must be "really close" to $\sqrt{N}$. As much as the lowest prime gets near to 1, the ratio $\frac{(1-k)^2}{2k}$ becomes larger, until the actual magnitude of this factorization method approaches $\mathcal{O}(N)$.

4.1 An Implementation Perspective

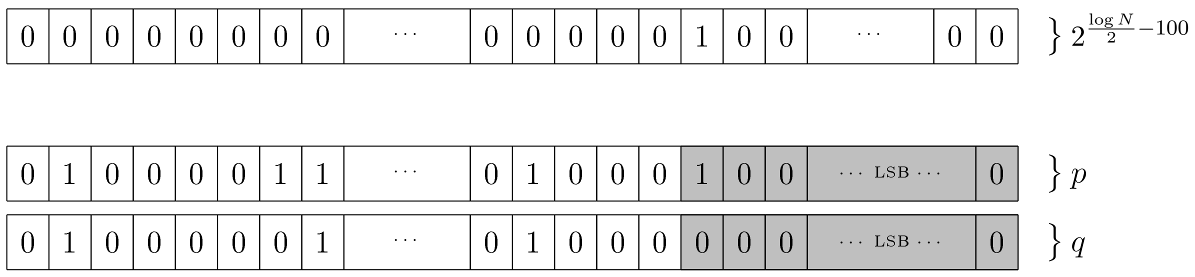
At each iteration, the i−th state is hold by the pair $\langle x, x^2 \rangle$.
The later step, described by $\langle x+1, (x+1)^2 \rangle$ can be computed efficiently considering the square of a binomial: $\langle x+1, x^2+2x+1 \rangle$. The upper-bound, instead, is reached when $\Delta = p - q = x + y - x + y = 2y > 2^{-100}\sqrt{N}$.

Algorithm 5 presents a simple implementation of this factorization method, taking into account the small optimizations aforementioned.

Algorithm 5 Fermat Factorization

```
 1: function FERMAT($N, e$)
 2:     $x \leftarrow \lfloor \sqrt{N} \rfloor$
 3:     $x' \leftarrow x \cdot x$
 4:     repeat
 5:         $x' \leftarrow x' + 2x + 1$
 6:         $x \leftarrow x + 1$
 7:         $y, rest \leftarrow isqrt(x' - N)$
 8:     until $rest \neq 0$ and $y < \frac{\sqrt{N}}{2^{101}}$        ▷ i.e., 4.1 holds?
 9:     if $rest = 0$ then
10:         $p \leftarrow x + y$
11:         $q \leftarrow x - y$
12:         return $p, q$
13:     else
14:         return nil
```

How to chose the upper limit? Our choice of keeping straight with the limits of the standard is a mere choice of commodity: we are interested in finding public keys not respecting the standard. Though, it is worth noting that what this limit *states* is that at least one of the most significant 100 bits should be different between the two primes:


0 0 0 0 0 0 0 0 ··· 0 0 0 0 0 1 0 0 ··· 0 0
$\} 2^{\frac{\log N}{2}-100}$
0 1 0 0 0 0 1 1 ··· 0 1 0 0 0 1 0 0 ··· LSB ··· 0
$\} p$
0 1 0 0 0 0 0 1 ··· 0 1 0 0 0 0 0 0 ··· LSB ··· 0
$\} q$

For example, in the case of a RSA key 1024, the binary difference between p and q has to be greater than 2^{412}, which means that, excluding corner-cases where the remainder is involved, there must be at least one difference in the top 100 most significant bits for the key to be considered safe.

4.2 Thoughts about a parallel solution

At first glance we might be willing to split the entire interval $\{\lceil\sqrt{N}\rceil, \ldots, N-1\}$ in equal parts, one per each node. However, this would not be any more efficient than the trial division algorithm, and nevertheless during each single iteration, the computational complexity is dominated by the square root *isqrt*() function, which belongs to the class $\mathcal{O}\left(\log^2 N\right)$, as we saw in section 3.3. Computing separatedly x^2 would add an overhead of the same order of magnitude $\mathcal{O}\left(\log^2 N\right)$, and thus result in a complete waste of resources.

Wiener's cryptanalysis method

Wiener's attack was first published in 1989 as a result of cryptanalysis on the use of short RSA secret keys [15]. It exploited the fact that it is possible to find the private key in *polynomial time* using continued fractions expansions whenever a good estimate of the fraction $\frac{e}{N}$ is known. More specifically, given $d < \frac{1}{3}\sqrt[4]{N}$ one can efficiently recover d only knowing $\langle N, e \rangle$.

The scandalous implication behind Wiener's attack is that, even if there are situations where having a small private exponent may be particularly tempting with respect to performance (for example, a smart card communication with a computer), they represent a threat to the security of the cipher. Fortunately, [15] §9 presents a couple of precautions that make a RSA key-pair immune to this attack, namely (i) making $e > \sqrt{N}$ and (ii) $gcd(p-1, q-1)$ large.

5.1 Background on Continued Fractions

Let us call *continued fraction* any expression of the form:

$$a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4 + \dots}}}}$$

Consider now any *finite continued fraction*, conveniently represented with the sequence $\langle a_0, a_1, a_2, a_3, \; \dots, a_n \rangle$. Any number $x \in \mathbb{Q}$ can be represented as a finite continued fraction, and for each $i < n$ there exists a fraction ${}^{h}\!/_{k}$ approximating x. By definition, each new approximation

$$\begin{bmatrix} h_i \\ k_i \end{bmatrix} = \langle a_0, a_1, \; \dots, a_i \rangle$$

is recursively defined as:

$$\begin{cases} a_{-1} = 0 \\ a_i = h_i // k_i \end{cases} \qquad \begin{cases} h_{-2} = 0 \\ h_{-1} = 1 \\ h_i = a_i h_{i-1} + h_{i-2} \end{cases} \qquad \begin{cases} k_{-2} = 1 \\ k_{-1} = 0 \\ k_i = a_i k_{i-1} + k_{i-2} \end{cases} \tag{5.1}$$

Among the prolific properties of such objects, Legendre in 1768 discovered that, if a continued fraction $f' = \frac{\theta'}{\kappa'}$ is an underestimate of another one $f = \frac{\theta}{\kappa}$, i.e.

$$|f - f'| = \delta \tag{5.2}$$

then for a δ sufficiently small, f' is *equal* to the n-th continued fraction expansion of f, for some $n \geq 0$ ([16] §2). Formally,

Theorem (Legendre). *If* $f = \frac{\theta}{\kappa}$, $f' = \frac{\theta'}{\kappa'}$ *and* $\gcd(\theta, \kappa) = 1$, *then*

$$\left| f' - \frac{\theta}{\kappa} \right| < \delta = \frac{1}{2\kappa^2} \quad \textit{implies that} \quad \begin{bmatrix} \theta' \\ \kappa' \end{bmatrix} = \begin{bmatrix} \theta_n \\ \kappa_n \end{bmatrix}, \quad \textit{for some } n \geq 0 \tag{5.3}$$

Two centuries later, first Wiener [15] and later Dan Boneh [1] leveraged this theorem in order to produce an algorithm able to recover f, having f'. The *continued fraction algorithm* is the following:

(i) generate the next a_i of the continued fraction expansion of f';
(ii) use 5.1 to generate the next fraction ${}^{h_i}/_{k_i}$ equal to $\langle a_0, a_1, \dots, a_{i-1}, a_i \rangle$
(iii) check whether ${}^{h_i}/_{k_i}$ is equal to f

5.2 Continued Fraction Algorithm applied to RSA

As we saw in 3.4, by construction the two exponents are such that $ed \equiv 1 \pmod{\varphi(N)}$. This implies that there exists a $k \in \mathbb{N} \mid ed = k\varphi(N) + 1$. This can be formalized to be the same problem we formalized in 5.3:

$$\begin{aligned} ed &= k\varphi(N) + 1 \\ \left| \frac{ed - k\varphi(N)}{d\varphi(N)} \right| &= \frac{1}{d\varphi(N)} \\ \left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| &= \frac{1}{d\varphi(N)} \end{aligned}$$

Now we proceed by substituting $\varphi(N)$ with N, since for large N, one approximates the other. We consider also the difference of the two, limited by $|\cancel{N} + p + q - 1 - \cancel{N}| < 3\sqrt{N}$. For the last step, remember that $k < d < {}^{1}/_{3}\sqrt[4]{N}$:

$$
\begin{aligned}
\left|\frac{e}{N}-\frac{k}{d}\right| &= \left|\frac{ed-kN}{Nd}\right| \\
&= \left|\frac{\cancel{ed}-kN-\cancel{k\varphi(N)}+k\varphi(N)}{Nd}\right| \\
&= \left|\frac{1-k(N-\varphi(N))}{Nd}\right| \\
&\leq \left|\frac{3k\sqrt{N}}{Nd}\right| = \frac{3k}{d\sqrt{N}} < \frac{3({}^1\!/_3\ \sqrt[4]{N})}{d\sqrt{N}} = \frac{1}{d\sqrt[4]{N}} < \frac{1}{2d^2}
\end{aligned}
$$

This demonstrates that the hypotesis of 5.3 is satisfied, and allows us to proceed with the continued fraction algorithm to converge to a solution [1].

We start by generating the $\log N$ continued fraction expansions of $\frac{e}{N}$, and for each convergent $\frac{k}{d}$, which by contruction is already at the lowest terms, we verify if it produces a factorization of N. First we check that $\varphi(N) = \frac{ed-1}{k}$ is an integer. Then we solve 5.4 in x in order to find p, q:

$$x^2 - (N - \varphi(N) + 1)x + N = 0 \tag{5.4}$$

The above equation is constructed so that the x coefficient is the sum of the two primes, while the constant term N is the product of the two. Therefore, if $\varphi(N)$ has been correctly guessed, the two roots will be p and q.

5.3 An Implementation Perspective

The algorithm is pretty straightforward by itself: we just need to apply the definitions provided in 5.1 and test each convergent until $\log N$ iterations have been reached. A Continued fraction structure may look like this:

```
typedef struct cf {
  bigfraction_t fs[3];  /* holding h_i/k_i, h_i-1/k_i-1, h_i-2/k_i-2 */
  short i;              /* cycling in range(0, 3) */
  bigfraction_t x;      /* pointer to the i-th fraction in fs */
  BIGNUM* a;            /* current a_i */
  BN_CTX* ctx;
} cf_t;
```

where `bigfraction_t` is just a pair of `BIGNUM`s $\langle h_i, k_i \rangle$. Whenever we need to produce a new convergent, we increment $i \pmod 3$ and apply the given definitions. The fresh convergent must be tested with very simple algebraic operations. It is worth noting here that 5.4 can be solved using the reduced discriminant formula, as p, q are odd primes:

$$\Delta = \left(\frac{N - \varphi(N) + 1}{2}\right)^2 - N$$
$$x_{\langle p,q \rangle} = -\frac{N - \varphi(N) + 1}{2} \pm \sqrt{\Delta}$$

Assuming the existence of the procedures `cf_init`, initializing a continued fraction structure, and `cf_next` producing the next convergent, we provide an algorithm for attacking the RSA cipher via Wiener:

Algorithm 6 Wiener's Attack

```
1: function WIENER($N, e$)
2:     $f \leftarrow$ cf_init($e, N$)
3:     for $\lceil \log N \rceil$ times do
4:         $k, d \leftarrow$ cf_next($f$)
5:         if $k \nmid ed - 1$ then continue
6:         $\varphi(N) \leftarrow (ed - 1) \ // \ k$
7:         if $\varphi(N)$ is odd then continue
8:         $b \leftarrow (N - \varphi(N) + 1) \gg 1$
9:         $\Delta, r \leftarrow isqrt(b^2 - N)$
10:        if $r \neq 0$ then continue
11:        $p \leftarrow b + \Delta$
12:        $q \leftarrow b - \Delta$
13:        break
14:    return $p, q$
```

Parallelism Parallel implementation of this specific version of Wiener's Attack is difficult, because the inner loop is inherently serial. At best, parallelism could be employed to split the task into a *constructor* process, building the f_n convergents, and many *consumers* receiving each convergent to be processed seperatedly. The first one arriving to a solution, broadcasts a stop message to the others.

Pollard's $p-1$ factorization method

Pollard's $p-1$ method, first published in [17] §4, takes relevance whenever the predecessor of any of the two primes can be seen as the product of many, relatively small, prime powers. The idea employed here is that if we find a Q satisfying $p-1 \mid Q$ then, due to Fermat's little theorem, $p \mid a^Q - 1$ and thus p can be found by applying $gcd(N, a^Q - 1) = p$. The same applies to the other prime q.

Now, the original problem has been replaced by the problem of generating a number Q sufficiently large to be a multiple of $p-1$. It would be fairly easy to set Q to be the factorial of a number K greater than the highest prime power factorizing $p-1$. That is to say, having $p-1 = \pi_0^{e_0} \cdots \pi_k^{e_k}$ where $k \geq 1$ and each $\pi_i \in \mathbb{P}$ so that $\pi_0^{e_0} < \cdots < \pi_k^{e_k} \leq K$ implies that $Q = K!$ is divided by *all* factors of $p-1$ and so $p-1$ itself divides Q.
Though, at this point all primes would be over-represented. What we actually need is the least common multiple of all numbers $\leq K$. We could easily obtain it via the popular formula $lcm(a,b) = \frac{ab}{\gcd(a,b)}$ and pay $\mathcal{O}(3 \log N)$ on each step, but there is something even better.

If we are provided with a pool of primes $\mathcal{P}$, we can guess a valuable Q manually tuning the powers of each prime, to the bounds strictly needed. We proceed as follows:

(i) take a random, initial b, and call it *base*;

(ii) take the i-th prime in $\mathcal{P}$, and call it π;

(iii) iteratively update the value of b via

$$b \leftarrow b^{\pi} \pmod{N} \tag{6.1}$$

for $e = \lceil \frac{\log N}{\log \pi} \rceil$ times. In this way we are sure to have taken the greatest prime power not exceeding N's bits.

(iv) select $Q = b - 1 \pmod{N}$ and check the *gcd* with N, hoping this leads to one of the two prime factors:

$$g = gcd(Q, N), \quad 1 < g < N\,. \tag{6.2}$$

If so, than we have finished, since g itself and $\frac{N}{g}$ are the two primes factorizing the public modulus. Otherwise, if $g = 1$ we go back to to (ii), since $p - 1 \nmid Q$ yet; if $g = N$ start back from scratch, as $pq \mid g$.

Complexity Let m be the maximum prime number in $\mathcal{P}$. Consider now the computational cost of each single step discussed above:

- computing the gcd of step (iv) costs $\mathcal{O}(\log N)$ each time is being performed;
- computing the e-th power of π in step (iii) is upperly bounded by the multiplication for $\log m$ bits;
- computing the exponentiation in b^{π} acts instead over $\pmod N$, hence belongs to $\mathcal{O}(\log N)$;
- all remaining steps are trivially $\mathcal{O}(1)$;

Therefore the time complexity for the above algorithm is $\mathcal{O}\left(m \log(m) \log^2(N)\right)$.
In the worst-case scenario, where p, q are *safe* primes[1], that becomes

$$\mathcal{O}\left(\sqrt{N} \log(\sqrt{N}) \log^2(N)\right) = \mathcal{O}\left(\sqrt{N} \log^3 N\right)$$

- assuming the prime pool $\mathcal{P}$ is big enough.
In the best-case scenario instead, where p is a Fermat prime, i.e. of the form $2^n + 1$, the computation is incredibly fast:

$$\mathcal{O}\left(\log^2 N\right)$$

Remark 6.0.1. What's crucial about this algorithm is the tradeoff that must be undertaken in the choice of $\mathcal{P}$. A pool with few primes would visibly reduce the running time of Pollard's $p-1$, but at the same time also the probability to reach a solution would be reduced.

6.1 An Implementation Perspective

Algorithm 7 illustrates a naïve implementation of Pollard's $p-1$ algorithm. Various refinements are possible. Because gcds are more expensive than multiplication $\pmod N$, we could for example avoid most of those computation by accumulating values of Q [17].

This means keeping a counter s, and multiply each new Q by itself in line 6. This way the $\gcd(Q, N) = g$ can be checked on regular intervals, for example whenever $s \mid 100$. If $g = 1$,we could backup the current state $\langle \pi, b \rangle$; if $g = N$, we roll back to the latest state b and π, and then proceed one by one. Even if this change might seem irrelevant as the asymptotic behaviour is the same for gcd and multiplication, reality shows that the latter is more efficient by a constant factor. There will be a chance to see this improvement in chapter 8, where the same trick is adopted to speed up the execution.

[1] A prime p is said to be a safe prime iff $\frac{p-1}{2}$ is prime.

Parallelism As the inner loop of the $p-1$ algorithm is intrinsecally serial, it is difficult to propose a good parallel solution. At best, parallelism can speed up the multiple precision operations by a small factor, but this topic goes beyond the scope of this study [18].

Still, it is possible to run the same instance of the proposed algorithm over multiple nodes, hoping that a different random base would pay back. Though, if both primes are safe, there is no way this soulution could improve the running time.

Algorithm 7 Pollard's $p-1$

```
Require: $\mathcal{P}$, a prime pool
 1: function FACTORIZE($N, b$)
 2:     for $\pi$ in $\mathcal{P}$ do                          ▷ step (ii)
 3:         $e \leftarrow \log \sqrt{N} // \log \pi$
 4:         for $e$ times do                                 ▷ step (iii)
 5:             $b \leftarrow b^{\pi} \pmod{N}$
 6:             $Q \leftarrow b - 1$
 7:             $g \leftarrow \gcd(Q, N)$                      ▷ step (iv)
 8:             if $g = N$ then return nil
 9:             else if $g > 1$ then return g
10:
11: function POLLARD($N, e$)
12:     repeat
13:         $b \overset{r}{\leftarrow} \mathbb{N}_{>2}$        ▷ step (i)
14:         $p \leftarrow$ FACTORIZE($N, b$)
15:     until $p \neq$ nil
16:     $q \leftarrow N // p$
17:     return $p, q$
```

Williams' $p + 1$ factorization method

Analogously to Pollard's $p - 1$ factorization described in chapter 6, this method will allow the determination of the divisor p of a number N, if p is such that $p+1$ has only small prime power divisors. This method was presented in [19] together with the results of the application of this method to a large number of composite numbers.

7.1 Background on Lucas Sequences

Let us call *Lucas Sequence* the recurrence relation with parameters τ, υ

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_n = \tau U_{n-1} - \upsilon U_{n-2} \end{cases} \qquad \begin{cases} V_0 = 2 \\ V_1 = \tau \\ V_n = \tau V_{n-1} - \upsilon V_{n-2} \end{cases}$$

For respectively different values of τ, υ, Lucas Sequences have specific names:

- $U(\tau = 1, \upsilon = -1)$ *Fibonacci numbers*;
- $V(\tau = 1, \upsilon = -1)$ *Lucas numbers*;
- $U(\tau = 3, \upsilon = 2)$ *Mersenne numbers.*

For our purposes, U_n is not necessary, and $\upsilon = 1$.[1] In order to simplify any later theorem, we just omit U_n, and assume $\upsilon = 1$. Therefore, the latter expression becomes:

$$\begin{cases} V_0 = 2 \\ V_1 = \tau \\ V_n = \tau V_{n-1} - V_{n-2} \end{cases} \tag{7.1}$$

Two foundamental properties interpolate terms of Lucas Sequences, namely *addition* and *duplication* formulas:

$$V_{n+m} = V_n V_m - V_{m-n} \tag{7.2}$$
$$V_{2n} = V_n^2 - 2 \tag{7.3}$$

[1] Williams justifies this choice stating that choosing to compute a U_n sequence is far more computationally expensive than involving V_n; for what concerns υ, that simplifies Lehmer's theorem with no loss of generality. For further references, see [19] §3.

All these identities can be verified by direct substitution with 7.1. What is interesting about the ones of above, is that we can exploit them to efficiently compute the product V_{hk} if we are provided with V_k by considering the binary representation of the number h. In other words, we can consider each bit of h, starting from second most significant one: if it is zero, we compute $\langle V_{2k}, V_{(2+1)k}\rangle$ using 7.3 and 7.2 respectively; otherwise we compute $\langle V_{(2+1)k}, V_{2(k+1)}\rangle$ using 7.2 and 7.3.

Notice that $V_{(2+1)k} = V_{2k+k} = V_{2k}V_k - V_k$.

Algorithm 8 Lucas Sequence Multiplier

```
1: function LUCAS($V, a, N$)
2:     $V_1 \leftarrow V$
3:     $V_2 \leftarrow V^2 - 2 \pmod N$
4:     for each bit $b$ in $a$ to right of the MSB do
5:         if $b$ is 0 then
6:             $V_2 \leftarrow V_1V_2 - V \pmod N$        ▷ by addition
7:             $V_1 \leftarrow V_1^2 - 2 \pmod N$         ▷ by duplication
8:         else if $b$ is 1 then
9:             $V_1 \leftarrow V_1V_2 - V \pmod N$        ▷ by addition
10:            $V_2 \leftarrow V_2^2 - 2 \pmod N$         ▷ by duplication
11:    return $V_1$
```

Finally, we need the following ([19] §2):

Theorem (Lehmer). *Let Δ be $\tau^2 - 4$; if p is an odd prime and the Legendre symbol $\varepsilon = (\Delta/p)$, then:*

$$V_{(p-\varepsilon)m} \equiv 2 \pmod p$$

Remark 7.1.1. From number theory we know that the probability that $P(\varepsilon = -1) = {}^1/_2$. There is no reason to restrict ourselves to $(\Delta/p) = -1$. In the alternative case of $\varepsilon = 1$, the factorization yields the same factors as Pollard's $p-1$ method, but slowerly. For this reason it is advisable to first attempt the attack presented in the previous chapter [19]whenever we look up for a $p-1$ factorization.

7.2 Dressing up

At this point the factorization proceeds just by substituting the exponentiation and Fermat's theorem with Lucas sequences and Lehmer's theorem introduced in the preceeding section. If we find a Q satisfying $p+1 \mid Q$ or $p-1 \mid Q$ then, due to Lehmer's theorem $p \mid V_Q - 2$ and thus $\gcd(V_Q - 2, N)$ is a non-trivial divisor of N.

(i) Take a random, initial τ and let it the *base* V_1.

(ii) Take the i-th prime in the pool $\mathcal{P}$, and call it π;

(iii) assuming the current state is V_k, compute the successive terms of the sequence using additions and multiplications formula, until you have $V_{\pi k}$.

(iv) just like with the Pollard $p-1$ method, repeat step (iii) for $e = \lceil \frac{\log N}{\log \pi} \rceil$ times;

(v) select $Q = V_k - 2 \pmod N$ and check the *gcd* with N, hoping this leads to one of the two prime factors:

$$g = gcd(Q, N), \quad 1 < g < N\,. \tag{7.4}$$

If so, than we have finished, since g itself and $\frac{N}{g}$ are the two primes factorizing the public modulus. Otherwise, if $g = 1$ we go back to to (ii), since $p - 1 \nmid Q$ yet; if $g = N$ start back from scratch, as $pq \mid g$.

Algorithm 9 Williams $p+1$ factorization

```
Require: $\mathcal{P}$, the prime pool
1: function FACTORIZE($N, \tau$)
2:     $V \leftarrow \tau$
3:     for $\pi$ in $\mathcal{P}$ do                                  ▷ step (i)
4:         $e \leftarrow \log\sqrt{N} // \log\pi$
5:         for $e$ times do
6:             $V \leftarrow$ LUCAS($V, \pi, N$)                      ▷ step (ii)
7:             $Q \leftarrow V - 2$
8:             $g \leftarrow \gcd(Q, N)$                               ▷ step (iii)
9:             if $g = 1$ then return nil
10:            else if $g > 1$ then return $g, N//g$
```

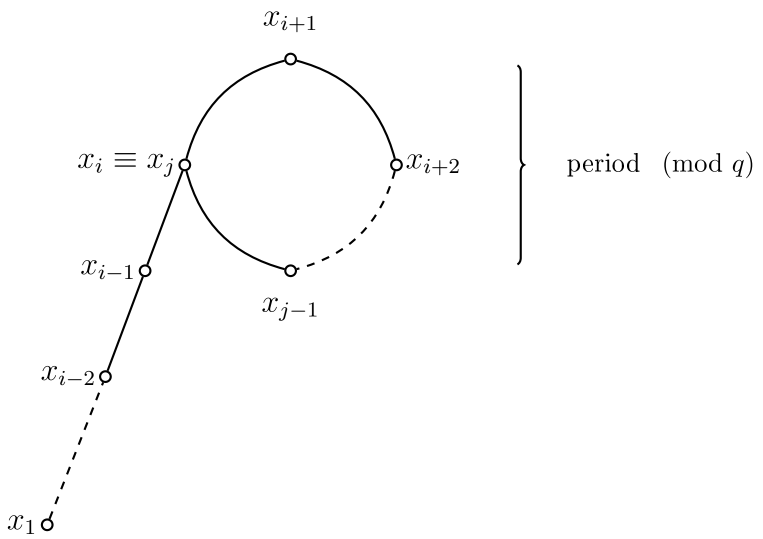
Pollard's ρ factorization method

A *Monte Carlo* factorization method, published by J. M. Pollard in [20], consists into identifying a periodically recurrent sequence of integers within a random walk (mod N) that could leak one of the two factors.

Consider a function f from $\mathcal{S}$ to $\mathcal{S}$, where $\mathcal{S} = \{0, 1, \ldots, q-1\}$ and $q \in \mathbb{P}$. Let s be a random element in $\mathcal{S}$, and consider the sequence

$$s,\ f(s),\ f(f(s)),\ \ldots$$

Since f acts over a finite set, it is clear that this sequence must eventually repeat, and become cyclic. We might diagram it with the letter ρ, where the tail represent the aperiodic part, or *epacts*, and the oval the cyclic part, or *period.*



Now, consider $N = pq$. Let $F(x)$ be any function generating pseudorandom integers $\langle x_1, x_2, \ldots\rangle$, and let $f(x) = F(x) \pmod q$. As we said above, without any luck, there will be a pair $\langle x_i, x_j\rangle$ generated by F such that $x_i \equiv x_j \pmod q$, but $x_i \neq x_j$.

Therefore, in order to factorize N, we proceed as follows: starting from a random s, we iteratively apply F reduced modulo N. Whenever we find a period, if $\gcd(x_i - x_j, N) > 1$ then we found a non-trivial factor of N.

Choosing the function Ideally, F should be easily computable, but at the same time random enough to reduce as much as possible the epacts [8] §5.2.1. Any quadratic function $F(x) = x^2 + b$ should be enough[1], provided that $b \in \mathbb{N} \setminus \{0, 2\}$. For example, [20] uses $x^2 - 1$, meanwhile we are going to choose $F(x) = x^2 + 1$.

Finding the period The trivial way to discover a period would be to test x_i with all $x_j, \quad j < i$. Though, in [9] §3.1, Knuth gives a simple and elegant algorithm, attributed to Floyd, for finding a multiple of the period. This algorithm is the same one finally adopted by Pollard in [20].

Theorem (Floyd). *Given an* ultimately periodic *sequence, in the sense that there exists numbers λ and μ for which the values:*

$$X_0, X_1, \ldots, X_\mu, \ldots, X_{\mu+\lambda-1}$$

are distinct, but $X_{n+\lambda} = X_n$ when $n \geq \mu$, then there exists an $\mu < n < \mu + \lambda$ such that $X_n = X_{2n}$.

Proof. First, if $X_n = X_{2n}$, then the sequence is obviously periodic from X_{2n} onward, possibly even earlier. Conversely, it is true that $X_n = X_m \quad (n \geq \mu)$ for $m = n + k\lambda, \quad k \in \mathbb{N}$. Hence, there will eventually be an n such that $X_n = X_{2n}$ if and only if $n - \mu$ is a multiple of λ. The first such value happens for $n = (\lambda + 1)\lfloor {}^{\mu}/_{\lambda} \rfloor$. $\square$

The immediate consequence of this is that we can find a collision simply by checking $\gcd(x_{2i} - x_i, N) > 1$ for incremental i.

Brent's Improvement In 1979, Brent discovered an entire family of cycle-finding algorithms whose optimal version resulted to be 36% faster than Floyd's one [21]. Instead of looking for the period of the sequence using $x_{2i} - x_i$, Brent considers $|x_j - x_{2^k}|$ for $3 \cdot 2^{k-1} < j \leq 2^{k+1}$, resulting in fewer operations required by the algorithm. Pragmatically, this boils down to compare:

$k = 0$	$j \in \{1 + 1 \ldots 2\}$	$\lvert x_1 - x_2 \rvert$
$k = 1$	$j \in \{3 + 1 \ldots 4\}$	$\lvert x_2 - x_4 \rvert$
$k = 2$	$j \in \{6 + 1 \ldots 8\}$	$\lvert x_4 - x_7 \rvert$, $\lvert x_4 - x_8 \rvert$
$k = 3$	$j \in \{12 + 1, \ldots 16\}$	$\lvert x_8 - x_{13} \rvert$, $\lvert x_8 - x_{14} \rvert$, ..., $\lvert x_8 - x_{16} \rvert$
$k = 4$	$j \in \{24 + 1, \ldots 32\}$	$\lvert x_{16} - x_{25} \rvert$, ..., $\lvert x_{16} - x_{32} \rvert$
$\vdots$	$\vdots$	$\vdots$

A Pollard's ρ variant that implements Brent's cycle-finding algorithm instead of Floyd's one runs around 25% faster on average [21].

[1] Note that this has been only empirically verified, and so far not been proved ([14], p. 177)

8.1 Complexity

[14] presents a nice proof of the *average* complexity of this algorithm, based on the birthday paradox.

The Birthday Paradox. *How many persons needs to be selected at random in order that the probability of at least two of them having the same birthday exceeds* $^1/_2$*?*

Solution. The probability that ϵ different persons have different birthdays is:

$$\left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\left(1-\frac{3}{365}\right)\cdots\left(1-\frac{\epsilon-1}{365}\right)=\frac{365!}{365^{\epsilon}(365-\epsilon)!}$$

This expression becomes $< {}^1/_2$ for $\epsilon \geq 23$. □

We can obviously substitute the 365 with any set of cardinality ζ to express the probability that a random function from $\mathbb{Z}_\epsilon$ to $\mathbb{Z}_\zeta$ is injective. However, back to our particular case, we want to answer the question:

How many random numbers do we have to run through before finding at least two integers equivalent $\mod q$*?*

Using the same reasoning presented above over the previously defined function $f(x) : \mathcal{S} \rightarrow \mathcal{S}$, we will discover that after $\approx 1.18\sqrt{q}$ steps the probability to have fallen inside the period is $^1/_2$. Since any of the two primes factoring N is bounded above by $\sqrt{N}$, we will find a periodic sequence, and thus a factor, in time $\mathcal{O}\left(\sqrt[4]{N}\right)$.

8.2 An Implementation Perspective

The initial algorithm described by Pollard [20] and consultable immediately below, looks for the pair $\langle x_i, x_{2i}\rangle$ such that $\gcd(x_{2i}-x_i, N) > 1$. This is achieved by keeping two variables x, y and respectively updating them via $x \leftarrow f(x)$ and $y \leftarrow f(f(y))$.

Remark 8.2.1. It is intresting to see how in its basic version, Pollard's ρ method just needs 3 variables to preserve the state. This places it among the most parsimonious factorization algorithms in terms of memory footprint.

An immediate improvement of this algorithm would be to occasionally compute Euclid's algorithm over the accumulated product to save some computation cycles, just as we saw in section 6.1. The next code fragment - algorithm 11 - adopts this trick together with Brent's cycle-finding variant ([21]§7).

Parallelism Unfortunately, a parallel implementation of the ρ algorithm would not provide a linear speedup.

Algorithm 10 Pollard's ρ factorization

```
function RHO($N, e$)
    $x \xleftarrow{r} \mathbb{N}$
    $y \leftarrow x$
    $g \leftarrow 1$
    while $g = 1$ do
        $x \leftarrow x^2 + 1 \pmod N$
        $y \leftarrow y^4 + 2y^2 + 2 \pmod N$
        $g \leftarrow gcd(|x - y|, N)$
    if $g = N$ then return nil
    else return $g, N//g$
```

The computation of the x_i sequence is intrinsecally serial; the only plausible approach to parallelism would be to try several different pseudorandom sequences, in which case m different machines processing m different sequences in parallel would be no more than $\mathcal{O}\left(\sqrt{m}\right)$ efficient ([18] §3).

Algorithm 11 Pollard-Brent's factorization

```
function RHO($N, e$)
    $r \leftarrow 1$
    $q \leftarrow 1$                                    ▷ the accumulated gcd
    $g \leftarrow 1$
    $m \leftarrow 100$                                  ▷ steps before checking for gcd
    $y \overset{r}{\leftarrow} \mathbb{N}_{<N}$
    while $g = 1$ do
        $x \leftarrow y$
        for $r$ times do
            $y \leftarrow y^2 + 1 \pmod N$
        $k \leftarrow 0$
        while $k \leq r$ and $g = 1$ do
            $ys \leftarrow y$                           ▷ backup state
            for $\min\{m, r - k\}$ times do             ▷ accumulate values to test later
                $y \leftarrow y^2 + 1 \pmod N$
                $q \leftarrow q \cdot |x - y| \pmod N$
            $k \leftarrow k + m$
            $g \leftarrow \gcd(q, N)$
        $r \leftarrow r \ll 1$
    if $g = N$ then                                     ▷ too far; fall back to latest epoch
        repeat
            $ys \leftarrow ys^2 + 1 \pmod N$
            $g \leftarrow \gcd(N, |x - ys|)$
        until $g > 1$
    if $g = 1$ then return nil
    else return $g, N//g$
```

Dixon's factorization method

[22] describes a class of "probabilistic algorithms" for finding a factor of any composite number, at a sub-exponential cost. They basically consists into taking random integers r in $\{1, \ldots, N\}$ and look for those where $r^2 \mod N$ is *smooth*. If enough are found, then those integers can somehow be assembled, and so a fatorization of N attemped.

9.1 Interlude

During the latest century there has been a huge effort to approach the problem formulated by Fermat 4.1 from different perspecives. This led to an entire family of algorithms, like *Quadratic Sieve*, *Dixon*, ...

The core idea is still to find a pair of perfect squares whose difference can factorize N, but maybe Fermat's hypotesis can be made weaker.

Kraitchick was the first one popularizing the idea that instead of looking for integers $\langle x, y \rangle$ such that $x^2 - y^2 = N$ it is sufficient to look for *multiples* of N:

$$x^2 - y^2 \equiv 0 \pmod{N} \tag{9.1}$$

and, once found, claim that $\gcd(N, x \pm y)$ are non-trial divisors of N just as we did in 4.1. Kraitchick did not stop here: instead of trying $x^2 \equiv y^2 \pmod{N}$ he kept the value of previous attempt, and tries to find *a product* of such values which is also a square. So we have a sequence

$$\langle x_0, \ldots, x_k \rangle \mid \forall i \leq k \quad x_i^2 - N \text{ is a perfect square} \tag{9.2}$$

and hence

$$\prod_i (x_i^2 - N) = y^2$$

that $\mod N$ is equivalent to:

$$y^2 \equiv \prod_i (x_i^2 - N) \equiv \big(\prod_i x_i\big)^2 \pmod{N} \tag{9.3}$$

and voilà our congruence of squares ([23] §4). For what concerns the generation of x_i with the property 9.2, they can simply be taken at random and tested using trial division.

Brillhart and Morrison later proposed ([24] p.187) a better approach than trial division to find such x. Their idea aims to ease the enormous effort required by the trial division. In order to achieve this. they introduce a *factor base* $\mathcal{B}$ and generate random x such that $x^2 - N$ is $\mathcal{B}$-smooth. Recalling what we anticipated in 3, $\mathcal{B}$ is a precomputed set of primes $p_i \in \mathbb{P}$. This way the complexity of generating a new x is dominated by $\mathcal{O}\left(|\mathcal{B}|\right)$. Now that the right side of 9.3 has been satisfied, we have to select a subset of those x so that their product can be seen as a square. Consider an *exponent vector* $v_i = (\alpha_0, \alpha_1, \dots, \alpha_{r-1})$ with $r = |\mathcal{B}| + 1$ associated with each x_i, where

$$\alpha_j = \begin{cases} 1 & \text{if } p_j \text{ divides } x_i \text{ to an odd power} \\ 0 & \text{otherwise} \end{cases} \tag{9.4}$$

for each $1 \leq j < r$. There is no need to restrict ourselves for positive values of $x^2 - N$, so we are going to use α_0 to indicate the sign -1 if negative, 0 otherwise. This benefit has a neglegible cost: we have to add the non-prime -1 to our factor base $\mathcal{B}$.

Let now $M \in \mathbb{F}_2^{(f \times r)}$, for some $f > r$, be the rectangular matrix having per each i-th row the v_i associated to x_i: this way each matrix element m_{ij} will be the j-th component of v_i. We are interested in finding set(s) of the subsequences of x_i whose product always have even powers (9.3). Turns out that this is equivalent to look for the set of vectors $\{w \mid wM = 0\} = \ker(M)$ by definition of matrix multiplication in $\mathbb{F}_2$.

Dixon Morrison and Brillhart's ideas of [24] were actually used for a slightly different factorization method, employing continued fractions instead of the square difference polynomial. Dixon simply ported these to the square problem, achieving a probabilistic factorization method working at a computational cost asymptotically better than all other ones previously described: $\mathcal{O}\left(\exp\{\beta(\log N \log\log N)^{1/2}\}\right)$ for some constant $\beta > 0$ [22].

9.2 Breaching the kernel

The following reduction procedure, extracted from [24], is a forward part of the Gauss-Jordan elimination algorithm (carried out from right to left), and can be used to determine whether the set of exponent vectors is linearly dependent.

For each v_i described as above, associate a *companion history vector* $h_i = (\beta_0, \beta_1, \ldots, \beta_{f-1})$, where for $0 \leq m < f$:

$$\beta_m = \begin{cases} 1 & \text{if } m = i \\ 0 & \text{otherwise} \end{cases}$$

At this point, we have all data structures needed:

Reduction Procedure

(i) Set $j = r - 1$;

(ii) find the "pivot vector", i.e. the first vector $v_i, \quad 0 \leq i < f$ such that $\alpha_j = 1$. If none is found, go to (iv);

(iii) (a) replace every following vector $v_m, \quad i < m < f$ whose rightmost 1 is the j-th component, by the sum $v_i \oplus v_m$;

   (b) whenever v_m is replaced by $v_i \oplus v_m$, replace also the associated history vector h_m with $h_i \oplus h_m$;

(iv) Reduce j by 1. If $j \geq 0$, return to (ii); otherwise stop.

Algorithm 12 formalizes concepts so far discussed, by presenting a function `ker`, discovering linear dependencies in any rectangular matrix $M \in \mathbb{F}_2^{(f \times r)}$ and storing dependencies into a *history matrix* H.

Remark 9.2.1. We are proceeding from right to left in order to conform with [24]. Instead, their choice lays on optimization reasons, which does not apply any more to a modern calculator.

Remark 9.2.2. The `yield` statement in line 12 of algorithm 12 has the same semantics as in the python programming language. It is intended to underline the fact that each $\{\mu \mid H_{i,\mu} = 1\}$ can lead to a solution for 9.2, and therefore their generation can be performed asynchronously.

9.3 An Implementation Perspective

Before gluing all toghether, we need one last building brick necessary for Dixon's factorization algorithm: a `smooth`(x) function. In our specific case, we need a function

Algorithm 12 Reduction Procedure

```
1:  function Ker($M$)
2:      $H \leftarrow \mathtt{Id}(f \times f)$                ▷ the initial $H$ is the identity matrix
3:      for $j = r - 1$ downto $0$ do                          ▷ reduce
4:          for $i = 0$ to $f - 1$ do
5:              if $M_{i,j} = 1$ then
6:                  for $i' = i + 1$ to $f - 1$ do
7:                      if $M_{i',k} = 1$ then
8:                          $M_{i'} = M_i \oplus M_{i'}$
9:                          $H_{i'} = H_i \oplus H_{i'}$
10:                 break
11:     for $i = 0$ to $f - 1$ do                    ▷ yield linear dependencies
12:         if $M_i = (0, \ldots, 0)$ then yield $\{\mu \mid H_{i,\mu} = 1\}$
```

that, given as input a number x, returns **nil** if $x^2 - N$ is not $\mathcal{B}$-smooth. Otherwise, returns a vector $v = (\alpha_0, \ldots, \alpha_r)$ such that each α_j is defined just as in 9.4. Once we have established $\mathcal{B}$, its implementation comes straightfoward.

How do we choose $\mathcal{B}$? It's not easy to answer: if we choose $\mathcal{B}$ small, we will rarely find $x^2 - N$ *smooth*. If we chose it large, attempting to factorize $x^2 - N$ with $\mathcal{B}$ will pay the price of iterating through a large set. [8] §6.1 finds a solution for this problem by employing complex analytic number theory. As a result, the ideal value for $|\mathcal{B}|$ is $e^{\sqrt{\ln N \ln \ln N}}$.

Algorithm 13 Discovering Smoothness

```
Require: $\mathcal{B}$, the factor base
1:  function smooth($x$)
2:      $v \leftarrow (\alpha_0 = 0, \ldots, \alpha_{|\mathcal{B}|} = 0)$
3:      if $x < 0$ then $\alpha_0 \leftarrow 1$
4:      for $i = 1$ to $|\mathcal{B}|$ do
5:          while $\mathcal{B}_i \mid x$ do
6:              $x \leftarrow x // \mathcal{B}_i$
7:              $\alpha_i \leftarrow \alpha_i \oplus 1$
8:      if $x = 1$ then
9:          return $v$
10:     else
11:         return nil
```

Algorithm 14 Dixon

```
Require: $\mathcal{B}$, the factor base
 1: function DIXON($N, e$)
 2:     $i \leftarrow 0$
 3:     $f \overset{r}{\leftarrow} \mathbb{N}_{>|\mathcal{B}|}$                ▷ finding linearity requires redundance
 4:     while $i < f$ do                                   ▷ search for suitable pairs
 5:         $x_i \overset{r}{\leftarrow} \mathbb{N}_{<N}$
 6:         $y_i \leftarrow x_i^2 - N$
 7:         $v_i \leftarrow \text{SMOOTH}(y_i)$
 8:         if $v_i \neq$ nil then $i \leftarrow i + 1$
 9:     $M \leftarrow \texttt{matrix}(v_0, \dots, v_{f-1})$
10:     for $\lambda = \{\mu_0, \dots, \mu_k\}$ in $\text{KER}(M)$ do          ▷ get relations
11:         $x \leftarrow \prod_{\mu \in \lambda} x_\mu \pmod N$
12:         $y, r \leftarrow isqrt(\prod_{\mu \in \lambda} y_\mu \pmod N)$
13:         $g \leftarrow \gcd(x + y, N)$
14:         if $1 < g < N$ then
15:             $p \leftarrow g$
16:             $q \leftarrow N // p$
17:             return $p, q$
```

Parallelism Dixon's factorization is ideally suited to parallel implementation. Similarly to other methods like ECM and MPQS, treated in [18] §6.1, we can *linearly* improve the running time by distributing across many nodes the discovery of $\mathcal{B}$-smooth numbers.

Depending on the granularity we desire - and the number of nodes available, we can even act on the `ker` function - but less easily. This idea would boil down to the same structure we discussed with Wiener's attack: one node - the *producer* - discovers linear dependencies, while the others - the *consumers* - attempt to factorize N.

Certainly, due to the probabilistic nature of this algorithm, we can even think about running multiple instances of the same program. This solution is fairly effective in proportion to the development cost.

An Empirical Study

Excluding Dixon's factorization method, all attacks analyzed so far exploit some peculiarities of a candidate RSA public key $\langle N, e \rangle$ in order to recover the private exponent. Summarizingly:

- Pollard's $p - 1$ attack works only if the predecessor of any of the two primes factorizing the public modulus is composed of very small prime powers;
- Williams' $p + 1$ attack works under similar conditions - on the predecessor or the successor of any of the two primes ;
- Fermat's factorization is valuable whenever the two primes p and q are really close to each other;
- Pollard's ρ method is best whenever one of the two primes is strictly lower than the other;
- Wiener's attack is guaranteed to work on small private exponents.

Dixon's factorization method instead, being a general-purpose factorization algorithm, can be employed to *measure* the strength of a RSA keypair: the more relations (satisfying 9.3) are found, the less it is assumed to be resistant.

Given these hypothesis, it has been fairly easy to produce valid RSA candidate keys that can be broken using the above attacks. They have been used to assert the correctness of the implementation.

On the top of that, there has been a chance to test the software under real conditions: we downloaded the SSL keys (if any) of the top one million visited websites, and survey them with the just developed software. This not only gave us the opportunity to survey the degree of security on which the internet is grounded today, but also led to a deeper understanding of the capacities and limits of the most widespread libraries offering crypto nowadays.

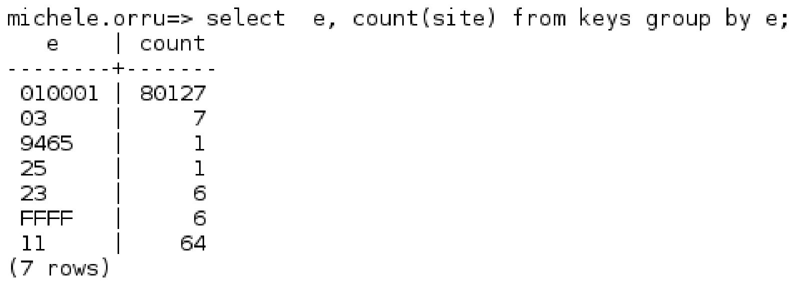
10.1 To skim off the dataset

What has been most scandalous above all was to discover that more than **half** of the most visited websites do **not** provide SSL connection over port 443 - reserved for HTTPS according to IANA [25]. To put it in numbers, we are talking about $533,000$ websites either unresolved or unreachable in 10 seconds. As a side note for this, many websites (like `baidu.com` or `qq.com`) keep a TCP connection open without writing anything to the channel, requiring us to adopt a combination of non-blocking socket with the `select()` system call in order to drop any empty communication. It would be interesting to investigate more on these facts, asking ourselves how many of those unsuccessful connections are actually wanted from the server, and how many dropped for censorship reasons; there is enough room for another project.

Of the remaining $450,000$ keys, 21 were using different ciphers than RSA. All others represent the dataset upon which we worked on.

10.2 To count

Once all valuable certificate informations have been stored inside a database, almost any query can be performed to get a statistically valuable measure of degree of magnitude to which some conditions are satisfied. What follows now is a list of commented examples that we believe are relevant parameters for understanding of how badly internet is configured today.



```
michele.orru=> select  e, count(site) from keys group by e;
   e    | count
--------+-------
 010001 | 80127
 03     |     7
 9465   |     1
 25     |     1
 23     |     6
 FFFF   |     6
 11     |    64
(7 rows)
```

The most prolific number we see here, 65537 in hexadecimal, is the fourth Fermat number and no other than the largest known prime of the form $2^{2^n}+1$. Due to its composition, it has been advised by NIST as default public exponent, and successfully implemented in most software, such as OpenSSL.

Sadly, a negligible number of websites is using low public exponents, which makes the RSA key vulnerable to Coppersmith's attack; though, this topic goes beyond the scope of this research and hence has not been analyzed further.

```
michele.orru=> select count(distinct (n, e)) from keys;
 count
-------
 57678
(1 row)
```

What is interesting to see here is that an enormous portion of our dataset shared the same public key, pushing down the number of expected keys of one order of magnitude. Reasons for this are mostly practical: it is extremely frequent to have blogs hosted on third-party services such as "Blogspot" or "Wordpress" which always provide the same X.509 certificate, as they belong to an unique organization. Though improbable, it is even possible that exists a millesimal portion of different websites sharing the same public key due to a bad cryptographically secure random number generator, and therefore also the same private key. Such a case has been already investigated in [26].

```
michele.orru=> select count(site) from certs
 where subject like '%localhost%' and
       issuer not like '%localhost%';
 count
-------
    58
(1 row)
```

Here we go. A suprisingly consistent number of websites provides certificates filled with dummy, wrong, or even testing informations.
Some do have non-printable bytes in the *common name* field.
Some are certified from authorities.
Some are even gonvernmental entities.

```
michele.orru=> select nbits, count(site) from keys group by nbits order by nbits;
 nbits | count
-------+-------
   512 |    47
   768 |     1
  1023 |     4
  1024 | 11103
  2014 |     1
  2028 |     1
  2047 |     3
  2048 | 66962
  2058 |     1
  2080 |     1
  2084 |     1
  2432 |    43
  2922 |     1
  3072 |     5
  3248 |     1
  4048 |     1
  4086 |     1
  4092 |     1
  4096 |  2033
  8192 |     1
(20 rows)
```

According to [27] §3, table 2, all RSA keys of bitlength less than 1024 are to be considered deprecated at the end of 2013 and shall no more be issued since the beginning of this year. Not differently from the above results, the remark has been globally adopted, yet still with a few exceptions: around a dozen of non-self-signed certificates with a 1024 RSA key appears to have been issued in 2014.

10.3 The proof and the concept

At the time of this writing, we have collected the output of only two mathematical tests performed in the university cluster.

Wiener. The attack described in chapter 5 was the first employed, being the fastest one above all others. Recalling the different public exponents we probed and discussed in the preceeding section (all ≤ 65537), we expected all private expenents to be $> {}^{1}/_{3}\sqrt[4]{N}$ and therefore not vulnerable to this particular version of Wiener's attack. Indeed, we found no weak keys with respect to this attack. Though, as pointed out in [1] §3, there is still the possibilty that the public keys we collected could be broken employing some variants of it.

GCD. On the wave of [26], whe attempted also to perform the gcd of every possible pair of dinstinct public modulus present in the dataset. In contrast to our expectations, this test led to no prime factor leaked, for any key pair. We have reasons to believe this depends on the relatively small size of our dataset, with respect to the one used in [26].

Conclusions

Everytime we surf the web, we share our communication channel with lots of entities around the globe. End-to-end encryption protocols such as TLS can provide the security properties that we often take as granted, like *confidentiality*, *integrity*, and *authenticity*; though, these holds only if we *trust* the authorities certifying the end entity.

There is this mindless thinking that whenever we see that small lock icon in the browser's url bar, somebody is telling us the connection is safe. There is some authority out there telling what to do, and we should be thinking more about what these authorities are and what they are doing. This issue is no more a technical problem, but instead is becoming more and more a social and political problem. It is our responsability as citzens to do something about that.

Bibliography

[1] D. Boneh, R. Rivest, A. Shamir, L. Adleman, *et al.*, "Twenty years of attacks on the rsa cryptosystem," *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.

[2] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Rfc 4158: Certification path building."

[3] T. Dierks and C. Allen, "Rfc 2246: The tls protocol version 1.0."

[4] P. E. Jesse, "Is the ssliverse a safe place? an update on eff's ssl observatory project." [27c3], 2010.

[5] S. Turner and T. Polk, "Rfc6176: Prohibiting secure sockets layer (ssl) version 2.0."

[6] S. Vaudenay, "Security flaws induced by cbc padding - applications to ssl, ipsec, wtls," pp. 534–546, 2002.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* The MIT Press, 3rd ed., 2009.

[8] R. Crandall, C. Pomerance, R. Crandall, and C. Pomerance, *Prime numbers: a computational perspective. Second Edition.* Cambridge, MA, USA: Birkhauser Boston Inc., 2005.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[10] R. Bombelli, *L'Algebra.* Mathematical Association of America, 1572.

[11] E. W. Dijkstra, *A Discipline of Programming.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1997.

[12] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[13] NIST, “Fips pub 186-3: Digital signature standard,” 2009.

[14] H. Riesel, *Prime Numbers and Computer Methods for Factorization.* Cambridge, MA, USA: Birkhauser Boston Inc., 1985.

[15] M. J. Wiener, “Cryptanalysis of short rsa secret exponents,” *IEEE Transactions on Information Theory*, vol. 36, pp. 553–558, 1990.

[16] I. Smeets, “On continued fraction algorithms,” 2010.

[17] J. M. Pollard, “Theorems on factorization and primality testing,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 76, pp. 521–528, 11 1974.

[18] R. P. Brent, “Parallel algorithms for integer factorisation,” *Number Theory and Cryptography (edited by JH Loxton), London Mathematical Society Lecture Note Series*, vol. 154, pp. 26–37, 1990.

[19] H. C. Williams, “A $p + 1$ method of factoring,” *Mathematics of Computation*, vol. 39, no. 159, pp. pp. 225–234, 1982.

[20] J. Pollard, “A monte carlo method for factorization,” *BIT Numerical Mathematics*, vol. 15, no. 3, pp. 331–334, 1975.

[21] R. P. Brent, “An improved monte carlo factorization algorithm,” *BIT Numerical Mathematics*, vol. 20, no. 2, pp. 176–184, 1980.

[22] J. D. Dixon, “Asymptotically fast factorization of integers,” *Mathematics of Computation*, vol. 36, no. 153, pp. pp. 255–260, 1981.

[23] A. Odlyzko, “Discrete logarithms: The past and the future,” *Towards a Quarter-Century of Public Key Cryptography*, pp. 59–75, 2000.

[24] M. A. Morrison and J. Brillhart, “A method of factoring and the factorization of $\mathcal{F}_7$,” *Mathematics of Computation*, vol. 29, no. 129, pp. 183–205, 1975.

[25] I. A. N. Authority, “Service names port numbers.” 2014.

[26] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, “Ron was wrong, whit is right.,” *IACR Cryptology ePrint Archive*, vol. 2012, p. 64, 2012.

[27] E. Barker and A. Roginsky, “Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths,” *NIST Special Publication*, vol. 800, p. 131A, 2011.