RWTH Aachen

Department of Computer Science
Technical Report

Information Hiding in the Public RSA Modulus

Stefan Wüller, Marián Kühnel, and Ulrike Meyer

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2015-11

RWTH Aachen · Department of Computer Science · June 2015

Information Hiding in the Public RSA Modulus

Stefan Wüller, Marián Kühnel, and Ulrike Meyer

Research Group IT-Security
RWTH Aachen, Germany
Email: {wueller, kuehnel, meyer}@itsec.rwth-aachen.de

Abstract. The manufacturer of an asymmetric backdoor for a public key cryptosystem manipulates the key generation process in such a way that he can extract the private key or other secret information from the user's public key by involving his own public/private key pair. All backdoors in major public key cryptosystems, including RSA, differ substantially in their implementation approaches and in their quality in satisfying backdoor related properties like confidentiality or concealment. While some of them meet neither of these two properties very well, others provide a high level of confidentiality but none of them is concealing, which limits their use for covert implementation. In this work we introduce two novel asymmetric RSA backdoors, both following the approach to embed bits of one of the RSA prime factors in the user's public RSA modulus. While our first backdoor provides confidentiality for a sufficiently large key length, it might be detected under certain circumstances. The second backdoor extends the first one such that it additionally provides concealment and is thus particularly suitable for covert implementation.

1 Introduction

Public key cryptosystems are used worldwide to provide confidentiality, integrity, and non-repudiation to electronic communication. This raises the question of whether those cryptosystems can be trusted in the case that they are implemented on a cryptographic device, i.e., in a kind of black box which inhibits the access to authenticity and integrity validation of its internal design. How can users of cryptographic devices preclude the existence of a backdoor which leaks secret information to a third party in order to recover the user's secret key?

Backdoors designed for asymmetric cryptosystems act up to the principle that the manufacturer of a cryptographic device modifies the underlying cryptosystem which enables him to reconstruct the target user's private key from the corresponding public key, i.e., the manufacturer has the exclusive ability to decrypt eavesdropped messages sent to the user and to sign messages in his place.

Generally, there can be distinguished between two purposes for the implementation of cryptographic backdoors: to illegitimately spy out the users of backdoored cryptographic devices or to design a legitimate auto-escrowing key system[1].

The background for the illegitimate distribution of backdoored cryptosystems, i.e., the user of a cryptographical device is unaware of its contamination, might be economic interest or the internal security of a country. Thus, companies—especially hard- and software manufacturer's—and the government

[1] An auto-escrowing key system enables a legitimate third party or the cooperation of a specified number of legitimate third parties to gain access to the private keys of involved users. Usually, the access to reconstruct private keys is only possible under controlled condition, e.g., a court decision.

or security agencies are considered to be potential stakeholders. Companies might benefit from spying on their customers in combination with data collection which might be used internally for advertising purposes or for selling-on to any prospects. Moreover—especially in the light of current events [Sch13,Maj13,Kno13]—it seems not to be far-fetched that a government forces or bribes companies of the hard- and software sector to embed backdoors into several of their products. If it is the case that the legal situation of a country prevents its government to implement a (forced) auto-escrowing key system, e.g., for privacy protection, the only remaining possibility of implementing such a system is a subliminal distribution. The cooperation between government and hard- and software manufacturer seems to be indispensable for this purpose—no matter if the cooperation is law enforced or basing on benefits or corruption.

The legitimate application of backdoored cryptosystems comprises the case where the users of cryptographic devices are aware of the modification made to the underlying cryptosystem. The rational behind legitimate backdoors is to design an auto-escrowing key system which enables private key retrieval for legitimate recipients of encrypted messages who lost their keys. Besides key retrieval services for ordinary users, companies are enabled to access proprietary encrypted files or mails of laid off or deceased employees while law enforcement agencies have the possibility to access the contents of encrypted messages attributed to suspects and criminals. In the USA, an auto-escrowing key system, the *Escrowed Encryption Standard*, was developed by the NSA in 1993 which addresses to an on-chip backdoored encryption scheme. Those chips were intended to be installed on telephones (*Clipper Chips*) and on computers (*Capstone Chips*) forced by law. Exploiting the backdoor, i.e., retrieving the secret key of a target, demands the cooperation of two state authorities which in turn require a judicial intervention to become active. The legal situation in the USA prohibited the practical implementation of the Escrowed Encrpytion Standard which was discarded 1996.

Cryptographic backdoors have to be well studied because of their security interfering behavior. The essential properties a backdoor can have are the following: the backdoored cryptosystem remains secure against attackers (*Confidentiality*); the manufacturer is able to reconstruct the private key from each user's public key which was generated with the backdoored cryptosystem (*Completeness*); the backdoored cryptosystem remains hidden from users who additionally are given access to an implementation of the corresponding honest cryptosystem (*Concealment*).

For all major cryptosystems, backdoors have been designed—first of all RSA due to its broad dissemination. Those backdoors differ substantially in their hiding and reconstruction techniques as well as their quality in satisfying Confidentiality and Concealment: in some neither of two properties are met very well (e.g., [YY96,YY97]), others satisfy Confidentiality but unfortunately do not meet the Concealment to a satisfying extent (e.g., [YY06]). In this work we propose two novel RSA backdoors RSA^{PHP}_{GEN1} and $RSA^{PHP'}_{GEN1}$ (Prime (P) Hiding (H) Prime (P)). The first backdoor, RSA^{PHP}_{GEN1}, modifies the RSA key generation function in such a way that the information required for the reconstruction of the private key is directly hidden in the user's RSA modulus. The second backdoor, $RSA^{PHP'}_{GEN1}$, additionally blinds the secret information to hide the backdoor from the user and thus from an (external) attacker. In order to reduce the amount of secret information hidden in the RSA modulus, $RSA_{GEN1}^{PHP'}$ takes advantage of Coppersmith's factorization attack [Cop97]. The analyses of both backdoors prove that RSA_{GEN1}^{PHP} is confidentiality preserving for sufficiently large key lengths, complete, but (under certain circumstances) not concealing whereas $RSA_{GEN1}^{PHP'}$ is confidentiality preserving for sufficiently large key lengths, complete, and concealing at the same time.

In order to give a proof of work and to conduct a running time analysis we implemented our new backdoors comprising the corresponding key recovery methods for OpenSSL.

The remainder of this work is organized as follows. Section 2 and 3 give an introduction to the cryptographic and mathematical foundations which are required to comprehend the functionality of RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ as well as the corresponding key reconstruction functions. After reviewing the most prominent backdoors and backdoor definitions from literature in Section 4, we introduce a more straight forward approach to analyze the security of a backdoor basing on the definition of essential backdoor related properties in Section 5. The introduction of the novel backdoors RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ as well as the corresponding security analysis is the focus of Section 6 which can be considered as the major contribution of this work. Section 7 presents the implementation of both backdoors as well as the corresponding key reconstruction functions followed by an evaluation of the running time. Finally, a conclusion is provided in Section 8.

2 Cryptographic Foundations

In this section we briefly refresh the terms of symmetric and asymmetric cryptography, outline the fundamental mathematical problems asymmetric cryptography relies on, and outline certain selected cryptosystems.

Since the focus lies on backdoors for asymmetric cryptosystems, we restrict ourselves to outline those cryptosystem whose infiltration by a backdoor is considered in sections below. In this work the principle of symmetric cryptography merely comes into play within the construction of a backdoor for an asymmetric cryptosystem.

Notations. We denote the set of all prime numbers of bit size l_n where l_n refers to the number of bits of integer n with Φ_{l_n}. To indicate that an element m is chosen uniformly at random from set M, we write $m \in_{rnd} M$. As usual we write φ to refer to Euler's totient function.

2.1 Symmetric Cryptography

The characteristic of symmetric encryption is that two communicating parties (Alice and Bob) who want to confidentially exchange messages share the same key which is used for encryption and decryption. Before any communication between Alice and Bob takes place, they have to negotiate a secret key over a secure channel. Usually, asymmetric encryption is used to solve this task (see Section 2.2). Suppose Alice wants to send a message encrypted with a symmetric encryption algorithm E to Bob. Alice uses E and the shared key k to encrypt the plaintext message m. E outputs the ciphertext $c = E(k, m)$ corresponding to

m. The message reaches Bob over an insecure channel. Bob uses the decryption algorithm D depending on E and his shared key to obtain the plaintext message $m = D(k, c)$.

2.2 Asymmetric Cryptography

The general idea of asymmetric encryption (also called public key encryption) is that each communication party possesses two keys: a public key for encryption and a private key for decryption. As the name suggests, the public key of every party is ideally publicly available to everyone who wants to send a message to a key owner. If Alice sends a message to Bob, she looks up Bob's public key inquiring a public directory and encrypts the message using it. Only with the help of the corresponding private key, a receiver is able to decrypt the message. If Bob's private key gets compromised, everyone else who can access the stolen key is able to decrypt messages issued for Bob until he revokes his key[2]. The most serious advantage of public key encryption is the circumvention of distributing a symmetric key.

Apart from encrypting messages, public key encryption can be used to sign messages. If Alice wants to sign a message m for Bob, she encrypts it with her private key $s = sign_{privKeyAlice}(m)$ and sends (m, s) to Bob. Bob applies Alice's public key to s and obtains m'. If $m = m'$, Bob knows that Alice is the valid creator of m.

The mathematical fundamentals necessary to implement public key encryption are one way functions, which base, e.g., on the factorization problem (see Section 2.2.1) or the discrete logarithm problem (see Section 2.2.2).

The authenticity of public keys is essential. Bob can not be sure if the public key he got from the public directory is a valid one of Alice. An attacker might have manipulated the directory and replaced his one with Alice's. To authenticate public keys, trusted third parties are involved. They issue certificates consisting of the user's identity and his public key, signed with their private keys. In combination with the public key, the certificate is published. When Bob looks up the public key for Alice he simultaneously checks its authenticity by verifying the corresponding certificate with the help of the trusted third party's public key. The whole public key system, called public key infrastructure (PKI), consists of methods to generate keys, to authenticate parties, to distribute, and verify public keys.

Since asymmetric encryption is computational more expensive than symmetric encryption, it is generally used to establish a symmetric key between communication parties. Subsequently, for message exchanges symmetric encryption is used.

2.2.1 The Factorization Problem.

Definition 2.1 (Factorization Problem). *The factorization problem for integers is the problem to compute the prime factors of a given integer.*

[2]Key revocation solves this problem only for messages encrypted with the novel key and only for the case that perfect forward secrecy is provided.

To date, it is not known if the factorization problem is hard, i.e., no efficient algorithm for solving the factorization problem has been published yet. Nevertheless, it is widely believed that is computationally not possible to factorize an integer with properly chosen large prime factors ($\geq$ 512 bits) of equal size [DK07].

2.2.2 The Discrete Logarithm Problem.

Let p be a prime number and g be a generator of $\mathbb{Z}_p$. For each $h \in \mathbb{Z}_p$ there exists an exponent $a \in \{0, 1, 2, ..., p-1\}$ such that

$$h \equiv g^a \ mod \ p.$$

a is called the discrete logarithm of h for base g w.r.t. $\mathbb{Z}_p$. The discrete logarithm problem is defined as follows:

Definition 2.2 (Discrete Logarithm Problem). *Let p, g, h, and a be defined as above. The problem to compute a given p, g, and h such that $h = g^a \ mod \ p$ is called the discrete logarithm problem.*

The computation of discrete logarithms is considered to be hard—to date, there are no polynomial time algorithms known which compute the discrete logarithm on the input of p, g, and h [Buc10].

2.2.3 Asymmetric Cryptosystems.

The following three asymmetric cryptosystems are those which are considered in the remaining sections w.r.t. the embedding of backdoors.

2.2.3.1 RSA. The RSA cryptosystem, named after its designers Rivest, Shamir, and Adleman, was the first published public key cryptosystem [DH76] and seems to be the most widely used one today [YY04].

To generate a key pair, two large primes of bitsize $l_n/2$ are randomly chosen and their product n is computed. The bitsize of p and q (also referred to as security parameter) is responsible for the security of the system and the cardinality of the message space. Today, l_n should be at least 1024 bits to guarantee confidentiality. Subsequently, another random number, the public exponent e, is chosen.[3] It is necessary that e satisfies the condition that $gcd(e, \varphi(n)) = 1$ that is e and $\varphi(n)$ are relatively prime. This is necessary to assure that the inverse of e exists in $\mathbb{Z}_{\varphi(n)}$. The pair (n, e) is published as the public key. The private exponent d which is the multiplicative inverse of e in $\mathbb{Z}_{\varphi(n)}$, i.e., $e \cdot d \equiv 1 \ mod \ \varphi(n)$, can be computed with the extended euclidean algorithm. The private key is constituted by the tuple (n, d).

If Bob wants to send a message m such that $1 < m < n$ and $gcd(m, n) = 1$ to Alice, he computes the ciphertext $c = m^e \ mod \ n$ using Alice's public key. Alice is able to decrypt the encrypted message c and receive the plaintext message $m = c^d \ mod \ n$.

It has been shown that the problem of computing the private key d from the public key (n, e) (known as the RSA problem) is equivalent to the factorization

[3] There exist implementations of RSA, e.g., the one in OpenSSL which use a fix e and choose p and q appropriately.

problem (see Section 2.2.1) [BMS84]. Thus, the security of RSA relies on the difficulty of finding the prime factors p and q of n.

The RSA cryptosystem can be divided into the functions `GEN`, `ENC`, and `DEC` where `GEN` can be divided further into `GEN1` - `GEN3`:

key generation:
$(p, q, n) \leftarrow \texttt{GEN1}(l_n) : p, q \in_{rnd} \Phi_{l_n/2}, l_p = l_q = l_n/2, n = p \cdot q$
$e \leftarrow \texttt{GEN2}(p, q) : e \in_{rnd} \mathbb{N}\backslash\{1, 2\}, gcd(e, \varphi(n)) = 1$
$d \leftarrow \texttt{GEN3}(p, q, e) : e \cdot d = 1 \ mod \ \varphi(n)$

encryption:
$c \leftarrow \texttt{ENC}(m, n, e) = m^e \ mod \ n$

decryption:
$m \leftarrow \texttt{DEC}(c, n, d) = c^d \ mod \ n$

2.2.3.2 Diffie-Hellman. The Diffie-Hellman protocol [DH76], proposed by Whitfield Diffie and Martin Hellman in 1976, was the inception of public-key cryptography and the first solution to the problem of how two parties that have never met before can negotiate a symmetric key over a public channel.

A sufficiently large prime p and a generator $g \in \mathbb{Z}_p^*$ form the public key shared between two parties: Alice and Bob. They are able to negotiate a secret symmetric key by conducting the following protocol. Alice chooses a random number r_A in the range $2 \leq r_A \leq p - 1$ and computes the value $A = g^{r_A} \ mod \ p$. Bob does the same obtaining r_B and B. The values A and B are exchanged over an insecure channel. Alice computes $K_A = B^{r_A} \ mod \ p$ and Bob analogously computes $K_B = B^{r_B} \ mod \ p$. It holds that $K_A = K_B$ since it can be shown that $(g^{r_A})^{r_B} \equiv (g^{r_B})^{r_A} \ mod \ p$ [Gal12]. The security of the Diffie-Hellman key exchange protocol is based on the *Diffie-Hellman Assumption* also known as the *Diffie-Hellman Problem* [DH76] which is closely related to the discrete logarithm problem (see Section 2.2.2), i.e., if someone is able to solve the discrete logarithm problem, he is able to solve the Diffie-Hellman Problem. The protocol only provides security against passive attackers but can be extended with an authentication technique basing on certificates to ensure robustness against active attackers, too [DK07]. The following functions have to be applied by each communication party except `GEN1`. `GEN1` has to be executed by only one party followed by publishing the result:

distributed key generation:
$(p, g) \leftarrow \texttt{GEN1}(l_p) : p \in_{rnd} \Phi_{l_p}, g \in_{rnd} \mathbb{Z}_p^*$
$(r_x, X) \leftarrow \texttt{GEN2}(p, g) : r_x \in_{rnd} \{0, ..., p - 1\}, X = g^{r_x} \ mod \ p$
$K \leftarrow \texttt{GEN3}(r_x, Y) : K = Y^{r_x} \ mod \ p$

If the Diffie-Hellman protocol is implemented over the group of elliptic curves, the protocol is called *Elliptic Curve Diffie-Hellman* (ECDH).

2.2.3.3 ElGamal. The ElGamal cryptosystem is a generalization of the Diffie-Hellman key exchange. The security of ElGamal depends on the discrete logarithm problem (see Section 2.2.2). The recipient, Bob, chooses a large prime p of bit length l_p in such a way that $p-1$ has a large prime factor and a generator $g \in \mathbb{Z}_p^*$. Subsequently, he computes $y = g^x \ mod \ p$, where x is a random integer in the range $0 \leq x \leq p-1$. (p, g, y) is Bob's public key while his private key is given by the tuple (p, g, x).

If Alice wants to send an encrypted message to Bob, she uses Bob's public key to compute the ciphertext $c = (c_1, c_2)$ of plaintext message $m \in \mathbb{Z}_p$: $c_1 = g^k \ mod \ p$, $c_2 = y^k \cdot m \ mod \ p$ where k is an random integer in the range $1 \leq k \leq p-1$. By using his private key, Bob first computes the inverse c_1^{-x} of c_1^x in $\mathbb{Z}_p$. With a second computation step, Bob is able to receive the plaintext:

$$m = c_1^{-x} \cdot c_2 \ mod \ p = y^{-k} \cdot y^k \cdot m \ mod \ p = m.$$

The ElGamal cryptosystem can be divided into three functions:

key generation:
$((p, g, y), (p, g, x)) \leftarrow \texttt{GEN}(l_p) : p \in_{rnd} \Phi_{l_p}, x \in_{rnd} \mathbb{N} \ and \ 0 < x \leq p-1, g \in_{rnd} \mathbb{Z}_p^*, y = g^x \ mod \ p$

encryption:
$(c_1, c_2) \leftarrow \texttt{ENC}(m, (p, g, y)) : k \in_{rnd} \mathbb{N}, 0 < k \leq p-1, c_1 = g^k \ mod \ p, c_2 = y^k \cdot m \ mod \ p$

decryption:
$m \leftarrow \texttt{DEC}(c, (p, g, x)) : m = c_1^{-x} \cdot c_2 \ mod \ p$

3 Mathematical Foundations

With his paper *Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities* [Cop97] (published in 1997), Coppersmith proposed a method for finding small roots of bivariate integer polynomials [Cop96] based on lattice reduction. As an application of his technique, he presented an attack on an RSA variant, referred to as Coppersmith's factorization attack, where the high- or low-order bits of one of the RSA prime factors are known to the attacker.

Eight years later, Coron presented a simpler approach for Coppersmith's method which is easier to implement and additionally heuristically extensible to multivariate polynomials [Cor04]. Nevertheless, this approach was less efficient than Coppersmith's algorithm.

In 2007, Coron came up with a novel direct approach [Cor07] for finding small roots of bivariate integer polynomial equations with the same complexity as achieved in [Cop96]. Both of Coron's approaches followed the technique proposed by Howgrave-Graham for the univariate case to simplify Coppersmith's method [HG97].

In this section we first give an introduction to lattice reduction by focussing on the LLL algorithm which is an important tool in the area of finding roots of integer polynomials. Subsequently, we present the results of Coron's paper *Finding Small Roots of Bivariate Integer Polynomial Equations: A Direct Approach* [Cor07] which plays a crucial role in the context of one of our backdoors we introduce in Section 6.

3.1 Lattices

Lattices—subgroups of the Euclidean vector space—are a fundamental tool in the area of cryptoanalysis of public key systems. There exists various attacks on knapsack cryptosystems, on RSA signatures, and on RSA variants involving lattices [Gal12].

Notations. Note that we write lattice vectors as row vectors, as it became a common practice. Let the notation of b_i^*, b_j^* denote that the vectors b_i and b_j are orthogonal.

Definition 3.1 (Lattice, Lattice Basis, Lattice Rank, Sublattice). *Let $\{b_1, ...b_\omega\}$ be a linearly independent set of vectors in $\mathbb{R}^n$ with $(n \geq \omega)$. Lattice L is generated by all linear integer combinations of the b_i vectors:*

$$L = \left\{ \sum_{i=1}^{\omega} n_i \cdot b_i \mid n_i \in \mathbb{Z} \right\} \tag{1}$$

The vectors $b_1, ..., b_\omega$ are called lattice basis (L^B) of L. The lattice rank ω is given by the number of vectors of L^B. A sublattice of L is a subset of row vectors of L which is a lattice itself.

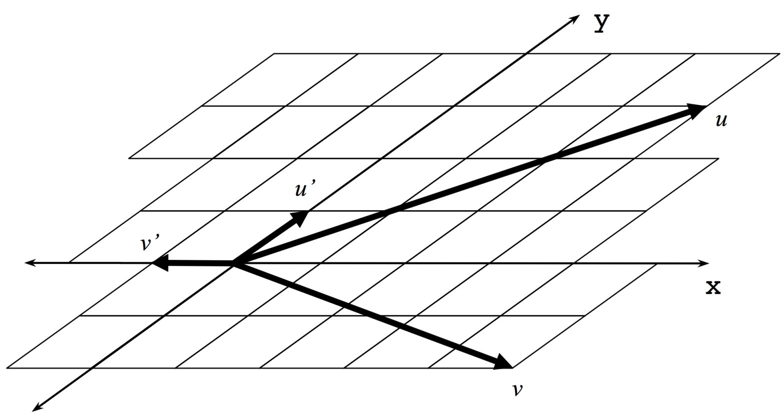

Fig. 1: Visualization of two different lattice bases ($\{u, v\}$ and $\{u', v'\}$) of a lattice in the 2-dimensional space

Consider Figure 1 depicting an extract of a lattice in the 2-dimensional space. Every vertex of a parallelogram represents one point of the lattice. The vectors u and v constitute a basis of the lattice, i.e., each point of the lattice can be reached by linear combinations of u and v. Another basis of the lattice is given by the vectors u' and v' which have the property to be orthogonal and of minimal length. In the following, we assume that $b_i \in \mathbb{Z}^n$ holds, as it is usually the case for cryptographic applications [Gal12].

The determinant of a lattice ($det(L)$) belongs to the most important numerical invariants attached to lattices [Len08]. Geometrically, the determinant of a lattice is the volume of the parallelepiped spanned by the lattice. The determinant of a lattice L is independent of the choice of the basis and is defined as the square root of the Gramian determinant of L (see Appendix A.4). For a full rank lattice, i.e., $n = \omega$, it holds that $det(L) = |det(L^B)| = |det(b_1, ..., b_\omega)|$.

A lot of computational problems are related to lattices. Some of them can be efficiently solved, others seem to be hard in general [Gal12]. The problem we are interested in is called the *shortest vector problem* (SVP) which seems to be hard.

Definition 3.2 (SVP). *The shortest vector problem is the problem to compute a non-zero vector $b \in L$ given the lattice basis L^B of lattice L such that the length of b is minimal.*

For lattice bases of rank 2 in $\mathbb{R}^2$ there exists an algorithm which solves the SVP in polynomial time [Gal12]. For higher dimensions there exists no polynomial time algorithm except heuristic algorithm which sufficiently reduce the vector length for practical purposes. In the following subsection the technique of *lattice basis reduction* is introduced which reduces the Euclidean norm (see Appendix A.2) of a lattice basis in polynomial time in order to approximate the result of the SVP.

3.1.1 Lattice Basis Reduction.

The goal to approximate the shortest vector problem will be clear in Section 3.2 when the relationship between the Euclidean norm of a lattice basis vector and the possibility to compute the roots of bivariate polynomials is introduced. Lattice basis reduction is a method to compute on input L^B another basis L^{RB} of L such that the Euclidean norm of the vectors of L^{RB} is smaller than the vectors of L^B and the vectors of L^{RB} are close to orthogonal. Orthogonality is an important property in the context of lattice basis reduction. This is because for orthogonal basis vectors the SVP is easy to solve. For sufficiently close to orthogonal basis vectors the SVP achieves adequate results as it is the case below where the orthogonal vectors are rounded to the closest integer. Quite often it is the case that the approximation of the SVP corresponds to the proper solution.

In the following, we present the LLL basis reduction algorithm [LLL82] named after their inventors Lenstra, Lenstra, and Lovász which plays an important role in practical applications for approximating the SVP, i.e., solving the SVP up to an exponential factor, but not guaranteeing that the shortest vector of the considered lattice is computed [Gal12].

The purpose of the LLL algorithm is to compute an *LLL-reduced basis* which is given by the following definition:

Definition 3.3 (LLL-reduced Basis). *Let $L^B = \{b_1, ..., b_\omega\}$ be an ordered lattice basis and*

$$\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}$$

the coefficients from the Gram-Schmidt orthogonalization process (see Appendix A.5). For a fixed $\delta \in (1/4, 1)$, $\{b'_1, ..., b'_\omega\}$ is called δ-LLL-reduced if it holds that

$$|\mu_{i,j}| \leq 0.5, \;\; 1 \leq j < i \leq \omega \;\; \textit{(Size reduced)}$$

and

$$||b_i^*||_2^2 = \langle b_i^*, b_i^* \rangle \geq (\delta - \mu_{i,i-1}^2) \cdot \langle b_{i-1}^*, b_{i-1}^* \rangle, \;\; 2 \leq i \leq \omega \;\; \textit{(Lovász condition).}$$

Unless otherwise specified, $\delta = 3/4$ is assumed. The LLL-reduced lattice basis is denoted with $L^{RB} = \{b'_1, ..., b'_\omega\}$.

The following theorem gives an upper bound for the shortest vector of an LLL-reduced basis.

Theorem 3.1. *Let $L^{RB} = \{b'_1, ..., b'_\omega\}$ be an LLL-reduced basis with $\delta = 3/4$ for a lattice $L \subset \mathbb{R}^n$, then the following inequality holds:*

$$||b'_1||_2 \leq 2^{(\omega-1)/4} \cdot det(L)^{1/\omega}.$$

Proof. See [LLL82].

Since lattice reduction plays a crucial role in the remaining work, we take a closer look at the interior of the LLL algorithm provided by Algorithm 3.1 basing on [Gal12]. The algorithm starts with computing a *Gram-Schmidt basis* (see Appendix A.5) of the input lattice basis L^B. The following computations are performed by iterating over the vectors of L^B.

The first condition of Definition 3.3 (Size reduced) is met by performing a size reduction, i.e., choosing suitable linear integer combinations of the vectors of L^B (Line 5-9 of Algorithm 3.1). Subsequently, the second condition of Definition 3.3 (Lovász condition) is checked (Line 10 of Algorithm 3.1). If the Lovász condition is not satisfied, i.e., b_i is not significantly longer than b_{i-1}, both vectors are swapped and backtracking is performed until the second condition is satisfied (line 12-19 of Algorithm 3.1).

It can be shown that the output of the LLL algorithm is an LLL-reduced lattice basis and by construction it holds that b'_1 is the shortest computed vector [LLL82].

Analysing the complexity of the LLL algorithm leads to Corollary 1.

Corollary 1. *Let $L \in \mathbb{Z}^n$ with $L^B = \{b_1, ..., b_\omega\}$ and let $X \in \mathbb{Z}_{\geq 2}$ such that $||b_i||_2^2 \leq X$ for $1 \leq i \leq \omega$. Then the LLL algorithm requires $\mathcal{O}(\omega^3 \cdot n \cdot log(X))$ arithmetic operations on integers of size $\mathcal{O}(\omega \cdot log(X))$. Using naive arithmetic, the running time is given by $\mathcal{O}(\omega^5 \cdot n \cdot log(X)^3)$ bit operations.*

Proof. See [Gal12].

Given that L^B is a full rank lattice basis, i.e., $n = \omega$, the following theorem can be formulated:

Theorem 3.2. *Let $L \in \mathbb{Z}^n$ be a lattice with basis $\{b_1, ..., b_n\}$ where the Euclidean norm of each basis vector is bounded by $X \in \mathbb{Z}_{\geq 2}$, then the LLL algorithm computes a vector b'_1 with $||b'_1||_2 \leq 2^{(n-1)/4} \cdot det(L)^{1/n}$ using $\mathcal{O}(n^6 \cdot log(X)^3)$ bit operations.*

Proof. Follows directly from combining Theorem 3.1 and Corollary 1.

Input: $L^B = \{b_1, ..., b_\omega\} \in \mathbb{Z}^n$
Output: $L^{RB} = \{b'_1, ..., b'_\omega\} \in \mathbb{Z}^n$
compute Gram-Schmidt basis $\{b^*_1, ..., b^*_\omega\}$ and coefficients $\mu_{i,j}$, $1 \leq j < i \leq \omega$.
compute $||b^*_i||_2^2$, $1 \leq i \leq n$.
set $k = 2$.
while $k \leq \omega$ **do**
  for $j = k - 1$ **downto** 1 **do**
    set $q_j = \lfloor(\mu_{k,j} + 0.5)\rfloor$.
    set $b_k = b_k - q_j b_j$.
    recompute $\mu_{k,j}$, $1 \leq j < k$.
  end
  if $||b^*_k||_2^2 \geq (\delta - \mu^2_{k,k-1}) \cdot ||b^*_{k-1}||_2^2$ **then**
    set $k = k + 1$.
  else
    swap b_k with b_{k-1}.
    recompute $b^*_k,\ b^*_{k-1}$.
    recompute $||b^*_k||_2^2,\ ||b^*_{k-1}||_2^2$.
    recompute $\mu_{k,j},\ \mu_{k-1,j}$, $1 \leq j < k$.
    recompute $\mu_{i,k},\ \mu_{i,k-1j}$, $k < i < \omega$.
    set $k = max\{2, k - 1\}$.
  end
end
set $\{b'_1, ..., b'_\omega\} = \{b_1, ..., b_\omega\}$.
return $\{b'_1, ..., b'_\omega\}$.

Algorithm 3.1: The LLL algorithm with $\delta = 3/4$

3.2 Finding Small Roots of Bivariate Integer Polynomial Equations

In this section we present the results of *Finding Small Roots of Bivariate Integer Polynomial Equations: A Direct Approach* [Cor07] by strictly separating between describing the algorithm and proving its functionality as opposed to [Cor04] and [Cor07]. Subsequently, the application on factorization and a sample calculation are provided to get a better understanding of Coron's algorithm and its parameters.

Given a bivariate polynomial $p(x, y) = \sum_{0 \leq i,j \leq \delta} p_{i,j} x^i y^j$ with $p_{i,j}, x, y \in \mathbb{Z}$ and maximum degree δ in each variable separately, the goal is to find its roots (x_0, y_0) such that $p(x_0, y_0) = 0$. Solving systems of multivariate polynomial equations (and thus computing the roots of bivariate polynomial equations) is assumed to be NP-hard.[4] Nevertheless, using the following algorithm and satisfying that $x_0 < X$, $y_0 < Y$, and $XY < W^{2/(3\delta)}$ with $W = max_{i,j}|p_{ij}|X^i Y^j$, the roots of the input polynomial bounded by X and Y can be computed. The existence of such an algorithm was shown by Coppersmith by proving the following theorem in [Cop97]:

[4]For solving systems of multivariate polynomials over a finite field instead over the integers the NP-hardness has been proven [GJ79].

Theorem 3.3. ***(Coppersmith).*** *Let $p(x, y)$ be an irreducible polynomial in two variables over $\mathbb{Z}$, of maximum degree δ in each variable, separately. Let X and Y be upper bounds on the desired integer solution (x_0, y_0), and let $W = max_{i,j}|p_{ij}|X^iY^j$. If $XY < W^{2/(3\delta)}$, then in time polynomial in $(log(W), 2^\delta)$, one can find all integer pairs (x_0, y_0) such that $p(x_0, y_0) = 0$, with $|x_0| \leq X$ and $|y_0| \leq Y$.*

We divide Coron's proof of Theorem 3.3 by first presenting the algorithm followed by the proof that the algorithm finds (x_0, y_0) in polynomial time.

Consider the set of polynomials

$$\mathcal{S} = \{s_{a,b}(x, y)|s_{a,b}(x, y) = x^a \cdot y^b \cdot p(x, y), 0 \leq a, b < k\} \tag{2}$$

with $k \in \mathbb{N}\backslash\{0\}$ (how to determine the value of k is described below).

The algorithm makes use of two indices $(i_0, j_0) \in \mathbb{N} \times \mathbb{N}$ with $0 \leq i_0, j_0 \leq \delta$ to define the matrix $S \in \mathbb{Z}^{k^2 \times k^2}$ whose rows are build from the coefficients of polynomials contained in $\mathcal{S}$ restricted to the monomials $x^{i_0+i}y^{j_0+j}$ for $0 \leq i, j, < k$. Each column of S is associated with a monomial. (i_0, j_0) has to be chosen according to Lemma 3.2 (see below) which, i.a., implies that the determinate of S is unequal to zero. After computing (i_0, j_0) and constructing S, n is set to $|det(S)|$.

Next, consider the second set of polynomials

$$\mathcal{R} = \{r_{i,j}(x, y)|r_{i,j}(x, y) = x^i \cdot y^j \cdot n, 0 \leq i, j < k + \delta\}. \tag{3}$$

From the coefficients of the polynomials in $\mathcal{S}$ and $\mathcal{R}$, a lattice $M \in \mathbb{Z}^{(k^2+(k+\delta)^2)\times(k+\delta)^2}$ of rank $(k + \delta)^2$ is constructed. Each column of M is associated with a monomial. Complying with [Cor07], S forms the upper left $k^2 \times k^2$ block of M which is done by reordering the monomials identifying the columns of M accordingly. Furthermore, the rows formed by the polynomials of $\mathcal{R}$ are ordered in such a way that the corresponding coefficients form a diagonal matrix (compare to the example depicted in Figure 2).

M is a matrix consisting of the blocks $S \in \mathbb{Z}^{k^2\times k^2}$, $T \in \mathbb{Z}^{k^2\times(\delta^2+2k\delta)}$, and $L \in \mathbb{Z}^{(k+\delta)^2\times(k+\delta)^2}$:

$$M = \begin{bmatrix} S & T \\ & L \end{bmatrix}.$$

The next goal is to compute a basis of a sublattice L_2 of L which has a smaller dimension than L. This is to reduce the calculation steps for the lattice reduction which is applied in a subsequent step.

Let $adj(S)$ be the adjugate matrix of S (see Appendix A.6) and $\omega = \delta^2 + 2k\delta$. Before computing L_2, the auxiliary matrix M_2 has to be computed:

$$M_2 = \begin{bmatrix} I_{k^2\times k^2} & 0_{k^2\times k^2} & 0_{k^2\times\omega} \\ adj(S)_{k^2\times k^2} & I_{k^2\times k^2} & 0_{k^2\times\omega} \\ 0_{\omega\times k^2} & 0_{\omega\times k^2} & I_{\omega\times\omega} \end{bmatrix} \cdot M = \begin{bmatrix} S_{k^2\times k^2} & T_{k^2\times\omega} \\ 0_{k^2\times k^2} & T'_{k^2\times\omega} \\ 0_{\omega\times k^2} & nI_{\omega\times\omega} \end{bmatrix}$$

whereupon L_2 can be extracted:

$$L_2 = \begin{bmatrix} T'_{k^2\times\omega} \\ nI_{\omega\times\omega} \end{bmatrix}. \tag{4}$$

	x^3y^2	x^2y^2	x^3y	x^2y	x^3y^3	x^2y^3	xy^3	xy^2	xy	x^3	y^3	x^2	y^2	x	y	1
$s_{1,1}(x,y)$	a_2	a_4	a_5	a_7	a_1	a_3	a_6	a_8	a_8	a_9	0	0	0	0	0	0
$s_{1,0}(x,y)$	a_1	a_3	a_2	a_4	0	0	0	a_6	a_8	a_5	0	a_7	0	a_9	0	0
$s_{0,1}(x,y)$	0	a_2	0	a_5	0	a_1	a_3	a_4	a_7	0	a_6	0	a_8	0	a_9	0
$s_{0,0}(x,y)$	0	a_1	0	a_2	0	0	0	a_3	a_4	0	0	a_5	a_6	a_7	a_8	a_9
$r_{3,2}(x,y)$	nX^3Y^2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$r_{2,2}(x,y)$	0	nX^2Y^2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$r_{3,1}(x,y)$	0	0	nX^3Y	0	0	0	0	0	0	0	0	0	0	0	0	0
$r_{2,1}(x,y)$	0	0	0	nX^2Y	0	0	0	0	0	0	0	0	0	0	0	0
$r_{3,3}(x,y)$	0	0	0	0	nX^3Y^3	0	0	0	0	0	0	0	0	0	0	0
$r_{2,3}(x,y)$	0	0	0	0	0	nX^2Y^3	0	0	0	0	0	0	0	0	0	0
$r_{1,3}(x,y)$	0	0	0	0	0	0	nXY^3	0	0	0	0	0	0	0	0	0
$r_{1,2}(x,y)$	0	0	0	0	0	0	0	nXY^2	0	0	0	0	0	0	0	0
$r_{1,1}(x,y)$	0	0	0	0	0	0	0	0	nXY	0	0	0	0	0	0	0
$r_{3,0}(x,y)$	0	0	0	0	0	0	0	0	0	nX^3	0	0	0	0	0	0
$r_{0,3}(x,y)$	0	0	0	0	0	0	0	0	0	0	nY^3	0	0	0	0	0
$r_{2,0}(x,y)$	0	0	0	0	0	0	0	0	0	0	0	nX^2	0	0	0	0
$r_{0,2}(x,y)$	0	0	0	0	0	0	0	0	0	0	0	0	nY^2	0	0	0
$r_{1,0}(x,y)$	0	0	0	0	0	0	0	0	0	0	0	0	0	nX	0	0
$r_{0,1}(x,y)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	nY	0
$r_{0,0}(x,y)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	n

Fig. 2: Lattice M for polynomial $a_1x^2y^2 + a^2x^2y + a_3xy^2 + a_4xy + a_5x^2 + a_6y^2a_7x + a_8y + a_9$ with $k = 2$ and $(i_0, j_0) = (2, 1)$

$L_2^B \in \mathbb{Z}^{\omega\times\omega}$ is obtained from the Hermite normal form (see Appendix A.6) of L_2 by removing the zero row vectors. Each column of L_2^B is multiplied with the upper bound X^iY^j for x^iy^j where i, j correspond to the powers of the monomial x^iy^j associated with the regarded column and obtain $L_2'^B$. Subsequently, the reduced lattice basis $L_2'^{RB}$ is computed by applying the LLL algorithm on $L_2'^B$.

The shortest row vector b_1 is extracted from $L_2'^{RB}$ in order to construct the polynomial $h(xX, yY)$ by associating the vector entries with their corresponding monomials. Dividing the coefficients of $h(xX, yY)$ by the appropriate divisor X^iY^j results in $h(x, y)$.

By construction of the polynomials in $\mathcal{S}$ and $\mathcal{R}$, for all $s_{a,b}(x, y) \in \mathcal{S}$ and $r_{i,j}(x, y) \in \mathcal{R}$ it holds that $s_{a,b}(x_0, y_0) = 0 \ mod \ n$ and $r_{i,j}(x_0, y_0) = 0 \ mod \ n$. Since $h(x, y)$ is a linear combination of the elements in $\mathcal{S}$ and $\mathcal{R}$, it holds that $h(x_0, y_0) = 0 \ mod \ n$. By satisfying the restrictions on $h(xX, yY)$ which are described below, $h(x_0, y_0) = 0$ holds over the integers.

Since S is invertible because $det(S) \neq 0$, it can be shown that $h(x, y)$ is not a multiple of $p(x, y)$ (see [Cor07]). Otherwise, $p(x, y)$ can not be a multiple of $h(x, y)$ because $p(x, y)$ is irreducible by assumption. It follows that $p(x, y)$ and $h(x, y)$ are algebraically independent and have at least one common root (x_0, y_0). The resultant $Q(x) = Res_y(p(x, y), h(x, y))$ returns a non-zero univariate integer polynomial with y eliminated and sharing the common roots of $p(x, y)$ and $h(x, y)$ for variable x, i.e., $Q(x_0) = 0$. With the knowledge of x_0, y_0 can be computed by solving $p(x_0, y) = 0$. The roots of $Q(x)$ and $p(x_0, y)$ can be computed by common root finding algorithms. Algorithm 3.2 gives an overview of the method described above.

Next, it is shown according to [Cor07] that the presented algorithm computes the roots (x_0, y_0) of $p(x, y)$ with $x_0 < X$, $y_0 < Y$ and performs the computation in polynomial time.

Due to the calculation steps of the lattice reduction algorithm $h(x, y)$ and $h(xX, yY)$ are linear combinations of $s_{a,b}(x, y)$, $r_{i,j}(x, y)$ and $s_{a,b}(xX, yY)$,

Input: $p(x,y), k, X, Y$
Output: (x_0, y_0)
Compute i_0, j_0 as defined in Lemma 3.2.
Construct M from $s_{a,b}(x,y)$, $r_{i,j}(x,y)$, column, and row permutation.
Compute M_2
$= [[I_{k^2\times k^2}, 0_{k^2\times k^2}, 0_{k^2\times\omega}], [adj(S)_{k^2\times k^2}, I_{k^2\times k^2}, 0_{k^2\times\omega}], [0_{\omega\times k^2}, 0_{\omega\times k^2}, I_{\omega\times\omega}]] \cdot M$
$= [[S_{k^2\times k^2}, T_{k^2\times\omega}], [0_{k^2\times k^2}, T'_{k^2\times\omega}], [0_{\omega\times k^2}, nI_{\omega\times\omega}]]$.
Define $L_2 = [T'_{k^2\times\omega}, nI_{\omega\times\omega}]$.
Obtain $L_2^B \in \mathbb{Z}^{\omega\times\omega}$ by computing the HNF of L_2.
Compute $L_2'^B$ by multiplying each column with the corresponding X^iY^j.
Compute $L_2'^{RB} = [[b_1], [b_2], ..., [b_\omega]] = LLL(L_2'^B)$, with $b_i \in \mathbb{Z}^{1\times\omega}$.
Construct $h(x,y)$ from b_1, dividing each coefficient h_{ij} of h by X^iY^j.
Compute $Q(x) = Res_y(h(x,y), p(xy))$.
Compute $x_0 = roots(Q(x))$.
Compute $y_0 = roots(p(x_0, y))$.
return (x_0, y_0).

Algorithm 3.2: Finding small roots of bivariate integer equations

$r_{i,j}(xX, yY)$, respectively. Basing on the construction of $s_{a,b}(x,y)$ and $r_{i,j}(x_0,y_0)$, these polynomials have the same roots in $\mathbb{Z}_n$ as $p(x,y)$ and thus, $h(x_0,y_0) = 0 \ mod \ n$.

The goal is to satisfy the restrictions $|x_0| < X$ and $|y_0| < Y$ such that $h(x_0,y_0) = 0$ holds over the integers. To determine appropriate bounds the following Lemma due to Howgrave-Graham is considered:

Lemma 3.1. ***(Howgrave-Graham).*** *Let $h(x,y) \in \mathbb{Z}[x,y]$ which is a sum of at most ω monomials. Suppose that $h(x_0,y_0) = 0 \ mod \ n$ where $|x_0| \le X$ and $|y_0| \le Y$ and $||h(xX,yY)|| < n/\sqrt{\omega}$. Then $h(x_0,y_0) = 0$ holds over the integers.*

As it was mentioned above, $L_2'^B$ is the same matrix as L_2^B with the difference that each column was multiplicated with the term X^iY^j where i and j coincide with the exponents of the monomial the regarded column is associated with. The reduced lattice base $L_2'^{RB}$ of $L_2'^B$ is a $\omega\times\omega$ matrix with $\omega = \delta^2 + 2\cdot k\cdot\delta$. $h(xX,yY)$ is constructed from the shortest row vector of $L_2'^{RB}$.

According to Theorem 3.2 $h(xX,yY)$ satisfies:

$$||h(xX,yY)|| \le 2^{(\omega-1)/4} \cdot det(L_2^B)^{1/\omega}. \tag{5}$$

With respect to Lemma 3.1 it has to be shown that

$$2^{(\omega-1)/4} \cdot det(L_2^B)^{1/\omega} \le \frac{n}{\sqrt{\omega}}. \tag{6}$$

As a result of some simple matrix calculations as presented in [Cor07] the determinant of $L_2'^B$ can be derived as:

$$det(L_2'^B) = n^{\omega-1} \cdot \frac{(XY)^{\delta+k-1\cdot(\delta+k)^2/(2-(k-1)\cdot k^2)/2}}{(X^{i_0}Y^{j_0})}$$

Combining this result with Inequality 6, exponentiate both sides with ω, followed by a division with $n^{\omega-1}$,

$$2^{\omega\cdot(\omega-1)/4} \cdot \frac{(XY)^{\delta+k-1\cdot(\delta+k)^2/(2-(k-1)\cdot k^2)/2}}{(X^{i_0}Y^{j_0})} \le \frac{n}{\omega^{\omega/2}} \tag{7}$$

has to be satisfied in order to apply Lemma 3.1 to $h(x,y)$. The following Lemma gives a lower and an upper bound for $n = |det(S)|$, the remaining term which has to be estimated.

Lemma 3.2. ***(Coron).*** *Given (u,v) such that $W = |p_{uv}|X^uY^u$, let indices (i_0, j_0) maximize the quantity $8^{(i-u)^2+(j-v)^2}|p_{ij}|X^iY^j$. Then*

$$\left(\frac{W}{X^{i_0}Y^{j_0}}\right)^{k^2} \cdot 2^{-6k^2\delta^2-2k^2} \leq |det(S)| \leq \left(\frac{W}{X^{i_0}Y^{j_0}}\right)^{k^2} \cdot 2^{k^2}. \tag{8}$$

Substituting n with its lower estimate, given by Lemma 3.3 and $\sqrt{\omega}$ by its upper estimate $2^{\omega/2}$ given by 7, the following equation has to be satisfied:

$$\begin{aligned} 2^{\omega\cdot(\omega-1)/4} \cdot (XY)^{(\delta+k-1)\cdot(\delta+k)^2/2-(k-1)\cdot k^2/2} &\leq W^{k^2} \cdot 2^{-6k^2\cdot\delta-2\cdot k^2} \cdot 2^{-\omega^2/2} \\ \Leftrightarrow XY \leq W^{\frac{k^2}{(\delta+k-1)\cdot(\delta+k)^2/2-(k-1)\cdot k^2/2}} &\cdot 2^{\frac{-6k^2\delta^2-2k^2-\omega^2/2-\omega\cdot(\omega-1)/4}{(\delta+k-1)\cdot(\delta+k)^2/2-(k-1)\cdot k^2/2}} \\ \Rightarrow XY < W^{\frac{2k^2}{\delta\cdot(3k^2+k\cdot(3\delta-2)+\delta^2-\delta)}} &\cdot 2^{-9\delta} \\ \Rightarrow XY < W^{2/(3\delta)-1/k} &\cdot 2^{-9\delta} \end{aligned}$$

By defining $k := \lfloor log(W) \rfloor$ and knowing or guessing the $\mathcal{O}(\delta)$ high or low order bits of x_0 or y_0, $XY < W^{2/(3\delta)}$ has to be satisfied.

To complete the proof of Theorem 3.3 it has to be shown that the roots (x_0, y_0) can be computed in polynomial time.

Since the runtime of Algorithm 3.2 is dominated by arithmetic operations of the lattice reduction (compare to section 3.1), a runtime of $\mathcal{O}(\omega^6 log(B)^3)$ is achieved by applying Theorem 3.2. ω is the column rank of the lattice (in our case $\omega = \delta^2 + 2k\delta$) and B is an upper bound for the values occurring in the lattice which has to be reduced. By reducing the columns of $L_2'^B$ by a modulo nX^iY^j computation with i, j chosen according to the exponents of the monomials x^iy^j associated with the respective column and calling up the upper bound of n given by Lemma 3.3, an upper bound for the entries of $L_2'^B$ is given by $\mathcal{O}(W^{k^2})$ and thus $B = W^{k^2}$. The running time for Algorithm 3.2 is in

$$\begin{aligned} \mathcal{O}((\delta^2+2k\delta)^6 \cdot log(W^{k^2})^3) &\overset{k\geq\delta}{=} \mathcal{O}((\delta^6k^6)\cdot k^6 \cdot log(W)^3) \\ &\overset{k=\lfloor log(W)\rfloor}{=} \mathcal{O}(\delta^6 \cdot log(W)^{15}) \end{aligned} \tag{9}$$

Altogether, Algorithm 3.2 achieves a polynomial runtime in $(2^\delta, log(W))$ which follows from Equation 9 and the fact that the $\mathcal{O}(\delta)$ high order bits of x_0 or y_0 have to be known.

Using Algorithm 3.2 for recovering the factorization of $N = p \cdot q$, $\delta = 1$ and thus a polynomial runtime in $\mathcal{O}(log(W))$.

3.2.1 Factorization Using Coron's Direct Approach. Algorithm 3.2 has an application on factorizing composites $N = p \cdot q$ with $p, q \in \Phi$. It is premised that half of the most or least significant bits of one of the prime factors of N are known and that p and q have the same order, i.e., $l_p = l_q$.

The following theorem has been proven by Coppersmith in [Cop97]:

Theorem 3.4. ***(Coppersmith).*** *Given $N = pq$ and the high- or low-order $1/4 \cdot ld(N)$ bits of p, one can recover the factorization of N in time polynomial in $log(N)$.*

First, we consider the case that the 1/2 high-order bits of p are known which are stored in variable P_0. By dividing N by P_0 the 1/2 high-order bits of q are obtained and variable Q_0 is set to the result. When $p \cdot q$ was computed it might have been that a borrow bit modified the upper order bits of N. Therefore the computation has to be repeated with Q_0 increased by 1 if the factorization with Q_0 fails. Let x and y be the unknown bits of p and q, respectively. N can be represented by the following equation:

$$N = (P_0 + x) \cdot (Q_0 + y).$$

This equation can be transferred into the bivariate polynomial

$$p(x, y) = (N - P_0 \cdot Q_0) + P_0 \cdot y + Q_0 \cdot x + xy \tag{10}$$

whose integer root represents the unknown bits of p and q. To compute the integer root of Polynomial 10, Algorithm 3.2 can be applied.

Coron proposed the following parameter values for applying his algorithm [Cor07]:

$$X = Y = N^{1/4},$$

$$W = P_0 \cdot X,$$

$$k = \lfloor log(W) \rfloor.$$

Note that if more than $1/4 \cdot ld(N)$ bits of p or q are known, the value of X and Y has to be adapted and k can be reduced appropriately.

If the low-order $1/4 \cdot ld(N)$ bits of p are known, the procedure to factorize N has to be slightly modified. Q_0 can by computed from the $1/4 \cdot ld(N)$ bits of p and N by applying the *Multivariate Hensel's Lemma* [GCL92] to obtain the following equation derived by Heninger and Shacham in [HS09]:

$$Q_0[i] \equiv (N - Q_{0,i-1} P_{0,i-1})[i] - P_0[i] \ mod \ 2.$$

The notation of $Var[i]$ denotes the i-th bit of integer Var. $Q_0[1]$ is initialized with 1 because q is a prime number. $Q_{0,i}$ and $P_{0,i}$ denote the value of Q_0 and P_0 through bit 0 to bit i.

Let l_p and l_{P_0} be the bit length of p and P_0, respectively. In [Cop97] Coppersmith defines

$$p(x, y) = 2^{l_{P_0}} xy + Q_0 x + P_0 y + (P_0 Q_0 - N)/2^{l_{P_0}}$$

such that the unknown bits x_0 and y_0 of

$$p = 2^{l_{P_0}} x_0 + P_0$$

and

$$q = 2^{l_{P_0}} y_0 + Q_0$$

are the integer roots of

$$p(x, y) = \frac{pq - N}{2^k}.$$

The upper bounds X and Y are defined by

$$|x_0| < X^{l_p - l_{P_0}},$$

$$|y_0| < Y = X.$$

Constructing the target polynomial $p(x, y)$ with the same approach as for the case that the high-order bits of p are known yields

$$p'(x, y) = (2^{l_{P_0}} + P_0) \cdot (2^{l_{P_0}} + Q_0) - N$$

which can not be used as input for Algorithm 3.2 because $p'(x, y)$ is not irreducible since all coefficients of $p'(x, y)$ have the common factor $2^{l_{P_0}}$ and thus the premises for Theorem 3.3 are not satisfied.

3.2.2 Sample Calculation on Factorization. Consider the natural number $N = 337237$ which is the product of the two prime numbers $p = 563$ and $q = 599$. According to Theorem 3.3 it is sufficient to know the high-order (or low-order) $1/4 \cdot ld(N)$ bits of p to factorize N in polynomial time using the algorithm introduced by Coron in [Cor07]. In practice there might be one or two additional bits to be known which is referable to lost information caused by roundings. By complying with Coron's proposed bounds of W, X and Y we obtain $W = 5328$ and $X = Y = 9$. We choose $k = 2$ to illustrate Algorithm 3.2. The computation of (i_0, j_0) gives $i_0 = 1$, $j_0 = 0$ which implicates the value $n = 122825015296$ as a result of determining the determinate of S.

Knowing the $1/4 \cdot ld(N) + 1$ high-order bits of p we have $P_0 = 560$ and $Q_0 = 592$. The polynomial its root we have to compute to factorize N results in $p(x, y) = -xy - 592x - 560y + 5717$.

Following the algorithm, we construct M from $s_{a,b}$ and $r_{i,j}$, moving the columns in such a way that the monomials $x^{i_0+i}y^{j_0+j}$ with $0 \leq i, j < k$ correspond to the matrix left-hand block, and alternating the row vectors formed by the $r_{i,j}$-polynomials that the coefficients are located on the diagonal of L:

$$\text{M} = \begin{array}{c|cccc|ccccc}
 & x^2y & x^2 & xy & x & x^2y^2 & xy^2 & y^2 & y & 1 \\
\hline
s_{1,1}(x,y) & -592 & 5717 & 0 & 0 & -1 & -560 & 0 & 0 & 0 \\
s_{1,0}(x,y) & -1 & -560 & -592 & 5717 & 0 & 0 & 0 & 0 & 0 \\
s_{0,1}(x,y) & 0 & -592 & 0 & 0 & 0 & -1 & -560 & 5717 & 0 \\
s_{1,1}(x,y) & 0 & -1 & 0 & -592 & 0 & 0 & 0 & -560 & 5717 \\
\hline
r_{1,1}(x,y) & nX^2Y & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
r_{1,0}(x,y) & 0 & X^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
r_{0,1}(x,y) & 0 & 0 & nXY & 0 & 0 & 0 & 0 & 0 & 0 \\
r_{0,0}(x,y) & 0 & 0 & 0 & nX & 0 & 0 & 0 & 0 & 0 \\
r_{2,2}(x,y) & 0 & 0 & 0 & 0 & nX^2Y^2 & 0 & 0 & 0 & 0 \\
r_{2,1}(x,y) & 0 & 0 & 0 & 0 & 0 & nXY^2 & 0 & 0 & 0 \\
r_{1,2}(x,y) & 0 & 0 & 0 & 0 & 0 & 0 & nY^2 & 0 & 0 \\
r_{2,0}(x,y) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & nY & 0 \\
r_{0,2}(x,y) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n \\
\hline
\end{array}$$

To compute the basis of sublattice L_2 we compute we first compute M_2 and transform $\left[T' \; nI_\omega\right]^T$ into *HNF* which gives us L_2^B after removing the zero row vectors:

$$L_2^B = \begin{bmatrix} 350464 & 110112 & 12981167616 & 52484009824 & 101789114624 \\ 0 & 350464 & 196259840 & 4635587328 & 80707653632 \\ 0 & 0 & 122825015296 & 0 & 0 \\ 0 & 0 & 0 & 122825015296 & 0 \\ 0 & 0 & 0 & 0 & 122825015296 \end{bmatrix}$$

Subsequently, the columns of L_2^B are multiplicated with the associated X^iY^j terms. Applying the LLL-algorithm on $L_2'^B$ gives us the reduced lattice basis:

$$L_2'^{RB} = \begin{bmatrix} 20694548736 & 8642580768 & 7253411328 & 17407971936 & -21061134080 \\ 22993943040 & -7883884224 & 25419930624 & -11204251200 & -6618863104 \\ -20694548736 & -8387092512 & 8643635712 & 24312314016 & -21056227584 \\ 20694548736 & 8898069024 & 23150458368 & 59128257888 & 59646519552 \\ 87376983552 & -160717649472 & -80451947520 & 50229165888 & 36136343040 \end{bmatrix}$$

From the shortest vector of $L_2'^{RB}$

$$b_1 = \begin{bmatrix} 20694548736 & 8642580768 & 7253411328 & 17407971936 & -21061134080 \end{bmatrix}$$

and dividing each entry by the corresponding X^iY^j term, we build the polynomial

$$h(x, y) = 3154176x^2y^2 + 11855392xy^2 + 89548288y^2 + 1934219104y - 21061134080$$

which has the same roots (x_0, y_0) as $p(x, y)$ with $|x_0| < X$ and $|y_0| < Y$. From the construction of $p(x, y)$ we know that for the root (x_0, y_0), we are interested in, both conditions hold. Computing the resultant $Q(x) = Res_y(h(x, y), p(x, y))$ gives us

$$\begin{aligned} Q(x) = 110542513766x^4 - 17195502141440x^3 + 53060406607872x^2 \\ - 872426083647488x + 2514473713139712. \end{aligned}$$

Applying a root-finding algorithm on $Q(x)$ and $p(x_0, y)$ we obtain the root $(3, 7)$ with which we can recover $p = P_0 + x_0$ and $q = Q_0 + y_0$ - the prime factors of N.

4 Related Backdoors in Cryptographic Systems

Among Anderson who was one of the firsts discussing tampering on key generation procedures of cryptosystems [And93], Young and Yung inaugurated the era of asymmetric[5] backdoors[6] for cryptosystems (especially for RSA) with their work [YY96] - [YY08] and their definition of a *Secretly Embedded Trapdoor With Universal Protection* (SETUP). They introduced a three person model involving the user, the manufacturer, and the (external) attacker. The manufacturer provides the user with a tampered publicly specified asymmetric cryptosytem (the SETUP) implemented on a cryptographic black-box device which itself is

[5]Note that the term *asymmetric* refers to the manufacturer's keys and not to the underlying (asymmetric) cryptosystem.

[6]Note that we use the term *backdoor* to refer to the mechanism which subverts a cryptosystem as well as to denote the whole backdoored cryptosystem depending on the context.

tamper-resistant after it has been issued. Furthermore, if the black-box device is successfully reverse engineered the reverse engineer should not be able to utilize the SETUP.[7] With the issued black-box device the user generates public and private key pairs as well as de- and encrypts messages. The output of the device can be observed by him. The SETUP contains the manufacturer's public key used to encrypt secret information which is included into the public output (the user's public key) of the black-box device. The encrypted secret information enables the manufacturer, after decrypting it, to efficiently reconstruct the user's private key.

As well as the manufacturer, the attacker aims to reconstruct the user's private key but only on the basis of the user's public key. By carefully designing the backdoor, the manufacturer strives to achieve that breaking the SETUP mechanism is as hard as breaking the underlying asymmetric cryptosystem, i.e., the manufacturer has the exclusive ability to derive the private keys of users which utilize the black-box device issued by him.

The properties an asymmetric backdoor has to fulfill in order to be considered as a SETUP (later called regular SETUP [YY97]) are listed in the following definition first introduced in [YY96].

Definition 4.1 ((Regular) SETUP). *A (regular) SETUP is defined as a modification made to a cryptosystem $\mathfrak{C}$ resulting in $\mathfrak{C}'$ fulfilling the following properties:*

1. ***(in/out)*** *Modifications on $\mathfrak{C}$ do not transgress the input and output specifications.*
2. ***(pub)*** *$\mathfrak{C}'$ comprises the manufacturer's public encryption function E.*
3. ***(priv)*** *$\mathfrak{C}'$ does not contain the manufacturer's private decryption function D.*
4. ***(hidden bits)*** *The public key output of $\mathfrak{C}'$ contains secret information encrypted with E where the manufacturer is able to derive the user's private key from.*

5a. ***(indistinguishability)*** *The output of $\mathfrak{C}$ and $\mathfrak{C}'$ are computationally indistinguishable (except by the manufacturer).*

6. ***(confidentiality)*** *Irrespective of whether or not the SETUP has been discovered, only the manufacturer is able to derive past and future keys.*

Later on, Young and Yung refined their definition of a SETUP in [YY97] by distinguishing between *weak* and *strong* SETUPs:

Definition 4.2 (Weak SETUP). *A weak SETUP is a regular SETUP with property* 5a *(indistinguishably) weakened:*

5b. ***(weak indistinguishability)*** *The output of $\mathfrak{C}$ and $\mathfrak{C}'$ are polynomially indistinguishable (except by the manufacturer and the user).*

Definition 4.3 (Strong SETUP). *A strong SETUP is a regular SETUP with two additional properties:*

[7] Actually, reverse engineering a black-box device means in the context of SETUPs that the reverse engineer is given the code of the SETUP and with this the public key of the manufacturer but nevertheless this excludes the information if the SETUP has been implemented on the present device.

7. ***(RE robustness)*** *Although the user is able to reverse engineer $\mathfrak{C}'$, he can not derive past and future keys. The output of $\mathfrak{C}$ and $\mathfrak{C}'$ remains computationally indistinguishable to the user.*
8. ***(uniformity)*** *The SETUP is identical in each device it is implemented.*

After there were denoted some deficiencies on Young's and Yung's previous work which violate the SETUP property of indistinguishability, the focus of succeeding backdoors lay on eliminating those deficiencies resulting in additional conditions a SETUP has to satisfy to particularly ensure indistinguishability.

The first issue, Young and Yung were aware of, was the lack of timing resistance mentioned in [CS03], i.e., basing on measurements of the running time of the operations performed by black-box devices it is possible to distinguish between $\mathfrak{C}$ and $\mathfrak{C}'$. Heretofore, all of the SETUPs proposed by Young and Yung have not been resistant against timing analysis. This issue led to an additional property a SETUP should satisfy w.r.t. condition $5a$ and $5b$ called *time-strong*.

Definition 4.4 (time-strong). *A SETUP is considered to be time-strong if on the basis of time measurements the user and the attacker are not able to distinguish between $\mathfrak{C}$ and $\mathfrak{C}'$.*

The second issue, also denoted in [CS03], concerns the distribution of the most significant bits (MSBs) of the user's public RSA modulus n w.r.t. a SETUP proposed by Young and Young in [YY96] which generates n uniformly at random. In fact, the MSBs of a number resulting from two factors of the same bit size are not uniformly distributed (see [CS03]). To avoid methods which are able to distinguish between $\mathfrak{C}$ and $\mathfrak{C}'$ by analyzing the distribution of n outputted by the black box device, a SETUP has to be MSB-strong otherwise property $5a/5b$ is not satisfied.

Definition 4.5 (MSB-strong). *An RSA-SETUP is considered to be MSB-strong if the distribution of the most significant bits of the RSA modulus converge to the distribution of a regular RSA modulus.*

Among asymmetric backdoors, there has been a lot of work on symmetric backdoors, whereas the roots can be attributed to Anderson [And93]. Due to the nature of symmetric cryptossystems, those backdoors do not belong to the group of SETUPs because of conditions 2 and 3 can not be satisfied.

Furthermore, symmetric backdoors rely on the assumption that it is not possible to reverse engineer the black-box device they are implemented on. Otherwise, the reverse engineer would be able to learn the symmetric key and thus could use the backdoor in the same way as the manufacturer does. Thus, RE-robustness can not inherently be satisfied by symmetric backdoors. However, all other SETUP properties mentioned before can be satisfied.

Notation. We write $a||b$ for the concatenation of a and b. $n\rfloor_l$ and $n\rceil_l$ indicate the l low and high order bits of integer n, respectively. To identify a backdoored cryptosystem we use the notation CS_f^v. CS refers to the target cryptosystem, f is the subfunction of CS which is affected by the backdoor, and v is the name for the backdoor to distinguish between backdoors for the same cryptosystem affecting the same subfunction. For example, RSA_{GEN1}^{PHP} refers to the RSA backdoor PHP which modifies `GEN1` of RSA.

Any cryptographic keys written in lower case letters are owned by the cryptosystem's user while upper case letter are used to indicate the manufacturer's keys.

4.1 Backdoors

In the following we discuss the mechanisms and the properties of the most prominent asymmetric and symmetric backdoors from literature.[8]

4.1.1 Asymmetric Backdoors.

Characteristically, asymmetric backdoors involve the manufacturer's public key E to hide information enabling the manufacturer to reconstruct the user's secret key using the manufacturer's private key D.

4.1.1.1 Backdoors for the Diffie-Hellman Key Exchange Protocol. $DH^{\beta_1}_{GEN2}$ is a strong SETUP for the Diffie-Hellman Key exchange protocol and was proposed by Young and Yung in [YY97]. More precisely, it is a $(1, 2)$-leakage system (see Appendix B.2) because there have to be two Diffie-Hellman key exchanges to enable the manufacturer to reconstruct one negotiated Diffie-Hellman key. To leak the necessary information, it is sufficient that $DH^{\beta_1}_{GEN2}$ is implemented for one communication party.

Let H be a cryptographically strong hash function, X and Y the manufacturer's private and public ElGamal key, and W an odd integer. W is used as a precaution measure to ensure that the user is not able to detect the presence of the SETUP in the case that H is discovered to be invertible. a and b are two integer values chosen in a way that $g_1 = g^{-Xb-W}$, $g_2 = g^{-Xb}$, and $g_3 = g^{1-aX}$ are generators in $\mathbb{Z}_p$ where p is the user's public key prime.

The first of the two DH key exchanges proceeds as usual with the difference that `GEN2` stores the randomly chosen value r_A (see Algorithm 4.1). The calculation steps in the second DH key exchange of $DH^{\beta_1}_{GEN2}$ are modified. First, a random value $t \in \{0, 1\}$ is chosen to compute the uniformly distributed value $z = g^{r_A - Wt} \cdot Y^{-ar_A - b}\ mod\ p$ in $\mathbb{Z}_p$. The value of t decides if W is involved into the computation or not. To obtain $r_{A'}$, the hash value $H(z)$ has to be computed. $A' = g^{r'_A}$ is send over a public channel to the other communication party in order to negotiate the DH key. Algorithm 4.2 gives an overview of the computation steps performed.

> **Input**: p, g
> **Output**: (r_A, A)
> 1 pick $r_A \in_{rnd} \mathbb{Z}_{p-1}$.
> 2 compute $A = g^{r_A}\ mod\ p$.
> 3 store r_A.
> 4 **return** (r_A, A).

Algorithm 4.1: Modified `GEN2` of $DH^{\beta_{1.1}}_{GEN2}$

[8] The properties are discussed as far as it has been done in the corresponding publications.

Input: $p, g, (Y, W, a, b)$
Output: (r'_A, A')
pick $t \in_{rnd} \{0, 1\}$.
compute $z = g^{r_A - Wt} \cdot Y^{-ar_A - b} \; mod \; p$.
compute $r'_A = H(z)$.
compute $A' = g^{r'_A} \; mod \; p$.
return (r'_A, A').

Algorithm 4.2: Modified `GEN2` of $DH^{\beta_{1.2}}_{GEN2}$

In order to reconstruct the latter DH key, the manufacturer needs to intercept the messages A, A', and message B' which is sent by the opposite communication party. Ultimately, an ElGamal encryption on z is accomplished. By utilizing Algorithm 4.3, the manufacturer is able to compute $r_{A'}$. Exponentiating B' with $r_{A'}$ modulo p yields the negotiated DH key of the second DH key exchange.

The two if-conditions in Algorithm 4.3 are necessary to determine the value of t which was used in $DH^{\beta_1}_{GEN2}$. If both if-conditions fail, the manufacturer has to assume the absence of the backdoor.

For the case $t = 0$, the manufacturer is able to compute z with the following equation:

$$z = z_1 = m_1 \cdot r^{-X} \; mod \; p = g^{r_A} \cdot g^{X(-ar_A - b)} = g^{r_A} \cdot Y^{-ar_A - b}.$$

Otherwise, for $t = 1$ the same computation is repeated by involving $z_2 = z_1 / g^W$ instead of z_1.

Input: $p, g, (X, W, a, b)$
Output: (r'_A)
compute $r = m_1^a \cdot g^b \; mod \; p$.
compute $z_1 = m_1 / r^X \; mod \; p$.
if $A' == g^{H(z_1)} \; mod \; p$ **then**
  compute $r'_A = H(z_1)$.
end
compute $z_2 = z_1 / g^W$.
if $A' == g^{H(z_2)} \; mod \; p$ **then**
  compute $r'_A = H(z_2)$.
end
return (r').

Algorithm 4.3: $DH^{\beta_1}_{GEN2}$ symmetric key reconstruction

$DH^{\beta_1}_{GEN2}$ can be extended to an $(l, l+1)$-leakage scheme by *chaining* the leaked values. For this purpose, the i-th application of the scheme ($i \leq l$) involves the previous r_A value in the computation of z. After the SETUP has been utilized l-times, a new random value r_A is chosen.

4.1.1.2 Backdoors for RSA. The following two RSA backdoors were proposed by Young and Yung in [YY96].

The first one ($RSA^{\beta_1}_{GEN2}$), a regular SETUP, is outlined in Algorithm 4.4. It is implemented into the RSA key generation function GEN2 and the hidden information to reconstruct the prime factors of the user's public modulus n is embedded within the public exponent $e = p^E \ mod \ N$, where (N, E) is the public key of the manufacturer which is available in GEN2 of $RSA^{\beta_1}_{GEN2}$. e and $\varphi(n)$ have to be relatively prime, otherwise GEN1 has to be invoked again to determine a new prime p of the same bit length.

Given the user's public key (n, e), the manufacturer is able to factorize n by utilizing his own private key (N, D): $p = e^D \ mod \ N$ (see Algorithm 4.5). This backdoor has the disadvantage that e is in size of N. This fact makes the backdoor inapplicable for several software which commonly require small public exponents as it is the case for PGP (Pretty Good Privacy) [Zim95]. Another disadvantage is, that if the user gets to know the public key of the manufacturer, e.g., by reverse engineering the SETUP device, he is able to distinguish between $\mathfrak{C}$ and $\mathfrak{C}'$ in polynomial time by checking if $e = p^E \ mod \ N$ holds.

Input: $(N, E), p, q$
Output: e
$e = p^E \ mod \ N$.
while $gcd(e, \varphi(n)) \neq 1$ **do**
  pick $p \in_{rnd} \Phi_{l_p}$.
  $e = p^E \ mod \ N$.
end
return e.

Algorithm 4.4: Modified GEN2 of $RSA^{\beta_1}_{GEN2}$

Input: $(n, e), (N, D)$
Output: p, q
given (n, e), compute $p = e^D \ mod \ N$.
compute $q = n \cdot p^{-1}$.
return p, q.

Algorithm 4.5: $RSA^{\beta_1}_{GEN2}$ prime factor reconstruction

The second RSA backdoor $RSA^{PAP}_{GEN1,2}$ is called *Pretty-Awful-Privacy* (PAP) which gives the manufacturer of PAP the exclusive ability to reconstruct the user's private keys. The rough idea of $RSA^{PAP}_{GEN1,2}$ is to hide one of the RSA prime factors into the user's public modulus n. Notice that the implementation of PAP modifies the RSA key generation functions GEN1 and GEN2.

$RSA^{PAP}_{GEN1,2}$ includes the manufacturer's public RSA key (N, E) and subverts RSA as described in the following. First, a random prime p of bitsize $l_N = l_n/2$ is drawn. Utilizing an invertible keyed pseudo-random function[9] F^E_K, p is randomized as long as the result p' is smaller than N by increasing K by i which is initialized with 0 and incremented for the case that p' is greater than or equal

[9] An invertible function to achieve pseudo-randomness including key K in its computation (not further specified in [YY96]). F^D_K is denoted to be the inverse of F^E_K.

to N. This can be done up to $B_1 = 16$ times whereupon a new p is drawn if the required condition of p' was not satisfied after at most B_1 iterations. Randomizing p is necessary to enable that p can be larger than N. Subsequently, p' is encrypted with the manufacturer's RSA key resulting in p''. p''' is obtained by applying another invertible keyed pseudo-random function G_K^E on p'' to ensure the randomness of p'''. A random number r of bitsize l_N is drawn and concatenated with p''' resulting in $X = p'''||r$. The pending value q is the result of X/p. If q does not pass a certain primality test, p'' is recomputed by applying $G_{K+1}^E(p'')$. This can be done up to $B_2 = 512$ times. If a prime q is not found by then, a new prime p is drawn and all previous computations have to be repeated. Finally, the user's public exponent e is set to 17 and increased by 2 as long as e and $\varphi(n)$ are relatively prime. `GEN3` computes the user's private key as usual on the base of the output of $RSA_{GEN1,2}^{PAP}$. Note that the bitsize of the generated keys is twice as long as the bitsize of the manufacturer's key because p and q both are l_N-bit quantities. $RSA_{GEN1,2}^{PAP}$ is illustrated in Algorithm 4.6.

Input: (N, E)
Output: (p, q, n, e)
pick $p \in_{rnd} \Phi_{l_N}$.
set $i = 0$.
repeat
  compute $p' = F_{K+i}^E(p)$.
  set $i = i + 1$.
until $p' \geq N$ *and* $i < 16$
if $i == 16$ **then**
  goto 1.
end
compute $p'' = p'^E \ mod \ N$.
set $j = 0$.
repeat
  compute $p''' = G_{K+j}^E(p'')$.
  pick $r \in_{rnd} \{0,1\}^{l_N}$.
  set $X = p'''||r$.
  compute $q = X/p$.
until $PrimalityTest(q) == True$ *and* $j < 512$
if $j == 512$ **then**
  goto 1.
end
compute $n = p \cdot q$.
set $e = 17$.
while $gcd(e, \varphi(n)) \neq 1$ **do**
  set $e = e + 2$.
end
return (p, q, n, e).

Algorithm 4.6: Modified `GEN1` and `GEN2` of $RSA_{GEN1,2}^{PAP}$

The prime factor reconstruction for $RSA^{PAP}_{GEN1,2}$ (see Algorithm 4.7) is sophisticated because the manufacturer does not know the exact values of key K which were used for randomization purposes but only the intervals they were chosen from. Therefore, the manufacturer has to compute all possible values until he finds a prime factor of n.

First, a variable U is set to the $l_n/2$ high order bits of n. Subsequently, all 512 possible values for p'' are computed applying $G^D_{K+j}(U)$ for each $j \in \{0, ..., B_2-1\}$. These values are decrypted with the manufacturer's private key. The resulting values which are candidates for p' are passed to F^D_{K+i} with $i \in \{0, ..., B_1 - 1\}$. For each candidate of p' at most 16 possible values are computed. This process is stopped once a prime factor of n is found; p and q are returned. If it was not possible to find the prime factor p, the same algorithm has to be applied on the value $U + 1$ because a borough bit might have modified the upper order bits of n when n was computed in $RSA^{PAP}_{GEN1,2}$.

```
Input: (n, e), (N, D)
Output: (p, q)
set repeat = False.
set U = n]_{l_n/2}.
set list_G = [ ].
for j <- 0 to 511 do
    compute p'' = G^D_{K+j}(U).
    append p'' to list_G.
end
set list_F = [ ].
for l in list_G do
    p' = l^D mod N.
    append p' to list_F.
end
set list_p = [ ].
for p in list_F do
    for i <- 0 to 15 do
        compute p = F^D_{K+i}(p).
        if n mod p == 0 then
            compute q = n/p.
            return (p, q).
        end
    end
end
if repeat == False then
    set repeat = True.
    set U = U + 1.
    goto 3.
end
return -1.
```

Algorithm 4.7: $RSA^{PAP}_{GEN1,2}$ prime factor reconstruction

$RSA^{PAP}_{GEN1,2}$ satisfies the properties of a regular SETUP except property $5a/5b$ because the MSB-strong property is not given as denoted in [CS03].

Furthermore, with the knowledge of the interior of $RSA^{PAP}_{GEN1,2}$, comprising the manufacturer's public key, the user is able to repeat the computation steps for q and compare the result to n. If the most significant bits of n coincide, the user is able to decide in polynomial time that the output was computed by $RSA^{PAP}_{GEN1,2}$ and thus RE-robustness is not given.

Nevertheless, $RSA^{PAP}_{GEN1,2}$ can be extended to $RSA^{PAP'}_{GEN1,2}$ fulfilling the RE-robustness property. The extension is implemented by involving the Probabilistic Bias Removal Method (see Appendix B.3) and $DH^{\beta_1}_{GEN2}$ in the generation process of p and q (see [YY97]). $RSA^{PAP'}_{GEN1,2}$ contains a public ElGamal key of the manufacturer instead of a public RSA key.

The next backdoor, RSA^{EC}_{GEN1}, proposed by Young and Yung in [YY06] satisfies the conditions of a strong SETUP. The focus of RSA^{EC}_{GEN1} lies on the lack of security the previous SETUPs from above entailed: the user's 1024-bit key provides only the security of the manufacturer's 512-bit key. Since in 2003 it was shown that a 576-bit RSA composite can be factorized [Wei03] (in 2005 a 640 bit RSA composite was factored [Wei05]), the SETUPs from [YY96] can not be considered to be secure anymore for the common key size of 1024 bit.

To overcome this issue RSA^{EC}_{GEN1} involves an *Elliptic Curve Diffie-Hellman* (ECDH) manufacturer key basing on the security of solving the *Elliptic Curve Discrete Logarithm Problem* [CFA+06].

Elliptic curves over the binary field $\mathbb{F}_{2^m}$ where m is a prime greater than 2 are used to enable a space efficient way to communicate prime factor bits of the user's private key from the SETUP device to the manufacturer using point compression (see below). The manufacturer has to choose two elliptic curves which can be described by a Weierstrass equation. Both curves $E_{a,b}$ and $E_{a',b}$ have to be twists of each other.

The utilization of two elliptic curves which are twists of each other is necessary to exploit the entire $\mathbb{F}_q$ for choosing elliptic curve points since usually only about the half of the elements in $\mathbb{F}_q$ represent x-coordinates on a single elliptic curve. For two twisted curves it holds that for each $x \in \mathbb{F}_{2^m}$ there exists an $y \in \mathbb{F}_{2^m}$ such that the point (x, y) either lies on $E_{a,b}$ or $E_{a',b}$. Young and Yung suggest $E_{0,b}$ and $E_{1,b}$ to be used by the manufacturer which resist known cryptographic attacks [YY06].

After selecting an appropriate pair of elliptic curves, the manufacturer creates an ECDH key pair by first choosing a *base point* (a generator) G_0 on $E_{0,b}(F_{2^m})$ and a base point G_1 on $E_{1,b}(F_{2^m})$. The private key consists of two randomly chosen integers (x_0, x_1) where $x_0 \in \{1, 2, ..., h_0\}$ and $x_1 \in \{1, 2, ..., h_1\}$. h_0 and h_1 are the cofactors of $E_{0,b}$ and $E_{1,b}$, respectively. The cofactor of an elliptic curve over a finite field can be computed by dividing the number of points of an elliptic curve by the order of G_0 and G_1, respectively. The order of a base point is defined to be the smallest scalar of the base point such that the product results in P_∞.

The manufacturer's public key is composed of G_0, G_1, $Y_0 = x_0 \cdot G_0$, and $Y_1 = x_1 \cdot G_1$.

To reduce the bit size needed to represent a point on an elliptic curve and thus utilizing the space offered by the SETUP to hide information to reconstruct the user's private key efficiently there exists the method of point compression. To represent a point (x, y) on $E(\mathbb{F}_{2^m})$, $m+1$ bits are necessary. The compressed point consists of x and an $ybit \in \{0, 1\}$. A decompression algorithm computes (x, y) from $x||ybit$. For more information on point compression see [CFA+06].

RSA^{EC}_{GEN1} is composed of different subfunctions (blocks) we will introduce subsequently. First, Young and Yung describe an algorithm `GenDHParamAndDHSecret` which is an essential subfunction of RSA^{EC}_{GEN1}. Including the manufacturer's public EC key (G_0, G_1, Y_0, Y_1), it computes an ECDH key exchange parameter (s_{pub}) and a secret ECDH key (s_{priv}) which both are represented as compressed points on one of the chosen elliptic curve pair of bit length $m+1$. By displaying s_{pub} in the upper order bits of the public RSA modulus n generated by RSA^{EC}_{GEN1}, the manufacturer is able to compute s_{priv} from s_{pub} and (x_0, x_1) by applying the `RecoverDHSecret` algorithm. This method enables the exchange of s_{priv} between the SETUP device and the manufacturer.

Before we start to describe how the RSA primes p and q are generated in RSA^{EC}_{GEN1}, we have to introduce three more functions involved in this process:

- $H(s, l, v)$ is a function which invokes a random oracle $\mathcal{R}$ (see Appendix B.1) on the input bit string s and returns v consecutive bits of $\mathcal{R}(s)$ starting at the l-th bit position.

- $A(p, l, e)$ returns *true* if p is a prime of bit length l with the uppermost bit set to 1 and $gcd(p, e) = 1$. Those primes are called acceptable primes [YY06]. Otherwise, if p is not an acceptable prime A returns *false*.

- $G(s, l, e)$ computes an acceptable prime. For this purpose, G calls $H(s, i, l/2)$ to compute the uppermost $l/2$-bits of p' with i is initially set to zero. The $l/2$ lowermost bits of p' are chosen uniformly at random. If p' is an acceptable prime, G returns $p = p'$. Otherwise, the whole computation is repeated and i is incremented by $l/2$.

Let e be an RSA exponent and let l_n be the bit length of the RSA modulus n which has to be computed beside the RSA primes p and q. Π_θ refers to the set of all permutations from $\{0, 1\}^\theta$ to $\{0, 1\}^\theta$ where θ is an even integer. π_θ is chosen uniformly at random from Π_θ and it is presumed that its inverse π_θ^{-1} can be computed efficiently.

RSA^{EC}_{GEN1} computes p and q on the input of s_{pub}, s_{priv}, l_n, and e as listed in Algorithm 4.8.

Input: $s_{pub}, s_{priv}, l_n, e$
Output: (p, q, n)
compute $p = G(s_{priv}, l_n, e)$.
pick $r_1 \in_{rnd} \{0,1\}^{\theta-(m+1)}$.
compute $t = \pi_\theta(r_1 || s_{pub})$.
pick $r_2 \in_{rnd} \{0,1\}^{l_n-\theta}$.
set $n_c = (t || r_2)$.
compute (q, r_c) such that $n_c = p \cdot q + r_c$ is satisfied.
if $A(q, l_n/2, e) == false$ **then**
  goto 2.
end
compute $n = p \cdot q$.
return (p, q, n).

Algorithm 4.8: `GenPrimes()`

Ultimately, RSA^{EC}_{GEN1} is composed of `GenDHParamAndDHSecret()` and `GenPrimes()`.

Input: (G_0, G_1, Y_0, Y_1)
Output: (p, q, n)
compute $(s_{pub}, s_{priv}) =$ `GenDHParamAndDHSecret`(G_0, G_1, Y_0, Y_1).
compute $(p, q, n) =$ `GenPrimes`(s_{pub}, s_{priv}).
return (p, q, n).

Algorithm 4.9: Modified `GEN1` of RSA^{EC}_{GEN1}

To recover the prime factors of the user's public modulus, the recovery algorithm utilizes Coppersmith's factorization attack which is used to factorize a composite $n = p \cdot q$ into its prime factors p and q by knowing the $1/4 \cdot ld(n)$ high- or low-bits of p or q.

The algorithm to recover p and q considers a potential borrow bit for the computation of n_c which would influence the recovery process of p or q. Thus, for the case that the primes could not be found for $b = 0$ the computation is repeated for $b = 1$ (see Algorithm 4.10). The other operations performed by the reconstruction method are derived analogously to the computation steps of RSA^{EC}_{GEN1} which merely have to be inverted.

```
Input: (n, e), (x_0, x_1)
Output: (p, q)
set t_1 = G(n, 0, θ).
for b = 0 to 1 do
    while j < upperBound do
        set t_2 = t_1 + b mod 2^θ.
        compute s_pub = G(π_θ^{-1}(t_2), θ − (m + 1), m + 1).
        compute s_priv = RecoverDHSecret(s_{pub,x_0,x_1}).
        set j = 0.
        compute u = G(R(s_priv), j · T, T).
        compute (p, q) = Coppersmith(u, n).
        if p · q = n then
            return (p, q).
        end
        set j = j + 1.
    end
end
return (p, q).
```

Algorithm 4.10: RSA^{EC}_{GEN1} prime factor reconstruction

Young and Yung propose a RSA^{EC}_{GEN1} configuration with $l_n = 768$, $m = 191$, $\theta = 320$, and $upperBound = \lceil l_n/4 \cdot 160 \cdot ln(2) \rceil$ to satisfy the presumed requirements and to reduce the probability of a potential borrow bit. In [YY08], Young and Yung propose an approach they call *algorithm engineering approach* for RSA^{EC}_{GEN1} to secure the strong SETUP against exposure on the basis of run time analysis, i.e., comparing the key generation running time of RSA and RSA^{EC}_{GEN1}. For this purpose RSA^{EC}_{GEN1} basically remains unmodified. Instead of calling OpenSSL routines, routines for prime testing were redesigned in order to accelerate the total running time of RSA^{EC}_{GEN1}. Following the algorithm engineering approach, Young and Yung have shown that their implementation of RSA^{EC}_{GEN1} into OpenSSL can be considered to be time-strong.

4.1.1.3 Backdoors for ElGamal. The following three Backdoors, embedded into the ElGamel cryptosystem, were conceived by Young and Yung [YY96] and expect that x and p or g is chosen by the key generation function.

The first proposed Backdoor, we call $ElGamal^{\beta_1}_{GEN}$ (see Algorithm 4.11), expects that x and g are chosen by the key generation function. A random integer x in the range of $0 \leq x \leq p-2$ is chosen as long as the result of applying a randomization function F^E_K on x encrypted with the manufacturer's public RSA key (N, E) is a generator in $\mathbb{Z}_p$. If such a value is found, g is set to it. The randomization function is used to broaden the domain of g.

The manufacturer's private key reconstruction method is shown in Algorithm 4.12. From the user's public key (p, g, y) and his own private RSA key (N, D), the manufacturer is able to compute x and thus the user's private key (p, g, x): first, he decrypts g utilizing F^D_K followed by decrypting the result with his private key.

Input: $p, (N, E)$
Output: $((p, g, y), (p, g, x))$
repeat
  pick $x \in_{rnd} \mathbb{N}$.
  compute $x' = x^E \ mod \ N$.
  compute $x'' = F_K^E(x')$.
until $x'' \leq p, \ x'' \in \mathbb{Z}_p^*$
set $g = x''$.
compute $y = g^x \ mod \ p$.
return $((p, g, y), (p, g, x))$.

Algorithm 4.11: Modified GEN of $ElGamal_{GEN}^{\beta_1}$

Input: $(p, g, y), (N, D)$
Output: (p, g, x)
compute $x' = F_K^D(g)$.
compute $x = x'^D \ mod \ N$.
return (p, g, x).

Algorithm 4.12: $ElGamal_{GEN}^{\beta_1}$ private key reconstruction

The second backdoor, called $ElGamal_{GEN}^{\beta_2}$, is implemented analogously to $ElGamal_{GEN}^{\beta_1}$ by interchanging the role of p and g. The termination condition for the loop is replaced by $x'' \in \Phi_{l_p}, \ x'' > g$.

The third backdoor $ElGamal_{GEN}^{\beta_1}$ which was introduced in [YY96] is a pure ElGamal system, i.e., there are no other cryptosystems involved in spite of ElGamal. This has the advantage that all necessary encryption routines are available in the host cryptosystem. It is assumed that both p and g can be chosen by the key generation function.

Let (P, G, Y) and (P, G, X) be the public and private key of the manufacturer, respectively. Instead of using RSA for encryption, the private key of the user is encrypted with ElGamal. First, the random values $x, k \in \mathbb{N}$ are generated where k and $P - 1$ have to be relatively prime. b is computed using the public key of the manufacturer: $b = Y^k \cdot x \ mod \ P$. A randomization function basing on ENC of ElGamal is used to meet the distribution required for b. After computing $a = G^k \ mod \ P$, it is checked that a and b satisfy the required conditions as presented in Algorithm 4.13. If this is not the case, a new k is randomly drawn and the depending values have to be recomputed. Otherwise, p is set to b, g is set to a, and y is computed as usual.

The private key of the user is reconstructed by decrypting p using the applied randomization function and the manufacturer's private key (P, G, X) (see Algorithm 4.14).

Input: $l_p, (P, G, Y)$
Output: $((p, g, y), (p, g, x))$
pick $x \in_{rnd} \mathbb{N}$.
repeat
  pick $k \in_{rnd} \mathbb{N}$, with $gcd(k, P-1) = 1$.
  compute $b' = Y^k \cdot x \ mod\ P$.
  compute $b = F_K^E(b')$.
  compute $a = G^k \ mod\ P$.
until $b \in \Phi_{l_p}, x < b, a < b$
set $p = b$.
set $g = a$.
compute $y = g^x \ mod\ p$.
return $((p, g, y), (p, g, x))$.

Algorithm 4.13: Modified `GEN` of $ElGamal_{GEN}^{\beta_3}$

Input: $(p, g, y), (P, G, X)$
Output: (p, g, x)
compute $b' = F_K^D(p)$.
compute $x = g^{-X} \cdot b' \ mod\ P$.
return (p, g, x)

Algorithm 4.14: $ElGamal_{GEN}^{\beta_3}$ private key reconstruction

4.1.2 Symmetric Backdoors. Symmetric backdoors involve the manufacturer's secret symmetric key K to hide information enabling him to reconstruct the user's private key. If symmetric backdoors are not secured against reverse engineering they are considered to be impractical.

The following three backdoors were proposed by Crépeau and Slakmon in [CS03] for the RSA key generation process. They have in common to hide secret bits in the user's public exponent e enabling the manufacturer to factorize n. Instead of hiding the secret bits by encryption with the manufacturer's public key as it is the case for asymmetric backdoors, keyed permutation mappings from the set of odd integers smaller than the user's public modulus into itself are used (see [CS03] for possible permutations). Key K on which a concrete permutation depends is only known by the manufacturer.

Unfortunately, it holds that the bit size of e is affected depending on which of the three backdoors is implemented.

The first backdoor, called RSA_{GEN2}^{HSD} (HSD - Hidden Small Private Exponent δ) and listed in Algorithm 4.15, relies on the attack on short RSA private exponents d where $d < n^{1/4}/3$ introduced by Wiener [Wie90] which was further improved by Boneh and Durfee in [BD00] who achieved a weaker bound on d: $d < n^{.292}$. First, on the basis of the two randomly chosen primes p and q by `GEN1` of RSA a weak exponent δ is chosen uniformly at random. The corresponding public exponent ϵ is the result of inverting δ modulo $\varphi(n)$. Utilizing the permutation function π_K the user's public exponent e is the result of applying $\pi_K(\epsilon)$.

If e and $\varphi(n)$ are relative primes, d is computed as usual. Otherwise, a new δ is chosen and the computation steps have to be repeated.

Input: p, q
Output: e
repeat
  pick $\delta \in_{rnd} \{2 \cdot k + 1 | k \in \mathbb{N}\}$ such that $gcd(\delta, \varphi(n)) = 1$ and $l_\delta \leq l_n/4$.
  compute $\epsilon = \delta^{-1} \ mod \ \varphi(n)$.
  compute $e = \pi_K(\epsilon)$.
until $gcd(e, \varphi(n)) = 1$
return e.

Algorithm 4.15: Modified `GEN2` of RSA^{HSD}_{GEN2}

From the user's public key (n, e) the manufacturer is able to compute the corresponding private key by recovering ϵ from applying $\pi_K^{-1}(e)$ and breaking the vulnerable public key (n, ϵ) to obtain δ. From the knowledge of δ and n, the manufacturer computes the prime factors of p and q of n. Given p, q, and e the manufacturer is able to compute d.

The main drawbacks of RSA^{HSD}_{GEN2} are that e has the same bit size as n and that e and thus d are not drawn randomly from the natural numbers, instead e is drawn from a smaller subset determined by the images of inverses modulo $\varphi(n)$ of δ-values with bit size smaller than $l_n/4$.

Input: (n, e)
Output: (p, q)
compute $\epsilon = \pi_K^{-1}(e)$.
compute $\delta = LowPrivateExponentAttack(n, \epsilon)$.
compute p, q from n, δ, ϵ.
return (p, q).

Algorithm 4.16: RSA^{HSD}_{GEN2} prime factor reconstruction

The next two backdoors, RSA^{HSPE}_{GEN2} (HSPE - Hidden Small Prime Public Exponent ϵ) and RSA^{HSE}_{GEN2} (HSE - Hidden Small Public Exponent ϵ), are very similar w.r.t. their construction. They distinguish in the bits of the private exponent δ which are hidden in the user's public exponent e. Thus, we merely take a closer look to the former one.

RSA^{HSPE}_{GEN2}, listed in Algorithm 4.17, relies on the low public prime exponent attack [BDF98]: For the case that $e \in [2^t, 2^{t+1}]$ where $t \in [l_n/4, l_n/2]$ is an integer and e is a prime it can be shown that in combination with the knowledge of the t most significant bits of d the user's private exponent d can be reconstructed in polynomial time [BDF98].

Input: p, q
Output: e
repeat
  pick $\epsilon \in \Phi_{l_n/4}$ such that $gcd(\epsilon, \varphi(n)) = 1$.
  compute $\delta = \epsilon^{-1} \; mod \; \varphi(n)$.
  compute $e = \pi_K(\delta\rceil^{l_n/4} || \epsilon)$.
until $gcd(e, \varphi(n)) = 1$
return e.

Algorithm 4.17: Modified `GEN2` of RSA^{HSPE}_{GEN2}

The manufacturer is able to reconstruct the user's private key from the public key (n, e) from extracting $\delta\rceil^{l_n/4}$ and computing ϵ by applying $\pi_K^{-1}(e)$. The low public prime exponent attack applied on n, $\delta\rceil^{l_n/4}$, and ϵ provides δ which allows the manufacturer to factorize n. Finally, the user's private key can be constructed from p and q.

Input: (n, e)
Output: (p, q)
compute $\delta\rceil^{l_n/4} || \epsilon = \pi_K^{-1}(e)$.
compute $\delta = LowPublicPrimeExponentAttack(n, \delta\rceil^{l_n/4}, \epsilon)$.
compute p, q from n, δ, ϵ.
return (p, q).

Algorithm 4.18: RSA^{HSPE}_{GEN2} prime factor reconstruction

Besides its simplicity, RSA^{HSPE}_{GEN2} has the same drawbacks as RSA^{HSD}_{GEN2} w.r.t. the restricted domain of e and thus d. Additionally, the running time of RSA^{HSPE}_{GEN2} deviates significantly from RSA due to the additional prime generation for ϵ. RSA^{HSE}_{GEN2} avoids this runtime overhead by exploiting another variant of the low public prime exponent attack described in [BDF98].

The forth RSA backdoor, $RSA^{HP}_{GEN1,2}$ (HP - Hidden Prime), proposed by Crépeau and Slakmon in [CS03] and listed in Algorithm 4.19 is a symmetric variant of $RSA^{PAP}_{GEN1,2}$ which was introduced in Section 4.1.1. The focus of $RSA^{HP}_{GEN1,2}$ lies on the proper distribution of the most significant bits of the user's public modulus n resulting from the multiplication of two randomly chosen primes which was ignored by Young and Yung in [YY96].

Crépeau and Slakmon achieve the proper distribution of the most significant bits of n by choosing a prime p and an odd number q' uniformly at random both of bit size $l_n/2$. The $l_n/8$ most significant bits of the product $p \cdot q' = n'$ are taken to be the $l_n/8$ most significant bits of the user's public modulus n. The remaining bits of n a determined by the permutated $l_n/4$ upper bits of p which enable the manufacturer to reconstruct p utilizing Coppersmith's factorization attack in combination with the $5l_n/8$ bits of n'. Subsequently, $q = \lfloor n/p \rfloor$ and n have to be adapted such that q is prime, e (determined in advance) and q are relatively prime, and $n = p \cdot q$. For this purpose, a random even number m of bit size $l_n/8$ which is xored with the $l_n/8$ least significant bits of q is chosen as long as the conditions from above are fulfilled. Since only the $l_n/8$ bits of q are modified, the hidden $l_n/4$ bits of p within n do not change by recomputing n with the new value of q.

Input: l_n
Output: (p, q, n, e)
pick $e \in_{rnd} \mathbb{N} \backslash \{1\}$.
pick $p \in_{rnd} \Phi_{l_n/2}$ such that $gcd(e, p-1) = 1$. pick $q' \in_{rnd} \{2 \cdot k + 1 | k \in \mathbb{N}\}$ of bit size $l_n/2$.
compute $n' = p \cdot q'$.
compute $n_1 = n' \rceil^{l_n/8}$, $n_2 = \pi_K(p\rceil^{l_n/4})$, $n_3 = n' \rfloor_{5l_n/8}$.
set $n = n_1 || n_2 || n_3$.
compute $q = \lfloor n/p \rfloor + (1 \pm 1)/2$ such that q is odd.
while $gcd(e, q-1)! = 1$ *and* $not(isPrime(q))$ **do**
    pick $m \in_{rnd} \{2 \cdot k | k \in \mathbb{N}\}$ of bit size $l_n/8$.
    recompute $q = q \oplus m$.
    recompute $n = p \cdot q$.
end
return (p, q, n, e).

Algorithm 4.19: Modified `GEN1` and `GEN2` of RSA^{HP}_{GEN2}

By applying the inverse permutation of π_K on the bits of $n\rceil^{3l_n/8}\rfloor_{l_n/4}$ which are extracted from the user's public key, the manufacturer obtains the upper $l_n/4$ bits of p. Applying Coppersmith's factorization attack on $p\rceil^{l_n/4}$ and n gives the prime factors of n, enabling the manufacturer to compute the user's private key.

Input: (n, e)
Output: (p, q)
compute $p\rceil^{l_n/4} = \pi_K^{-1}(n\rceil^{3l_n/8}\rfloor_{l_n/4})$.
compute $(p, q) = Coppersmith(p\rceil^{l_n/4}, n)$.
return (p, q).

Algorithm 4.20: RSA^{HP}_{GEN2} prime factor reconstruction

5 Backdoor Formalization Revision

In her dissertation [Arb08], Arboit devised a general backdoor definition oriented towards the SETUP definition of [YY06] which is more precise w.r.t. the properties a backdoor has to fulfill. In the following, we orientate ourselves towards Arboit's backdoor definition in combination with our own extensions in order to analyze the security of RSA^{PHP}_{GEN1} and $RSA^{PHP'}_{GEN1}$ in Section 6.

The three person model introduced in Section 4 stays the same in the remainder except that we use the term *user* and *internal attacker* interchangeable. Analogously to the definition of $\mathfrak{C}$ and $\mathfrak{C}'$, we define $\mathfrak{G}$ and $\mathfrak{G}'$ to be the honest and the backdoored cryptosystem with emphasizing its key generation purpose. k_{pub} and k_{priv} refer to the user's public and private key, respectively. $\mathcal{K}_{\mathfrak{G}}$ denotes the key space of $\mathfrak{G}$ and $\mathcal{K}_{\mathfrak{G}'}$ the one of $\mathfrak{G}'$. Let $\mathfrak{B}$ and $\mathfrak{B}'$ be the blackbox devices provided by the manufacturer where $\mathfrak{G}$ and $\mathfrak{G}'$ are implemented on, respectively.

We consider the black box device to be a physical oracle the user can access to generate keys as well as to de- and encrypt messages. The user has access to its input and output as well as physical measures, e.g., the time spend for key generation or the memory consumption. Additionally, we expect that the user has access to the source code of $\mathfrak{G}$ and $\mathfrak{G}'$ particularly including the manufacturer's public key if $\mathfrak{G}'$ is an asymmetric backdoor. Given two blackbox devices $\mathfrak{B}$ and $\mathfrak{B}'$ the user tries to distinguish between $\mathfrak{B}$ and $\mathfrak{B}'$ basing on statistical analyses w.r.t. the inputs and outputs of both devices and on side channel analyses by performing external physical measures.

According to [Arb08], each $\mathfrak{G}'$ can be modeled by the composition of an information compression function $\mathcal{C}$, an encryption function $\mathcal{E}$, and an embedding function $\mathcal{M}$:

$$k'_{pub} = \mathcal{M} \circ \mathcal{E} \circ \mathcal{C}(k'_{priv}),$$

where k'_{pub} is either k_{pub} itself or a component of k_{pub} and k'_{priv} is either equal to k_{priv}, a part of the private key, or information allowing to compute k_{priv}. Breaking a backdoor down in those three functions facilitates the analysis and the comparison of backdoors. For the case that $\mathcal{M}$, $\mathcal{E}$, or $\mathcal{C}$ is probabilistic, a sufficient amount of random bits are assumed to be provided by an implicit parameter.

The purpose of function $\mathcal{C}(k'_{priv}) = k_C$ is to extract a piece of information k_C from k'_{priv} such that if k_C is communicated through k_{pub} to the manufacturer, he is able to recover k_{priv} from this piece of information.

Definition 5.1 (Information Compression Function $\mathcal{C}$). *$\mathcal{C}$ is an arbitrary function of k'_{priv} which extracts sufficient information k_C from its input in order to efficiently reconstruct k_{priv} from the knowledge of k_C and k_{pub} only.*

Function $\mathcal{E}$ is an asymmetric encryption function which is applied to k_C in order to hide this sensitive piece of information from unauthorized parties except the manufacturer who possesses the appropriate secret key K or K_{priv} which allows to decrypt $\mathcal{E}(k_C)$.

Definition 5.2 (Encryption Function $\mathcal{E}$). *$\mathcal{E}$ is a symmetric or asymmetric encryption function of $k_C = \mathcal{C}(k'_{priv})$ where the output value distribution of $k_E = \mathcal{E}(k_C)$ is computationally indistinguishable from uniform.*

The third function, $\mathcal{M}$, accomplishes the embedding of $\mathcal{E} \circ \mathcal{C}(k'_{priv})$ into k_{pub} chosen in such a way that the backdoored keys ideally have the same statistical properties as the honest keys.

Definition 5.3 (Embedding Function $\mathcal{M}$). *$\mathcal{M}$ is an invertible function applied on $\mathcal{E} \circ \mathcal{C}(k'_{priv})$ determining the bit positions of the embedding and the probabilistic assignation of those bits of k'_{pub} which are not involved in the embedding of backdoor information.*

We refer to the function applied on k_{pub} by the manufacturer in order to reconstruct k'_{priv} as

$$\mathfrak{R}_{\mathfrak{G}'} = \mathcal{C}^{-1} \circ \mathcal{E}^{-1} \circ \mathcal{M}^{-1}(k_{pub}).$$

Emerged from literature, the essential properties a backdoor can have are *Confidentiality*, *Completeness*, and *Concealment* [YY96,YY97,Arb08].

Confidentiality considers if the confidentiality of the underlying cryptosystem is influenced through the existence of a backdoor. Evaluating this property involves analyzing if the effective key size of the user's key generated with $\mathfrak{B}'$ corresponds to the desired key size.

Definition 5.4 (Confidentiality). *The external attacker and the user are not able to invert* $\mathcal{M} \circ \mathcal{E} \circ \mathcal{C}(k'_{priv})$.

Completeness considers whether or not the manufacturer is able to reconstruct the user's private key from the corresponding public key for each public key which was generated with a backdoor distributed by the manufacturer.

Definition 5.5 (Completeness). *For each* k_{pub} *generated with* $\mathfrak{G}'$*, the manufacturer is able to reconstruct* k'_{priv} *from* k_{pub} *by computing* $\mathfrak{R}_{\mathfrak{G}'}(k_{pub}) = \mathcal{C}^{-1} \circ \mathcal{E}^{-1} \circ \mathcal{M}^{-1}(k_{pub}) = k'_{priv}$.

Concealment considers to what extent a backdoor is traceable for the external attacker and the user. This requires the analysis of a set of subproperties Concealment directly depends on.

Definition 5.6 (Concealment). *Given access to* $\mathfrak{B}$ *and* $\mathfrak{B}'$*, for the external attacker and the user it is not possible to distinguish between* $\mathfrak{B}$ *and* $\mathfrak{B}'$ *by analyzing the following related subproperties:*

KCC*: Analyzing the keyspaces* $\mathcal{K}_{\mathfrak{G}}$ *and* $\mathcal{K}_{\mathfrak{G}'}$ *does not contribute to efficiently distinguish between* $\mathfrak{B}$ *and* $\mathfrak{B}'$ *(Keyspace (K) Cardinality (C) Consistency (C)).*
DC*: Analyzing the distribution of the bits of keys generated with* $\mathfrak{G}$ *and* $\mathfrak{G}'$ *does not contribute to efficiently distinguish between* $\mathfrak{B}$ *and* $\mathfrak{B}'$ *(Bit Distribution (D) Consistency (C) of Generated Keys).*
VCC*: Differences w.r.t. the correlation of variables in* $\mathfrak{G}$ *and* $\mathfrak{G}'$ *do not contribute to efficiently distinguish between* $\mathfrak{B}$ *and* $\mathfrak{B}'$ *(Variable (V) Correlation (C) Consistency (C)).*
CC*: Differences w.r.t. the time complexities of* $\mathfrak{G}$ *and* $\mathfrak{G}'$ *do not contribute to efficiently distinguish between* $\mathfrak{B}$ *and* $\mathfrak{B}'$ *(Complexity (C) Consistency (C)).*
AM*:* $\mathfrak{B}'$ *does not make use of additional memory for purposes of concealment (Absence (A) of Additional Memory (M) Usage for Purposes of Concealment).*
RT*: The statistical properties of the key generation running times of* $\mathfrak{G}$ *and* $\mathfrak{G}'$ *do not significantly deviate from each other (Running (R) Time (T)).*

The subproperties of Concealment can be divided into three groups: key related, algorithm related, and side channel related properties (see Figure 3). In the following we give an informal description of how to evaluate each of these subproperties based on [Arb08]. Subsequently, we present a classification scheme for the defined backdoor related properties.

Key Related Properties:

KCC. In practice, modifications made to an honest key generator $\mathfrak{G}$ to hide secret information within k_{pub} typically affects the key space $\mathcal{K}_{\mathfrak{G}'}$ of $\mathfrak{G}'$ which

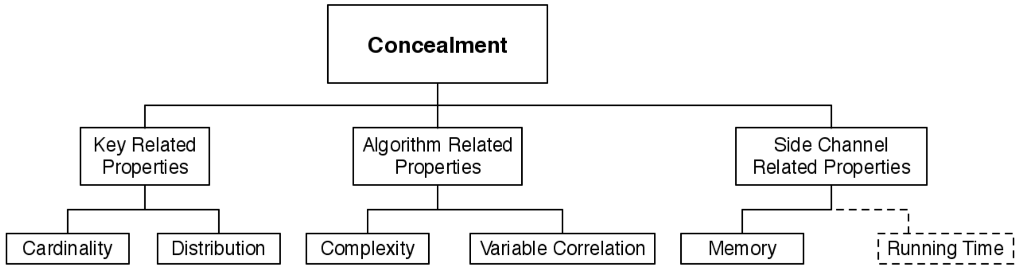

Fig. 3: The composition of Concealment.

might contribute to efficiently distinguish between $\mathfrak{B}$ and $\mathfrak{B}'$. To evaluate KCC for $\mathfrak{G}'$ w.r.t. the security parameter s of $\mathfrak{G}$, the ratio of cardinality of $\mathcal{K}_{\mathfrak{G}'}$ and $\mathcal{K}_{\mathfrak{G}}$ is computed: $\mathcal{R}(s) = \mathcal{N}_{\mathfrak{G}'}(s)/\mathcal{N}_{\mathfrak{G}}(s)$. In practice, the ratio of cardinality for RSA is of the form $\mathcal{R}(s) = 2^{c(s)\cdot s} \cdot f(s)$ where $c(s) \leq 0$ and $log_2(1/f(s)) \in o(s)$, i.e., $1/f(s)$ is negligible [Arb08].

DC. DC considers to which degree the bits of k_{pub} are influenced by the embedding of k_E. For $\mathfrak{G}$ and $\mathfrak{G}'$, it is not necessarily the case that all bits in k_{pub} have the same distribution [CS03]. If some bits of k_{pub} in $\mathfrak{G}'$ underlie another distribution than the appropriate bits in $\mathfrak{G}$, then this may reveal the existence of the backdoor by analyzing generated keys. To evaluate DC for $\mathfrak{G}'$, we compare the distributions of the concatenation of the influenced bits of k_{pub} by $\mathcal{M}(k_E)$ w.r.t. $\mathfrak{G}$ and $\mathfrak{G}'$ via their statistical distance referred to as $\mathcal{D}_{\mathfrak{G}'} \in [0,1]$. If $\mathcal{D}_{\mathfrak{G}'}$ is negligible, the distributions of k_{pub} are considered to be statistically indistinguishable for $\mathfrak{G}$ and $\mathfrak{G}'$.

Algorithm Related Properties:

VCC. VCC concerns *key regeneration* [CS03,Arb08] which is the possibility of the user to generate a new key pair keeping certain key parameters fixed while others are refreshed. In $\mathfrak{G}'$, there may exist key components of successively generated keys which are more correlated than those in $\mathfrak{G}$. Furthermore, VCC covers the case where no key regeneration is necessary since the backdoor can be revealed solely on the basis of the key parameters. We distinguish the case whether or not the manufacturer's key (K/K_{pub}) is publicly known.

CC. CC considers the time complexity $\mathcal{T}_{\mathfrak{G}'}$ of $\mathfrak{G}'$. If the time complexity of $\mathfrak{G}'$ is linear in the time complexity of $\mathfrak{G}$, then adapting the hardware of $\mathfrak{B}$ or $\mathfrak{B}'$ for matching the running times of $\mathfrak{G}$ and $\mathfrak{G}'$ is feasible. According to [Arb08], the complexity of a backdoored RSA key generator in terms of the complexity of an honest RSA key generator can be expressed in practice by $\mathcal{T}_{\mathfrak{G}'} = t_1^a \cdot t(\mathcal{F})^b + t_2^c$, where t_1 and t_2 refer to the complexities of the key parameter generations in $\mathfrak{G}$, a, b, and c are constant integer exponents, and $t(\mathcal{F})$ denotes the time complexity of a non-instantiated function $\mathcal{F}$ which does not occur in $\mathfrak{G}$. The complexity of key generators of other cryptosytems can be expressed similarly (see [Arb08]).

Side Channel Related Properties:

AM. There are two types of memory a backdoor might make use of in order to hide itself from the user: non-volatile memory (NM) and volatile memory (VM). Backdoors which transmit information of the previous key generation in order to reconstruct private keys make use of non-volatile memory. Other backdoors require volatile memory to satisfy the variable correlation property under key regeneration analysis. The usage of NM is easier to detect than the usage of VM. Since it is less discreet, it may require more system resources, and if it is reset then the variable correlation and the functioning of the backdoor might be influenced.

RT.[10] RT captures statistical properties of the key generation running times which can not be achieved by analyzing CC. In order to evaluate this property for $\mathfrak{G}'$ the mean μ and the coefficient of variation c_v of the key generation running times of $\mathfrak{G}$ and $\mathfrak{G}'$ are compared. We write $\mu_{\mathfrak{G}}$, $c_{v,\mathfrak{G}}$ and $\mu_{\mathfrak{G}'}$, $c_{v,\mathfrak{G}'}$ in order to refer to the median and the coefficient of variation of the key generation running times of $\mathfrak{G}$ and $\mathfrak{G}'$, respectively.

		failed	*poor*	*good*
Confidentiality		effective keysize is reduced more than 50%	effective keysize is at most halved	no reduction of effective keysize
Completeness		$\neg\forall k_{pub}\ \mathfrak{R}_{\mathfrak{G}'}(k_{pub}) = k'_{priv}$	—	$\forall k_{pub}\ \mathfrak{R}_{\mathfrak{G}'}(k_{pub}) = k'_{priv}$
Concealment	KCC (RSA)	$c(s) \leq -3/2$	$-3/2 < c(s) < -1/2$	$c(s) \geq -1/2$
	DC	$\mathcal{D}_{\mathfrak{G}'} > 0$	$\mathcal{D}_{\mathfrak{G}'} \approx 0$ with restrictions on the bounds of $\mathcal{K}_{\mathfrak{G}'}$	$\mathcal{D}_{\mathfrak{G}'} \approx 0$
	VCC	variable correlation w/o K_{pub}	variable correlation given K_{pub}	no var. correlation
	CC	$\mathcal{T}_{\mathfrak{G}'}$ is at least quadratic in $\mathcal{T}_{\mathfrak{G}}$	$\mathcal{T}_{\mathfrak{G}'}$ is more than linear but less than quadratic in $\mathcal{T}_{\mathfrak{G}}$	$\mathcal{T}_{\mathfrak{G}'}$ is linear in $\mathcal{T}_{\mathfrak{G}}$
	AM	NM (and VM) is used	only VM is used	neither VM nor NM is used
	RT	$\frac{\lvert\mu_{\mathfrak{G}'}-\mu_{\mathfrak{G}}\rvert}{\mu_{\mathfrak{G}}} \geq 0.1$ or $\frac{\lvert c_{v,\mathfrak{G}'}-c_{v,\mathfrak{G}}\rvert}{c_{v,\mathfrak{G}}} \geq 0.1$	$\frac{\lvert\mu_{\mathfrak{G}'}-\mu_{\mathfrak{G}}\rvert}{\mu_{\mathfrak{G}}} < 0.1$ and $\frac{\lvert c_{v,\mathfrak{G}'}-c_{v,\mathfrak{G}}\rvert}{c_{v,\mathfrak{G}}} < 0.1$	$\mu_{\mathfrak{G}} \approx \mu_{\mathfrak{G}'}$ and $c_{v,\mathfrak{G}} \approx c_{v,\mathfrak{G}'}$

Table 1: Classification scheme for backdoor related properties.

[10]We declare the property of running time to be an optional feature of Concealment because it seems to be unavoidable that the modifications made to $\mathfrak{G}$ resulting in $\mathfrak{G}'$ involve a significant increase of the key generation running time.

The classification scheme we use in order to evaluate the defined backdoor related properties (see Table 1) is inspired by [Arb08]. The bounds of $c(s)$ used to evaluate KCC are restricted to RSA backdoors. Those bounds have to be adapted accordingly for backdoors of other cryptosystems (cf. [Arb08]). If a backdoor related property is classified with *good* we consider the property to be satisfied while a property classified with *failed* is considered to be unsatisfied. The additional classifier *poor* allows for a more subtle differentiation between different backdoors w.r.t. their properties. Additionally to the classification given in Table 1, we characterize KCC to be *asymptotically exact* if $lim_{s\to\infty}c(s) = 0$ holds.

6 Novel RSA Backdoors

In this section we present two new asymmetric backdoors: RSA^{PHP}_{GEN1} and $RSA^{PHP'}_{GEN1}$. We use the acronym PHP—Prime Hiding Prime—to refer to the essential mechanism behind both backdoors which is to hide information of the first RSA prime factor in the second RSA prime factor. After describing the mechanics of both backdoors in Section 6.1 and 6.2, a comparative security analysis considering the backdoor properties discussed in Section 5 follows.

6.1 Design of RSA^{PHP}_{GEN1}

The first backdoor we want to introduce is called RSA^{PHP}_{GEN1}. The RSA key generation function `GEN1` is modified in such a way that it involves the manufacturer's public RSA key (N, E) to hide secret information in one of the prime factors of the user's public modulus n. This information can be used by the manufacturer in combination with his private key (N, D) to reconstruct the user's private key. Note that the bit length of N has to be equal to $l_n/2$ in order to generate primes of the appropriate bit size.

First, the random prime p of bit size $l_n/2$ is chosen. If the encryption of p with the manufacturer's public key and the subsequent multiplication with the inverse of p modulo N results in a prime of $\Phi_{l_n/2}$, the pending prime q is set. Otherwise, a fresh random prime p is picked and the computation steps are repeated. If q is found, n is computed as usual. The RSA key generation functions `GEN2` and `GEN3` stay unmodified. The computation steps of RSA^{PHP}_{GEN1} are listed in Algorithm 6.1.

Input: $l_n, (N, E)$
Output: p, q, n
repeat
  pick a $p \in_{rnd} \Phi_{l_n/2}$.
  compute $q = p^E \cdot p^{-1} \mod N$.
until q *is prime.*
compute $n = p \cdot q$.
return p, q, n.

Algorithm 6.1: Modified `GEN1` of RSA^{PHP}_{GEN1}

From a user's public key generated with RSA^{PHP}_{GEN1}, the manufacturer is able to compute one of the prime factors of n by decrypting n with his private key D:

$$n^D \equiv (p \cdot q)^D \equiv (p \cdot p^E \cdot p^{-1})^D \equiv p^{E \cdot D} \equiv p \ mod\ N.$$

Thus, the manufacturer is able to factorize n and to reconstruct the user's corresponding private key. The prime factor reconstruction of RSA^{PHP}_{GEN1} is given by Algorithm 6.2.

Input: $(n, e), (N, D)$
Output: p, q
compute $p = n^D \ mod\ N$.
compute $q = n \cdot 1/p$.
return p, q.

Algorithm 6.2: RSA^{PHP}_{GEN1} factor reconstruction

In order to motivate our second backdoor $RSA^{PHP'}_{GEN1}$, we anticipate one crucial deficiency of RSA^{PHP}_{GEN1}. If the internal attacker comes into possession of the manufacturer's public key (N, E), he can distinguish between key pairs generated with RSA^{PHP}_{GEN1} and RSA efficiently (see Section 6.3). To overcome this issue, we provide an extension of RSA^{PHP}_{GEN1} in the next section.

6.2 Design of $RSA^{PHP'}_{GEN1}$

$RSA^{PHP'}_{GEN1}$ is an extension of RSA^{PHP}_{GEN1} eradicating the addressed weakness of RSA^{PHP}_{GEN1}. To that end, $l_n/4$ bits are chosen uniformly at random and concatenated with the $l_n/4$ least significant bits of p. The result is denoted as p' (see Algorithm 6.3, Line 3 and 4). Subsequently, q is computed in the same way as in RSA^{PHP}_{GEN1} with the difference that instead of encrypting p with the manufacturers public RSA key as it is done in line 3 of Algorithm 6.1, p' is encrypted. This modification eradicates the weakness of RSA^{PHP}_{GEN1} (see Section 6.3.2). Another advantage of $RSA^{PHP'}_{GEN1}$ is, that if the computed value of q fails the primality test, p has not to be regenerated: it is sufficient to choose another random value for x. The remaining computation steps of RSA^{PHP}_{GEN1} coincide with $RSA^{PHP'}_{GEN1}$.

Input: $l_n, (N, E)$
Output: p, q, n
1 pick $p \in_{rnd} \Phi_{l_n/2}$.
2 **repeat**
3   pick a random x of the bit size $l_n/4$.
4   set $p' = x \mid p\rfloor_{l_n/4}$.
5   compute $q = (p')^E \cdot p^{-1} \ mod\ N$.
6 **until** q *is prime.*
7 compute $n = p \cdot q$.
8 **return** p, q, n.

Algorithm 6.3: Modified `GEN1` of $RSA^{PHP'}_{GEN1}$

To reconstruct the prime factors of the user's public modulus n which was generated by $RSA_{GEN1}^{PHP'}$, first the manufacturer decrypts n with his private RSA key and obtains p':

$$n^D \equiv (p \cdot q)^D \equiv (p \cdot p'^E \cdot p^{-1})^D \equiv p'^{E \cdot D} \equiv p' \ mod\ N.$$

The lower $l_n/4$ bits of p' correspond to the $l_n/4$ least significant bits of one of the prime factors of n. Applying Coppersmith's factorization attack on n and $p\rfloor_{l_n/4}$, the manufacturer is able to factorize n and thus to compute the corresponding user's private key. Algorithm 6.2 summarizes the computation steps to factorize n with the knowledge of (N, D).

Input: $(n, e), (N, D)$
Output: p, q
compute $x \mid p\rfloor_{l_n/4} = p' = n^D \ mod\ N$.
compute $(p, q) = Coppersmith(n, p\rfloor_{l_n/4})$.
return p, q.

Algorithm 6.4: $RSA_{GEN1}^{PHP'}$ factor reconstruction

Remark. For RSA_{GEN1}^{PHP} it is not necessary to satisfy the condition $p < N$ in order to properly decrypt p. Since N and p are of bit size $l_n/2$, at no time the condition $2N < p$ is violated. If $p \geq N$ and p is encrypted with the manufacturer's public key $p^E \ mod\ N \equiv (p \ mod\ N)^E \ mod\ N$, the original value of p can be obtained by slightly modifying the decryption function of the manufacturer:

$$p = (n^D \ mod\ N) + N.$$

The same argument holds for integer p' used in $RSA_{GEN1}^{PHP'}$ to hide $p\rfloor_{l_n/4}$ within n.

6.3 Security Analysis

This section provides the security analysis of RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ basing on the backdoor related properties defined in Section 5. Subsequently, both established backdoors are compared to those backdoors presented in Section 4.

For estimating the keyspace of RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ in our security analysis, we use the prime number theorem which is defined as follows:

Theorem 6.1 (Prime Number Theorem). *Let $\pi(n)$ be the number of all primes less than or equal to n. It holds that $\pi(n) \sim \frac{n}{ln(n)}$.*

6.3.1 Security Analysis of $\boldsymbol{RSA_{GEN1}^{PHP}}$.

For RSA_{GEN1}^{PHP}, we set k_{priv}, $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{M}$ as follows: k'_{priv} equals the first prime factor of n. $\mathcal{C}$ is chosen to be the identity since RSA_{GEN1}^{PHP} communicates the entire information of p to the manufacturer: $\mathcal{C}(p) = p = k_C$. $\mathcal{E}(k_C)$ corresponds to the RSA encryption of k_C with the manufacturer's public key (N, E): $\mathcal{E}(k_C) = k_C^E \ mod\ N = k_E$.

k_E is embedded into the user's public RSA modulus n by applying $\mathcal{M}(k_E) = (k_E \cdot p^{-1}\ mod\ N) \cdot p = n = k'_{pub}$.

In the following, we analyze the properties of RSA^{PHP}_{GEN1} according to the definitions given in Section 5.

Confidentiality (*poor*). Assume that an external attacker is in possession of (n, e) and (N, E). The external attacker is able to invert $\mathcal{M}$ by computing $k_E = n\ mod\ N$. k_E is the encryption of p with the manufacturer's public RSA key of bit length l_N which is half of the length of the user's RSA key of bit length l_n. To invert $\mathcal{E}$, i.e., to compute p and with that k_{priv}, the external attacker has to break RSA of key size l_N instead of key size l_n. Thus, to ensure that k_{priv} remains secret the bit size of N has to be chosen large enough. Since the security of the user's RSA key pair depends on N, Confidentiality is classified as *poor*. If, however, the external attacker does not know (N, E), then the user's private key retains the intended security.

Completeness (*good*). The manufacturer is able to compute k'_{priv} from k_{pub} by applying $\mathcal{C}^{-1} \circ \mathcal{E}^{-1} \circ \mathcal{M}^{-1}(k_{pub})$, where $\mathcal{M}^{-1}(k_{pub} = (n, e)) = n\ mod\ N$, $\mathcal{E}^{-1} = k_E^D\ mod\ N$, and $\mathcal{C}^{-1}(k_C) = \mathcal{C}(k_C)$. The composition of the three functions can be simplified to

$$\mathfrak{R}_{RSA^{PHP}_{GEN1}}(k_{pub} = (n, e)) = n^D\ mod\ N = p = k'_{priv}.$$

According to the classification scheme presented in Table 1, Completeness is classified as *good*.

Concealment. To analyze Concealment for RSA^{PHP}_{GEN1}, we have to consider the subproperties Concealment directly depends on (see Section 5).

KCC (good, asym. exact). Instead of considering the cardinality of $\mathcal{K}_{RSA^{PHP}_{GEN1}}$, it is sufficient to analyze the space q is chosen from: Since RSA^{PHP}_{GEN1} does not influence the choice of the public and private exponents e, d, $\mathcal{N}_{RSA,e}$ is equal to $\mathcal{N}_{RSA^{PHP}_{GEN1},e}$ and $\mathcal{N}_{RSA,d}$ is equal to $\mathcal{N}_{RSA^{PHP}_{GEN1},d}$. The difference of $\mathcal{N}_{RSA,n}$ and $\mathcal{N}_{RSA^{PHP}_{GEN1},n}$ directly depends on the cardinality of the set of possible primes for q in RSA^{PHP}_{GEN1} referred to as Λ. Thus, the task to determine the difference between $\mathcal{N}_{RSA}$ and $\mathcal{N}_{RSA^{PHP}_{GEN1}}$ is reduced to determine the difference between $\Phi_{l_n/2}$ and Λ. For RSA, the security parameter s corresponds to half of the bits of the key, whereas the length of the key is considered to be l_n.

Let $N \approx 2^{l_n/2}$ so that p is randomly chosen from $\Phi_{l_n/2}$ in RSA^{PHP}_{GEN1}. We can estimate the cardinality of the set of all primes by applying the prime number theorem:

$$|\Phi_{l_n/2}| = \pi(2^{l_n/2}) - \pi(2^{l_n/2-1}) \approx \frac{1}{2} \cdot \pi(2^{l_n/2}) \approx 2^{l_n/2-1-log_2(l_n/2)}.$$

Let Ψ be defined as follows:

$$\Psi = \{f(p) | f(p) = p^E \cdot p^{-1}\ mod\ N,\ p \in \Phi_{l_n/2}\},$$

where $p^E \cdot p^{-1}\ mod\ N = f(p) = p^{E-1}\ mod\ N$. For

$$\Psi^* = \{f(x) | f(x) = p^{E-1}\ mod\ N, x \in \mathbb{Z}_N\},$$

we have

$$|\Psi^*| = (1 + \frac{p-1}{gcd(p-1, E-1)}) \cdot (1 + \frac{q-1}{gcd(q-1, E-1)}).$$

Since $E-1$ is even, $|\Psi^*|$ is maximal for the case that $E-1$ satisfies

$$gcd(p-1, E-1) = gcd(q-1, E-1) = 2. \tag{11}$$

Since the manufacturer can freely choose E, we assume Equation 11 in the following and thus $|\Psi^*| > N/4$. This result can be used to estimate the cardinality of Ψ as $|\Psi| \approx |\Phi_{l_n/2}|/4$.

Assuming that the elements in Ψ are close to be uniformly distributed over $\mathbb{Z}_N$, about half of the elements in Ψ are $l_n/2$-bit integers. Let $\Psi' \subset \Psi$ be the set of all elements in Ψ of bit length $l_n/2$: $\Psi' := \{\psi \mid \psi \in \Psi,\ l_\psi = l_n/2\}$. It holds that $|\Psi'| \approx 1/2 \cdot |\Psi| = 2^{l_n/2-4-log_2(l_n/2)}$.

The probability that an element in Ψ' is a prime can be estimated by dividing the cardinality of the set of all $l_n/2$-bit primes by the cardinality of all $l_n/2$-bit integers:

$$\mathbb{P}(q \in \Phi_{l_n/2} | q \in_{rnd} \mathbb{N}_{l_n/2}) = \frac{|\Phi_{l_n/2}|}{|\mathbb{N}_{l_n/2}|} = \frac{2^{l_n/2-1-log_2(l_n/2)}}{2^{l_n/2} - 2^{l_n/2-1}} = 2^{-log_2(l_n/2)}.$$

Finally, the number of primes in Ψ' can be estimated by

$$|\Lambda| = |\Psi'| \cdot \mathbb{P}(q \in \Phi_{l_n/2} | q \in_{rnd} \mathbb{N}_{l_n/2}) = 2^{l_n/2-4-2\cdot log_2(l_n/2)}.$$

To evaluate the cardinality of Λ w.r.t. $\Phi_{l_n/2}$ we compute $\mathcal{R}(l_n/2)$:

$$\mathcal{R}(l_n/2) = \frac{\mathcal{N}_{RSA^{PHP}_{GEN1}}(l_n/2)}{\mathcal{N}_{RSA}(l_n/2)} = \frac{2^{l_n/2-4-2\cdot log_2(l_n/2)}}{2^{l_n/2-1-2\cdot log_2(l_n/2)}} = 2^{-3-log_2(l_n/2)} = 2^{c(l_n/2)\cdot l_n/2},$$

where $c(l_n/2) = (-3 - log_2(l_n/2))/(l_n/2)$. Assume that the security parameter is larger or equal than 512 bits. We have that $c(l_n/2) \geq c(512) = -0.02343$ and thus KCC is rated with *good*. Note that for increasing l_n values, the whole term converges to 0. Thus, KCC is considered to be *asymptotic exact*.

DC (good). The bits of k_{pub} in RSA^{PHP}_{GEN1} which are influenced by the backdoor are the bits of n whereas the generation of the public exponent e is not influenced. Thus, to show that $\mathcal{D}_{RSA^{PHP}_{GEN1}} \approx 0$, it is sufficient to show that the distribution of the bits of p and q generated with RSA^{PHP}_{GEN1} is close to the corresponding distribution for the case that p and q are generated with RSA.

As, for instance, it is the case in [The03], we assume that the primes p and q in RSA are generated by first choosing a random odd $l_n/2$ bit integer followed by testing it for primality. If the chosen odd integer fails the applied primality test, a new random $l_n/2$-bit odd integer is drawn or the former prime is incremented by two. In any case, the primes generated in RSA are uniformly distributed in $\Phi_{l_n/2}$.

In RSA^{PHP}_{GEN1}, the first prime p is chosen in the same way as in RSA. Assume that $N \approx 2^{l_n/2}$ and that all odd values $q = p^E \cdot p^{-1}\ mod\ N$ with $l_q < l_n/2$ are discarded. According to the considerations of KCC w.r.t. Λ, we have that for a sufficiently large security parameter, q is close to be uniformly distributed over

$\Phi_{l_n/2}$ and thus the distribution of the bits of q generated with RSA^{PHP}_{GEN1} is close to the distribution of bits of q generated with RSA. Consequently, $\mathcal{D}_{RSA^{PHP}_{GEN1}} \approx 0$ and DC is classified as *good*.

VCC (failed). The user is able to distinguish between RSA^{PHP}_{GEN1} and RSA by analyzing the correlation of variables for RSA^{PHP}_{GEN1} because the generation of q depends on p. Hence, if the user fixes p or q and initiates a key regeneration, the pending prime will be identical to the corresponding prime of the previous key generation. If the user additionally knows the manufacturer's public key (N, E), he is able to distinguish between RSA^{PHP}_{GEN1} and RSA without performing a key regeneration. From his private key (n, d) the user is able to compute p and q [May04]. To proof the presence of the RSA^{PHP}_{GEN1} backdoor, the user solely has to check if $p = q^E \cdot q^{-1} \ mod \ N$ or $q = p^E \cdot p^{-1} \ mod \ N$ holds. According to the classification scheme presented in Table 1, VCC is classified as *failed*.

CC (failed). Let t_p and t_q be the time complexity of RSA for the generation of p and q where $t_p = t_q$. Let $\mathcal{F} = p^E \cdot p^{-1} \ mod \ N$. Then the time complexity for the generation of p and q in RSA^{PHP}_{GEN1}, denoted by $\mathcal{T}_{RSA^{PHP}_{GEN1},p,q}$, leads to:

$$\mathcal{T}_{RSA^{PHP}_{GEN1},p,q} \approx t_q \cdot (t_p + t(\mathcal{F})) = t_q \cdot t(\mathcal{F}) + t_p^2 > t_q + t_p \approx \mathcal{T}_{RSA,p,q}$$

For large l_p, l_q, $t(\mathcal{F})$ is negligible w.r.t. t_p and t_q since $t(\mathcal{F}) \in \mathcal{O}(n^2)$ and $t_p, t_q \in \mathcal{O}(l_p^4/log(l_p))$ [JPV00]. CC for RSA^{PHP}_{GEN1} is classified as *failed* because $\mathcal{T}_{RSA^{PHP}_{GEN1},p,q}$ is quadratic in $\mathcal{T}_{RSA,p,q}$.

AM (good). RSA^{PHP}_{GEN1} does not use additional memory in order to avoid the correlation of variables or to communicate backdoor information to the manufacturer. Thus, AM is rated with *good*.

RT (failed). A detailed evaluation of the running time analysis of RSA^{PHP}_{GEN1} is given in Section 7. For the sake of completeness of the security analysis, a synopsis of the evaluation is given at this point. To evaluate the running time of RSA^{PHP}_{GEN1} we accomplished time measurements of 1000 1024-bit key generations with RSA^{PHP}_{GEN1} and RSA, respectively. The analysis revealed a significant difference between the statistical properties of RSA and RSA^{PHP}_{GEN1} w.r.t. the running time. We measured a mean running time of $0.03859s$ for RSA and a mean running time of $12.2393s$ for RSA^{PHP}_{GEN1} which differ by a factor of ≈ 3100. For keys of larger key sizes the deviation of the mean runtime for RSA^{PHP}_{GEN1} from the mean runtime of RSA increases rapidly. The coefficient of variation of RSA^{PHP}_{GEN1} resulted in 0.9442 which is almost 40% above the result for RSA which was determined to be 0.5665. Consequently, by analyzing the statistical properties of the key generation running time of a given blackbox device the presence of RSA^{PHP}_{GEN1} can definitely be ascertained. Thus, the running time property of RSA^{PHP}_{GEN1} has to be evaluate with *failed*.

6.3.2 Security Analysis of $RSA^{PHP'}_{GEN1}$.

The security analysis of $RSA^{PHP'}_{GEN1}$ proceeds analogously to the analysis of RSA^{PHP}_{GEN1}. We set k'_{priv}, $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{M}$ as follows: k'_{priv} equals to the first prime factor p of n generated by $RSA^{PHP'}_{GEN1}$.

Confidentiality			bisected effective key size	◯
Completeness			$\mathfrak{R}_{RSA^{PHP}_{GEN1}}(k_{pub}) = n^D \ mod \ N$	$\surd$
Concealment	key related properties	KCC	*asym. exact*	$\surd$
		DC	$\mathcal{D}_{RSA^{PHP}_{GEN1}} \approx 0$	$\surd$
	algorithm rel. properties	VCC	variable correlation	$\times$
		CC	quadratic in $\mathcal{T}_{RSA,p,q}$	$\times$
	side channel rel. properties	AM	no additional memory	$\surd$
		RT	significant deviation	$\times$

Table 2: Overview of the properties of RSA^{PHP}_{GEN1} ($\surd$: *good*, ◯: *poor*, $\times$: *failed*)

The information compression function $\mathcal{C}$ concatenates the $l_n/4$ least significant bits of p with a $l_n/4$-bit value $x \in_{rnd} \{0,1\}^{l_n/4}$: $\mathcal{C}(p) = x \ || \ p\rfloor_{l_n/4}$. $\mathcal{E}$ and $\mathcal{M}$ are defined analogously to RSA^{PHP}_{GEN1}.

In the following, we analyze the properties of $RSA^{PHP'}_{GEN1}$ according to the definitions given in Section 5.

Confidentiality (*poor*). For $RSA^{PHP'}_{GEN1}$ and RSA^{PHP}_{GEN1}, the analysis of Confidentiality is identical: k_C is encrypted with an $l_n/2$-bit RSA key which bisects the effective key size of $RSA^{PHP'}_{GEN1}$. Thus, Confidentiality is rated with *poor*.

Completeness (*good*). To extract k'_{priv} from k_{pub}, the manufacturer has to apply $\mathcal{C}^{-1} \circ \mathcal{E}^{-1} \circ \mathcal{M}^{-1}(k_{pub})$. $\mathcal{M}^{-1}$ and $\mathcal{E}^{-1}$ are computed in the same way as for RSA^{PHP}_{GEN1}. Inverting $\mathcal{C}$ corresponds to the reconstruction of the prime factor p. This is achieved by applying Coppersmith's factorization attack on n and on the $l_n/4$ least significant bits of k_C, since $k_C\rfloor_{l_n/4} = p\rfloor_{l_n/4}$. Coppersmith's factorization attack returns both prime factors of n. One of them equals to k'_{priv}.

Let $Coppersmith'(n, p\rfloor_{l_n/4})$ be a modified version of $Coppersmith(n, p\rfloor_{l_n/4})$ which only returns prime factor p of n with $p\rfloor_{l_n/4} = k_C\rfloor_{l_n/4}$. The definition of $\mathcal{C}^{-1}$ is given by

$$\mathcal{C}^{-1}(k_C) = Coppersmith'(n, k_C\rfloor_{l_n/4}).$$

Aggregating the computation steps of $\mathcal{C}^{-1} \circ \mathcal{E}^{-1} \circ \mathcal{M}^{-1}(k_{pub})$ yields the prime factor reconstruction function of the manufacturer:

$$\mathfrak{R}_{RSA^{PHP'}_{GEN1}}(k_{pub} = (n,e)) = Coppersmith'(n, (n^D \ mod \ N)\rfloor_{l_n/4}) = p = k'_{priv}.$$

Knowing one of the prime factors of the user's public modulus, the manufacturer is able to compute the user's private key k_{priv}. According to the classification scheme presented in Table 1, Completeness is classified as *good*.

Concealment. To analyze Concealment of $RSA^{PHP'}_{GEN1}$ we have to consider the subproperties Concealment directly depends (see Section 5).

KCC (good, asym. exact). To analyze KCC of $\mathcal{K}_{RSA^{PHP'}_{GEN1}}$ it is again sufficient to analyze the set q is chosen from. The RSA security parameter s equals to the bit length of the prime factors p and q of n which is $l_p = l_q = l_n/2$.

We assume that the manufacturer's public modulus N is of size $2^{l_n/2}$. The first prime factor p of n is chosen randomly from $\Phi_{l_n/2}$, where $|\Phi_{l_n/2}| \approx 2^{l_n/2-1-log_2(l_n/2)}$ (see Subsection 6.3.1). In order to compute the cardinality of

$$\Theta = \{p' \mid p' = x \mid\mid p\rfloor_{l_n/4},\ x \in_{rnd} \mathbb{N}_{l_n/4},\ p \in_{rnd} \Phi_{l_n/2}\},$$

we have to determine the cardinality of $\mathbb{N}_{l_n/4}$ where x is randomly chosen from as well as the cardinality of the set Υ formed by all possible values for $p\rfloor_{l_n/4}$. The cardinality of $\mathbb{N}_{l_n/4}$, corresponds to

$$|\mathbb{N}_{l_n/4}| = 2^{l_n/4} - 2^{l_n/4-1} = 2^{l_n/4-1}.$$

Υ is defined as follows:

$$\Upsilon = \{p\rfloor_{l_n/4} \mid p \in \Phi_{l_n/2}\}.$$

Since p is a prime, we can deduce that Υ does not contain any integer divisible by 2 and 5. All other $l_n/4$-bit integers are potential elements in Υ. Truncating the $l_n/4$ least significant bits of a $l_n/2$-bit prime can be considered to be a mapping from $\Phi_{l_n/2}$ to Υ. Due to the substantial difference in size of the domain and the image of the mapping as well as the unpredictable distribution of prime numbers we can assume that approximately all $l_n/4$-bit integers which are coprime to 2 and 5 appear in Υ. Thus, we can estimate the cardinality of Υ by the following formula:

$$|\Upsilon| \approx 2^{l_n/4-1} - \frac{2^{l_n/4}}{10} \approx 2^{l_n/4-1} - 2^{l_n/4-3} = \frac{3}{4} \cdot 2^{l_n/4-1} \approx 2^{l_n/4-1.4}.$$

Now, we are able to compute the cardinality of Θ by multiplying the cardinality of $\mathbb{N}_{l_n/4}$ and Υ:

$$|\Theta| = |\mathbb{N}_{l_n/4}| \cdot |\Upsilon| \approx 2^{l_n/4-1} \cdot 2^{l_n/4-1.4} = 2^{l_n/2-2.4}.$$

Analogous to the analysis of RSA^{PHP}_{GEN1} we define Ψ and Ψ':

$$\Psi = \{f(p,p') \mid f(p,p') = (p')^E \cdot p^{-1}\ mod\ N\},$$

$$\Psi' = \{\psi \mid \psi \in \Psi, l_\psi = l_n/2\},$$

with p and p' defined as above. Similarly as for RSA^{PHP}_{GEN1} we can estimate the cardinality of Ψ and Ψ' by $|\Psi| \approx |\Theta|/4$ and thus $|\Psi'| \approx 1/2 \cdot |\Psi|$.

Finally, we can estimate the cardinality of the set of the possible prime candidates for the second prime factor q of n. In order to do this, we follow the analysis of KCC for RSA^{PHP}_{GEN1} and multiply $|\Psi'|$ by the probability that an element in Ψ' is prime:

$$\begin{aligned}|\Lambda| \approx |\Psi'| \cdot \mathbb{P}(q \in \Phi_{l_n/2} \mid q \in_{rnd} \mathbb{N}_{l_n/2}) &= 2^{l_n/2-3.4} \cdot 2^{-log_2(l_n/2)} \\ &= 2^{l_n/2-5.4-log_2(l_n/2)}.\end{aligned}$$

To evaluate the cardinality of Λ w.r.t. $\Phi_{l_n/2}$ we compute $\mathcal{R}(l_n/2)$:

$$\mathcal{R}(l_n/2) = \frac{\mathcal{N}_{RSA^{PHP'}_{GEN1}}(l_n/2)}{\mathcal{N}_{RSA}(l_n/2)} = \frac{2^{l_n/2-5.4-log_2(l_n/2)}}{2^{l_n/2-1-log_2(l_n/2)}} = 2^{-4.4} = 2^{c(l_n/2)\cdot l_n/2},$$

where $c(l_n/2) = -4.4/(l_n/2)$. For a secure usage of RSA, $s = l_n/2 \geq 512$ is required and we have that $c(l_n/2) \geq c(512) = -0.008593$ wherefore KCC is rated with *good*. For $RSA^{PHP'}_{GEN1}$, KCC is *asymptotic exact*, too. Compared to RSA^{PHP}_{GEN1}, the convergence of $c(s)$ to 0 is even faster by a logarithmic factor.

DC (good). DC (good). The analysis of DC for $RSA^{PHP'}_{GEN1}$ can be carried out analogously to RSA^{PHP}_{GEN1} which leads to the fact that DC is classified as *good*.

VCC (good). The user is not able to distinguish between $RSA^{PHP'}_{GEN1}$ and RSA based on analyzing the correlation of variables for $RSA^{PHP'}_{GEN1}$. We verify this assertion in two steps: first we argue that with key regeneration in combination with fixing key parameters a distinction is not possible whereupon we show that the same holds if additionally (N, E) is known to the user.

Since the user's private and public exponent (e,d) are not affected by $RSA^{PHP'}_{GEN1}$, we can exclude them from analysis. It remains to show that there is no variable correlation between p and q which enables the user to distinguish between the backdoored key generator and the honest one. Suppose the user fixes one of the prime factors of n. Recall from Section 6.2 how the second RSA prime factor is generated for $RSA^{PHP'}_{GEN1}$. Since the fresh prime factor cannot be efficiently distinguished from a randomly generated one, there is no connection between p and q which can be computed efficiently.

If additionally the manufacturer's public key is known to the user, he might be able to distinguish between $RSA^{PHP'}_{GEN1}$ and RSA if he is able to guess the random integer $x \in \{0,1\}^{l_n/4}$ which has been used for key generation in $RSA^{PHP'}_{GEN1}$. The user can check if his guess was correct if the following equation is satisfied:

$$(x \mid\mid p\rfloor_{l_n/4})^E \cdot p^{-1} \; mod \; N = q. \tag{12}$$

Since x is chosen uniformly at random, the best strategy for the user to guess x is equivalent to iterating through all possible x values until Equation 12 is satisfied. In the context of RSA, this approach is computationally infeasible since otherwise it would be possible to factorize RSA modules by first guessing $l_n/4$ bits of one of the prime factors of n followed by applying Coppersmith's factorization attack on those bits and n. Since factoring in general is hard, the user is not able to distinguish between $RSA^{PHP'}_{GEN1}$ and RSA on the basis of analyzing the correlation of variables for $RSA^{PHP'}_{GEN1}$ given (N, E). Overall, VCC is classified as *good*.

CC (good). Let t_p and t_q be the time complexity of RSA for generating the prime factors p and q of n where $t_p = t_q$. For $RSA^{PHP'}_{GEN1}$, we can specify $\mathcal{F}$ explicitly by $\mathcal{F} = \mathcal{C}(p)^E \cdot p^{-1} \; mod \; N$. The time complexity for the generation of p and q in $RSA^{PHP'}_{GEN1}$ can be estimated as follows:

$$\mathcal{T}_{RSA^{PHP'}_{GEN1},p,q} \approx t_p + t_q \cdot t(\mathcal{F}) \approx t_p + t_q \approx \mathcal{T}_{RSA,p,q}$$

As for RSA_{GEN1}^{PHP}, for large l_p, l_q, $t(\mathcal{F})$ is negligible w.r.t. t_p and t_q since $t(\mathcal{F}) \in \mathcal{O}(n^2)$ and $t_p, t_q \in \mathcal{O}(l_p^4/log(l_p))$ [JPV00]. According to the classification scheme in Table 1, CC for $RSA_{GEN1}^{PHP'}$ is rated as *good*.

AM (good). $RSA_{GEN1}^{PHP'}$ does not use additional memory to avoid the correlation of variable or to communicate backdoor information to the manufacturer. Thus, AM is rated with *good*.

RT (failed). A detailed evaluation of the running time analysis of $RSA_{GEN1}^{PHP'}$ is given in Section 7. For the sake of completeness of the security analysis, a synopsis of the evaluation is given at this point. To evaluate the running time of $RSA_{GEN1}^{PHP'}$ we accomplished time measurements of 1000 1024-bit key and 2048-bit key generations with $RSA_{GEN1}^{PHP'}$ and RSA, respectively. For 1024-bit key generation we measured a mean running time of $0.03959s$ for RSA and a mean running time of $0.2308s$ for $RSA_{GEN1}^{PHP'}$, differing by a factor of ≈ 6. For $RSA_{GEN1}^{PHP'}$ the coefficient of variation was ascertained by 0.8601 which is more than 30% above the ascertained value for RSA (0.559). For 2048-bit key generation we achieved similar results. The mean running time resulted in $2.3095s$ for $RSA_{GEN1}^{PHP'}$ and $0.2625s$ for RSA, differing by a factor of ≈ 9. As expected, the difference between the coefficient of variation which could be ascertained by 0.8995 and 0.6695 for $RSA_{GEN1}^{PHP'}$ and RSA decreases for increasing the key size. Although the running time of $RSA_{GEN1}^{PHP'}$ could be reduced significantly w.r.t. RSA_{GEN1}^{PHP}, altogether the running time property for $RSA_{GEN1}^{PHP'}$ has to be classified as *failed* for 1024- and 2048-bit key generation because of the non neglectable increase of variation of the running time compared to RSA.

Confidentiality			bisected effective key size	$\bigcirc$
Completeness			$\mathfrak{R}_{RSA_{GEN1}^{PHP'}}(k_{pub}) =$ $Coppersmith'(n, (n^D \ mod \ N)\rfloor_{l_n/4})$	$\surd$
Concealment	key related properties	KCC	*asym. exact*	$\surd$
		DC	$\mathcal{D}_{RSA_{GEN1}^{PHP}} \approx 0$	$\surd$
	algorithm rel. properties	VCC	no variable correlation	$\surd$
		CC	linear in $\mathcal{T}_{RSA,p,q}$	$\surd$
	side channel rel. properties	AM	no additional memory	$\surd$
		RT	significant deviation	$\times$

Table 3: Overview of the properties of $RSA_{GEN1}^{PHP'}$ ($\surd$: *good*, $\bigcirc$: *poor*, $\times$: *failed*)

6.3.3 Discussion. Table 4 provides a comparative overview of the analysis results for the backdoor related properties of RSA_{GEN1}^{PHP}, $RSA_{GEN1}^{PHP'}$, as well as the prominent backdoors which were discussed in Section 4. The analysis results of the backdoors presented in Section 4 have been extracted from [Arb08]. Although no backdoor can be considered to be confidential, complete, and concealing at the same time, it is noticeable that w.r.t. the statistical backdoor properties $RSA_{GEN1}^{PHP'}$ is superior to the other backdoors. Particularly, the achieved results for KCC for $RSA_{GEN1}^{PHP'}$ are outstanding since the difference between $\mathcal{N}_{RSA_{GEN1}^{PHP'}}$ and $\mathcal{N}_{RSA}$ can be neglected for a sufficiently large security parameter. We are not aware of any other RSA backdoors achieving *asymptotic exactness* for KCC. In addition, as opposed to the reference backdoors, $RSA_{GEN1}^{PHP'}$ is the only backdoor which satisfies all mandatory subproperties of Concealment.

The only disadvantage of $RSA_{GEN1}^{PHP'}$ is the bisection of the effective key size leading to the classification as poor w.r.t. Confidentiality.

7 Implementation and Evaluation

To give a proof of concept and to accomplish a timing analysis for the proposed backdoors involving a comparison with the timing results of an unmodified RSA version, we embedded RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ into the RSA implementation of OpenSSL[11] version 1.0.1e. OpenSSL can be seen as two tools in one: a cryptographic library and an SSL (Secure Socket Layer) toolkit, both written in C. The library provides implementations of the industry's best-regarded algorithms like encryption algorithms, message digest algorithms, message authentication codes, and a pseudo number generator [VMC02]. Additionally, OpenSSL comes with a command line interface which provides access to much of its functionality. We use the OpenSSL command line interface below for instance to generate RSA key pairs. The RSA implementation of OpenSSL can be found in `openssl-1.0.1e/crypto/rsa`. The source file `rsa_gen.c` in the `rsa` folder contains the relevant code for generating the RSA key pair components, i.e., e, p, q, n, d.

Basically, a call of the RSA key generator works as follows:

1. Unless otherwise specified, e is set to 65537[12].
2. p and q are generated subsequently without considering the properties of safe primes[13] by default.
3. n is computed by multiplying p and q.
4. d is computed by inverting $e \ mod \ \varphi(n)$.

According to the design of RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ the code for generating p and q in `rsa_gen.c` has to be modified to implement the backdoors into OpenSSL.

To support arithmetic and the representation of large integers as it is necessary for RSA, OpenSSL provides the BN (Big Number) library which declares routines for the `BIGNUM` data type. The size of a number a variable of type `BIGNUM` can hold is only restricted by the available memory [VMC02].

[11] http://www.openssl.org/

[12] 65537 is a *Fermat prime* which has favorable properties concerning the security of RSA and the computation time for encryption.

[13] A safe prime is a prime which satisfies a set of security relevant properties which are not satisfied by every prime.

Backdoor			Confidentiality	Completeness	Concealment					
Name	Alg.	sym./asym.			KCC	DC	VCC	CC	RT	AM
$DH^{\beta_1}_{GEN2}$	4.1,4.2	asym.	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\surd$	$\surd$	—	$\times$
$RSA^{\beta_1}_{GEN2}$	4.4	asym.	$\bigcirc$	$\surd$	$\times$	$\bigcirc$	$\times$	$\bigcirc$	—	$\surd$
$RSA^{PAP}_{GEN1,2}$	4.6	asym.	$\bigcirc$	$\surd$	$\bigcirc$	$\bigcirc$	$\surd$	$\surd$	—	$\surd$
$RSA^{PAP'}_{GEN1,2}$	—	asym.	$\bigcirc$	$\surd$	$\bigcirc$	$\bigcirc$	$\surd$	$\surd$	—	$\surd$
RSA^{EC}_{GEN1}	4.10	asym.	$\surd$	$\surd$	$\bigcirc$	$\bigcirc$	$\surd$	$\bigcirc$	$\surd$	$\surd$
$ElGamal^{\beta_1}_{GEN}$	4.11	asym.	$\bigcirc$	$\surd$	$\times$	$\times$	$\bigcirc$	$\bigcirc$	—	$\surd$
$ElGamal^{\beta_2}_{GEN}$	—	asym.	$\bigcirc$	$\surd$	$\times$	$\times$	$\bigcirc$	$\bigcirc$	—	$\surd$
$ElGamal^{\beta_3}_{GEN}$	4.13	asym.	$\surd$	$\surd$	$\surd$	$\bigcirc$	$\times$	$\times$	—	$\surd$
RSA^{HSD}_{GEN2}	4.15	sym.	$\surd$	$\surd$	$\times$	$\bigcirc$	$\bigcirc$	$\surd$	—	$\times$
RSA^{HSPE}_{GEN2}	4.17	sym.	$\surd$	$\surd$	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\surd$	—	$\surd$
RSA^{HSE}_{GEN2}	—	sym.	$\surd$	$\surd$	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\surd$	—	$\surd$
RSA^{HP}_{GEN2}	4.19	sym.	$\surd$	$\surd$	$\bigcirc$	$\surd$	$\bigcirc$	$\surd$	—	$\surd$
RSA^{PHP}_{GEN1}	6.3	asym.	$\bigcirc$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$
$RSA^{PHP'}_{GEN1}$	6.1	asym.	$\bigcirc$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$

Table 4: Overview of the properties of discussed backdoors ($\surd$: *good*, $\bigcirc$: *poor*, $\times$: *failed*, —: not available)

In the following sections, we will successively elucidate the implementation of RSA^{PHP}_{GEN1} and $RSA^{PHP'}_{GEN1}$, the corresponding methods to reconstruct private keys from public keys which were generated with one of our backdoors, and conclusively we will present an evaluation. The evaluation consists of the comparison of measured running times of RSA^{PHP}_{GEN1} and $RSA^{PHP'}_{GEN1}$ with those of the unmodified RSA implementation in OpenSSL and additionally a discussion of the running times of the corresponding private key reconstruction functions. The source code for the key reconstruction functions of both backdoors is included in Appendix C.

7.1 Implementation of RSA^{PHP}_{GEN1}

We generated a 512-bit and a 1024-bit RSA key pair for the manufacturer and included the corresponding public keys into the modified version of rsa_gen.c. Exemplary, the backdoor is implemented for 1024-bit and 2048-bit RSA key generation, i.e., if the user generates a 1024-bit or a 2048-bit RSA key pair with the underminded OpenSSL version, a backdoored key is returned. For other key lengths RSA keys are generated trustworthy.

In doing so, the generation of p and q in `rsa_gen.c` was replaced by the computation steps depicted in Algorithm 6.1. The implementation is straightforward—for each required mathematical operation, there exists a corresponding BN routine—and thus not discussed explicitly.

7.1.1 Private Key Reconstruction Function. The python script to reconstruct the user's private key is included in Appendix C.1. It is called by

```
./RPK_RSA_GEN1_PHP.py pubKey recPrivKey manPrivKey
```

where `pubKey` is the user's public key, `recPrivKey` is the name of the file the corresponding reconstructed user's private key is stored to and `manPrivKey` is the manufacturer's private key which is needed to decrypt the user's public modulus.

We use the python package *PyCrypt*[14] to access an RSA implementation. The following listing constitutes the main functionality of `RPK_RSA_GEN1_PHP`.

```
#reconstruct the hidden prime factor p
p = pow(n,D,N)

#compute q
q = n/p

#compute d
ePhi = (p-1)*(q-1)
d = modInv(e,ePhi)
```

Listing 1.1: Private key reconstruction

First n is decrypted to obtain p. By dividing n by p, we obtain q. Knowing the prime factors of n one can efficiently compute $\varphi(n)$ and d, the inverse of e modulo $\varphi(n)$.

[14] https://www.dlitz.net/software/pycrypto/

From n, e, d, p, q an RSA instance is created which provides a method to export the private key in the PEM[15] format compliant to the PKCS#1[16] specification. Finally, the user's private key is written to `recPrivKey` (see Listing 1.2).

```
#construct the user's RSA instance from n,e,d,p,q
rsaUser = RSA.construct((n,e,d,p,q))

#write the user's private key to the output file
extractedKey = rsaUser.exportKey('PEM',pkcs=1)
privKeyUser = file(recPrivKey, "w")
privKeyUser.write(extractedKey)
privKeyUser.write("\n")
```

Listing 1.2: Private key export

In the following, we demonstrate the combination of RSA_{GEN1}^{PHP} and the private key reconstruction function. We consider only keys in PEM format because it is the default format of OpenSSL for RSA keys. Other common formats can be converted to PEM using the OpenSSL command line tool.

Before the user of RSA_{GEN1}^{PHP} can encrypt or sign messages he has to create an RSA key pair:

```
openssl genrsa -out privKey.pem 1024
openssl rsa -in privKey.pem -pubout -out pubKey.pem
```

Afterwards, the user publishes his public key to enable communication parties to send him encrypted messages or verify messages containing his signature. With the knowledge of the user's public key and his own private key, the manufacturer is able to reconstruct the user's private key. For this purpose the manufacturer uses the key reconstruction method `RPK_RSA_GEN1_PHP.py` from above:

```
./RPK_RSA_GEN1_PHP.py pubKey.pem recPrivKey.pem manPrivKey.pem
```

The reconstructed private key is stored to `recPrivKey.pem`. Now, the manufacturer is able to decrypt eavesdropped public key encrypted messages issued for the target user and to sign messages in the user's place.

If instead of a pure public key a certificate was published, the manufacturer can simply extract the user's public key by the following command and proceed as described above:

```
openssl x509 -inform pem -in crt.pem -pubkey -out crt.pem
```

Other certificate formats can be converted to PEM format using OpenSSL. For example a certificate in DER[17] format can be converted to the PEM format by the following command:

```
openssl x509 -inform der -in crt.der -out crt.pem -outform pem
```

[15] http://tools.ietf.org/html/rfc5280
[16] http://tools.ietf.org/html/rfc3447
[17] http://www.itu.int/rec/T-REC-X.690/en

7.1.2 Evaluation. To compare the key generation running time (RT) of RSA^{PHP}_{GEN1} and RSA, we did time measurement for 1000 1024-bit key generation on a 3.1 GHz PC under OSX.

As one might expect from the complexity analysis of RSA^{PHP}_{GEN1} in Section 6, the runtime difference between RSA^{PHP}_{GEN1} and RSA is significant. The mean runtime for 1024 bit RSA key generation of the unmodified RSA version resulted in $0.03859s$ while the mean runtime of RSA^{PHP}_{GEN1} is $12.2393s$. On average, the generation of RSA^{PHP}_{GEN1} backdoored keys takes ≈ 3100 times longer than usual which makes the exposure of the backdoor trivial using timing analysis.

To compare the variation of the key generation time of both algorithms we use the coefficient of variation, a normalized measure of variation. The coefficient of variation of RSA and RSA^{PHP}_{GEN1} resulted in 0.5665 and 0.9442, respectively. The coefficient of variation of RSA^{PHP}_{GEN1} is almost 40% higher than for RSA, i.e., the variation around the mean value is 40% higher. Thus, even if the hardware of the black box device is adapted for the implementation of RSA^{PHP}_{GEN1}, the backdoor can be detected by analyzing the variation of the computation times for key generation.

For the key generation of 2048-bit keys the difference of the mean running time between RSA^{PHP}_{GEN1} and RSA increases rapidly, while the difference of the coefficient of variation decreases only neglectable so that we can conclude that overall the evaluation does not improve when increasing the keysize.

The running time of `RPK_RSA_GEN1_PHP.py` is less than 0.5 seconds and its complexity corresponds to the complexity of RSA decrpytion. As well as the complexity, the running time of the private key reconstruction function is not relevant w.r.t. side channel attacks, merely a polynomial running time in the input parameters has to be assured.

7.2 Implementation of $RSA^{PHP'}_{GEN1}$

As for RSA^{PHP}_{GEN1} we exemplary implement $RSA^{PHP'}_{GEN1}$ for 1024-bit and 2048-bit RSA key generation by modifying `rsa_gen.c`. We include the manufacturer's 512-bit and 1024-bit public key into `rsa_gen.c` and replace the generation procedure of the second RSA prime q for the 1024-bit and the 2048-bit key case. In the following we discuss Listing 1.3 which depicts the C code for the backdoored prime generation of q.

```
do
  {
  /* generate x */
  BIGNUM *x = NULL;
  x = BN_new();
  BN_rand(x,bitsx,0,0);

  BIGNUM *lsbp = NULL;
  lsbp = BN_new();
  BN_copy(lsbp,rsa->p);

  /* compute the lp/2 least significant bits of p */
  int i = 0;
  for(i = bitsp - 1; i >= (bitsp - bitsx); i--)
    {
```

```
16      BN_clear_bit(lsbp,i);
17      }

19    /* concatenate x with the lp/2 least significant bits of p
20    * first do a bitspl-leftshift of x  */
21    BIGNUM *xPrime = NULL;
22    xPrime = BN_new();
23    if(!BN_lshift(xPrime,x,bitspl)) goto err;

25    /* concatenation by addition */
26    BIGNUM *pPrime = NULL;
27    pPrime = BN_new();
28    if(!BN_add(pPrime,xPrime,lsbp)) goto err;

30    /* compute q */
31    BIGNUM *tmp1 = NULL;
32    tmp1 = BN_new();
33    BIGNUM *tmp2 = NULL;
34    tmp2 = BN_new();
35    if(!BN_mod_exp(tmp1, pPrime, E, N, ctx)) goto err;
36    if(!BN_mod_inverse(tmp2, rsa->p, N, ctx)) goto err;
37    if(!BN_mod_mul(rsa->q,tmp1,tmp2,N,ctx)) goto err;
38    } while((BN_is_prime_ex(rsa->q,BN_prime_checks,NULL,NULL) ==
          0) || (BN_cmp(rsa->p, rsa->q) == 0) || BN_num_bits(rsa->q)
           != 1024); /* repeat until q is a 1024 bit prime */
```

Listing 1.3: Backdoored generation of the second RSA prime for 2048-bit keys

First, a random number `x` of bit size `bitsx` (512/1024 for 1024/2048-bit RSA keys) is generated (Listing 1.3, Line $3-6$). This random number has to be concatenated with the $l_n/4$ least significant bits of the previously generated prime p to obtain p' (compare to Algorithm 6.3). Next, we compute the $l_n/4$ least significant bits of p by clearing the $l_n/4$ most significant bits (Listing 1.3, Line $12-17$). Since the BN-library only provides standard arithmetic routines for the `BIGNUM` datatype, we have to implement the concatentation $x||p\rfloor_{l_n/4}$ by first performing a 256/512 bit leftshift of `x` followed by an addition of p' (Listing 1.3, Line $19-28$). The backdoored prime q is computed in three steps: p' is encrypted with the manufacturer's public key; next, the inverse of p modulo N is computed; both results are multiplied modulo N which results in q (Listing 1.3, Line $35-37$).

The whole procedure is repeated until q passes the primality test, q is a 512/1024 bit integer, and $p \neq q$ (Listing 1.3, Line 38).

7.2.1 Private Key Reconstruction Function. The implementation of the private key reconstruction function for $RSA_{GEN1}^{PHP'}$ is more involved than the one for RSA_{GEN}^{PHP}. This is because Coppersmith's factorization attack has to be involved into the factorization process of $n = p \cdot q$ from the knowledge of n and the $l_{n/4}$ least significant bits of p or q. First, we will introduce the implementation of the factorization attack which is implemented as a Sage[18] module (see Appendix C.3), followed by the presentation of the Sage script which allows the

[18] http://www.sagemath.org/

reconstruction of the user's private RSA key by calling the factorization attack as a subfunction (see Appendix C.2).

To get access to Coppersmith's factorization attack we implemented Coron's algorithm on finding small roots of bivariate integer polynomial equations which was extensively discussed in Section 3. The implementation follows the outlined computation steps presented with Algorithm 3.2. We implemented Coron's algorithm in Sage to get access to complex mathematical algorithms, e.g., a lattice reduction algorithm or an algorithm to compute the resultant of two polynomials. Basing on code snippets, we successively discuss the most important computation steps of Coron's algorithm in order to factorize n. The variable names are chosen analogously to the variables used for introducing Coron's algorithm in Section 3. Note that for the following part describing the implementation of Coron's algorithm N refers to the user's public modulus whereas n refers to the absolute value of the determinant of Matrix S introduced in Section 3.

Coron's algorithm expects the following input parameters:

- `N`: the user's public modulus which should be factorized in order to compute the user's private key
- `P0`: the $l_{n/4}$ least significant bits of one of the prime factors of n
- `k`: the parameter to control the dimension of the lattice which has to be reduced (given `addBits`, `k` has to be determined experimentally)
- `addBits`: notifies how many additional bits of p are known (in order to keep k small)

Given `P0` as input parameter, we can compute the $l_{N/4}$ least significant bits of q, referred to as `Q0`, by applying the multivariate Hensel lemma as described in Section 3 (see Listing 1.4).

```
nob = int(math.ceil(0.25 * _numberOfBits(N))) + addBits
pi = 1
qi = 1
bits = nob
for i in range(1,nob):
  b = multHensel(pi,qi,N,P0,i)
  if (P0 >> i)&1:
    pi = _setBit(pi,i)
  else:
    pi = _clearBit(pi,i)
  if b == 1:
    qi = _setBit(qi,i)
  else:
    qi = _clearBit(qi,i)
Q0 = qi
```

Listing 1.4: Compute `Q0` by applying the multivariate Hensel lemma

`_numberOfBits(a)` computes the bit length of integer `a`. The value of `pi` and `qi` which are successively constructed within the for-loop correspond to the `i` least significant bits of `P0` and `Q0` where `i` is the iterator of the for-loop. From `pi` and `qi` the $(i+1)$-bit of `Q0` is determined by calling `multHensel(pi,qi,N,P0,i)`. The auxillary function `_setBit(n,i)` sets the i-th bit of integer `n` to 1. `_clearBit(n,i)` works vice versa.

From `P0` and `Q0` the bivariate polynomial `pxy` can be computed:

```
pxy = 2^bitLengthQ0 * x * y + Q0 * x + P0 * y + int((P0*Q0 - N)
      /2^bitLengthQ0)
```

The roots of `pxy` (x_0, y_0) are later used to determine the prime factors of `N`. Note that degree δ of `pxy` is equal to 1.

According to [Cor07], we compute `X`, `Y`, the upperbounds for `x0` and `y0`, and `W` which bounds the product of `X` and `Y` (see Listing 1.5).

```
X = 2^(bitLengthQ - bitLengthQ0)
Y = X
W = abs(int(pxy.coefficient({x:1,y:0})))*X
```

Listing 1.5: Computing the upper bounds for the roots of `pxy`

The indices `i0` and `j0` which describe matrix `S` are computed by maximizing $8^{(i-u)^2+(j-v)^2}|p_{ij}|X^iY^j$ according to Lemma 3.2. In the first nested loop (see Line 4-9 of Listing 1.6) we compute the indices `u` and `v` which maximize $|p_{uv}|X^uY^v$. The second nested loop computes `i0`, `j0` by maximizing $8^{(i-u)^2+(j-v)^2}|p_{ij}|X^iY^j$ for $0 \leq i, j < k$.

```
1  print "\n\ncomputing (i0,j0)..."
2    maxW = 0
3    maxUV =  (0,0)
4    for u in range(0,delta+1):
5      for v in range(0,delta+1):
6        W = abs(int(pxy.coefficient({x:u,y:v})))*X^u*Y^v
7        if W > maxW:
8          maxW = W
9          maxUV = (u,v)
10   u,v = maxUV
11   maxV = 0
12   maxIJ = (0,0)
13   for i in range(0,delta+1):
14     for j in range(0,delta+2):
15       V = 8^((i-u)^2 + (j-v)^2) * abs(int(pxy.coefficient({x:i,y
            :j})))*X^i*Y^j
16       if V > maxV:
17         maxV = V
18         maxIJ = (i,j)
19   i0, j0 = maxIJ
```

Listing 1.6: Computation of the indices `i0` and `j0`

Next, matrices `S` and `M` have to be constructed. In doing so, we first compute the upper part of `M` consisting of the monomials of the polynomials $s_{a,b} \in \mathcal{S}$. Listing 1.7 outlines the code for constructing matrix `S`. From `i0` and `j0` the row indices of `S` are computed and stored to `selectedCols`. `unselectedCols` contains the row indices which were excluded from `S`. Subsequently, the polynomials $s_{a,b} \in \mathcal{S}$ are constructed from which the upper part of `M` is formed. The rows of `M` which correspond to the monomials of `pxy` are ordered according to [Cor07]. The row orderings of `S` and `M` are provided by the variables `listK` and `listKDelta` of type `list`, respectively. `S` can be obtained from the upper part of `M` by extracting those rows which are stored in `selectedCols`. In the last computation step of Listing 1.7, `n` is set to the absolute value of the determinant of `S`.

```
#compute row number of monomials with x^(i0 + i) * y^(j0 + j) for 0 <= i,j < k (selectedColumns)
columnCounter = (k+delta)^2 - 1
selectedCols = []
unselectedCols = []
for i in range(0,(k+delta)^2):
  a,b = listKDelta[i]
  if _monomialSelection(a,b,i0,j0,k):
    selectedCols.append(columnCounter)
  else:
    unselectedCols.append(columnCounter)
  columnCounter = columnCounter - 1
selectedCols.reverse()
unselectedCols.reverse()

#compute the polynomials sab
s = []
for i in range(0,k^2):
  a, b = listK[i]
  s.append(x^a * y^b * pxy)
s.reverse()

#compute upper part of MM consting of the row vectors of sab's
rowCounter = 0
for se in s:
  vector = []
  for i in range(0,(k+delta)^2):
    a,b = listKDelta[i]
    vector.append(se.coefficient({x:a,y:b}))
  vector.reverse()
  M.set_row(rowCounter,vector)
  rowCounter = rowCounter + 1

S = M.matrix_from_rows_and_columns([0..(k^2 - 1)],selectedCols)

#compute the determinant of SS
n = S.det().abs()
```

Listing 1.7: Construction of matrix `S`

To complete the computation of `M` the polynomials $r_{i,j} \in \mathcal{R}$ are computed. The monomials of each polynomial provide the pending rows of `M` which are appended to the upper part of `M` ordered according to [Cor07] (see Listing 1.8).

```
#compute polynomials rij
r = []
for iterator in range(0,(k + delta)^2):
  i, j = listKDelta[iterator]
  r.append(x^i  * y^j * n)
r.reverse()

#compute the lower part of M
#current value of rowCounter is used to continue the construction of M
for ra in r:
```

```
  vector = []
  for iterator in range(0,(k + delta)^2):
    i,j = listKDelta[iterator]
    vector.append(ra.coefficient({x:i,y:j}))
  vector.reverse()
  M.set_row(rowCounter,vector)
  rowCounter = rowCounter + 1
```

Listing 1.8: Construction of matrix `M`

From an elongated sequence of matrix multiplications as described in Section 3 `L2Prime` which corresponds to $L_2'^B$ in Section 3 is computed (see Appendix C.3). It is not necessary to compute the HNF of `L2Prime` before applying the LLL algorithm because the computation of the basis for the input matrix is an inherent operation of the implementation of the LLL algorithm we use. Thus, the LLL algorithm can directly be applied on `L2prime`. To obtain the reduce lattice base we have to cut the 0 rows of the resulting matrix:

```
L2Red = L2Prime.LLL().matrix_from_rows_and_columns([k^2..(k^2 +
       delta^2 + 2 * k * delta - 1)],[0..(delta^2 + 2 * k
       * delta - 1)])
```

From the first row of `L2Red` which equals to the shortest row vector of `L2Red` (referred to as $L_2'^{RB}$ in Section 3) we extract the coefficients of the corresponding monomials to construct the bivariate polynomial `hxy` which has the same integer roots as `pxy` (see Listing 1.9).

```
#read coefficiants of hxy
hcoeffs = L2Red.row(0).list()

#create hxy
columnCounter = (k + delta)^2 - 1
hxy = 0 * x * y
hcoeffs.reverse()
coeffCounter = 0
for iterator in range(0,(k+delta)^2):
  i,j = listKDelta[iterator]
  if columnCounter in unselectedCols:
    hxy = hxy + ((hcoeffs[coeffCounter]//X^i)//Y^j) * x^i * y^j
    coeffCounter = coeffCounter + 1
  columnCounter = columnCounter - 1
if debug : print "\n\nhxy:";print hxy
```

Listing 1.9: Construction of polynomial `hxy`

Subsequently, we compute the resultant of `pxy` and `hxy`. We obtain a non-zero univariate integer polynomial with root `x0`.

There might be the case that more than one root has been found by the algorithm. For each root `x0` we have to compute $p(x_0, y)$. From the collected pairs (`x0`,`y0`) we select the unique one with $x_0 \geq 0$, $y_0 \geq 0$ which is the root we were looking for (see Appendix C.3). Finally, if the root of `pxy` could be computed succesfully we obtain the prime factors of `N` by concatenating `x0` and `P0` as well as `y0` and `Q0`.

```
#compute the prime factors of N
p = P0 + targetRoot[0]*2^bitLengthQ0
q = Q0 + targetRoot[1]*2^bitLengthQ0
return (p,q)
```

Listing 1.10: Computation of the prime factors of `N`

Next, we describe the Sage script to reconstruct the user's private key which calls `coronFactorization()` as a subfunction (for the remaining section, N denotes the manufacturer's and n denotes the user's public modulus). The script is called by

```
sage RPK_RSA_GEN1_PHPP.sage pubKey recPricKey manPrivKey addBits k
```

where `pubKey`, `recPricKey`, and `manPrivKey` correspond to the user's public key, the file name for the reconstructed private key, and the manufacturer's private key. `addBits` indicates the number of additional bits known from p w.r.t. $p\rfloor_{l_{n/4}}$. Parameter k enables to reduce the dimension of matrix M depending on the number of additional bits known. As for the implementation of RSA_{GEN1}^{PHP}, we use the python package *PyCrypt* to access an RSA implementation.

After reading D and N from the manufacturer's private key as well as n and e from the user's public key, the least significant bits of p are computed by decrypting n and dismissing the $l_{n/4}$ most significant bits of the decrypted value. If n, k, `addBits`, and $P0 = p\rfloor_{l_{n/4}}$ satisfy the required restrictions for Coron's algorithm given in Section 3 we are able factorize n by calling `coronFactorization(n,P0,k, addBits)`. To allow $p \in \Phi_{l_{n/2}} > N$ we have to consider a case differentiation for the result of the first call of `coronFactorization()` which is illustrated in Figure 4. If the first call of `coronFactorization()` successfully computes p and q, we have to distinguish two cases: If $p \cdot q = n$, the prime factors of n are returned. Otherwise, it may be the case that $p > N$. Nevertheless, we can compute the proper value $P0$ by adding N to p'. `coronFactorization()` is called again with the updated value of `P0`. For the case that `coronFactorization()` succeeds, n could be factorized in a second attempt; p and q are returned. Otherwise, an error message is outputted.

If the first call of `coronFactorization()` failed it might be the that $p > N$, too. In that case, the reconstruction algorithm proceeds analogously to the treated case that the first call of `coronFactorization()` was successfully but $n \neq p \cdot q$.

If the factorization of n finally has been successful, eventually we have to switch p and q because OpenSSL requires $p \geq q$ for private RSA keys. Subsequently, the user's private exponent can be computed and his private key is exported in PEM format compliant to the PKCS#1 specification and written to `recPrivKey` as described for `RPK_RSA_GEN1_PHP.py`.

In the following, we provide an example[19] for the interaction between $RSA_{GEN1}^{PHP'}$ and `RPK_RSA_GEN1_PHPP.sage`. First, the user generates a 2048-bit RSA key pair using the following commands

```
openssl genrsa -out privKey.pem 2048
openssl rsa -in privKey.pem -pubout -out pubKey.pem
```

[19] In this example 38 additional bits of p are contained in p'.

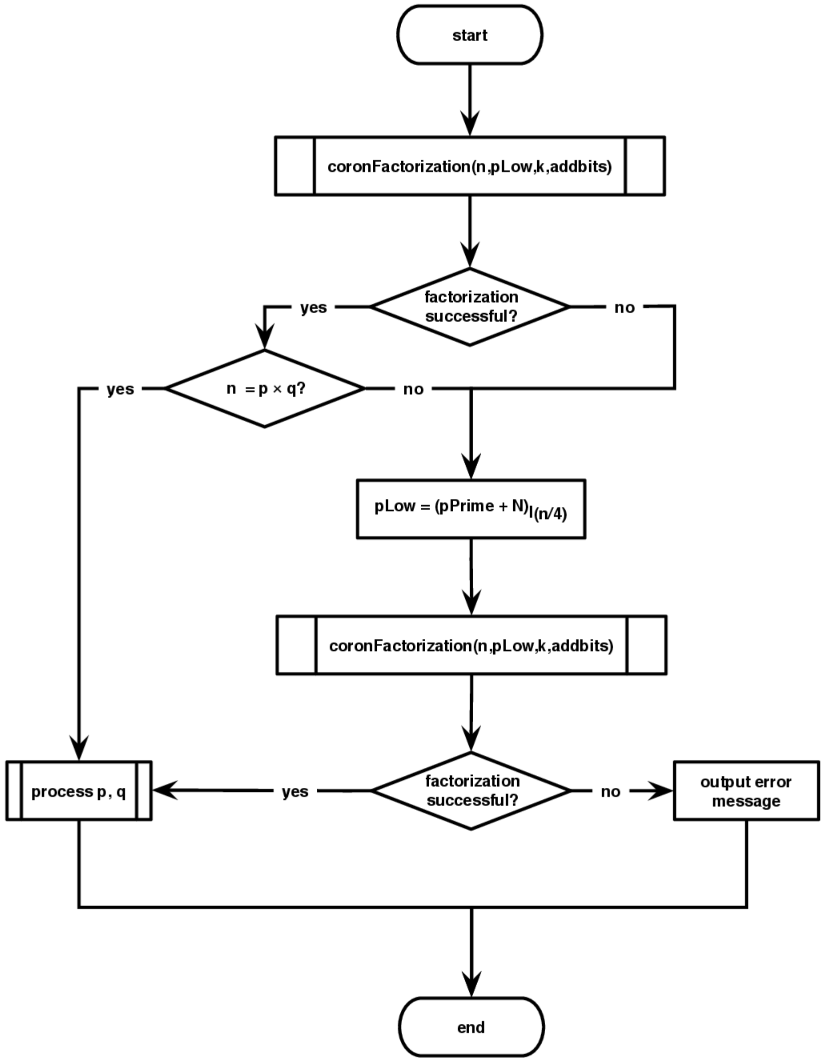

Fig. 4: Calling flow of `coronFactorization()`

and publishes his public key. From the user's public key and his own private key, the manufacturer is able to compute the user's private key by calling `RPK_RSA_PHPP.sage` with the following parameters:

```
sage RPK_RSA_GEN1_PHPP.sage pubKey.pem recPrivKey.pem
              manPrivKey.pem 38 7
```

For the reconstructed private key, a new file called `recPrivKey.pem` is created with which the manufacturer is able to decrypt messages of the target user and to sign messages in his place.

7.2.2 Evaluation. To compare the key generation running time (RT) of $RSA_{GEN1}^{PHP'}$ and RSA, we did time measurements for 1000 1024-bit and 2048-bit key generations on a 3.1 GHz PC under OSX.

The mean running time for 1024-bit key generations of the unmodified RSA version resulted in $0.03959s$ while the mean running time of $RSA_{GEN1}^{PHP'}$ could be estimated with $0.2308s$. On average, $RSA_{GEN1}^{PHP'}$ takes ≈ 6 times longer for generating 1024-bit keys. This difference might not necessarily give rise to suspection of the existence of a backdoor without performing a side channel analysis.

The coefficient of variation of $RSA_{GEN1}^{PHP'}$ was ascertained with 0.8601 which is more than 30% higher than the corresponding value of RSA which was ascertained with 0.559.

For the 2048-bit key generation we achived similar results. The deviation of the mean running time between $RSA_{GEN1}^{PHP'}$ ($2.3095s$) and RSA ($0.2625s$) increases slightly which can be described by the factor of ≈ 9. In contrast, the difference of the coefficient of variation for RSA_{GEN1}^{PHP} (0.8995) and RSA (0.6695) decreases, but the deviation still exceeds 20%.

As for RSA_{GEN1}^{PHP}, the substantial differences of the variation between $RSA_{GEN1}^{PHP'}$ and RSA inhibit the possibility to adapt the key generation running time of $RSA_{GEN1}^{PHP'}$ to RSA for arbitrary key sizes by hardware modification.

The box plots for the key generation running times for 1024-bit and 2048-bit keys (see Figure 5) provide additional statistical measures and give a graphical illustration of the measurements which confirm our results.

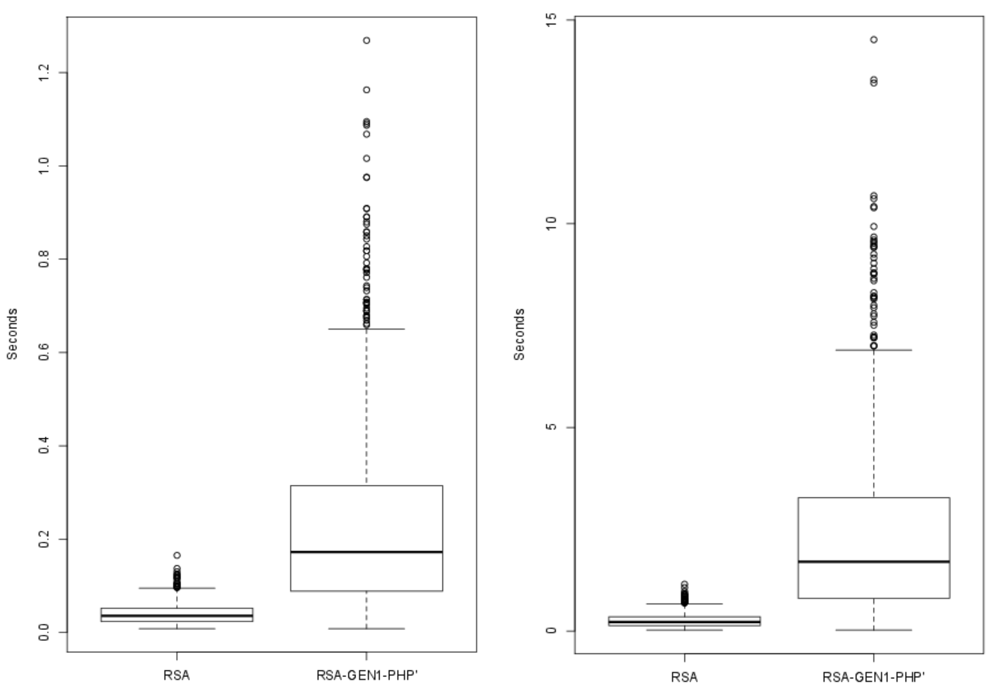

Fig. 5: Boxplot for the running time of RSA and $RSA_{GEN1}^{PHP'}$ for generating 1024-bit keys (left) and 2048-bit keys (right)

Table 5 presents the determined values of each considered statistical meassure of $RSA_{GEN1}^{PHP'}$ compared to those of RSA for 1024-bit and 2048-bit key generation, respectively. For a p-quantile it holds that at least $p \cdot 100\%$ of all measured values

	1024-bit keys		2048-bit keys	
stat. measure	RSA	$RSA_{GEN1}^{PHP'}$	RSA	$RSA_{GEN1}^{PHP'}$
mean [s]	0.03959	0.2308	0.2625	2.3095
median [s]	0.0355	0.172	0.221	1.703
coef. of variation	0.559	0.8601	0.6695	0.8995
maximum [s]	0.165	1.269	1.15	14.52
minimum [s]	0.008	0.008	0.022	0.023
0.25-quantile [s]	0.023	0.088	0.132	0.8058
0.75-quantile [s]	0.052	0.31425	0.352	3.269
quartile range [s]	0.029	0.2262	0.22	2.4633

Table 5: Statitical measures for the key generation running time of $RSA_{GEN1}^{PHP'}$ and RSA

are smaller than or equal to the determined value of the quantile. The 0.25-quantile and the 0.75-quantile are indicated by the upper and lower border of the box for each box plot. Within this box 50% of all meassurements are located.

The median (the horizontal line in a box) is more resistant against outliers than the mean. Since even for the median the running time of RSA and $RSA_{GEN1}^{PHP'}$ significantly deviates, we can conclude that for the mean running time difference between RSA and $RSA_{GEN1}^{PHP'}$ the contribution of outliers is not decisive. In each box plot, outliers can be identified by those meassurements which lie beyond the whiskers.

The complexity of the private key reconstruction function is determined by the complexity of Coron's algorithm which is polynomial in k (for the application of factoring a composite of two prime factors we have that $\delta = 1$). To achieve running times $< 1h$ for the factorization of N, in practice it is the case that additional bits of p have to be known. Note that this does not have a pratical implication on the properties of $RSA_{GEN1}^{PHP'}$ as discussed in Section 6.

The results of practical experiments with our implementation of Coron's algorithm on a 3.1 GHz PC under OSX are summarized in Table 6. Consider the first row of the 1024-bit key block of Table 6 which has to be read as follows: a 1024-bit composite N of two prime factors, both of bitsize $l_{N/2}$, is factorized in $2s$ for $k = 5$ and 280 bits of p known, i.e., 24 additonal bits of p are known. The dimension of the lattice base which is reduced by applying the LLL-algorithm is equal to 11 and the running time for LLL equals approximatelly to the running time of the whole factorization process.

If it is premised that no additional bits of p are known, practical experiments showed that it is more efficient to guess some bits of p instead of setting $k = \lfloor log(W) \rfloor$. The amount of bits which have to be guessed depends on the keysize which in turn is responsible for the relationship between k, the minimal number of additional bits which have to be known, and the running time to factorize N. For keys of bitsize 512, it is possible to factorize N on average in $\approx 9\ min$ by guessing 10 bits and choosing $k = 6$:

$$\frac{1}{2} \cdot \frac{2^{10} \cdot 1s}{3600} \approx 9\ min$$

If the key size equals to 1024 bits the best running time to factorize N can be achieved by guessing 14 bits with $k = 9$ which is on average $\approx 11\ days$.

The optimal setting to factorize 2048-bit keys has been determined to guess 23 additional bits with $k = 11$: an average running time of ≈ 110000 *days* has to be expected.

The estimated minimal running times for factoring without additional bits known assume that for $RSA^{PHP'}_{GEN1}$ it holds that the number of prime factors for the user's public modulus which are larger that the manufacturer's public modulus can be neglected. If this is not the case, we have to assume that for each guess of the unknown bits of p `coronFactorization()` has to be called twice (see Figure 4). Thus, the estimated running times have to be doubled.

Note that the job of guessing the correct missing bits of p can be split into a set of single jobs which can be processed in parallel since each guess is indepedent of previous and upcoming guesses. Thus, the estimated running time of the private key recovery function of $RSA^{PHP'}_{GEN1}$ can approximately be reduced by a factor of $1/\#processors$.

N	k	bits of p given	dimension of L'^{B}_{2}	LLL	factorization
512	4	142	9	< 1 s	< 1 s
512	5	140	11	< 1 s	< 1 s
512	6	138	13	< 1 s	< 1 s
512	7	137	15	2 s	3 s
512	8	136	17	12 s	16 s
512	9	135	19	28 s	48 s
512	11	134	23	2 min	3 min
512	12	133	25	4 min	7 min
1024	5	280	11	2 s	2 s
1024	6	276	13	6 s	7 s
1024	7	273	15	17 s	19 s
1024	8	271	17	42 s	55 s
1024	9	270	19	2 min	2 min
1024	10	269	21	4 min	5 min
1024	11	268	23	7 min	11 min
1024	12	267	25	15 min	21 min
2048	5	559	11	6 s	7 s
2048	6	552	13	21 s	23 s
2048	7	546	15	1 min	1 min
2048	8	543	17	3 min	3 min
2048	9	540	19	6 min	8 min
2048	10	537	22	14 min	17 min
2048	11	535	23	30 min	38 min

Table 6: Running times for factoring $N = p \cdot q$ given the $l_{N/2}$ least significant bits of p using Coron's algorithm implemented with Sage on a 3.1 GHz PC under OSX

8 Conclusion

The primary goal of this paper has been to introduce, implement, and analyze two novel RSA backdoors: RSA^{PHP}_{GEN1} and $RSA^{PHP'}_{GEN1}$. The basic mechanism of both backdoors which distinguishes them from a lot of existing RSA backdoors is to hide the first RSA prime factor into the second one. $RSA^{PHP'}_{GEN1}$ is an extension of RSA^{PHP}_{GEN1}. It invokes Coppersmith's factorization attack to bisect the number of bits of the first prime factor which have to be known by the manufacturer—and thus the bits which have to be hidden in the second prime factor of n—to reconstruct the user's private key. Due to the modifications distinguishing RSA_{GEN1}^{PHP} from $RSA_{GEN1}^{PHP'}$ additional backdoor related properties could be satisfied or at least could be improved.

In order to present our results, we first introduced context-relevant cryptographic and mathematical foundations in Section 2 and 3. Subsequently, we gave a wide-ranged overview of existing backdoors for different cryptosystems based on various ideas and concepts in Section 4. To provide a comprehensive and comparative security analysis for RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$, it was necessary to revise the established definition of a SETUP in Section 5. In Section 6 we finally introduced RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$, provided the corresponding security analysis basing on Section 5, and compared our results to the properties of existing backdoors which were introduced in Section 4. To be able to evaluate RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ w.r.t. their functionality and run time behavior, we implemented both backdoors into OpenSSL. The implementation and its evaluation was extensively discussed in Section 7.

The essential contributions of this work can be divided into those which concern the comparative security analysis and those concerning the implementation of RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$.

The security analysis of both novel backdoors revealed a high correlation w.r.t. the properties of an honest RSA implementation. Particularly, the second proposed backdoor, $RSA_{GEN1}^{PHP'}$, can be considered to be unique w.r.t. the quality it satisfies the subproperties of Concealment compared to previously published prominent RSA backdoors. The implementation of RSA_{GEN1}^{PHP} and $RSA_{GEN1}^{PHP'}$ in combination with the corresponding private key recovering functions give a proof of work for our novel backdoors and would allow a direct comparison of the key generation running time to other backdoors as soon as corresponding results will be available.

Appendices

A Fundamentals of Linear Algebra

A.1 Inner Product

Let $v = (v_1, ..., v_n)$ and $w = (w_1, ..., w_n)$ be two vectors of the same length over the same field. The inner product of v and w, written $\langle v, w \rangle$, is defined as

$$\langle v, w \rangle = \sum_{i=1}^{n} v_i w_i.$$

A.2 Euclidean Norm

Let v be a vector in the Euclidean space $\mathbb{R}^n$. The Euclidean norm of v written $||v||_2$ is defined as

$$||v||_2 = \sqrt{\langle v, v \rangle}$$

A.3 Adjugate Matrix

The adjugate matrix of a square matrix $M \in \mathbb{K}^{n \times n}$ is defined as the transpose of the cofactor matrix of M:

$$adj(M) = Cof(M)^T$$

The entries of cofactor matrix $Cof(M)$, called cofactors $\tilde{m}_{ij}$ of $M = (m_{ij})_{ij}$, can be computed with the following formula:

$$\tilde{m}_{ij} = (-1)^{i+j} \cdot Minor_{ij}.$$

The Minor $Minor_{ij}$ is defined to be the determinant of the submatrix M' of M which is obtained after deleting the i-th row and the j-th column.

An important property of the adjugate matrix is the following one:

$$M \cdot adj(M) = adj(M) \cdot M = det(M) \cdot E.$$

A.4 Gramian Determinant

Let $M \in \mathbb{K}^{m \times n}$. The Gramian determinant is defined to be the determinant of the Gramian matrix of M $(M^T M)$:

$$Gram(M) = det(M^T M).$$

The Gramian determinant of M is equal to zero iff the columns of M are linearly independent.

A.5 Gram-Schmidt Algorithm

A Gram-Schmidt orthogonalization algorithm computes an orthogonal basis $\{b_1^*, ..., b_\omega^*\}$ (Gram-Schmidt basis) on an input basis $\{b_1, ..., b_\omega\}$ iteratively by computing

$$b_i^* = b_i - \sum_{j=1}^{i-1} \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \cdot b_j^*$$

for $2 \leq i \leq n$ and $b_1^* = b_1$.

A.6 Hermite Normal Form

According to [Gal12] an $n \times m$ integer matrix $M = M_{i,j}$ $(0 \leq i \leq n,\ 0 \leq j \leq m)$ is in Hermite nomal form (HNF) if there exists an integer $r \in [1, n]$ and a strictly increasing map $f : \{1, ..., n - r\} \rightarrow \{1, ..., m\}$ such that:

1. the last r rows of M are zero,
2. $0 \leq M_{j,f(i)} < M_{i,f(i)}$ for $1 \leq j < i$ and $M_{j,f(i)} = 0$ for $i < j \leq n$.

The HNF of a matrix is of rank m and unique. Algorithms for computing the HNF which have a polynomial runtime in the input size $(\mathcal{O}(n^2 \log M))$ are restricted to elementary row operations which include the addition of rows to each other, multiplying a row with a non-zero constant, and interchanging rows.

B Backdoor Related Terms and Tools

B.1 Random Oracle Model

A random oracle is a theoretical model which idealizes the properties of hash functions. The random oracle computes an infinitely random bit string $\mathcal{R}(s)$ on an input bit string s of finite length. For each input string, the random oracle remembers the output such that for the same input the resulting random bit stream is always the same.

Definition B.1 (Random Oracle). *A random oracle $\mathcal{R}(s) : \{0, 1\}^* \rightarrow \{0, 1\}^\infty$ computes on an input bit string of arbitrary length an infinitely bit string with each bit chosen uniformly at random and independent from s.*

Random oracles are a common tool to prove the security of cryptosystems which rely on the properties of involved cryptographic hash functions. Those security proofs are accomplished by assuming that the used hash functions have the same properties as a random oracle.

Since a random oracle is a theoretical model which does not exist in practice we refer to protocols as defined in [BR93] when random oracles are used in algorithms presented above.

B.2 Leakage Scheme

The term of *leakage scheme* was introduced in [YY97], and describes the *leakage bandwidth* of the system, i.e, how many times it has to be utilized to leak a certain amount of secret information to the manufacturer.

Definition B.2 (Leakage Scheme). *A (m, n)-leakage scheme is a SETUP mechanism that leaks m keys/secret messages over n keys/messages with $m \leq n$.*

B.3 Probabilistic Bias Removal Method

In [YY97] and [YY04] Young and Yung proposed a method called *Probabilistic Bias Removal Method* (PBRM) solving the *biasing problem* which was discovered by Jo Schueth in [YY96] and reported in the public news group *sci.crypt* [YY97]. In general, the biasing problem occurres when a random number is drawn from the uniformly distributed set $\{1, 2, ..., R\}$ but actually the random number has to be drawn from the uniformly distributed set $\{1, 2, ..., S\}$ with $S > R > S/2$ (values in $\{R+1, R+2, ..., S\}$ are drawn with probability zero). In the area of Kleptography this problem occurres, e.g., when the manufacturer's public key (N, E) with $N \in \{0,1\}^{l_N}$ is used to decrypt the user's key and the result is treated as uniformly drawn from $\{0, 1, ..., 2^{l_N}\}$.

The Probabilistic Bias Removal Method solves this problem: from a given uniformly random value $x \in \{1, 2, ..., R\}$ an uniformly random value from $\{1, 2, ..., S\}$ can be easily derived by calling `PBRM(R,S,x)` which is given by Algorithm B.1. Additionally, the access to an unbiased coin c is necessary.

Input: R, S, x
Output: (e, x')

set $e = 1$.
set $x' = 0$.
if $x \leq S - R$ *and* $c = 1$ **then**
    set $x' = x$.
end
if $x \leq S - R$ *and* $c = 0$ **then**
    set $x' = S - 1 - x$.
end
if $x > S - R$ *and* $c = 1$ **then**
    set $x' = x$.
end
if $x > S - R$ *and* $c = 0$ **then**
    set $e = -1$.
end
return (e, x').

Algorithm B.1: The Probabilistic Bias Removal Method

Since there is the possibility for PBRM to fail which is the case if `PBRM(R,S,x)` returns $e = -1$ and the method has to be invoked again, PBRM belongs to category of *Monte Carlo* algorithms.

The following lemma argues the correctness of PBRM. The appropriate proof is detailed in [YY04].

Lemma B.1. *Let $S > R > S/2$, let x be a value chosen uniformly at random from $\{1, 2, ..., R\}$, and let (e, x') =* `PBRM(R,S,x)`*. If $e = 1$ then x' is uniformly distributed in $\{1, 2, ..., S\}$.*

C Program Code

C.1 RSA^{PHP}_{GEN1}: Private Key Reconstruction Function

```
#!/usr/bin/python
#This script reconstructs private keys from public keys generated with
    RSA_GEN1_PHP
from Crypto.PublicKey import RSA
import sys
```

```
def extendedEuclideanAlgorithm(n, m):
    if n == 0:
        return (m, 0, 1)
    else:
        g, y, x = extendedEuclideanAlgorithm(m % n, n)
        return (g, x - (m // n) * y, y)

def modInv(a, m):
    g, x, y = extendedEuclideanAlgorithm(a, m)
    if g != 1:
        print "Error: inverse of e mod Phi(n) does not exist."
        sys.exit(1)
    else:
        return x % m

debug = True
print "recontructing private key..."

if len(sys.argv) < 4:
  sys.exit("Usage: %s input-public-key-file output-private-key-file manufacturer-private-key-file" % sys.argv[0])

pubKeyUserInput = sys.argv[1]
privKeyUserOutput = sys.argv[2]
privKeyManufacturerInput = sys.argv[3]

pubKeyUser = open(pubKeyUserInput,'r')
privKeyManufacturer = open(privKeyManufacturerInput,'r')

#read D and N from the manufacturer's private key
rsaManufacturer = RSA.importKey(privKeyManufacturer.read())
D = rsaManufacturer.d
N = rsaManufacturer.n

#read n and e from the user's public key
rsaUserPub = RSA.importKey(pubKeyUser.read())
n = rsaUserPub.n
e = rsaUserPub.e

#reconstruct the hidden prime factor p
p = pow(n,D,N)
if n % p != 0:
    p = p + N
if debug : print "\n\np:"; print p;

#compute q
q = n/p
if debug : print "\n\nq:"; print q;

#adaption to openSSL rsa_gen.c
if p < q:
    tmp = q
    q = p
    p = tmp

#compute d
ePhi = (p-1)*(q-1)
d = modInv(e,ePhi)
if debug : print "\n\nd:"; print d;

#construct the user's RSA instance from n,e,d,p,q
rsaUser = RSA.construct((n,e,d,p,q))

#write the user's private key to the output file
extractedKey = rsaUser.exportKey('PEM',pkcs=1)
if debug : print "\n\nextracted key:"; print extractedKey;
privKeyUser = file(privKeyUserOutput, "w")
privKeyUser.write(extractedKey)
```

```
privKeyUser.write("\n")

pubKeyUser.close
privKeyManufacturer.close()
privKeyUser.close()
```

C.2 $RSA^{PHP'}_{GEN1}$: Private Key Reconstruction Script

```
#!/usr/bin/python
#This script reconstructs private keys from public keys generated with
    RSA_GEN_PHP'
from Crypto.PublicKey import RSA
from sage.all_cmdline import *
import c2007Module
import math
import sys

def extendedEuclideanAlgorithm(n, m):
    if n == 0:
        return (m, 0, 1)
    else:
        g, y, x = extendedEuclideanAlgorithm(m % n, n)
        return (g, x - (m // n) * y, y)

def modInv(a, m):
    g, x, y = extendedEuclideanAlgorithm(a, m)
    if g != 1:
        print "Error: inverse of e mod Phi(n) does not exist."
        sys.exit(1)
    else:
        return x % m

def clearBit(n, offset):
    mask = ~(int(1) << offset)
    return(n & mask)

def numberOfBits(n):
    return (int(math.log(n,2)) + 1)

debug = True

print "recontructing private key..."

if len(sys.argv) < 6:
    sys.exit("Usage: %s input-public-key-file output-private-key-file
        manufacturer-private-key-file additional-bits k" % sys.argv[0])

pubKeyUserInput = sys.argv[1]
privKeyUserOutput = sys.argv[2]
privKeyManufacturerInput = sys.argv[3]
addBits = int(sys.argv[4])
k = int(sys.argv[5])

pubKeyUser = open(pubKeyUserInput,'r')
privKeyManufacturer = open(privKeyManufacturerInput,'r')

#read D and N from the manufacturer's private key
rsaManufacturer = RSA.importKey(privKeyManufacturer.read())
D = rsaManufacturer.d
N = rsaManufacturer.n

#read n and e from the user's public key
rsaUserPub = RSA.importKey(pubKeyUser.read())
n = long(rsaUserPub.n)
e = long(rsaUserPub.e)
secPar = numberOfBits(n)/2
```

```python
#compute the hidden least significant bits of p
pPrime = pow(n,D,N)
pLow = pPrime
for i in range(secPar-1,secPar-secPar/2+addBits-1,-1):
    pLow = clearBit(pLow,i)
if debug : print "\n\nlower bits of p:"; print pLow;

#compute p and q basing on the least significant bits of p
result = c2007Module.coronFactorization(n,pLow,k,addBits)
if result == -1:

    #if p' might be larger than N
    pLow = pPrime + N

    #compute least significant bits of p'
    for i in range(secPar-1,secPar-secPar/2+addBits-1,-1):
        pLow = clearBit(pLow,i)
    if debug : print "\n\nlower bits of p:"; print pLow;

    result = c2007Module.coronFactorization(n,pLow,k,addBits)
    if result == -1:
        print "\n\nError: n can not be factorized"
        pubKeyUser.close()
        sys.exit(1)
    else:
     (p,q) = result
else:
    (p,q) = result

    #check if the computed roots are the primefactors of n
    if n == p * q:
        (p,q) = result
    else:

        #p' might be larger than N
        pLow = pPrime + N

        #compute least significant bits of p'
        for i in range(secPar-1,secPar-secPar/2+addBits-1,-1):
            pLow = clearBit(pLow,i)
        if debug : print "\n\nlower bits of p:"; print pLow;

        result = c2007Module.coronFactorization(n,pLow,k,addBits)
        if result == -1:
            print "\n\nError: n can not be factorized"
            pubKeyUser.close()
            sys.exit(1)
        else:
            (p,q) = result
p = long(p)
q = long(q)

#adaption to openSSL rsa_gen.c
if p < q:
    tmp = q
    q = p
    p = tmp

#compute d
ePhi = (p-1)*(q-1)
d = long(modInv(e,ePhi))
if debug : print "\n\nd:"; print d;

#construct the user's RSA instance from n,e,d,p,q
rsaUser = RSA.construct((n,e,d,p,q))

#write the user's private key to the output file
extractedKey = rsaUser.exportKey('PEM',pkcs=1)
```

```
if debug : print "\n\nextracted key:"; print extractedKey;
privKeyUser = file(privKeyUserOutput , "w")
privKeyUser.write(extractedKey)
privKeyUser.write("\n")

pubKeyUser.close()
privKeyManufacturer.close()
privKeyUser.close()
```

C.3 $RSA^{PHP'}_{GEN1}$: Coron's Factorization Algorithm

```
from itertools import *
import sys
import copy
import math

#compute the column indices which are used to compute matrix S
def _monomialSelection(a,b,i0,j0,k):
    return (a >= i0) and (b >= j0) and (a - i0 < k) and (b - j0 < k)

#set the order of monomials
def _createCombinationList(maxVal):
  list = []
  cx = 0
  cy = 0
  base = 0
  max = maxVal
  maxit = maxVal
  for i in range(0,max):
    list.append((base,base))
    for j in range(1,maxit+1):
      cy = cy + j
      list.append((cx,cy))
      cy = base
      cx = cx + j
      list.append((cx,cy))
      cx = base
    maxit = maxit - 1
    base = base + 1
    cx = base
    cy = base
  list.append((max,max))
  return list

#compute the number of bits for the input
def _numberOfBits(n):
  return (int(math.log(n,2)) + 1)

#returns integer with bit at position offset set to 0
def _clearBit(n, offset):
  mask = ~(int(1) << offset)
  return(n & mask)

#returns integer with bit at position offset set to 1
def _setBit(int_type, offset):
  mask = int(1) << offset
  return (int_type | mask)

#multivariate Hensel lemma
def multHensel(p,q,N,P0,i):
  return ((((N - p*q) >> i)&1) - ((P0 >> i)&1)) % 2

#implementation of Coron's algorithm for finding small roots of bivariate integer equations from 2007 adapted for the application of factorization
def coronFactorization(N,P0,k,addBits):
```

```
  debug = False

  #define polynomial ring over the integers
  R.<x,y> = ZZ[]

  #compute Q0 by applying the multivariate Hensel lemma
  nob = int(math.ceil(0.25 * _numberOfBits(N))) + addBits
  pi = 1
  qi = 1
  bits = nob
  for i in range(1,nob):
    b = multHensel(pi,qi,N,P0,i)
    if (P0 >> i)&1:
      pi = _setBit(pi,i)
    else:
      pi = _clearBit(pi,i)
    if b == 1:
      qi = _setBit(qi,i)
    else:
      qi = _clearBit(qi,i)
  Q0 = qi

  #compute the target polynomial p(x,y)
  bitLengthQ = int(_numberOfBits(N)/2)
  bitLengthQ0 = int(_numberOfBits(N)/4) + addBits
  pxy = 2^bitLengthQ0 * x * y + Q0 * x + P0 * y + int((P0*Q0 - N)/2^bitLengthQ0)
  print "\ncompute the roots of the polynomial: " + str(pxy)

  #assume that this algorithm is only used for bivariate polynomials of degree 1
  delta = 1

  #reduce the upper bounds of x0,y0 according to the additional bits known
  X = 2^(bitLengthQ - bitLengthQ0)
  Y = X

  W = abs(int(pxy.coefficient({x:1,y:0})))*X
  if debug : print "\n\nX: " + str(X) + " Y: " + str(Y) + " W: " + str(W)

  #definition of matrix dimensions
  M = matrix(ZZ,(k^2 + (k + delta)^2),(k + delta)^2)
  S = matrix(ZZ, k^2, k^2)

  #access the monomial order
  listKDelta = _createCombinationList(k + delta - 1)
  listK = _createCombinationList(k - 1)
  if debug : print "\n\nlistKDelta:";print listKDelta
  if debug : print "\n\nlistK:";print listK

  #compute i0j0
  print "\n\ncomputing (i0,j0)..."
  maxW = 0
  maxUV =  (0,0)
  for u in range(0,delta+1):
    for v in range(0,delta+1):
      W = abs(int(pxy.coefficient({x:u,y:v})))*X^u*Y^v
      if W > maxW:
        maxW = W
        maxUV = (u,v)
  u,v = maxUV
  maxV = 0
  maxIJ = (0,0)
  for i in range(0,delta+1):
    for j in range(0,delta+2):
      V = 8^((i-u)^2 + (j-v)^2) * abs(int(pxy.coefficient({x:i,y:j})))*X^i*Y^j
```

```
      if V > maxV:
        maxV = V
        maxIJ = (i,j)
  i0, j0 = maxIJ
  if debug : print "(i0,j0) = (" + str(i0) + "," + str(j0) + ")"

  #compute row number of monomials with x^(i0 + i) * y^(j0 + j) for 0
      <= i,j < k
  columnCounter = (k+delta)^2 - 1
  selectedCols = []
  unselectedCols = []
  for i in range(0,(k+delta)^2):
    a,b = listKDelta[i]
    if _monomialSelection(a,b,i0,j0,k):
      selectedCols.append(columnCounter)
    else:
      unselectedCols.append(columnCounter)
    columnCounter = columnCounter - 1
  selectedCols.reverse()
  unselectedCols.reverse()

  #compute the polynomials sab
  s = []
  for i in range(0,k^2):
    a, b = listK[i]
    s.append(x^a * y^b * pxy)
  s.reverse()

  #compute upper part of MM consting of the row vectors of sab's
  rowCounter = 0
  for se in s:
    vector = []
    for i in range(0,(k+delta)^2):
      a,b = listKDelta[i]
      vector.append(se.coefficient({x:a,y:b}))
    vector.reverse()
    M.set_row(rowCounter,vector)
    rowCounter = rowCounter + 1

  S = M.matrix_from_rows_and_columns([0..(k^2 - 1)],selectedCols)

  #compute the determinant of SS
  n = S.det().abs()

  if debug : print "\n\nS:";print S.str();
  if debug : print "\n\nabsolute value of determinant of SS:";print n

  #compute polynomials rij
  r = []
  for iterator in range(0,(k + delta)^2):
    i, j = listKDelta[iterator]
    r.append(x^i  * y^j * n)
  r.reverse()

  #compute the lower part of M
  #current value of rowCounter is used to continue the construction of
      M
  for ra in r:
    vector = []
    for iterator in range(0,(k + delta)^2):
      i,j = listKDelta[iterator]
      vector.append(ra.coefficient({x:i,y:j}))
    vector.reverse()
    M.set_row(rowCounter,vector)
    rowCounter = rowCounter + 1
  if debug : print "\n\nM:";print M.str()

  #prepare M to compute L2
  SelCols = M.matrix_from_rows_and_columns([0..(k^2-1)],selectedCols)
```

```
    UnselCols = M.matrix_from_rows_and_columns([0..(k^2-1)],
        unselectedCols)
    L = M.matrix_from_rows_and_columns([k^2..((k^2 + (k + delta)^2) - 1)
        ],        [0..((k + delta)^2 - 1)])

    #reorder columns of M
    M2 = SelCols.augment(UnselCols)
    M2 = block_matrix([M2,L],nrows=2)
    if debug : print "\n\nM2:";print M2.str()

    #define auxiliary matrices
    print "\n\ncompute adjugate matrix of S..."
    SAdj = S.adjoint()
    IdK2 = matrix.identity(k^2)
    IdOmega = matrix.identity(delta^2 + 2 * k * delta)
    ZerosK2 = matrix(ZZ,k^2,k^2,[])
    ZerosOmega = matrix(ZZ,delta^2 + 2 * k * delta,delta^2 + 2 * k *
        delta,[])
    ZerosK2Omega = matrix(ZZ,k^2,delta^2 + 2 * k * delta,[])
    ZerosOmegaK2 = matrix(ZZ,delta^2 + 2 * k * delta,k^2,[])
    if debug : print "SAdj:";print SAdj.str()

    M3 = block_matrix([IdK2,ZerosK2,ZerosK2Omega,-SAdj,IdK2,ZerosK2Omega,
        ZerosOmegaK2,ZerosOmegaK2,IdOmega],ncols = 3)
    if debug: print "\n\nM3:";print M3.str()

    M4 = M3 * M2
    if debug : print "\n\nM4:";print M4.str()

    L2 = M4.matrix_from_rows_and_columns([k^2,..,((k^2 + (k + delta)^2) -
         1)],      [k^2,..,((k + delta)^2 - 1)])
    if debug : print "\n\nL2:";print L2

    #compute L2' by multiplying each colum with the corresponding X^iY^j
        term
    listKDeltaRev1 = copy.deepcopy(listKDelta)
    listKDeltaRev1.reverse()
    for i in range(0,(k+delta)^2):
      for j in range(0,(delta^2 + 2 * k * delta)):
        a, b = listKDeltaRev1[unselectedCols[j]]
        L2[i,j] = L2[i,j] * X^a * Y^b
    L2Prime = L2
    if debug : print "\n\nL2':";print L2Prime


    #perform the lattice reduction
    #it is not necessary to compute the basis of L2 first
    print "\n\nperform lattice reduction..."
    L2Red = L2Prime.LLL().matrix_from_rows_and_columns([k^2..(k^2 + delta
        ^2 + 2 * k * delta - 1)],[0..(delta^2 + 2 * k * delta - 1)])
    if debug : print "L2Red':";print L2Red

    #read coefficiants of hxy
    hcoeffs = L2Red.row(0).list()

    #create hxy
    columnCounter = (k + delta)^2 - 1
    hxy = 0 * x * y
    hcoeffs.reverse()
    coeffCounter = 0
    for iterator in range(0,(k+delta)^2):
      i,j = listKDelta[iterator]
      if columnCounter in unselectedCols:
        hxy = hxy + ((hcoeffs[coeffCounter]//X^i)//Y^j) * x^i * y^j
        coeffCounter = coeffCounter + 1
      columnCounter = columnCounter - 1
    if debug : print "\n\nhxy:";print hxy

    #compute the resultant
```

```
  print "\n\ncomputing resultant..."
  Qx = pxy.resultant(hxy,y)
  if debug : print "resultant:";print Qx

  #compute the root of the resultant
  print "\n\ncomputing roots of resultant..."
  P.<x> = ZZ[]
  Qx1 = P(Qx)
  rootsX = Qx1.roots(multiplicities = False)
  if len(rootsX) == 0:
    return -1
  if debug : print "\n\n-----------------------------"
  targetRoot = []

  #there can be more than one root
  for x0 in rootsX:
    P2.<y> = ZZ[]
    px0y = P2(pxy(x0,y))
    if debug : print "px0y:";print px0y
    rootsY = px0y.roots(multiplicities = False)
    if len(rootsY) == 0:
      if debug : print "\n\nno roots computed"
      if debug : print "-----------------------------"
      continue
    #compute the root of px0y
    print "\n\ncomputing roots of px0y..."
    y0 = rootsY[0] #attention: there can be more than one root!
    root = (x0,y0)
    if x0 >= 0 and y0 >= 0:
      targetRoot.append(x0)
      targetRoot.append(y0)
    if debug : print "\n\ncomputed root:";print root
    if debug : print "-----------------------------"
  if len(targetRoot) == 0:
    return -2
  print "\n\ntarget root: (" + str(targetRoot[0]) + "," + str(
      targetRoot[1]) + ")"

  #compute the prime factors of N
  p = P0 + targetRoot[0]*2^bitLengthQ0
  q = Q0 + targetRoot[1]*2^bitLengthQ0

  print "\n\nprime factors of n: " +  "p = " + str(p) + ", q = " + str(
      q)

  #return targetRoot
  return (p,q)
```

References

[And93] R. J. Anderson. A Pratical RSA Trapdoor. *Journal of Electronics Letters*, 29, 1993.

[Arb08] G. Arboit. *Two Mathematical Security Aspects of the RSA Cryptosystem*. PhD thesis, McGill University - School of Computer Science, 2008.

[BD00] D. Boneh and G. Durfee. Cryptanalysis of RSA with Private Key d Less Than N0.292. *IEEE Transactions on Information Theory*, 46:1339–1349, 2000.

[BDF98] D. Boneh, G. Durfee, and Y. Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In *Advances in Cryptology – ASIACRYPT 1998*, volume 1514 of *Lecture Notes in Computer Science*, pages 25–34. Springer Berlin Heidelberg, 1998.

[BMS84] E. Bach, G. Miller, and J. Shallit. Sums of Divisors, Perfect Numbers, and Factoring. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC 1984, pages 183–190, 1984.

[BR93] M. Bellare and P. Rogaway. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS 1993, pages 62–73, 1993.

[Buc10] J. Buchmann. *Einführung in die Kryptographie*. Springer, 5 edition, 2010.

[CFA+06] R. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Niguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperellitptic Curve Cryptography*. Chapman & Hall/CRC, 2006.

[Cop96] D. Coppersmith. Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. In *Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques*, volume 1070 of *EUROCRYPT 1996*, pages 178–189. Springer Berlin Heidelberg, 1996.

[Cop97] D. Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10:233–260, 1997.

[Cor04] J.-S. Coron. Finding Small Roots of Bivariate Integer Polynomial Equations Revisited. In *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 492–505. Springer Berlin Heidelberg, 2004.

[Cor07] J.-S. Coron. Finding Small Roots of Bivariate Integer Polynomial Equations: A Direct Approach. In *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 379–394. Springer Berlin Heidelberg, 2007.

[CS03] C. Crépeau and A. Slakmon. Simple Backdoors for RSA Key Generation. In *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 403–416. Springer Berlin Heidelberg, 2003.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[DK07] H. Delfs and H. Knebl. *Introduction to Cryptography*. Springer, 2007.

[Gal12] S. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

[GCL92] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[HG97] N. Howgrave-Graham. Finding Small Roots of Univariate Modular Equations Revisited. In *Proceedings of the 6th IMA International Conference on Cryptography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pages 131–142. Springer Berlin Heidelberg, 1997.

[HS09] N. Heninger and H. Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2009.

[JPV00] M. Joye, P. Paillier, and S. Vaudenay. Efficient Generation of Prime Numbers. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 340–354. Springer Berlin Heidelberg, 2000.

[Kno13] F. Knoke. RSA bestreitet Millionen-Deal mit der NSA. http://www.spiegel.de/netzwelt/web/it-firma-rsa-dementiert-10-millionen-deal-mit-nsa-a-940620.html, 2013.

[Len08] H. W. Lenstra. Lattices. *Algorithmic Number Theorie: Lattices, Number Fields, Curves, and Cryptography*, pages 127–181, 2008.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring Polynomials with Rational Coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[Maj13] M. Majica. Sicherheitsfirma RSA warnt vor sich selbst. http://www.zeit.de/digital/datenschutz/2013-09/rsa-bsafe-kryptografie-nsa, 2013.

[May04] A. May. Computing the RSA Secret Key is Deterministic Polynomial Time Equivalent to Factoring. In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 213–219. Springer Berlin Heidelberg, 2004.

[Sch13] J. Schmidt. Todesurteil für Verschlüsselung in den USA. http://www.heise.de/security/artikel/Todesurteil-fuer-Verschluesselung-in-den-USA-1972561.html, 2013.

[The03] The OpenSSL Project. OpenSSL: The Open Source Toolkit for SSL/TLS. www.openssl.org, April 2003.

[VMC02] J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL*. O'Reilly, 1 edition, 2002.

[Wei03] E. Weisstein. RSA-576 Factored. http://mathworld.wolfram.com/news/ 2003-12-05/rsa/, 2003.

[Wei05] E. Weisstein. RSA-640 Factored. http://mathworld.wolfram.com/news/ 2005-11-08/rsa-640/, 2005.

[Wie90] M. J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36:553–558, 1990.

[YY96] A. Young and M. Yung. The Dark Side of Black-Box Cryptography or: Should We Trust Capstone? In *Advances in Cryptology – CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg, 1996.

[YY97] A. Young and M. Yung. Kleptography: Using Cryptography Against Cryptography. In *Advances in Cryptology – EUROCRYPT 1997*, volume 1233 of *Lecture Notes in Computer Science*, pages 62–74. Springer Berlin Heidelberg, 1997.

[YY04] A. Young and M. Yung. *Malicious Cryptography - Exposing Cryptovirologie*. Wiley, 2004.

[YY06] A. Young and M. Yung. A Space Efficient Backdoor in RSA and Its Applications. In *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 128–143. Springer Berlin Heidelberg, 2006.

[YY08] A. Young and M. Yung. A Timing-Resistant Elliptic Curve Backdoor in RSA. In *Information Security and Cryptology*, volume 4990 of *Lecture Notes in Computer Science*, pages 427–441. Springer Berlin Heidelberg, 2008.

[Zim95] P. Zimmermann. Building in Big Brother. chapter Pretty Good Privacy: Public Key Encryption for the Masses, pages 93–107. Springer-Verlag New York, Inc., 1995.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2012-01 Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie: Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-14 Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures

2013-16 Carsten Otto: Java Program Analysis by Symbolic Execution

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

2013-20 Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models

2014-01 * Fachgruppe Informatik: Annual Report 2014

2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide

2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata

2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic

2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video

2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations

2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014

2014-14 Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks

2014-15 Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code

2014-16 Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results

2015-01 * Fachgruppe Informatik: Annual Report 2015

2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications

2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.