# DÉVELOPPEMENT LOGICIELS CRYPTOGRAPHIQUES

**Project:** RSA Key Reconstruction

**Enseignant:** M. Christophe CLAVIER

**Réalisé par:**
BOUABDALLAOUI Adam
DERFOUFI Houssam
DUSHIME Neysa
**Promotion: M2 MATHS & INFO CRYPTIS**

# Contents

# List of Figures

## 1 Introduction to RSA

RSA is a widely used encryption algorithm that has successfully withstood over 30 years of cryptanalytic attacks and plays a crucial role in securing sensitive information transmitted over networks. It is named after its inventors, Ronald Rivest, Adi Shamir, and Leonard Adleman, who introduced the algorithm in 1977. Unlike symmetric key cryptography, where the same key is used for both encryption and decryption, RSA uses a pair of keys, a public key for encryption and a private key for decryption. This allows anyone to encrypt a message using the public key, but only the holder of the private key can decrypt it, ensuring confidentiality.

Public and private keys in RSA are generated as follows [4]:

1. **Key Generation:**
   - Choose two large, distinct, random prime numbers $p$ and $q$.
   - Compute $N = pq$, which will be used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.
   - Compute $\phi(N) = (p-1)(q-1)$, where $\phi$ is Euler's totient function.
   - Choose an integer $e \in (\mathbb{Z}/N\mathbb{Z})^*$ such that $1 \leq e \leq N-1$ and $\gcd(e, \phi(N)) = 1$.
   - Compute $d$, the modular inverse of $e \mod \phi(N)$: $ed \equiv 1 \pmod{\phi(N)}$.

   `PUBLIC KEY:` the pair $(N, e)$. `PRIVATE KEY:` the triplet $(p, q, d)$
2. **Encryption:**
   - The message $m$, assumed to be a number smaller than $N$, is encrypted using the public key $(N, e)$ by computing the ciphertext $c = m^e \mod N$.
3. **Decryption:**
   - The ciphertext $c$ is decrypted using the private key $(N, d)$ by computing the message $m = c^d \mod N$.

The security of RSA is based on the practical difficulty of factoring the product of two large prime numbers with current algorithms and computing power. This makes it exceedingly difficult to derive the private key $d$ from the public key $e$ and the modulus $N$.

The robustness and efficiency of RSA stem not only from its key generation and encryption/decryption processes but also from its ability to use optimizations such as the Chinese Remainder Theorem (CRT) for faster decryption. Moreover, RSA inherently includes certain redundancies and relationships between the components of the private key $(p, q, d, d_p, d_q, q_p^{-1})$, as described in the PKCS#1 standard, which can be leveraged for optimized computations and also form the basis for certain types of cryptographic attacks if not properly secured.

In what follows, we will use $\mathbf{pk} = (N, e)$ as an RSA public key with the corresponding secret key $\mathbf{sk} = (p, q, d, d_p, d_q, q_p^{-1})$. The last three components of $\mathbf{sk}$ are defined as $d_p = d \mod (p-1)$, $d_q = d \mod (q-1)$, and $q_p^{-1} = q^{-1} \mod p$.

## 2 RSA Private Key Reconstruction from Random Key Bits [3] (Heninger-Shacham Algorithm)

The Heninger-Shacham algorithm [3] focuses on the recovery of RSA secret keys from partial or incomplete data. This algorithm is significant in the field of cryptanalysis, especially in regards to correcting errors or erasures in RSA keys. It recovers RSA key parameters (such as $p, q, d, d_p, d_q$) using a bit-by-bit approach and by building a search tree. It is designed to operate in situations where some bits of the secret key are known or correct, thereby allowing the complete key to be reconstructed by exploiting the redundancy and mathematical relationships between different components of the RSA key.

The algorithm uses relationships such as:

$$\begin{aligned} N &= pq && (1)\\ ed &= 1 \mod \phi(N) = 1 + k\phi(N) = 1 + k(N - p - q + 1) && (2)\\ ed_p &= 1 \mod p - 1 = 1 + k_p(p - 1) && (3)\\ ed_q &= 1 \mod q - 1 = 1 + k_q(q - 1) && (4) \end{aligned}$$

For small values of $e$, we can compute each of these three variables $k$, $k_p$, and $k_q$ even with a heavily degraded version of $d$.

### 2.1 RSA Key Reconstruction Process

1. **Bit-by-Bit Reconstruction:** The algorithm operates iteratively, reconstructing the private key one bit at a time. For each bit index $i$, it considers all combinations of values for the $i$-th bit of $p$, $q$, $d$, $d_p$, and $d_q$.
   It starts with the least significant bits, exploiting the fact that $p$ and $q$ are large prime numbers and must therefore be odd. This knowledge immediately corrects certain bits of $d_p$, $d_q$, and $d$ based on the known bits of $p$ and $q$.
   A crucial aspect of the reconstruction is understanding how a change in the $i$-th bit of $p$ or $q$ affects the corresponding bits in $d_p$, $d_q$, and $d$. This understanding is used to correct other bits and to build the solution bit by bit.

   According to [3], let $p[i]$ be the $i$-th bit of $p$, where the least significant bit is bit 0, and similarly for the bits $q$, $d$, $d_p$, $d_q$. Let $\tau(x)$ be the exponent of the largest power of 2 that divides $x$.

   As $p$ and $q$ are large prime numbers, they are odd, which means the least significant bit (the 0-th bit) for $p$ and $q$ must be 1 ($p[0] = q[0] = 1$). As a result, $2|p-1$, so $2^{1+\tau(k_p)}|k_p(p-1)$. Thus, by reducing (3) modulo $2^{1+\tau(k_p)}$, we have:

   $$ed_p = 1 \mod 2^{1+\tau(k_p)}$$

   Since we know $e$, this allows us to immediately correct the least significant bits of $d_p$, that is, $1 + \tau(k_p)$.

Similar arguments using (4) and (2) allow us to correct respectively the $1 + \tau(k_p)$ and $2 + \tau(k)$ bits of $d_q$ and $d$.

2. **Enumeration and Pruning of Solutions:** For each bit index $i$, the algorithm generates all possible combinations of the $i$-th bit for $p$, $q$, $d$, $d_p$, and $d_q$ that are consistent with the previously reconstructed bits (from 1 to $i - 1$). It retains a candidate combination only if it satisfies the constraints established by the relations among the key components.
   The number of possible solutions is initially large, but the algorithm eliminates incorrect solutions using the constraints. This significantly reduces the number of candidates to consider.

3. **Projecting Solutions:** For each bit slice $i$, the algorithm must determine how to project a solution for slice $i - 1$ into a possible solution for slice $i$. While there might seem to be many possibilities for each new bit (up to $2^5 = 32$), the constraints among the key components significantly reduce the number of viable solutions. In most cases, the number of possible solutions is at most 2, and for a sufficiently large $\delta$, it is often less.

4. **Reaching the Solution:** The algorithm proceeds in this iterative manner until it has considered all bits up to $i = \frac{n}{2}$. At this point, the factorization of $N$ is revealed in one or more of the candidate solutions.

The efficiency of the algorithm depends on the fraction $\delta$ of the RSA private key bits known in advance. The higher $\delta$ is, the fewer possibilities need to be considered at each step, and the more quickly the algorithm converges to the correct solution. The Heninger-Shacham algorithm [3] is particularly effective when the known bits are uniformly randomly distributed across the key components.

This algorithm offers an efficient method for reconstructing RSA private keys from a portion of their bits, exploiting the redundancy of information in their standard storage and the intrinsic structure of RSA keys. This makes it particularly useful in scenarios where an RSA key is partially corrupted or partially revealed, as in the case of "cold boot" attacks [1].

A *cold boot* attack on RSA keys exploits the fact that a computer's volatile memory (RAM) does not immediately lose its data when the power is cut off, but instead, the data fades slowly. When RSA keys are stored in memory, they can potentially be recovered by this method if the attacker has quick physical access to the computer. Here's how the cold boot attack can be applied to RSA keys:

1. **RAM Capture:**

   - The computer is either physically restarted, or the power is cut off and then quickly restored.

- The attacker then launches a minimalistic program or operating system that does not overwrite the crucial parts of RAM where the RSA key might reside.
- In some cases, the attacker might cool the RAM (e.g., with compressed air) to slow down data degradation.

2. **RSA Key Search:**
   - After the restart, the attacker's program scans the memory for patterns matching a potential RSA key.
   - Since RSA keys are large (often 2048 bits or more) and have a specific structure, they can sometimes be distinguished from other data in memory.

3. **Reconstruction Challenges:**
   - The data recovered from RAM can be incomplete or partially corrupted due to data degradation after the shutdown.
   - The attacker might need to apply reconstruction techniques to recover the complete key. This is where research like that of Heninger and Shacham becomes relevant, where they propose methods for reconstructing private RSA keys even with a significant part of the bits missing or corrupted.

4. **Impact on RSA Security:**
   - If the attacker succeeds in recovering the RSA private key, they can decrypt all messages encrypted with the corresponding public key, sign messages, or carry out other actions for which the private key is necessary.
   - This completely compromises the security of the cryptographic system based on that RSA key.

## 2.2 Runtime Analysis

The expected runtime to reconstruct an $n$-bit RSA key is $O(n^2)$, with very high probability, given a fraction $\delta$ of the bits of $p$, $q$, $d$, $d_p$, and $d_q$.
It depends on the number of partial keys examined. The algorithm employs constraints to limit this number, generating and evaluating partial solutions for each bit slice.

The analysis examines how the constraints impact the branching behavior of the algorithm, characterizing the number of solutions generated at each step and providing a bound on the total number of branches examined.

Mathematical tools are utilized to provide an upper bound on the total number of keys examined during the entire operation of the algorithm, demonstrating its efficiency.

Different mathematical tools used:

1. **Multivariate Hensel's Lemma (Lemma 4.2 from document [3]):** This tool is used to characterize the conditions under which a solution to the constraint equations modulo a power of two can be "lifted" to a solution modulo a higher power of two. It aids in understanding how to generate the $i$-th bit of a partial solution based on bits 0 to $i - 1$.

2. **Chebyshev's Inequality:** This statistical tool is used to relate the probability that the sum of random variables deviates from its mean. In this context, it is used to estimate the probability that the total number of keys examined deviates from its expected value.

3. **Bounds on the Variance of a Sum of Random Variables:** A bound is applied to the variance of the sum of random variables (representing the number of partial solutions examined at each stage). The bound is based on the maximum of the individual variances of the random variables and is used in conjunction with Chebyshev's Inequality to estimate the probability of deviation from the expectation.

4. **Probability Generating Functions:** Generating functions are used to compute the expectation and variance of the number of incorrect solutions generated at each step of the algorithm. These functions encapsulate the distribution of the random variables and allow for the computation of moments like expectation and variance elegantly.

# 3 Block-wise Threshold-Based Vector Correction

## 3.1 Explanation of the Block-wise Threshold-Based Vector Correction Algorithm

The block-wise threshold-based vector correction algorithm [2] is an iterative method designed to reconstruct unknown tuples of bit vectors (denoted as $\mathbf{x}$), considering only a corrupted version of these vectors (denoted as $\tilde{\mathbf{x}}$), and public information about $\mathbf{x}$ that doesn't allow for direct extraction of $\mathbf{x}$.

## 3.2 Generic Description of the Algorithm

The algorithm proceeds through the following phases:

1. **Initialization Phase:** Utilize public information to compute an initial partial solution for the vector $x$. This phase is optional and may result in an empty

string as the sole partial solution.

2. **Expansion Phase:** Each partial solution is extended to include the next $t$ most significant bits for each component of $\mathbf{x}$ $(\mathbf{x}_1, \ldots, \mathbf{x}_m)$. This could theoretically generate up to $2^{mt}$ new partial solutions, but the goal is to significantly reduce this number using public information.

3. **Pruning Phase:** For each new partial solution from the expansion phase, the number of matches of the extended $mt$ bits with the corresponding bits of the corrupted version $\tilde{\mathbf{x}}$ is counted. If this number is less than a certain threshold parameter $C$, the partial solution is discarded.

4. **Finalization Phase:** The remaining partial solutions are examined using public information to check if any of them match the desired $\mathbf{x}$.

The efficiency of the algorithm heavily relies on the appropriate choice of the block size $t$. A $t$ that is too large exponentially increases the number of partial solutions generated, making the processing task challenging. Conversely, a $t$ that is too small may prevent the efficient pruning of incorrect partial solutions, thereby unnecessarily increasing the computation time and memory space required. The algorithm aims to remove enough incorrect solutions while retaining the correct partial solution with high probability, thereby ensuring iterative progression toward the correct solution without being overwhelmed by incorrect candidates.

## 3.3 Explanation of the Error Correction Algorithm for RSA Keys [2]

This section details the specification of the block-wise threshold-based vector correction method applied to error correction for RSA keys. The goal is to reconstruct an unknown RSA secret key $\mathbf{sk} = (p, q, d, d_p, d_q)$ from its corrupted version $\tilde{\mathbf{sk}} = (\tilde{p}, \tilde{q}, \tilde{d}, \tilde{d}_p, \tilde{d}_q)$, using the corresponding public key $(N, e)$. Here is a detailed overview of the steps in this algorithm:

**Inputs and Objective:**

- **Inputs**: RSA public key $(N, e)$ and corrupted version of RSA secret key $\tilde{\mathbf{sk}} = (\tilde{p}, \tilde{q}, \tilde{d}, \tilde{d}_p, \tilde{d}_q)$, with error rate $\delta$.

- **Objective**: Reconstruct the correct RSA secret key $\mathbf{sk} = (p, q, d, d_p, d_q)$.

**Initialization Phase:**

- Utilize public information to obtain an initial approximation or partial solution of $\mathbf{sk}$.

- This stage may start with an empty string or an approximation based on available information.

- $(k, k_p, k_q) \leftarrow \text{Init}(N, e)$

- $\text{Slice}(0) \leftarrow \text{Mount}(e, k, k_p, k_q)$

**Iterative Process:**
For $i = 1$ to $\lceil \frac{n/2-1}{t} \rceil$

- **Expansion Phase:** For each candidate $(p', q', d', d'_p, d'_q)$ with slices from 0 to $(i-1)t$ extend the candidate $t$ times with the procedure Expand(.) from Heninger-Shacham. This results in $2^t$ new candidates differing in the slices $(i-1)t+1, \cdots, it$.

- **Pruning Phase:** For each new candidate $(p', q', d', d'_p, d'_q)$, count the number of bits in the extended slices $(i-1)t+1, \cdots, it$ that agree with the corresponding bits of $\tilde{\mathbf{sk}}$. If this number is below a certain threshold $C$, discard the solution.

**Finalization Phase:** For each candidate **sk'** $= (p', q', d', d'_p, d'_q)$, check all RSA identities (1)–(4). If all equations are verified, output **sk'**.

**OUTPUT:** $\mathbf{sk} = (p, q, d, d_p, d_q)$

The algorithm proceeds by expanding, pruning, and verifying candidate solutions for each bit slice $t$ until the correct secret key is reconstructed or all solutions are eliminated.

The algorithm cleverly uses public information during the expansion phase to limit the number of new candidate solutions to $2^t$ for the $5t$ new bits, instead of the naive $2^{5t}$ candidates. This is made possible by the use of the Expand(.) procedure from Heninger and Shacham, which leverages public information to guide the expansion process.

Ultimately, the algorithm has an expected polynomial time complexity in $n$, provided the error rate $\delta$ is less than $0.237 - \epsilon$, where $\epsilon$ is a small positive number, and the threshold parameter $C$ is chosen to maximize the probability of the correct partial solution's survival at each pruning phase.

## 3.4 Parameter Choices and Analysis of Success / Runtime

This part of the document [2] provides a detailed analysis of the error correction algorithm for RSA keys and shows how this analysis can be generalized to other cases where the attacker has erroneous information about a subset of the RSA secret key parameters.

### 3.4.1 Comprehensive Analysis for the RSA Case

- **Modeling Bit Correspondence:** We define a random variable $X_c$ representing the number of corresponding bits between the erroneous version of the RSA secret key $\tilde{\mathbf{sk}}$ and a correct partial solution. The distribution of $X_c$ is modeled as a binomial distribution with parameters $5t$ (number of bits in a block) and the probability $1 - \delta$ (where $\delta$ is the error rate).

- **Heuristic for Incorrect Partial Solutions:** For incorrect partial solutions, the algorithm assumes that each expansion of an incorrect candidate results in an addition of $5t$ uniformly random bits (Heuristic 2 from document [2]), relying on a similar heuristic assumption made by Heninger and Shacham.

- **Choice of Threshold $C$:** The threshold $C$ is chosen to significantly separate the distributions of the random variables $X_c$ (for correct solutions) and $X_b$ (for incorrect solutions), in order to maximize the probability of pruning bad solutions while retaining the good solution.

- **Main Theorem:** The Main Theorem 3 from document [2] is presented, indicating that under certain conditions (notably a proper choice of $t$, $C$, and a bound on the error rate $\delta$), the error correction algorithm corrects the erroneous secret key in expected polynomial time and with a high probability of success. The theorem also shows that the error rate tolerable by the algorithm can be close to 0.237 for large values of $n$.

## 4 Comparative Analysis of the Two Methods

### 4.1 Objective and Approach

**Heninger-Shacham Algorithm (RSA Private Key Reconstruction from Random Key Bits [3]):**

- **Objective:** The goal of reconstructing RSA private keys from random key bits, as described in the Heninger-Shacham algorithm, is to demonstrate the possibility of recovering partially corrupted or incomplete RSA private keys. This can occur in situations where keys are damaged, partially erased, or when a portion of the bits is uncertain due to defects in the storage or transmission of keys.

  The Heninger-Shacham algorithm exploits the mathematical structure of RSA keys and uses factorization techniques to reconstruct the private key from a fraction of its bits. This has significant implications for computer security, as it shows that partial loss of key data does not necessarily lead to a total loss of security and that in some cases, these keys can be reconstructed and potentially exploited.

In the paper by Nadia Heninger and Hovav Shacham [3], their algorithm is designed to reconstruct RSA private keys from partial key bits, exploiting the redundancy in the typical storage format of an RSA private key. The algorithm is particularly relevant for "cold boot" attacks, where a secret key stored in a computer's memory can be partially recovered due to DRAM remanence. It demonstrates that even with only a fraction (about 27%) of the key bits randomly known, it is possible to reconstruct the RSA private key with remarkable efficiency.

- **Approach:** The algorithm [3] uses an iterative approach to reconstruct the RSA private key. It starts by identifying known bits of the key and then uses specific mathematical relationships inherent to RSA to progressively predict the missing bits. This method leverages the structure of the RSA key and its arithmetic properties to reduce the search space of possible solutions, making reconstruction practically feasible even with a significant number of missing bits.

**Block-wise Threshold-Based Vector Correction Algorithm:**

- **Objective:** The objective of the Block-wise Threshold-Based Vector Correction algorithm is to correct errors in erroneous RSA private keys using public information, such as the RSA public key, to aid in the reconstruction of the correct private key. This algorithm is specifically tailored to scenarios where RSA private keys have been corrupted, with a certain error rate in the key bits.

- **Approach:** The algorithm [2] uses a strategy of expansion, pruning, and verification by blocks of bits. The algorithm extends partial solutions to include the next most significant bits and uses a threshold parameter to prune unlikely solutions. The final phase checks if the candidate solution matches the correct RSA key. This method is suited for correcting errors in secret keys by relying on the RSA public key and aims to maximize the probability of reconstruction while minimizing the number of incorrect solutions considered.

## 4.2 Complexity and Efficiency

**Heninger-Shacham Algorithm:**

- **Complexity:** The complexity of the Heninger-Shacham algorithm for reconstructing RSA private keys from random key bits [3] is quadratic with respect to the key size ($O(n^2)$), provided that at least 0.27 fraction of the key bits are known. This implies that the time required to reconstruct the entire key grows proportionally to the square of the key's length.

  The runtime depends on the fraction $\delta$ of the key bits known. The algorithm is especially effective when the known bits are uniformly randomly distributed across the key components.

- **Efficiency:** The algorithm [3] is efficient in reconstructing the key when a sufficiently large number of bits are known, but its performance can decrease if the fraction $\delta$ of the known bits is low.

  The efficiency of the Heninger-Shacham algorithm is significant, enabling the reconstruction of RSA private keys with only a fraction (about 27%) of the key bits known. This efficiency makes the algorithm a powerful and practical method for recovering partially corrupted or erased RSA keys, highlighting the importance of protecting private keys against any partial exposure.

**Block-wise Threshold-Based Vector Correction Algorithm:**

- **Complexity:** The runtime is polynomial in $n$, but heavily depends on the parameter $t$ (block size) and hence $\epsilon$. The complexity can increase significantly even for small error terms $\epsilon$.

- **Efficiency:** The algorithm [2] is robust for correcting corrupted keys, provided that the error rate $\delta$ is below a certain threshold. For instance, for error rates below 0.237, the algorithm is capable of recovering the secret key in expected polynomial time, with a probability of success close to 1. However, in practice, it might be challenging to achieve error rates close to the theoretical limit due to the explosion of runtime for small error terms $\epsilon$.

## 4.3 Probability of Success

**Heninger-Shacham Algorithm [3]:**

- The Heninger-Shacham algorithm [3] successfully reconstructs an RSA key with a very high probability when at least 27% of the key bits are known. The algorithm's behavior exhibits a sharp "threshold boundary": below this 27% threshold, the number of keys examined increases exponentially, while above it, the increase remains close to linear. This threshold marks a critical point where the algorithm's efficiency changes drastically, demonstrating its strong dependence on the fraction of known key bits.

**Block-wise Threshold-Based Vector Correction Algorithm [2]:**

- The probability of success is also high, especially if the threshold parameter $C$ is correctly chosen to prune unlikely solutions while retaining the correct solution. The algorithm ensures that the correct solution survives each pruning phase with a probability close to 1.

## 4.4 Conclusion

The comparative analysis reveals that both the Heninger-Shacham algorithm [3] and the Block-wise Threshold-Based Vector Correction Algorithm [2] are effective for reconstructing RSA private keys. The former excels with a minimum of 27% of known bits, offering quadratic-time reconstruction. The latter is effective for error rates below 0.237 and employs a block-wise reconstruction strategy.

Both algorithms address slightly different scenarios of RSA key reconstruction or correction and offer sophisticated approaches to solving these problems. The Heninger-Shacham algorithm is more focused on reconstructing keys from known bits, while the Block-wise Threshold-Based Vector Correction Algorithm concentrates on correcting errors in corrupted keys. Each method has its own advantages and disadvantages in terms of complexity, efficiency, and probability of success, and the choice between them may depend on the specifics of the RSA key scenario and practical constraints.

## Advantages and Disadvantages of the Two Algorithms

**Heninger-Shacham:**

- **Advantages:**
  - Efficient with only 27% of known bits.
  - Robust approach to reconstruction with quadratic complexity.
- **Disadvantages:**
  - Requires a significant proportion of known bits.
  - Less effective for keys with a high error rate.

**Block-wise Threshold-Based Vector Correction Algorithm:**

- **Advantages:**
  - Can handle keys with error rates up to 0.237.
  - Utilizes a block-wise approach for methodical correction.
- **Disadvantages:**
  - Potentially more complex due to the block-wise approach.
  - Efficiency is dependent on the error rate and can be limited if the rate is too high.

## 5 Quick review on DRAMs

A foundational premise of [3] and [2] is that DRAMs retain memory states for a short period after power-off, gradually fading over time. This characteristic, due to the physical properties of DRAMs—which we'll explore in detail—enables a vulnerability to cold boot attacks. By rapidly cooling DRAMs with liquid nitrogen to approximately -50°C immediately after shutdown, it is possible to preserve the memory state long enough to extract its contents. While some data corruption occurs, it is generally minimal, allowing for the recovery and correction of cryptographic keys even with some level of degradation.

This section aims to succinctly explain the fundamentals of DRAMs and their operation at the physical layer. Our objective is to provide readers with a comprehensive understanding of how DRAMs work and why they are susceptible to cold boot attacks, offering a clear perspective on the execution of such attacks against this memory type.

## 5.1 Memory cell structure

1. **Transistor: The Switch**: Think of the transistor as a switch controlling the flow of electrical current. It has three main parts: the source, gate, and drain. When the gate is open (switch is on), it allows the capacitor to discharge if no current is entering from the source.

2. **Capacitor: The Storage Unit**: This is an electronic component capable of quickly storing and releasing charges. Within a DRAM, capacitors are incredibly small but essential for storing data bits through their charge state.

3. **Ground: The Common Reference Point**: Ground serves as a reference point in an electrical circuit. It's crucial for the operation of DRAMs.

   **The Memory Cell: The Fundamental Storage Unit** Combining the above components—transistor, capacitor, and connection to ground—forms a memory cell, the most basic storage unit in DRAM "see 1". It stores a single bit of data: '1' when the capacitor is charged and '0' when it's not.

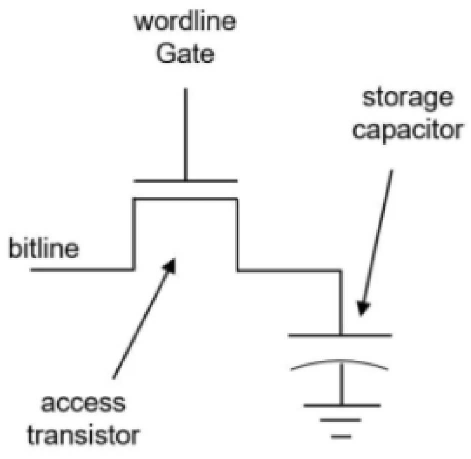

Figure 1: A memory cell.

## 5.2 word lines, bit lines, and memory arrays

1. **Bit Lines**: Bit lines are integral to the operation of memory cells in DRAM. Each bit line is precharged to half of a reference voltage, enabling the detection of a memory cell's state—whether it's holding a binary 1 (charged) or 0

(discharged). This detection is based on the voltage differential caused by the slight charging or discharging of the cell's capacitor. Essentially, the bit line can discern the stored bit by observing how the presence or absence of charge affects its voltage level.

2. **Word Lines**: Word lines play a crucial role in accessing memory cells. They activate a row of memory cells, making them ready for reading or writing. This activation is achieved by opening the transistor gates within the memory cells, which, in turn, allows the capacitors to influence the bit lines. Consequently, this interaction permits the sense amplifier to determine the state (charged or discharged) of each accessed memory cell.

3. **Sense Amplifier**: The sense amplifier operates closely with bit lines to accurately read the state of memory cells. By recharging the bit lines, the sense amplifier can identify whether a cell is charged or discharged, effectively reading the stored data. Additionally, it implements a crucial refresh mechanism to counteract the inevitable charge leakage from the capacitors. This leakage would otherwise lead to data loss over time. In DRAM systems, differential sense amplifiers are often used; they determine a cell's state by analyzing the voltage difference across the bit line, enabling precise data retrieval and maintenance.

4. **Memory Arrays**: At the heart of DRAM technology are memory arrays, where memory cells are systematically arranged in a two-dimensional grid. These cells are interconnected by word lines and bit lines, allowing for organized access and management of data. A typical DRAM memory array contains billions of cells, each connected to a network of hundreds of thousands of bit and word lines. Such an arrangement facilitates the efficient storage and retrieval of data, as illustrated in 2.

## 5.3 READ/WRITE mechanism

In the figure 3 , we delve into the intricacies of a memory cell architecture, capable of representing a total of 64 unique memory values. This capability is rooted in the design of the memory address bus, which consists of 6 lines. The reason for choosing 6 lines lies in their ability to encode up to $2^6 = 64$ distinct binary values, perfectly matching the number of memory values we aim to represent.

1. **Row Address Buffer**: This component comprises a set of latches connected directly to the memory address bus. Its primary function is to isolate a specific row for access. Given that each row is encoded with 8 bits, we require only 3 lines to select from $2^3 = 8$ possible rows. This efficient encoding scheme underscores the buffer's role in narrowing down the selection to a single row within the memory matrix.

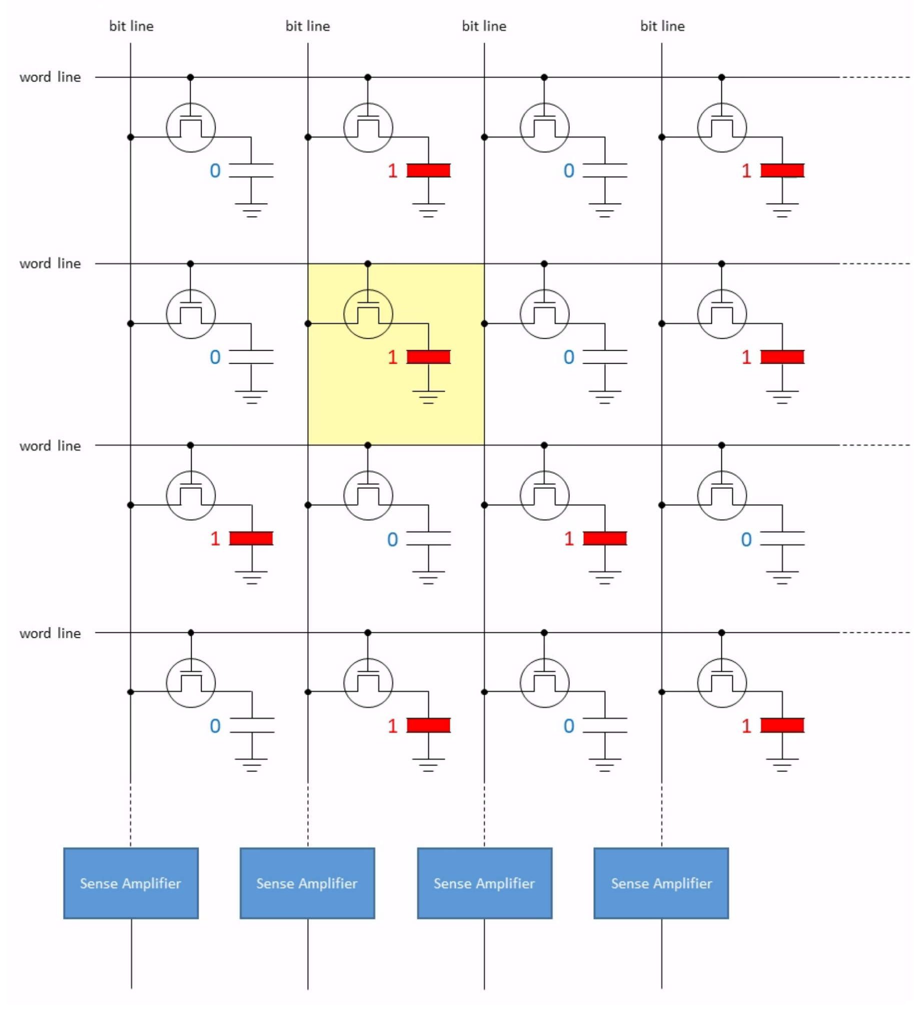

Figure 2: A memory array. source: [5]

2. **Column Address Buffer**: Mirroring the functionality of the Row Address Buffer, the Column Address Buffer also features a set of latches. However, its focus is on the memory array's columns. It facilitates the selection of a specific column, and similar to the row selection mechanism, 3 lines are sufficient here—owing to the 8-bit encoding of columns, which allows for $2^3 = 8$ distinct binary values and, consequently, column selections.

3. **Row Address Decoder**: This component is tasked with deciphering the specific row address as determined by the Row Address Buffer. It translates the binary value provided into a precise selection of one row out of the possible options, enabling direct access to the desired memory location.

4. **Column Address Decoder**: Parallel in function to the Row Address Decoder, the Column Address Decoder interprets the input from the Column Address Buffer. It selects the exact column based on the provided binary value, thus pinpointing the specific memory location within the selected row.

Together, these components orchestrate the selection process within the memory's 2-dimensional grid, allowing for precise and efficient access to any of the 64

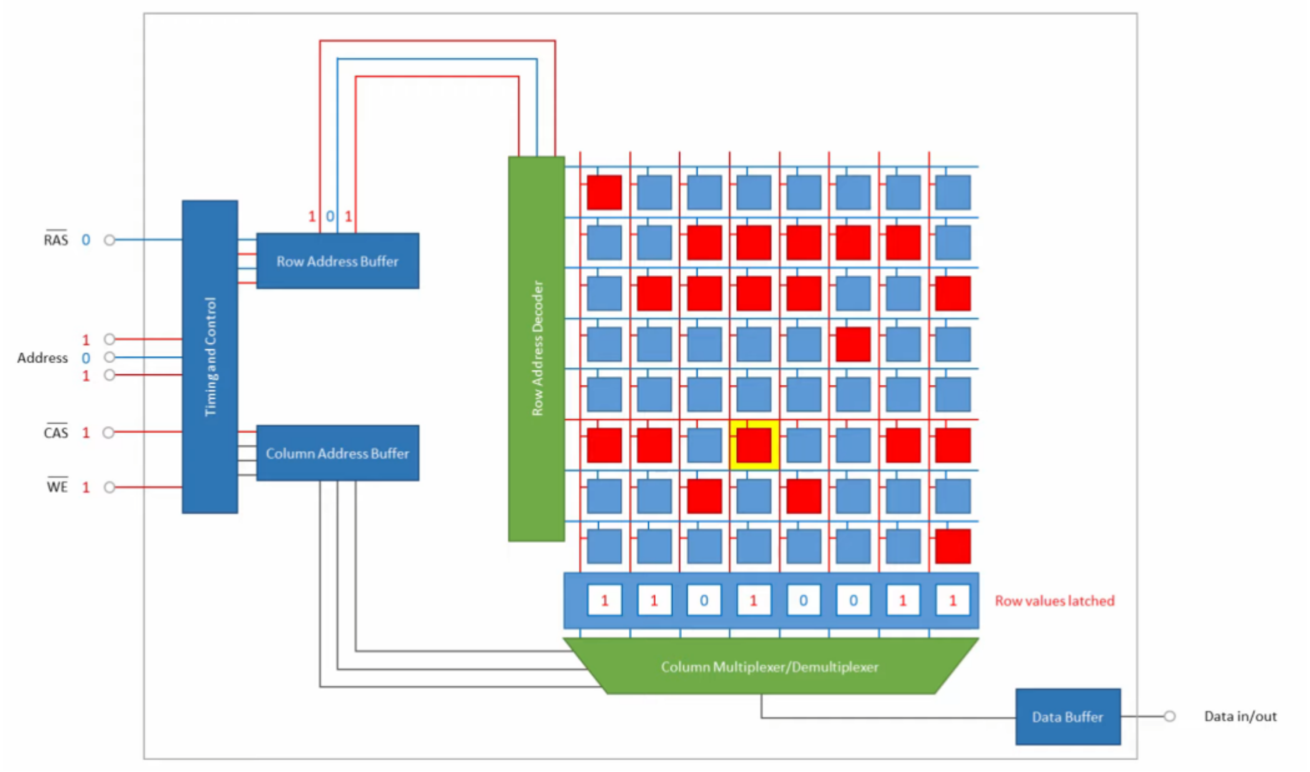

Figure 3: A memory array in action

memory values represented in the system.

## 6 Implementation of Heninger-Shacham [3] algorithm

Our project aimed to pragmatically implement the Heninger-Shacham algorithm, a method devised for RSA key reconstruction. The implementation was based solely on our understanding; some deviations may have occurred during coding, but the final goal of fully reconstructing the keys was achieved. Utilizing Ubuntu 22.04.3 LTS as our operating system, we developed our solution in C—a choice motivated by the language's efficiency and control over system resources. Given the algorithm's demand for handling large integers, we integrated the GNU Multiple Precision Arithmetic Library (GMP) to manage these computations effectively.

The core implementation of the Heninger-Shacham algorithm was encapsulated within approximately 350 lines of code. Together with auxiliary functions, the total codebase expanded to around 600 lines, illustrating the complexity and robustness of our solution. Additionally, we designed a supplementary program capable of generating RSA keys, complete with all necessary components $(N, e, d, p, q, d_p, d_q)$, to facilitate testing and demonstration.

To simulate real-world scenarios where keys might be partially compromised, we also developed a tool for key degradation. This utility introduces errors into the RSA keys at a user-defined rate, such as a 30% corruption rate, enabling us to evaluate the resilience and efficiency of the Heninger-Shacham algorithm under conditions of key impairment.

This section will delve into the technical specifics of our implementation process, challenges encountered, and the solutions devised to overcome them, providing a comprehensive overview of our implementation.

## 6.1 Generating/Degrading the RSA keys

We implemented a simple yet efficient program that generates for us correct RSA keys with all components $(N, e, d, p, q, d_p, d_q)$, the name of the file is *key_gen.c*, situated in the */keys* folder.

We also implemented a simple program that flips bits with a fixed probability which can be inputed by the user, the name of the file is *degrade.c*.

Both files can be compiled in the following way:

**$ gcc -o outputName filename.c -lgmp**

## 6.2 Reconstructing $k$

The component of our implementation tasked with identifying $k$, given a compromised version of $d$, is encapsulated within the file $Heninger\_part1.c$ situated in the folder named $RSA\_Key\_Reconstruction$. This program systematically enumerates all potential $d'$ values using the previously described method. It iterates through the bits of each $d'$, comparing them against the corresponding bits of $d_{corrupted}$. The variant $d'$ that exhibits the highest number of matching bits is determined to possess the correct $k'$ and is subsequently selected.

This approach was chosen for its fidelity to real-world attack scenarios, offering a practical simulation of how an adversary might attempt to reconstruct $k$ by exploiting the vulnerabilities inherent in a partially compromised $d$.

## 6.3 Correcting the Most Significant half bits of $d_{corrupted}$

When the right value of k is computed as described above, the same program file $Heninger\_part1.c$, corrects all the most significants bits of $d_{corrupted}$ by copying from the computed $d'$, and injecting them in $d_{corrupted}$, for me specifications please visit the file, the functions are well named and structured.

## 6.4 Computing $k_p$ *and* $k_q$

Computing $k_p$ and $k_q$ requires the use of some extra algorithms to calculate the square root in a setting of congruence, more specifically given the following quadratic congruence $r^2 \equiv n \mod p$, and more specifically in the case of p being an odd prime the equation have 2 solutions, that can be computed using Tonelli Shanks or Cipolla algorithm from which we chose to use the first [6], it was first described by Adi Shamir in 1980.

The logic is we have the following quadratic equation which we want to solve (ie: find the roots):

$k_p^2$ - [k(N - 1) + 1] $\cdot$k$_p$ - k $\equiv 0 \mod e$

The best way we found is transforming the qudratic setup in the following form:

$$\left[k_p - \left(\frac{k\cdot(N-1)+1}{2}\right)\right]^2 \equiv k + \left[\left(\frac{k\cdot(N-1)+1}{2}\right)\right]^2 \mod e$$

However, the following value:

[k $\cdot(N-1)+1$]

is an odd number and thus cannot be devided by 2, But since we are in a congruence setup and we already know that e is an odd prime, we can modify our equation to the following and it will be correct:

$$\left[k_p - \left(\frac{k\cdot(N-1)+1+e}{2}\right)\right]^2 \equiv k + \left(\frac{k\cdot(N-1)+1+e}{2}\right)^2 \mod e$$

at this point what is left to do is compute the value of k + $\left(\frac{k\cdot(N-1)+1+e}{2}\right)^2$

then pass it on to the tonneli shanks function along side $e$ as the modulo value, after we got our 2 results all what's left is to add the left side of the equation

$$\left(\frac{k\cdot(N-1)+1+e}{2}\right)$$

Which will yield the correct values of both $k_p$ and $k_q$, please note that in our context we use the tonelli shanks algorithm as a black box.

## 6.5 branche and prune

The algorithm proceeds in an iterative manner, progressively rebuilding the private key one bit after another. At each step for a given bit index $i$, it evaluates every possible value for the $i$-th bit of $p$, $q$, $d$, $d_p$, and $d_q$.

Initiating with the least significant bits, it leverages the knowledge that $p$ and $q$, being large prime numbers, are inherently odd. This insight allows for immediate adjustments to certain bits of $d_p$, $d_q$, and $d$ based on the already determined bits of $p$ and $q$.

A pivotal element in the key reconstruction process is the comprehension of the impact that alterations in the $i$-th bit of $p$ or $q$ have on the corresponding bits in $d_p$, $d_q$, and $d$. Utilizing this knowledge, the algorithm methodically corrects additional bits and incrementally constructs the solution, bit by bit.

Heninger and Shacham introduce the concept of bit slices for each bit index $i$, denoted as $Slice(i)$, which encapsulates the corresponding $i$-th bits of $p$, $q$, and the adjusted bits of $d$, $d_p$, and $d_q$ factoring in the offsets given by $\tau(k)$, $\tau(k_p)$, and $\tau(k_q)$ respectively:

$$Slice(i) := (p[i], q[i], d[i+\tau(k)], d_p[i+\tau(k_p)], d_q[i+\tau(k_q)]).$$

The initial bit slice, $Slice(0)$, is determined through the operation $Mount(e, k, k_p, k_q)$, which calculates the first bit slice following the previously outlined steps. This operation sets:

$$Slice(0) \leftarrow (1, 1, d[\tau(k)], d_p[\tau(k_p)], d_q[\tau(k_q)]),$$

where the computation of the last three components becomes straightforward once the values of $k$, $k_p$, and $k_q$ are established. The computational cost of $Mount(\cdot)$ is deemed negligible in scenarios involving RSA with small public exponents.

**Extending Solutions**: Given a partial solution $sk_0 = (p_0, q_0, d_0, dp_0, dq_0)$ up to $Slice(i-1)$, the goal is to advance the calculation to encompass all potential solutions $(p, q, d, d_p, d_q)$ for $Slice(i)$. Heninger and Shacham demonstrate that by applying a multivariate adaptation of Hensel's Lemma to the relevant equations, one can derive specific identities that facilitate this computation.

$$p[i] + q[i] = (N - p_0q_0)[i] \mod 2 \tag{5}$$
$$d[i + \tau(k)] + p[i] + q[i] = (k(N+1) + 1 - k(p_0 + q_0) - ed_0)[i + \tau(k)] \mod 2 \tag{6}$$
$$d_p[i + \tau(k_p)] + p[i] = (k_p(p_0 - 1) + 1 - ed_0^p)[i + \tau(k_p)] \mod 2 \tag{7}$$
$$d_q[i + \tau(k_q)] + q[i] = (k_q(q_0 - 1) + 1 - ed_0^q)[i + \tau(k_q)] \mod 2 \tag{8}$$

And pruned the result based on the information we know about the keys using a recursive process.

For some reasons, our implementation generates correctly all the bist up until it finishes $d_q$ and $d_p$, and the rest of bits in p, q, d don't get completed, but all the previous ones are correct, this pushed us to use the following equations:

q $= \frac{e \cdot d_q - 1}{k_q} + 1$

p $= \frac{e \cdot d_p - 1}{k_p} + 1$

We calculate the values of p and q in the above manner and check against N, if the multiplication is equal to the previous, the process halts and we have succsesfully recovered our keys.

We tested the algorithm on various sizes 512 bits and 1024 bits with errors ranging between 20% and 70%, the results we got were almost all under 2 seconds of computation sometimes going even under 1 second, this proves the efficiency of Heninger-Shacham algorithm.

## 7 Conclusion

The project, focused on the implementation of the Heninger-Shacham algorithm, presented a unique set of challenges. It required a detailed understanding of each step to ensure accuracy, as interpreting the authors' intentions accurately was crucial. Some concepts, when viewed from different perspectives, led to varied results, many of which initially missed the mark. Through a process of trial, error, and adaptation, we managed to refine our approach and align our outcomes more closely with expected results.

The effort was not only technical but also deeply engaging, pushing us to explore the algorithm's intricacies and potential improvements. Our codebase, as it stands, reflects our journey through these challenges. It's functional and meets the project's objectives, yet we see significant room for enhancements.

Looking ahead, there's a clear path for making the codebase more efficient and user-friendly. Future work could focus on optimizing the algorithm's performance, simplifying the interface for ease of use, and perhaps even expanding its functionality to cover related cryptographic challenges.

In wrapping up, this project has laid a solid foundation for further exploration. It offers an invitation to both our team and the broader community to build upon what we've achieved, improving and extending the application of the Heninger-Shacham algorithm. Our experience underscores the importance of perseverance and adaptability in research, qualities that we will carry forward into future projects.

## References

[1] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[2] W. Henecka, A. May, and A. Meurer. Correcting errors in rsa private keys. *Lecture Notes in Computer Science. Springer, Berlin, Heidelberg*, vol 6223., 2010.

[3] Nadia Heninger and Hovav Shacham. Reconstructing rsa private keys from random key bits. *Lecture Notes in Computer Science. Springer, Berlin, Heidelberg*, vol 5677, 2009.

[4] Malika More, Jean Mailfert, and Gisèle Provost. Cours de cryptographie à clé publique : La méthode rsa, 2012. `http://www.irem.univ-bpclermont.fr/IMG/pdf/3CoursRSA.pdf`.

[5] Computer Science. Dynamic random access memory (dram). part 1: Memory cell arrays. YouTube, 2020. YouTube video. Available online: `https://www.youtube.com/watch?v=x3jGqOrXXc8`.

[6] Wikipedia. Tonelli-shanks algorithm. Wikipedia, 1980. `https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm`.