Security Analysis of Cryptographic Primitives Used in Blockchain

DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Cryptology and Security

submitted by

Atosi Das
(Roll No. CrS2202)

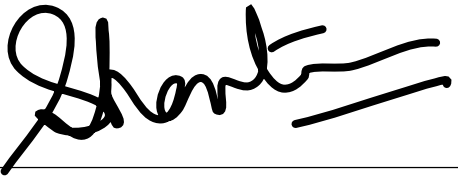
under guidance of
Sc. Shantanu, Scientist E, SAG Labs, DRDO
Prof. Goutam K. Paul, Associate Professor, CSRU, ISI Kolkata

Cryptology and Security Research Unit
Indian Statistical Institute, Kolkata

Certificate

This is to certify that the report entitled **"Security Analysis of Cryptographic Primitives Used in Blockchain"**, submitted by **Atosi Das** to the Indian Statistical Institute, Kolkata, for the award of the degree of Master of Technology in Cryptology and Security, is a record of the original, bona fide research work carried out by him under our supervision and guidance. The report has reached the standards fulfilling the requirements of the regulations related to the award of the degree.

The results contained in this report have not been submitted in part or in full to any other University or Institute for the award of any degree or diploma to the best of our knowledge.



Shantanu,
Scientist E,
SAG Labs, DRDO

Goutam Paul,
Cryptology and Security Research Unit(CSRU),
Indian Statistical Institute, Kolkata

Acknowledgments

I would like to express my profound gratitude to my advisor, Sc. Shantanu, Scientist E, SAG Labs, DRDO, for his patient guidance, enthusiastic encouragement, and useful critiques needed for this research work. He motivated me with great insight and innovative research ideas which helped me pursue good research and complete this dissertation. I would also like to thank my institute supervisor Dr. Goutam K. Paul, Associate Professor, CSRU, ISI Kolkata, for his constant support, guidance, and encouragement. It was a privilege for me to work under such great supervisors who consistently supported me both academically and personally.

Finally, I want to thank all my teachers, friends, and family members for their constant support and guidance.

Atosi Das
CrS2202
MTech Student
Cryptology and Security
Indian Statistical Institute, Kolkata

Abstract

This thesis conducts a comprehensive security analysis of cryptographic primitives within the context of blockchain technology, focusing on MD5, Elliptic Curve Cryptography (ECC), and digital signatures.

The first objective was to evaluate the security of MD5. The analysis revealed that MD5, once widely used, is vulnerable to collision attacks, leading to its deprecation in favor of more secure hash functions like SHA-256 for blockchain applications.

The second objective focused on the security and efficiency of ECC. While ECC demonstrates strong security features and efficiency, the study identified potential vulnerabilities related to curve selection and possible attacks. The importance of secure implementation practices to fully leverage ECC's benefits in blockchain was emphasized.

The third objective was to examine the role of digital signatures in blockchain. Digital signatures are crucial for ensuring transaction integrity and non-repudiation. The analysis highlighted challenges such as key management complexities and the necessity for robust algorithmic implementations to enhance security.

In conclusion, this thesis provides valuable insights into the security landscape of cryptographic primitives in blockchain. It underscores the need to implement robust cryptographic standards to mitigate identified vulnerabilities in MD5 and ECC, improve digital signature algorithms and key management practices, and continuously monitor and adapt to emerging threats and advancements in cryptographic technology for sustainable blockchain development.

Keywords: Hash functions, MD5, SHA256, Public-key cryptography, Elliptic curve, Digital Signature

Contents

Chapter 1

Introduction

1.1 Background of Blockchain Technology

Blockchain is a decentralized and distributed digital ledger that records transactions in a secure, transparent, and tamper-proof manner. It consists of a chain of blocks, where each block contains a list of transactions. Each transaction is grouped into a block, cryptographically linked to the previous one, forming an immutable chain. Key components of blockchain include distributed ledgers, consensus mechanisms like Proof of Work (PoW) and Proof of Stake (PoS), and cryptographic hash functions. Beyond cryptocurrencies, blockchain technology is being explored for use in supply chain management, healthcare, finance, and voting systems, highlighting its potential to revolutionize various industries by enhancing trust, efficiency, and transparency.

1.2 Importance of Cryptographic Primitives in Blockchain

Cryptographic primitives are the fundamental building blocks that ensure blockchain technology's security, integrity, and functionality. These primitives include hash functions, digital signatures, elliptic curve cryptography, etc.

1. **Hash Functions**

    (a) **Data Integrity and Immutability**: Hash functions generate

a fixed-size hash value from input data, ensuring that any alteration in the input will produce a completely different hash. In blockchain, each block contains the previous block's hash, creating a secure link that makes it computationally infeasible to alter the data without detection.

(b) **Efficient Data Verification**: Hash functions enable quick and efficient data verification, which is essential for the consensus mechanisms and the overall operation of blockchain networks.

2. **Digital Signatures**

   (a) **Authentication and Authorization**: Digital signatures, based on asymmetric cryptography, provide a way to authenticate the identity of participants and authorize transactions. In blockchain, users sign transactions with their private keys, which can be verified by others using the corresponding public keys.

   (b) **Non-Repudiation**: Digital signatures ensure that a transaction cannot be denied by the signer once it has been submitted and included in the blockchain, thus providing accountability.

3. **Public-Key Cryptography**

   (a) **Secure Transactions**: Public-key cryptography underlies the security of digital signatures and the confidentiality of transactions. It enables secure communication between parties in the blockchain network without the need for pre-shared secrets.

   (b) **Key Management**: Each user in a blockchain network has a pair of cryptographic keys (private and public). Public-key cryptography ensures that users can securely generate, store, and manage these keys, which are essential for signing transactions and accessing blockchain resources.

1.3 Structure of the Thesis

This thesis begins with an introduction to blockchain technology, highlighting its decentralized and transparent nature for securely recording transactions. It underscores the pivotal role of cryptographic primitives—hash functions,

digital signatures, and elliptic curve cryptography (ECC)—in ensuring data integrity, authentication, and non-repudiation within blockchain networks. The subsequent section delves into each primitive, detailing hash functions' function in generating unique data identifiers and facilitating efficient verification. Digital signatures are explored for their role in authenticating transactions and providing non-repudiation guarantees, crucial for maintaining operational integrity. ECC is examined for its efficient encryption capabilities and secure key exchange mechanisms, essential in protecting sensitive information and transactions.

A comprehensive security analysis follows, evaluating vulnerabilities and strengths of these primitives in blockchain contexts. Hash functions are scrutinized for susceptibility to collision attacks and scalability issues, while digital signatures face assessment regarding forgery risks and key management complexities. ECC is analyzed for curve selection vulnerabilities and efficiency trade-offs. The discussion section synthesizes findings, comparing strengths and weaknesses across primitives and addressing emerging challenges like quantum computing threats. The conclusion underscores the necessity of robust cryptographic standards to mitigate vulnerabilities, proposing future research directions for enhancing security and scalability in blockchain applications.

Chapter 2

Cryptographic Primitives in Blockchain

2.1 Hash Functions: Definition and Properties

2.1.1 Definition

A hash function is a mathematical algorithm that transforms an input (or 'message') into a fixed-size string of bytes, typically in the form of a hexadecimal number. The output, known as the hash value or digest, is unique to each unique input. Hash functions are fundamental components of many cryptographic protocols and play a crucial role in ensuring data integrity and security in blockchain technology.

2.1.2 Properties of Hash Functions

- **Deterministic**: A hash function must always produce the same output (hash value) for the same input. This consistency ensures that the hash value can reliably represent the input data.
- **Fast Computation**: The hash function should be able to quickly process an input and generate the hash value. Efficiency is important for practical applications, especially in blockchain networks where large volumes of data are processed.

- **Pre-image Resistance**: Given a hash value, it should be computationally infeasible to find the original input. This property ensures that even if an attacker knows the hash value, they cannot reverse-engineer it to discover the input data.
- **Small Changes in Input Produce Large Changes in Output (Avalanche Effect)**: A slight change in the input data should result in a significantly different hash value. This property is crucial for detecting any alterations in the input data, as even a minor change will produce a completely different hash.
- **Collision Resistance**: It should be computationally infeasible to find two different inputs that produce the same hash value. This property ensures the uniqueness of the hash value for each unique input, preventing hash collisions that could compromise data integrity.
- **Second Pre-image Resistance**: Given an input and its hash value, it should be computationally infeasible to find a different input that produces the same hash value. This property prevents an attacker from finding an alternate input that results in the same hash, ensuring data authenticity.
- **Fixed Output Length**: Regardless of the size of the input, the hash function should always produce a fixed-length output. This uniformity is essential for consistency in cryptographic applications and data storage.
- **Uniform Distribution**: The hash function should produce hash values that are uniformly distributed across the output space. This property minimizes the likelihood of collisions and ensures a balanced representation of input data.

2.1.3 Applications in Blockchain

- **Data Integrity**: Hash functions ensure the integrity of transactions and blocks in the blockchain. Each block contains the hash of the previous block, creating a secure and tamper-evident chain.

- **Proof of Work (PoW)**: In PoW consensus mechanisms, miners solve complex mathematical puzzles based on hash functions to validate transactions and create new blocks.
- **Digital Signatures**: Hash functions are used in conjunction with digital signatures to ensure the authenticity and integrity of transactions.

2.1.4 Examples

1. **SHA-256**: The SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function in blockchain technology, particularly in Bitcoin. It generates a 256-bit (32-byte) hash value from an input of any length. SHA-256 exemplifies the properties of a secure hash function, making it suitable for ensuring the integrity and security of blockchain data.
2. **MD5 Hash Function**: The MD5 (Message Digest Algorithm 5) hash function was once widely used for ensuring data integrity and producing unique hash values. However, MD5 is no longer considered secure for cryptographic purposes due to several vulnerabilities:
   - **Collision Vulnerability**: MD5 is prone to collision attacks, where two different inputs produce the same hash value. This vulnerability compromises the uniqueness and reliability of the hash value, making MD5 unsuitable for security-sensitive applications.
   - **Pre-image and Second Pre-image Attacks**: Advances in computational power and cryptographic analysis have shown that MD5 is vulnerable to pre-image and second pre-image attacks. These attacks undermine the hash function's ability to prevent reverse-engineering of the input data and ensure data authenticity.
   - **Weak Security for Modern Applications**: The 128-bit output size of MD5 is no longer sufficient to provide robust security in the face of modern computational capabilities. More secure hash functions like SHA-256, with a 256-bit output, offer better protection against brute-force attacks.

   Due to these limitations, MD5 is not used in blockchain technology, where security and data integrity are paramount. Instead, more secure

hash functions like SHA-256 are preferred to ensure the robustness and reliability of blockchain systems.

2.2 Digital Signatures: Definition and Properties

2.2.1 Definition

A digital signature is a cryptographic mechanism that provides a way to verify the authenticity and integrity of a digital message or document. It involves the use of asymmetric cryptography, where a pair of keys (a private key and a public key) is used to create and verify the signature. The digital signature serves as a digital equivalent of a handwritten signature or a stamped seal, but it offers far more inherent security.

2.2.2 Properties of Digital Signatures

- **Authentication**: Digital signatures verify the identity of the sender. When a document is signed with a sender's private key, the recipient can use the sender's public key to confirm that the signature was indeed created by the sender. This ensures that the message comes from a legitimate source.
- **Integrity**: Digital signatures ensure that the content of the message or document has not been altered since it was signed. Any modification to the signed data will invalidate the signature, thus alerting the recipient to the tampering.
- **Non-repudiation**: Once a sender has signed a document, they cannot deny having signed it later. The unique properties of the digital signature, tied to the sender's private key, provide undeniable proof of the sender's involvement.
- **Efficiency**: Creating and verifying digital signatures should be computationally efficient, enabling their use in real-time applications without significant delays.

- **Unforgeability**: It should be computationally infeasible for an attacker to forge a valid digital signature without access to the signer's private key. This ensures that only the legitimate owner of the private key can create a valid signature.

2.2.3 How Digital Signatures Work

1. **Key Generation**: The signer generates a pair of cryptographic keys – a private key and a public key. The private key is kept secret, while the public key is shared with anyone who needs to verify the signature.

2. **Signing**: To sign a message, the signer creates a hash of the message using a cryptographic hash function. The hash value is then encrypted with the signer's private key to create the digital signature.

3. **Verification**: The recipient of the signed message uses the signer's public key to decrypt the digital signature and obtain the hash value. The recipient also generates a hash of the received message. If the two hash values match, the signature is verified, confirming the authenticity and integrity of the message.

2.2.4 Applications in Blockchain

- **Transaction Verification**: In blockchain technology, digital signatures are used to verify the authenticity of transactions. Each transaction is signed by the sender's private key, ensuring that it was authorized by the owner of the cryptocurrency.

- **Smart Contracts**: Digital signatures are essential in executing and validating smart contracts. Parties to a smart contract sign the contract terms, ensuring that the agreement is authentic and has not been tampered with.

- **Consensus Mechanisms**: Some consensus mechanisms, like Proof of Stake (PoS), use digital signatures to validate the identities of validators and ensure that only legitimate nodes participate in the consensus process.

2.2.5 Example: ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA is a widely used digital signature algorithm based on elliptic curve cryptography. It offers high security with shorter key lengths compared to other algorithms like RSA, making it efficient and suitable for use in blockchain technology. ECDSA is used in Bitcoin and other cryptocurrencies to sign transactions and verify their authenticity.

2.3 Public Key Cryptography: Definition and Properties

2.3.1 Definition

Public key cryptography, also known as asymmetric cryptography, is a cryptographic system that uses pairs of keys: public keys, which can be disseminated widely, and private keys, which are known only to the owner. This method is used to secure data, provide digital signatures, and establish secure communication channels. The security of public key cryptography relies on the computational difficulty of certain mathematical problems.

2.3.2 Properties of Public Key Cryptography

- **Asymmetry**: Public key cryptography uses two different keys—a public key for encryption and a private key for decryption. This asymmetry allows secure communication even if the public key is widely known.
- **Public and Private Key Pair**: The public key can be shared openly and is used to encrypt data or verify digital signatures. The private key is kept secret and is used to decrypt data or create digital signatures. The keys are mathematically linked, but it is computationally infeasible to derive the private key from the public key.
- **Confidentiality**: Messages encrypted with the recipient's public key can only be decrypted by the recipient's private key. This ensures that only the intended recipient can read the message.

- **Authentication**: A message signed with a sender's private key can be verified by anyone using the sender's public key. This provides assurance that the message was indeed created by the sender and has not been tampered with.
- **Non-repudiation**: Public key cryptography provides non-repudiation, meaning that a sender cannot deny sending a message they signed with their private key. The digital signature uniquely identifies the sender.
- **Integrity**: Public key cryptography ensures the integrity of a message. Any alteration to the message will invalidate the digital signature, alerting the recipient to potential tampering.
- **Scalability**: Public key cryptography scales well for large networks, as public keys can be distributed freely and used to establish secure communications without the need for a shared secret key.
- **Key Management**: Managing keys in a public key infrastructure (PKI) involves creating, distributing, storing, and revoking keys. The infrastructure must be secure to ensure the authenticity and integrity of public keys.

2.3.3 How Public Key Cryptography Works

1. **Key Generation**: The user generates a pair of keys—a public key and a private key. The public key can be shared openly, while the private key is kept secure.
2. **Encryption and Decryption**: To send an encrypted message, the sender encrypts the message using the recipient's public key. The recipient then decrypts the message using their private key.
3. **Digital Signatures**: To sign a message, the sender creates a hash of the message and encrypts the hash with their private key. The recipient can verify the signature by decrypting the hash with the sender's public key and comparing it to a hash of the received message.

2.3.4 Applications in Blockchain

- **Transaction Security**: In blockchain technology, public key cryptography ensures that transactions are securely encrypted and can only be accessed by authorized parties.
- **Identity Verification**: Public key cryptography is used to verify the identity of participants in the blockchain network, ensuring that only legitimate participants can create and validate transactions.
- **Digital Signatures**: Public key cryptography underpins digital signatures used in blockchain transactions. Each transaction is signed with the sender's private key and verified with the sender's public key.
- **Smart Contracts**: Public key cryptography enables secure and verifiable execution of smart contracts. Parties to a contract can sign the contract with their private keys, ensuring authenticity and non-repudiation.

2.3.5 Example: RSA (Rivest-Shamir-Adleman) Algorithm

The RSA algorithm is one of the most widely used public key cryptosystems. It relies on the mathematical difficulty of factoring large composite numbers. RSA keys are typically 2048 bits or longer, providing strong security. RSA is used for secure data transmission, digital signatures, and key exchange.

2.3.6 Example: Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a form of public key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC offers equivalent security to traditional public key cryptosystems like RSA but with much smaller key sizes, resulting in faster computations and reduced storage requirements. This makes ECC particularly attractive for use in resource-constrained environments such as mobile devices and embedded systems.

How ECC Works

1. **Key Generation**: A user generates a private key (a random integer) and a public key (a point on the elliptic curve obtained by multiplying

the private key with a generator point on the curve).

2. **Encryption and Decryption**:
    - To encrypt a message, the sender uses the recipient's public key and an ephemeral private key to generate a shared secret, which is then used to encrypt the message.
    - The recipient uses their private key and the sender's ephemeral public key to generate the same shared secret and decrypt the message.
3. **Digital Signatures**:
    - To sign a message, the signer generates a signature using their private key and the message hash.
    - To verify the signature, the verifier uses the signer's public key and the message hash to ensure the signature is valid.

Properties of ECC

- **Smaller Key Sizes**: ECC achieves the same level of security as RSA with significantly smaller key sizes. For example, a 256-bit key in ECC provides comparable security to a 3072-bit key in RSA.
- **Efficiency**: Smaller key sizes lead to faster computations, reduced bandwidth usage, and lower storage requirements.
- **Security**: ECC is based on the elliptic curve discrete logarithm problem, which is currently considered hard to solve, providing strong security guarantees.

Applications in Blockchain

- **Cryptocurrency Wallets**: Many cryptocurrencies, including Bitcoin and Ethereum, use ECC (specifically, the ECDSA algorithm) for generating public-private key pairs, signing transactions, and verifying signatures.
- **Smart Contracts**: ECC is used in smart contract platforms to secure contract execution and ensure the integrity of transactions.

Chapter 3

Security Analysis of Digital Signatures: A Case Study on ECDSA

3.1 Introduction

Digital signatures are crucial for ensuring the authenticity and integrity of messages in blockchain technology. The Elliptic Curve Digital Signature Algorithm (ECDSA) is widely used for this purpose due to its efficiency and strong security properties. However, like any cryptographic algorithm, ECDSA is vulnerable to certain types of attacks if not implemented correctly. One such attack is the related nonce attack, which can compromise the security of ECDSA.

3.2 Related Nonce Attack on ECDSA

A nonce in ECDSA is a random value used during the signature generation process. The security of ECDSA relies on the randomness and uniqueness of this nonce. If nonces are reused or related in any way, an attacker can potentially recover the private key. The paper "A Novel Related Nonce Attack for ECDSA" discusses an advanced method to exploit related nonces, demonstrating the critical importance of proper nonce generation.

3.3 Implementation and Analysis of Related Nonce Attack on ECDSA

```sage
#!/usr/bin/env sage

from sage.all import GF, PolynomialRing
import hashlib
import ecdsa
import random

def separator():
    print("-" * 150)


####################
# global parameters #
####################

# choose any curve
usedcurve = ecdsa.curves.SECP256k1
# usedcurve = ecdsa.curves.NIST521p
# usedcurve = ecdsa.curves.BRAINPOOLP160r1

print("Selected curve :")
print(usedcurve.name)
separator()

# the private key that will be guessed
g = usedcurve.generator
d = random.randint(1, usedcurve.order - 1)
print("TYPES: ", type(g), type(d))

pubkey = ecdsa.ecdsa.Public_key( g, g * d )
privkey = ecdsa.ecdsa.Private_key( pubkey, d )
print("Private key :")
print(d)
separator()

# N = the number of signatures to use, N >= 4
# the degree of the recurrence relation is N-3
# the number of unknown coefficients in the recurrence equation is N-2
# the degree of the final polynomial in d is 1 + Sum_(i=1)^(i=N-3)i

N = 7
assert N >= 4

#########################################################
# nonces and signature generation with recurrence relation #
#########################################################

# first, we randomly generate the coefficients of the recurrence relation
a = []
for i in range(N-2):
    a.append(random.randint(1, usedcurve.order - 1))

# then, we generate the N nonces
k = []
# the first one is random
k.append(random.randint(1, usedcurve.order - 1))
# the other ones are computed with the recurrence equation
```

```
# the other ones are computed with the recurrence equation
for i in range(N-1):
    new_k = 0
    for j in range(N-2):
        new_k += a[j]*(k[i]**j) % usedcurve.order
    k.append(new_k)

# sanity check to see if we generated the parameters correctly
# print(k[1] % usedcurve.n)
# print((a[1]*k[0] + a[0]) % usedcurve.n)
# assert k[1] == ((a[1]*k[0] + a[0]) % usedcurve.n)

# then, we generate the signatures using the nonces
h = []
sgns = []
for i in range(N):
    digest_fnc = hashlib.new("sha256")
    digest_fnc.update(b"recurrence test ")
    digest_fnc.update(i.to_bytes(1, 'big'))
    h.append(digest_fnc.digest())
    # get hash values as integers and comply with ECDSA
    # strangely, it seems that the ecdsa module does not take the leftmost bits of hash
    # if hash size is bigger than curve... perahps is because i use low level functions
    if usedcurve.order.bit_length() < 256:
        h[i] = (int.from_bytes(h[i], "big") >> (256 - usedcurve.order.bit_length())) % usedcurve.order
    else:
        h[i] = int.from_bytes(h[i], "big") % usedcurve.order
    sgns.append(privkey.sign( h[i], k[i] ))


# get signature parameters as arrays
s_inv = []
s = []
r = []
for i in range(N):
    s.append(sgns[i].s)
    r.append(sgns[i].r)
    s_inv.append(ecdsa.numbertheory.inverse_mod(s[i], usedcurve.order))


#######################################
# generating the private-key polynomial #
#######################################

# declaring stuff for manipulating polynomials with SAGE
Z = GF(usedcurve.order)
R = PolynomialRing(Z, names=('dd',))
(dd,) = R._first_ngens(1)

# the polynomial we construct will have degree 1 + Sum_(i=1)^(i=N-3)i in dd
# our task here is to compute this polynomial in a constructive way starting from the N signatures in the given list order
# the generic formula will be given in terms of differences of nonces, i.e. k_ij = k_i - k_j where i and j are the signature indexes
# each k_ij is a first-degree polynomial in dd
# this function has the goal of returning it given i and j
```

```python
def k_ij_poly(i, j):
    hi = Z(h[i])
    hj = Z(h[j])
    s_invi = Z(s_inv[i])
    s_invj = Z(s_inv[j])
    ri = Z(r[i])
    rj = Z(r[j])
    poly = dd*(ri*s_invi - rj*s_invj) + hi*s_invi - hj*s_invj
    return poly
# the idea is to compute the polynomial recursively from the given degree down to 0
# the algorithm is as follows:
# for 4 signatures the second degree polynomial is:
# k_12*k_12 - k_23*k_01
# so we can compute its coefficients.
# the polynomial for N signatures has degree 1 + Sum_(i=1)^(i=N-3)i and can be derived from the one for N-1 signatures
# let's define dpoly(i, j) recursively as the dpoly of degree i starting with index j
def dpoly(n, i, j):
    if i == 0:
        return (k_ij_poly(j+1, j+2))*(k_ij_poly(j+1, j+2)) - (k_ij_poly(j+2, j+3))*(k_ij_poly(j+0, j+1))
    else:
        left = dpoly(n, i-1, j)
        for m in range(1,i+2):
            left = left*(k_ij_poly(j+m, j+i+2))
        right = dpoly(n, i-1, j+1)
        for m in range(1,i+2):
            right = right*(k_ij_poly(j, j+m))
        return (left - right)
def print_dpoly(n, i, j):
    if i == 0:
        print('(k', j+1, j+2, '*k', j+1, j+2, '-k', j+2, j+3, '*k', j+0, j+1, ')', sep='', end='')
    else:
        print('(', sep='', end='')
        print_dpoly(n, i-1, j)
        for m in range(1,i+2):
            print('*k', j+m, j+i+2, sep='', end='')
        print('-', sep='', end='')
        print_dpoly(n, i-1, j+1)
        for m in range(1,i+2):
            print('*k', j, j+m, sep='', end='')
        print(')', sep='', end='')
print("Nonces difference equation :")
print_dpoly(N-4, N-4, 0)
print(' = 0', sep='', end='')
print()
separator()
poly_target = dpoly(N-4, N-4, 0)
print("Polynomial in d :")
print(poly_target)
separator()

d_guesses = poly_target.roots()
print("Roots of the polynomial :")
print(d_guesses)
separator()
# check if the private key is among the roots
for i in d_guesses:
    if i[0] == d:
        print("key found!!!")
```

3.4 Analysis

The above implementation demonstrates how related nonces can be exploited to recover the private key in ECDSA. This highlights the critical importance of generating secure, random, and unique nonces for each signature. The

related nonce attack can compromise the entire security of a cryptographic system if nonces are not properly managed.

Chapter 4

Security Analysis of Public-Key Cryptography

4.1 Security Analysis of Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is widely used in blockchain and other cryptographic applications due to its efficiency and strong security properties. However, like any cryptographic system, ECC is susceptible to various attacks if not implemented or used correctly.

4.2 Attacks on Elliptic Curve Cryptography (ECC)

4.2.1 Baby-step Giant-step Method

The Baby-step Giant-step method is an algorithm used to solve the discrete logarithm problem efficiently. In ECC, this problem involves finding k given $Q = kP$, where P is a point on the elliptic curve and Q is the result of multiplying P by k.

The Baby-step Giant-step method is a classic algorithm used to solve the discrete logarithm problem efficiently. In the context of ECC:

- **Description**: The algorithm divides the problem into two phases:

  - **Baby steps**: Precompute a table of m points $P, 2P, 3P, \ldots, (m-1)P$.
  - **Giant steps**: For each i, compute iP and check if $iP = Q$ for some i.

- **Complexity**: The time complexity of the Baby-step Giant-step method is $O(\sqrt{n})$, where n is the order of the curve's group. This makes it feasible for moderate-sized groups but impractical for large groups like those used in modern ECC implementations.

- **Security Implications**:
  - ECC curves with large prime orders make the Baby-step Giant-step method computationally impractical due to the high $\sqrt{n}$ complexity.
  - Effective mitigation involves choosing ECC parameters (curve and base point) with large prime orders, ensuring the discrete logarithm problem remains difficult to solve using such methods.

Implementation of Baby-step Giant-step Method

Python code snippet for implementing the Baby-step Giant-step method.

```python
from math import isqrt
from collections import defaultdict

# Define the elliptic curve and point addition, multiplication functions

class EllipticCurve:
    def __init__(self, a, b, p):
        self.a = a
        self.b = b
        self.p = p

    def add(self, P, Q):
        if P == (None, None):
            return Q
        if Q == (None, None):
            return P
        if P == Q:
            return self.double(P)
        if P[0] == Q[0] and (P[1] + Q[1]) % self.p == 0:
            return (None, None)

        lam = (Q[1] - P[1]) * pow(Q[0] - P[0], -1, self.p) % self.p
        x = (lam**2 - P[0] - Q[0]) % self.p
        y = (lam * (P[0] - x) - P[1]) % self.p
        return (x, y)

    def double(self, P):
        if P == (None, None):
            return (None, None)

        lam = (3 * P[0]**2 + self.a) * pow(2 * P[1], -1, self.p) % self.p
        x = (lam**2 - 2 * P[0]) % self.p
        y = (lam * (P[0] - x) - P[1]) % self.p
        return (x, y)

    def multiply(self, P, n):
        result = (None, None)
        addend = P

        while n:
            if n & 1:
                result = self.add(result, addend)
            addend = self.double(addend)
            n >>= 1

        return result

def baby_step_giant_step(curve, P, Q, N):
    m = isqrt(N) + 1

    # Baby steps: compute and store iP for 0 <= i < m
    baby_steps = {}
    current = (None, None)
    for i in range(m):
        baby_steps[current] = i
        current = curve.add(current, P)
```

```python

    # Compute mP
    mP = curve.multiply(P, m)

    # Giant steps: find j such that Q - jmP is in the baby steps list
    giant_step = Q
    for j in range(m):
        if giant_step in baby_steps:
            i = baby_steps[giant_step]
            k = i + j * m
            return k % N
        giant_step = curve.add(giant_step, (mP[0], -mP[1] % curve.p))

    return None  # No solution found

# Example usage
a = 2
b = 3
p = 97
curve = EllipticCurve(a, b, p)

P = (3, 6)
Q = (80, 10)
N = 5  # Prime order of P

k = baby_step_giant_step(curve, P, Q, N)
print(f"k = {k}")
```

4.2.2 Pollard's Rho Algorithm

Pollard's Rho algorithm is another method for solving the discrete logarithm problem, particularly effective when the prime factors of the curve's order are small.

- **Description**: The algorithm uses a random walk in the group of points on the curve:
  - Randomly generate points and iteratively apply a function until a collision is detected.
  - Use the detected collision to infer properties about the discrete logarithm.
- **Complexity**: Pollard's Rho algorithm has a time complexity of $O(\sqrt{n})$, similar to the Baby-step Giant-step method. It is effective when factors of n are small but becomes less efficient as n increases.
- **Security Implications**:
  - Similar to the Baby-step Giant-step method, the security of ECC against Pollard's Rho depends on the size of the group's order n.
  - Choosing curves with large prime orders and regularly updating ECC parameters mitigate the risk of attacks using Pollard's Rho.

Implementation of Pollard's Rho Algorithm

Python code snippet for implementing Pollard's Rho algorithm.

```python
import random
from math import gcd

# Define the elliptic curve and point addition, multiplication functions

class EllipticCurve:
    def __init__(self, a, b, p):
        self.a = a
        self.b = b
        self.p = p

    def add(self, P, Q):
        if P == (None, None):
            return Q
        if Q == (None, None):
            return P
        if P == Q:
            return self.double(P)
        if P[0] == Q[0] and (P[1] + Q[1]) % self.p == 0:
            return (None, None)

        lam = (Q[1] - P[1]) * pow(Q[0] - P[0], -1, self.p) % self.p
        x = (lam**2 - P[0] - Q[0]) % self.p
        y = (lam * (P[0] - x) - P[1]) % self.p
        return (x, y)

    def double(self, P):
        if P == (None, None):
            return (None, None)

        lam = (3 * P[0]**2 + self.a) * pow(2 * P[1], -1, self.p) % self.p
        x = (lam**2 - 2 * P[0]) % self.p
        y = (lam * (P[0] - x) - P[1]) % self.p
        return (x, y)

    def multiply(self, P, n):
        result = (None, None)
        addend = P

        while n:
            if n & 1:
                result = self.add(result, addend)
            addend = self.double(addend)
            n >>= 1

        return result

# Function for Pollard-Rho algorithm

def pollard_rho(curve, P, Q, N):
    def f(X, a, b):
        x, _ = X
        if x % 3 == 0:
            return (curve.add(X, P), (a + 1) % N, b)
        elif x % 3 == 1:
            return (curve.add(X, X), (2 * a) % N, (2 * b) % N)
```

```
            else:
                return (curve.add(X, Q), a, (b + 1) % N)

    x = (None, None)
    a = b = 0
    X = (None, None)
    A = B = 0

    for _ in range(N):
        x, a, b = f(x, a, b)
        X, A, B = f(*f(X, A, B))

        if x == X:
            r = (a - A) % N
            if r == 0:
                continue

            k = (b - B) * pow(r, -1, N) % N
            if (curve.multiply(P, k) == Q):
                return k

    return None

# Example usage
a = 2
b = 3
p = 97
curve = EllipticCurve(a, b, p)

P = (3, 6)
Q = (80, 10)
N = 5  # Prime order of P

k = pollard_rho(curve, P, Q, N)
print(f"k = {k}")
```

4.3 Analysis

These implementations demonstrate the application of the Baby-step Giant-step method and Pollard's Rho algorithm for attacking the discrete logarithm problem in Elliptic Curve Cryptography. Understanding these algorithms and their potential vulnerabilities helps in designing ECC systems resilient against such attacks.

Chapter 5

Security Analysis of Hash Functions

5.1 Security Analysis of Hash Functions, with Focus on MD5

5.1.1 MD5 Hash Function

MD5 (Message Digest Algorithm 5) is a widely used cryptographic hash function designed by Ronald Rivest in 1991. Over time, vulnerabilities have been discovered that compromise its security for cryptographic applications.

- **Description**: MD5 produces a 128-bit hash value from an input message of arbitrary length. It operates by repeatedly mixing data in 512-bit blocks and performing a series of modular additions and rotations.
- **Vulnerabilities**:
  - **Collision Vulnerability**: MD5 is vulnerable to collision attacks, where two different inputs can produce the same hash value.
  - **Preimage Vulnerability**: It is also susceptible to preimage attacks, where an attacker can find an input that hashes to a specific output.
  - **Speed**: MD5 hashes can be computed very quickly, making it susceptible to brute-force attacks.

- **Security Concerns**: Due to these vulnerabilities, MD5 is no longer considered secure for cryptographic applications such as digital signatures or certificates.

5.1.2 Wang's Attack on MD5

Wang Xiaoyun and Hongbo Yu discovered significant weaknesses in MD5, leading to their attack in 2005.

- **Description**: Wang's attack exploits the structure and vulnerabilities of MD5 to find collisions efficiently.
- **Implementation**: Let's implement Wang's attack to demonstrate how collisions can be generated in MD5.

5.1.3 Analysis

MD5, once widely used, is now deprecated due to its vulnerabilities, including susceptibility to collision attacks demonstrated by Wang's attack. Understanding these weaknesses is crucial for choosing secure hash functions in cryptographic applications.

Chapter 6

Discussion

6.1 Summary of Key Findings

This chapter summarizes the key findings related to MD5, ECC (Elliptic Curve Cryptography), and digital signatures within the context of blockchain security. Each cryptographic primitive is analyzed for vulnerabilities, strengths, and implications for blockchain technology.

6.1.1 MD5

MD5, once widely used for hashing, is now deprecated due to vulnerabilities such as collision attacks demonstrated by Wang's attack. Its impact on blockchain security highlights the need for stronger cryptographic hash functions like SHA-256.

6.1.2 Elliptic Curve Cryptography (ECC)

ECC offers strong security with smaller key sizes, making it ideal for resource-constrained environments in blockchain. However, concerns about curve selection vulnerabilities necessitate careful implementation and ongoing research into secure elliptic curves.

6.1.3 Digital Signatures

Digital signatures ensure transaction authenticity and non-repudiation in blockchain. Analysis of digital signature algorithms like ECDSA reveals their

robustness but underscores the importance of secure key management practices.

6.2 Implications for Blockchain Security

The implications of MD5 vulnerabilities, ECC strengths and concerns, and digital signature functionality impact the overall security posture of blockchain systems. Recommendations include phasing out MD5, enhancing ECC implementations, and improving digital signature key management to bolster blockchain security.

6.3 Recommendations for Future Research

Future research should focus on addressing the identified limitations and advancing the state-of-the-art in cryptographic primitives for blockchain security. Areas for exploration include post-quantum hash functions, secure elliptic curve design, and advancements in digital signature algorithms and key management practices.

Chapter 7

Conclusion

7.1 Restate the Objectives

This thesis aimed to conduct a thorough security analysis of cryptographic primitives, focusing on MD5, ECC (Elliptic Curve Cryptography), and digital signatures within the context of blockchain technology.

7.2 Summarize the Main Findings

The findings from this thesis highlight significant insights into the security of cryptographic primitives in blockchain:

7.2.1 MD5

MD5, once widely used, was found to be vulnerable to collision attacks, necessitating its deprecation in favor of more secure hash functions like SHA-256 in blockchain applications.

7.2.2 Elliptic Curve Cryptography (ECC)

ECC demonstrated strengths in security and efficiency but raised concerns about curve selection vulnerabilities and potential attacks. Secure implementation practices are essential for leveraging ECC's benefits in blockchain.

7.2.3 Digital Signatures

Digital signatures play a crucial role in ensuring transaction integrity and non-repudiation in blockchain. Challenges identified include key management complexities and the need for robust algorithmic implementations.

7.3 Concluding Remarks

In conclusion, this thesis provides valuable insights into the security landscape of cryptographic primitives in blockchain. Moving forward, it is imperative to: - Implement robust cryptographic standards to mitigate vulnerabilities identified in MD5 and ECC. - Enhance digital signature algorithms and key management practices to strengthen blockchain security. - Continuously monitor and adapt to emerging threats and advancements in cryptographic technology for sustainable blockchain development.

Bibliography

Appendix A

Appendix: Supplementary Material

1. Cryptographic primitives in blockchains
2. A Novel Related Nonce Attack for ECDSA
3. A Study of General Attacks on Elliptic Curve Discrete Logarithm Problem over Prime Field and Binary Field
4. A Meaningful MD5 Hash Collision Attack