Post-Quantum Digital Signatures via Non-Interactive Zero-Knowledge-Proofs of k-Colorable Graphs

Wimar Widiarto - 13525009
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: wimarwidiarto123@gmail.com , 13525009@std.stei.itb.ac.id

***Abstract*—**The emergence of fault-tolerant quantum computing poses an existential threat to contemporary public-key infrastructure, as algorithms such as RSA and ECC rely on mathematical problems that are solvable in polynomial time via Shor's Algorithm. To address this vulnerability, this paper explores the development of digital signatures based on the NP-complete *k*-colorable graph problem, which is mathematically guaranteed to resist quantum-computational attacks. The proposed scheme implements a Non-Interactive Zero-Knowledge Proof to facilitate secure, standalone digital signatures without requiring interactive communication between the prover and verifier. By integrating the Fiat-Shamir Transformation, the protocol converts an interactive proof system into a static signature, effectively binding the proof to the message. The implementation demonstrates that iterative verification rounds can reduce the probability of successful forgery to an infinitesimally small value. Experimental testing confirms that the protocol maintains high integrity, successfully verifying authentic signatures while accurately rejecting modified or unauthorized inputs

Keywords—quantum, cryptography, zero-knowledge-proofs, k-colorable graphs

I. INTRODUCTION

The landscape of modern cybersecurity is built on the assumption that certain mathematical problems are computationally intractable. Currently, most widely used digital signature algorithms, such as RSA and Elliptic Curve Cryptography (ECC), rely on the foundational difficulty of integer factorization and the discrete logarithm problem. While these problems are classically secure, the rapid advancement of quantum computing poses an existential threat to these protocols. The development of fault-tolerant quantum hardware threatens to transform the digital security landscape by enabling the execution of Shor's Algorithm, which allows a quantum computer to solve these specific number-theoretic problems in polynomial time. Once a quantum computer achieves a sufficient number of logical qubits, the mathematical "trapdoor" that secures our current public-key infrastructure will effectively vanish, enabling attackers to forge digital signatures and compromise data integrity.

To mitigate this looming threat, the cryptographic community is actively exploring Post-Quantum Cryptography, with a particular focus on problems classified as NP-Complete. This paper aims to use *k*-colorable graph problems as an alternative to to the integer factorization problem since the *k*-colorable graph problem ($k \geq 3$) is considered NP-Complete which means it is mathematically guaranteed to require super polynomial (effectively exponential) time to be solved in the worst case. This means that it is resistant to attempts to solve it even with Quantum Computers.

II. THEORETICAL BACKGROUND

A. Digital Signature

Digital signatures are a fundamental pillar of modern cybersecurity, serving as the functional equivalent of handwritten signatures or wax seals in the physical world. While a physical signature can be easily forged or altered, a digital signature utilizes complex cryptographic algorithms to link an identity to a specific piece of data. Its primary function is to provide three core security guarantees: authenticity, which proves the message originated from the claimed sender; integrity, which ensures the data has not been modified in transit; and non-repudiation, which prevents the signer from later denying the validity of the message. These guarantees are essential for secure digital communication, ranging from financial transactions and software updates to legally binding contracts [1].

The process of generating a digital signature begins with the signer using their private key, which must be kept strictly confidential. First, the original message is passed through a cryptographic hash function, such as SHA-256. This function creates a unique, fixed-length string of characters known as a "message digest," which acts as a digital fingerprint of the data. Even the slightest alteration to the original document would result in a completely different hash value. The signer then encrypts this digest using their private key. The resulting encrypted data is the digital signature, which is then attached to the message and sent to the recipient.

To verify the signature, the receiver performs a two-part mathematical test. First, they use the sender's public key which is openly available to decrypt the attached signature, thereby revealing the original message digest. Simultaneously, the receiver takes the received message and passes it through the exact same hash function used by the sender to generate their own version of the digest. By comparing these two values, the receiver can determine the status of the data: if the hashes match perfectly, the message is confirmed to be authentic and untampered. If the hashes do not match, it serves as a cryptographic warning that the data was either modified or that the sender's identity is not legitimate.

Various algorithms provide the mathematical foundation for these signatures, with RSA and ECDSA being the most prominent. RSA relies on the difficulty of factoring large prime numbers, a method that has stood the test of time for its reliability. In contrast, the Elliptic Curve Digital Signature Algorithm (ECDSA) leverages the algebraic structure of elliptic curves. Because ECDSA can achieve the same level of security as RSA but with significantly smaller keys, it has become the preferred standard for modern applications where computational efficiency and memory are critical, such as in mobile devices, IoT hardware, and blockchain technology.

B. Non-Interactive Zero-Knowledge-Proofs

At its core, a Zero-Knowledge Proof (ZKP) is a cryptographic protocol that enables a prover to convince a verifier that a specific statement is authentic without revealing the underlying secret [2]. Traditionally, this requires an interactive "challenge-response" dialogue: the prover sends a commitment, the verifier issues a random challenge, and the prover returns a response. This process must be repeated multiple times to ensure statistical confidence, which creates significant bottlenecks for modern, asynchronous infrastructure like decentralized networks or offline digital signatures, where the verifier and prover may not be able to engage in a back-and-forth conversation.

Non-Interactive Zero-Knowledge Proofs (NIZKs) resolve these limitations by condensing the entire multi-round dialogue into a single, static message [3]. This transformation allows a prover to generate a signature that functions as a self-contained, verifiable proof that can be checked by anyone at any time without further communication. To enable this, NIZKs typically rely on a Common Reference String (CRS) or Common Random String model, where both parties have trusted access to a pre-generated, uniformly distributed random string [4]. This shared randomness provides an impartial foundation that ensures the verifier's challenges are unpredictable and consistent.

By utilizing the Fiat-Shamir Heuristic, this framework allows the prover to generate challenges locally, effectively "binding" the proof to the message being signed. This development is crucial for post-quantum security because it allows complex, NP-complete problems to be used as the basis for digital signatures. Because the verification is non-interactive and mathematically tied to the specific message, this approach creates a robust, quantum-resistant signature scheme that eliminates the need for live stateful interaction while maintaining the high security standards required for future-proof communication.

Non-Interactive Zero-Knowledge-Proofs must satisfy three fundamental criteria [3]

a. Completeness: If the statement is true and the prover is honest, a legitimate proof string will always cause the verifier to accept.
b. Soundness: If the statement is false, no malicious prover can generate a forged proof string that will deceive the verifier to accept the signature.
c. Zero Knowledge: The proof string leaks absolutely no computational knowledge regarding the witness

C. Fiat-Shamir Transformation

Fiat-Shamir Transformation or Fiat-Shamir Heuristics is a technique in cryptography used to convert an interactive proof system to a non-interactive system [5].

A standard interactive "public coin" system follows three-steps.

a. Commitment: The prover provides a preliminary message (commitment) to the prover
b. Challenge: The verifier chooses a truly random challenge from a public set and sends it to prover
c. The prover computes a response based challenge, commitment, and secret witness/private key

The Fiat-Shamir Transformation eliminates the verifier's active role by forcing the prover to generate a challenge themselves, but in a way that prevents cheating. This process is done by putting a concatenation of the statement being proved(x) and the commitment(a) through a hash function [5].

$$e = H(x \,||\, a)$$

Because the hash function is unpredictable, the prover cannot predict what e will be until they have committed to a. The prover then computes a response as normal. The entire proof/response is sent to the verifier alongside the a or e. The verifier receives the single message from the prover and hashes x and a on their own to compute e, and compares the result to the given e. If the resulting e is the same as the e given by the prover, then the message is authentic. If not, the message is not authentic [5].

D. Probability Problem and Power of Iteration

The Probability Problem arises because a single round of a zero-knowledge proof is often statistically weak. If the graph used in the protocol has E edges, the soundness error, the probability that a prover can trick the verifier to accept a false proof, in a single round is proportional to 1/|E|. The Power of

Iteration lies in the exponential decay of the cheating probability. Because each round of the Fiat-Shamir-transformed protocol is independent (due to the random oracle properties of the hash function), the trials are independent events.

$P(Cheating\ Success) = \left(1 - \frac{1}{|E|}\right)^n$, where E is the number of edges in the graph and n is the number of repetition or ”rounds”.

E. Graph Theory

Graph theory is a foundational branch of discrete mathematics and computer science dedicated to studying graphs; any system that involves interconnected elements that can be modeled, analyzed, and optimized [6].

A graph G is formally defined as an ordered pair $G = (V, E)$, where V represents a set of vertices and E represents a set of edges that connect pairs of vertices. There are three types of graphs [6].

a. Undirected Graphs: A graph where edges have no orientation. The edge (u,v) is identical to the edge (v,u) [7].
b. Directed Graphs: A graph where edges have specific direction. The edge (u,v) denotes a link from u to v, but not vice versa [7].
c. Weighted Graphs. A graph where each edge is assigned a numerical value or “weight” [7].

F. k-Colorable Graphs Problem

A k-Coloring of graph $G = (V, E)$ is a map c:V→S, such that $c(v) \neq c(w)$ whenever v and w are adjacent. The elements S are called available colors [7]. Typically, a graph is considered k–colorable if there exist a coloring function c:V→{1,2,...,k} such that every edge $(u, v) \in E$, the condition $c(u) \neq c(v)$ holds true.

The complexity of determining whether a graph is k-colorable varies based on the value of k. For $k=1$, it is trivial since only a null graph is 1-colorable. For $k=2$, the problem can be solved in polynomial time using Breadth-First Search (BFS) or Depth-First Search (DFS). For any value of $k \geq 3$, the problem is considered NP-Complete [8].

III. Implementation Method

A. Basic Description

This digital signature scheme utilizes a Non-Interactive Zero-Knowledge Proof (NIZK) to provide post-quantum security. By leveraging the NP-complete nature of the k-coloring problem, the system allows a prover to demonstrate possession of a valid secret coloring for a public graph G without revealing the underlying color assignment. To facilitate widespread application, the protocol employs the Fiat-Shamir Transformation, which effectively replaces the need for live interaction between prover and verifier of an interactive proof with a cryptographic hash. This transformation binds the signature to the message itself, enabling the prover to generate a standalone signature that any verifier can check at any time without further interaction.

B. Setup

The security of the scheme is established through a rigorous initialization process.

1. Public Key: The public key consists of a large and complex graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. The graph used must be sufficiently large and dense to ensure that solving the k-colorability of it computationally unfeasible even for a quantum computer
2. Private Key: The prover must possess a valid k-coloring of the public graph, an assignment of colors from the set {1,2,…,k} to each vertex in V, such that no two adjacent vertices share the same color.

C. Signature Generation

To sign a message M, the prover will need to do the following steps.

1. Shuffle: The prover will randomly select a permutation of the k-coloring. This ensures that even if a cheater observes the revealed colors, they cannot map them back to the original private key.
2. Commitment: For every $v \in V$, the prover hashes the coloring of the vertex (combined with a random salt) to create a commitment which essentially locks the colors to the node for the current iteration.
3. Hash Challenge: The prover will use Fiat-Shamir transformation on the concatenation of all the commitments and the message to compute a challenge hash.
$$Challenge = H(Commitments \,||\, M)$$
4. Select Edges: The resulting challenge hash is used to pseudo randomly select a set of edges from the graph.
5. Package: The prover will construct a package containing the commitments, challenge_edges, and keys (used to unlock specific commitments as proof)

D. Verifying Signature

To verify a message M, the verifier will need to do the following steps.

a. Recompute: on receiving the signature, the verifier first recomputes the challenge string using the provided commitments and the message M provided by the prover.
b. Compare: The verifier then compares this recomputed value with the challenge included in the signature package. If the two match, it proves the prover followed the deterministic Fiat-Shamir process.

c. Validate: The verifier will then use the keys given to unlock the specific commitments. They then confirm the opened values are consistent with the original commitments and verify that the revealed colors are valid and different (obeying the graph coloring rule).
d. Decision: If all conditions are met, the message is accepted

E. Repetition

To ensure the integrity of the cryptographic process and mitigate the risk of a "cheating" prover the protocol incorporates an iterative verification mechanism. Rather than relying on a single, isolated transaction, the Signature Generation and Signature Verification processes are executed over *N* independent rounds. In this iterative framework, the prover must demonstrate their knowledge of the secret k-coloring solution in every single iteration. This process functions as a zero-knowledge proof of knowledge, where the repetition serves to reduce the probability of a successful forgery to an infinitesimally small value based on the Probability Problem and the Power of Iteration.

IV. IMPLEMENTATION IN PYTHON

The implementation of this protocol was done in python utilizing 2 main libraries, hashlib for the cryptographic hashing (SHA-256) and secrets for cryptographically secure pseudo-random number generation. These built-in libraries were selected to ensure auditability and to avoid vulnerabilities associated with third-party dependencies.

The graph is represented as an edge list (a list of tuples), providing an efficient way to iterate over connections. For simplicity, the implementation uses a small 4 vertices and 7 edge graph. The coloring mappings utilize dictionaries, offering O(1) average time complexity for vertex-to-color lookups, which is essential for rapid proof generation and verification.

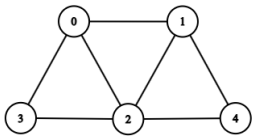
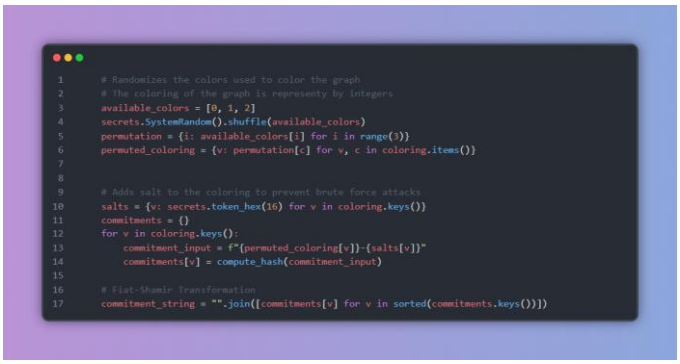


Fig 4.1 Graph Used as Public Key in Implementation

A. Commitment Generation

To prevent brute-force attacks on the commitment hashes, every vertex assignment includes a 128-bit salt generated via secrets.token_hex. This prevents an adversary from pre-computing hashes for all possible color combinations. We prioritize secrets over random because the latter is deterministic and unsuitable for cryptographic operations.



```
# Randomizes the colors used to color the graph
# The coloring of the graph is representy by integers
available_colors = [0, 1, 2]
secrets.SystemRandom().shuffle(available_colors)
permutation = {i: available_colors[i] for i in range(3)}
permuted_coloring = {v: permutation[c] for v, c in coloring.items()}


# Adds salt to the coloring to prevent brute force attacks
salts = {v: secrets.token_hex(16) for v in coloring.keys()}
commitments = {}
for v in coloring.keys():
    commitment_input = f"{permuted_coloring[v]}-{salts[v]}"
    commitments[v] = compute_hash(commitment_input)

# Fiat-Shamir Transformation
commitment_string = "".join([commitments[v] for v in sorted(commitments.keys())])
```

Fig 4.2 Commitment Generation Code

B. Fiat-Shamir Transformation, Edge Selection, and Proof Packaging

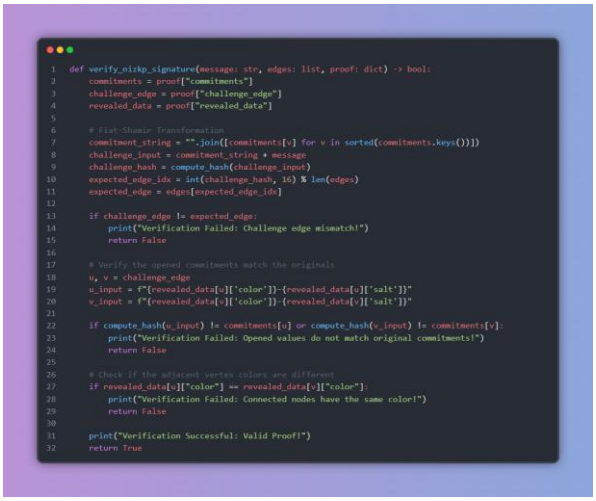
The transformation was implemented by concatenating the serialized commitments with the input message. The resulting hex digest is converted to an integer and mapped to a graph edge. This ensures that the challenge is both deterministic and unpredictable to the prover. A proof package is then generated in the form a dictionary that contains commitments, challenge edges, and the revealed data

```
# Fiat-Shamir Transformation
commitment_string = "".join([commitments[v] for v in sorted(commitments.keys())])
challenge_input = commitment_string + message
challenge_hash = compute_hash(challenge_input)

# Map the hash to a specific edge index
challenge_edge_idx = int(challenge_hash, 16) % len(edges)
challenge_edge = edges[challenge_edge_idx]

# Construct the Proof Package
u, v = challenge_edge
proof = {
    "commitments": commitments,
    "challenge_edge": challenge_edge,
    "revealed_data": {
        u: {"color": permuted_coloring[u], "salt": salts[u]},
        v: {"color": permuted_coloring[v], "salt": salts[v]}
    }
}
```

Fig 4.3 Fiat-Shamir Transformation, Edge Selection, and Proof Packaging Code

C. Verification Process

The verification process starts by reconstructing the challenge edge using Fiat-Shamir Transformation and confirming that the edges computed are the same as the ones given. It then checks if the recomputed commitments of the challenge edges and the given ones are equal. Finally, it checks if the colors of the edges follow the graph coloring rule.



```
def verify_nizkp_signature(message: str, edges: list, proof: dict) -> bool:
    commitments = proof["commitments"]
    challenge_edge = proof["challenge_edge"]
    revealed_data = proof["revealed_data"]

    # Fiat-Shamir Transformation
    commitment_string = "".join([commitments[v] for v in sorted(commitments.keys())])
    challenge_input = commitment_string + message
    challenge_hash = compute_hash(challenge_input)
    expected_edge_idx = int(challenge_hash, 16) % len(edges)
    expected_edge = edges[expected_edge_idx]

    if challenge_edge != expected_edge:
        print("Verification Failed: Challenge edge mismatch!")
        return False

    # Verify the opened commitments match the originals
    u, v = challenge_edge
    u_input = f"{revealed_data[u]['color']}-{revealed_data[u]['salt']}"
    v_input = f"{revealed_data[v]['color']}-{revealed_data[v]['salt']}"

    if compute_hash(u_input) != commitments[u] or compute_hash(v_input) != commitments[v]:
        print("Verification Failed: Opened values do not match original commitments!")
        return False

    # Check if the adjacent vertex colors are different
    if revealed_data[u]["color"] == revealed_data[v]["color"]:
        print("Verification Failed: Connected nodes have the same color!")
        return False

    print("Verification Successful: Valid Proof!")
    return True
```

Fig 4.4 Verification Function

V. TESTING

To validate the integrity and effectiveness of the proposed Non-Interactive Zero-Knowledge (NIZK) signature scheme, a testing framework was established to evaluate the protocol's performance across a diverse range of scenarios. These test cases were designed to simulate both authorized and unauthorized inputs, ensuring that the scheme functions reliably under ideal conditions while remaining resilient against adversarial attempts to manipulate the system. By systematically iterating through these inputs, we confirmed that the protocol correctly manages signature verification and, crucially, accurately identifies even the most subtle inconsistencies within the input data.

The experimental phase involved subjecting the system to a series of "positive" or accepted tests, where valid signatures generated by legitimate private keys were submitted for verification. These tests aimed to establish a baseline for successful identification, ensuring that authorized signatures are consistently accepted without false negatives. Complementing these are the "negative" or rejected tests, which involve the deliberate introduction of malicious or corrupted inputs. This included data that had been intentionally modified post-signature, signatures generated with incorrect or non-matching private keys, and completely malformed cryptographic packets. In each instance, the system was required to cross-reference the input against the stored public parameters, consistently triggering a rejection mechanism when the data failed to satisfy the underlying zero-knowledge proof requirements.

A. Accepted Cases

The signature given to the verifier should be accepted so verification will be successful.

Test Case Input:

Message: "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat "

N = 3

```
❯ uv run simpleconcept.py
Message to be signed: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor inc
ididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco lab
oris nisi ut aliquip ex ea commodo consequat
Number of signing and verifying repetition: 3
[TEST] Round 0
[Test] Commitment String Signature:
0a008e1344df68f4f95860701d1716ec70f8f9bef0fcf0410fcefbcd35fb6c413ec8174578d35ead3aa7761599528d0deb9062c6
707ad28fe9cd10a93dce8610b86a4612159eaf4ad6d4d4555e53d1d78f6b777b22d109a462187549ae643fd8f62d3625b6200de5
37e62fcf11fd64af4ba4c7cb8a114f750108c8ed21cc0a14
[Test] Commitment String Verify:
0a008e1344df68f4f95860701d1716ec70f8f9bef0fcf0410fcefbcd35fb6c413ec8174578d35ead3aa7761599528d0deb9062c6
707ad28fe9cd10a93dce8610b86a4612159eaf4ad6d4d4555e53d1d78f6b777b22d109a462187549ae643fd8f62d3625b6200de5
37e62fcf11fd64af4ba4c7cb8a114f750108c8ed21cc0a14
Verification Successful: Valid Proof!
====Verification Successful====
[TEST] Round 1
[Test] Commitment String Signature:
476c5d9717feece6f36d2d8743de5cf2d42695f15a65b7561c9d5b240c334be98c084c8bd63bdaa7ffa97534921832fed4a0426
f5563bf2b19e37c4d23e7876736da3c056baff43947bfb18f9a7774362e8331fb72d00ae847390f661aa780c8a061a1a7b2c596a
3f26dba3ad2e177002883be7cff707382e350eb9061cd2e6
[Test] Commitment String Verify:
476c5d9717feece6f36d2d8743de5cf2d42695f15a65b7561c9d5b240c334be98c084c8bd63bdaa7ffa97534921832fed4a0426
f5563bf2b19e37c4d23e7876736da3c056baff43947bfb18f9a7774362e8331fb72d00ae847390f661aa780c8a061a1a7b2c596a
3f26dba3ad2e177002883be7cff707382e350eb9061cd2e6
Verification Successful: Valid Proof!
====Verification Successful====
```

Fig 4.5 Result of Test Case Success Part 1

```
[TEST] Round 2
[Test] Commitment String Signature:
48ff9a3d5c77299edee066236a2099bc313d883b85a944827c742ba8796020d784d9336a594e3bb99284e4b0d556ec236ef3c4fc
991498b17b9b6908c8aa6bac62db76adb84f8a3ad68e8bf4e9adabb78060d54b4cdef5eca83d1b1eeaffef4a1fda786e290c6fcb
b4413f7c5f8c2ca8c5635ce03f51d7eb8a0796b0fd3e55a5
[Test] Commitment String Verify:
48ff9a3d5c77299edee066236a2099bc313d883b85a944827c742ba8796020d784d9336a594e3bb99284e4b0d556ec236ef3c4fc
991498b17b9b6908c8aa6bac62db76adb84f8a3ad68e8bf4e9adabb78060d54b4cdef5eca83d1b1eeaffef4a1fda786e290c6fcb
b4413f7c5f8c2ca8c5635ce03f51d7eb8a0796b0fd3e55a5
Verification Successful: Valid Proof!
====Verification Successful====
```

Fig 4.5 Result of Test Case Success Part 2

B. Rejected Cases

1. Test Case Failure 1

The signature given to the verifier should be rejected since the message given to the verifier is different from given to the prover

Message Prover: "Discrete Mathematics"

Message Verifier: "Calculus"

N = 2

```
❯ uv run simpleconcept.py
Message to be signed: Discrete Mathematics
Number of signing and verifying repetition: 2
[TEST] Round 0
[Test] Commitment String Signature:
b093d1630c85b8b09fddbdeb39e79b08e587e00a530e8cf932e5b17b03429018ac9e5e9d645a05570e7ae2f8fd7b3cf139506106
630aeda0214acc32e38dc3b8679d7b4e0c246b8a7d473cee73e60485a97556078a355f651b1d0097be9a1a04dd487c3843fc4b5a
ebbae5ef33ff73bf2cb8c63aad5c4c08f076f71921fa0887
[Test] Commitment String Verify:
b093d1630c85b8b09fddbdeb39e79b08e587e00a530e8cf932e5b17b03429018ac9e5e9d645a05570e7ae2f8fd7b3cf139506106
630aeda0214acc32e38dc3b8679d7b4e0c246b8a7d473cee73e60485a97556078a355f651b1d0097be9a1a04dd487c3843fc4b5a
ebbae5ef33ff73bf2cb8c63aad5c4c08f076f71921fa0887
Verification Failed: Challenge edge mismatch!
```

Fig 4.5 Result of Test Case Failure 1

2. Test Case Failure 1

The signature given to the verifier should be rejected since the message given to the verifier is different from given to the prover by 1 character

Message Prover: "ABCD"

Message Verifier: "ABCS"

N = 2

```
❯ uv run simpleconcept.py
Message to be signed: ABCD
Number of signing and verifying repetition: 2
[TEST] Round 0
[Test] Commitment String Signature:
4ff5e17aa9ffa204f5fa6659f990a7a602c110cd3f9866cdd4cb4d956cadcee21246f52076d8ca1beb940c6747c3b200f831e76b
87bac0d804482a82db4504f2745f5c2ee54f1effe0241eef6c5a8d2ffc76be252c8adc09d15155f68abb751267fef986b138deed
8594acc84328471f614906b42660ab1bd59df51c8a2cacd93
[Test] Commitment String Verify:
4ff5e17aa9ffa204f5fa6659f990a7a602c110cd3f9866cdd4cb4d956cadcee21246f52076d8ca1beb940c6747c3b200f831e76b
87bac0d804482a82db4504f2745f5c2ee54f1effe0241eef6c5a8d2ffc76be252c8adc09d15155f68abb751267fef986b138deed
8594acc84328471f614906b42660ab1bd59df51c8a2cacd93
Verification Failed: Challenge edge mismatch!
====Verification Failed====
[TEST] Round 1
[Test] Commitment String Signature:
67374621f5f5822908697eb3e60b3d308fdc6c9a6f6cbc097333bb5a5b0a15ec161f85fbbfc5b8aac18e7f94d19a17afe6bd79cc
1a5f1ae548571b6cfeec969fd3f44634423dff3745df12238dad58c6b7ad41cb0d45aaded9ec053804e7d2e2ea86363453444496bf
132f984cd245fff3cb6250214a64159718e671c111ae8c95
[Test] Commitment String Verify:
67374621f5f5822908697eb3e60b3d308fdc6c9a6f6cbc097333bb5a5b0a15ec161f85fbbfc5b8aac18e7f94d19a17afe6bd79cc
1a5f1ae548571b6cfeec969fd3f44634423dff3745df12238dad58c6b7ad41cb0d45aaded9ec053804e7d2e2ea86363453444496bf
132f984cd245fff3cb6250214a64159718e671c111ae8c95
Verification Failed: Challenge edge mismatch!
====Verification Failed====
```

Fig 4.5 Result of Test Case Failure 2

VI. EVALUATION

The evaluation of the implemented program demonstrates a high degree of technical diligence in its adherence to secure cryptographic practices. By exclusively utilizing Python's built-in libraries, the implementation prioritizes auditability and eliminates potential attack vectors associated with third-party dependencies. The design choices reflect a focus on performance, as the use of an edge list for graph representation and dictionary-based mappings for color lookups allows for an average time complexity of O(1), which is critical for ensuring that proof generation and verification remain efficient even as the graph complexity scales.

Security was further bolstered by incorporating a 128-bit salt for every vertex assignment during the commitment phase. This addition of a salt, generated via secrets.token_hex, serves as a vital safeguard against pre-computation attacks where an adversary might attempt to build hash tables for all possible color combinations. By prioritizing the secrets library over standard random modules, the program ensures that the underlying values are non-deterministic and suitable for cryptographic operations, thereby creating a robust defense against brute-force attempts on commitment hashes.

The testing phase confirms that the protocol functions reliably under varied conditions, successfully differentiating between authentic and forged signatures. In scenarios where the message and graph coloring matched, the verification process consistently returned a "Valid Proof!" status. Conversely, the system demonstrated precise sensitivity to data integrity by rejecting signatures that were modified by even a single character, as well as cases where the message signed by the prover failed to correspond to the one presented to the verifier. This testing confirms that the Fiat-Shamir transformation and the edge selection mechanisms are correctly implemented to maintain the strict integrity of the digital signature.

VII. CONCLUSION

The research concludes that the proposed non-interactive zero-knowledge proof (NIZK) scheme provides a viable pathway toward quantum-resistant digital security. By leveraging the NP-complete difficulty of the k-colorable graph problem, the scheme bypasses the algebraic vulnerabilities inherent in RSA and ECC, which are susceptible to Shor's algorithm on future quantum hardware.

The integration of the Fiat-Shamir Transformation effectively resolves the communication bottlenecks associated with traditional interactive proofs. By transforming an interactive system into a static, standalone signature, the protocol achieves the flexibility required for modern asynchronous environments, such as decentralized networks. Ultimately, by enforcing strict iterative verification rounds, the protocol maintains a mathematically sound security threshold that reduces the probability of successful forgery to an infinitesimally small value, ensuring the scheme remains resilient against both classical and quantum-computational adversaries.

ACKNOWLEDGMENT

The author of this paper gives praise and gratitude to God who has given the author the opportunity to complete this paper entitled "Post-Quantum Digital Signatures via Non-Interactive Zero-Knowledge-Proofs of k-Colorable Graphs". The author would also like to express his deepest gratitude to the lecturer of this Discrete Mathematics class, Prof. Dr. Ir. Rinaldi, M.T for all the lessons and motivations given throughout the lectures and the making of this paper.

REFERENCES

[1] "Digital Signature Standard (DSS)," Feb. 2023. doi: 10.6028/NIST.FIPS.186-5.

[2] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof Systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, Feb. 1989, doi: 10.1137/0218012.

[3] H. Wu and F. Wang, "A Survey of Noninteractive Zero Knowledge Proof System and Its Applications," The Scientific World Journal, vol. 2014, pp. 1–7, 2014, doi: 10.1155/2014/560484.

[4] M. Blum, A. De Santis, S. Micali, and G. Persiano, "Noninteractive Zero-Knowledge," SIAM Journal on Computing, vol. 20, no. 6, pp. 1084–1118, Dec. 1991, doi: 10.1137/0220068.

[5] A. Fiat and A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems," in Advances in Cryptology — CRYPTO' 86, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 186–194. doi: 10.1007/3-540-47721-7_12.

[6] K. H. Rosen, Discrete mathematics and its applications. McGraw-Hill Education, 2019.

[7] R. Diestel, *Graph Theory*, vol. 173. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. doi: 10.1007/978-3-662-53622-3.

[8] D. W. Matula, G. Marble, and J. D. Isaacson, "GRAPH COLORING ALGORITHMS," in Graph Theory and Computing, Elsevier, 1972, pp. 109–122. doi: 10.1016/B978-1-4832-3187-7.50015-5.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2025

Wimar Widiarto 13525009