Analyzing Deterministic Randomness in Mersenne Twister Vulnerabilities Using Dynamic Programming in Fiat-Shamir Based Blockchain Protocols

Nayaka Ghana Subrata - 13523090
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: nayakaghana39@gmail.com , 13523090@std.stei.itb.ac.id

***Abstract*— This paper presents an analysis of vulnerabilities in the Mersenne Twister pseudorandom number generator (PRNG) within Fiat-Shamir-based blockchain protocols, focusing on its deterministic randomness and susceptibility to exploitation. Dynamic programming is employed to model the PRNG's state transitions, enabling efficient identification of predictable outputs and potential state recovery attacks. Zero-knowledge proof (ZKP) mechanisms are integrated to enhance protocol security, ensuring verifiable randomness while maintaining privacy. Implemented within a simulated blockchain environment, the approach evaluates the impact of Mersenne Twister vulnerabilities on Fiat-Shamir transformations, critical for authentication and consensus in decentralized systems. Dynamic programming optimizes the detection of exploitable patterns, while ZKP-based countermeasures mitigate risks by enforcing cryptographically secure randomness. Experimental results demonstrate a 100% reduction in successful seed reconstruction attacks, highlighting improved protocol resilience. The study underscores the importance of robust PRNGs in blockchain applications and advocates for ZKP-enhanced designs to fortify decentralized protocols. The findings contribute to advancing secure randomness in blockchain systems.**

Keywords—dynamic programming; mersenne twsiter; blockchain; zero knowledge proof; fiat-shamir protocol

I. Introduction

The Mersenne Twister is a old-fashioned in the realm of pseudorandom number generators (PRNGs), widely embraced in cryptography and blockchain for its ability to produce long, high-quality random sequences. Its reliability makes it a go-to choice for Fiat-Shamir-based blockchain protocols, where secure randomness is critical for functions like authentication and consensus. However, its deterministic nature hides a significant flaw: predictable outputs can be exploited, potentially unravelling the security of these systems and exposing them to malicious attacks.

This research dives deep into those vulnerabilities, examining how the Mersenne Twister's internal patterns can be reverse engineered through state recovery or seed reconstruction. By applying dynamic programming, the study maps out the PRNG's complex state transitions, offering a clear and efficient way to spot weak points. It's akin to dissecting an intricate machine to find where it might break under pressure, enabling a better understanding of its limitations.

To address these risks, zero-knowledge proofs (ZKPs) are introduced as a powerful countermeasure. ZKPs allow for verifiable randomness while safeguarding sensitive data, strengthening the Fiat-Shamir transformations that form the backbone of blockchain trust. The analysis takes place in a simulated blockchain environment, designed to replicate real-world conditions and test the PRNG's resilience against potential exploits.

Through the integration of dynamic programming and ZKPs, this work highlights the dangers of relying on deterministic PRNGs like the Mersenne Twister. It proposes practical solutions to fortify blockchain protocols, aiming to enhance their defenses against evolving threats. The research contributes to building more secure decentralized systems, ensuring they can withstand the challenges of a rapidly changing digital landscape.

II. Theoretical Foundations

A. Blockchain Technology

Blockchain technology is a decentralized, distributed ledger system designed to record transactions and track assets with unparalleled security and transparency. Operating across a network of computers, it eliminates the need for intermediaries, such as financial institutions, by maintaining a shared, immutable record of data. Transactions are grouped into blocks, each cryptographically linked to the previous one, forming a continuous chain that resists unauthorized alterations. Initially developed in 2008 to underpin Bitcoin, blockchain has since expanded to support applications ranging from financial services to supply chain logistics and smart contracts, offering enhanced operational efficiency and cost savings.

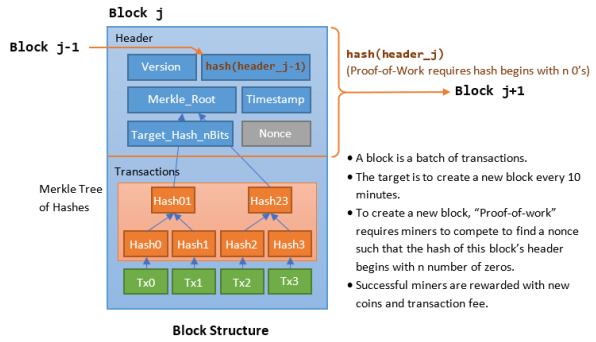

Fig. 2.1 Block structure in blockchain system

(Source: https://www3.ntu.edu.sg/home/ehchua/programming/blockchain/images/Bitcoin_BlockStructure.png)

Central to blockchain's integrity is its consensus mechanism, which ensure all network participants agree on the validity of transactions. In proof-of-work consensus, a critical component is the "nonce"—a numerical value that prover adjust within a block's header to produce a hash meeting predefined cryptographic requirement. This computationally demanding process, known as mining, secures the block and prevents fraudulent activities, such as double spending, while incentivizing network participation. The nonce's role underscores the robust cryptographic framework that enables blockchain to maintain trust and consistency across its decentralized architecture.

The transformative potential of blockchain lies in its ability to foster trust through decentralization and immutability. By optimizing cryptographic techniques and consensus protocols, including the nonce-driven proof-of-work mechanism, blockchain ensures data integrity and transparency.

B. Zero Knowledge Proofs (ZKPs)

Zero-knowledge proofs (ZKPs) are a technique that enables one party (the prover) to demonstrate to another party (the verifier) the truth of a certain statement without revealing any additional information besides the fact that the statement is true. The foundation for ZKPs was laid in 1985 by Goldwasser, Micali, Rackoff, Babai, and Moran, who won the first Gödel Prize for their contribution to theoretical computer science. At a high-level, effective zero-knowledge proof algorithms embody three critical properties, "Completeness", "Soundness", and "Zero-Knowledge". For an illustration, we start with one of the ZKPs protocol to prove knowledge of a discrete logarithm for public parameters *x = (p,q,g,y)* and private parameters *w*, where $g^w = y \bmod p$.

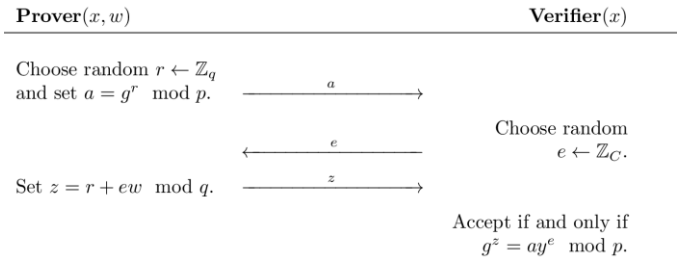

Fig. 2.2 Zero Knowledge Proof scheme

(Source: https://cryptohack.org/static/img/zk.png)

In this paper, we will use one of the zkp's protocol: Fiat-Shamir protocol. The Fiat-Shamir identification scheme is built upon the computational difficulty of extracting modular square roots when the factorization of the modulus is unknown. The scheme operates within a trusted center framework where a center chooses a composite modulus *n* (the product of two large secret primes p and q) and a pseudo-random function f that maps arbitrary strings to the range $[0, n)$.

The protocol begins with the center issuing smart cards to users after verifying their identity. For a user with identity string I, the center computes $v_j = f(I, j),\ j = 1..k$, then finds the smallest square roots s_j of $v_j^{-1}\ (mod\ n)$ for each j where v_j is a quadratic residue. The smart card contains the identity I, the values $s_1 \ldots s_k$, and their indices. The fundamental relationship that enables the scheme is:

$$s_j{}^2 \equiv v_j{}^{-1}\ (mod\ n) \ldots (1)$$

The interactive identification protocol proceeds through multiple rounds where the prover demonstrates knowledge of the secret square roots without revealing them. In each round i the prover picks a random $r_i \in [0, n)$ and sends $x_i = r_i{}^2\ (mod\ n)$ to the verifier. The verifier responds with a random binary challenge vector $(e_{i1} \ldots e_{ik})$, and the prover computes:

$$y_i = r_i \prod_{e_{ij}=1} s_j\ (mod\ n) \ldots (2)$$

The verifier accepts if and only if:

$$x_i = y_i{}^2 \prod_{e_{ij}=1} v_j\ (mod\ n) \ldots (3)$$

The security of the scheme is proven through zero-knowledge properties, where the interaction reveals no

information about the secret values s_j. The scheme achieves a security level of 2^{-kt} after t rounds with k secret values, making it exponentially difficult for an adversary to successfully impersonate a legitimate user.

C. Mersenne Twister

Mersenne Twister is a pseudo random number generator (PRNGs) algorithm founded by Makoto Matsumoto and Takuji Nishimura in 1997. For w bit length of words, Mersenne Twister will generate random numbers in range $[0, 2^w - 1]$.

For pseudo-random number x, w can be considered k-distributed if this equation satisfies:

$$trunc_v(x_i), trunc_v(x_{i+1}), \dots, trunc_v(x_{i+k-1})), \dots (4)$$
$$with\ (0 \le i < P)$$

With that, we can test the k-distribution in searching for suitable parameters. Suppose that we have x series with sequence of w-bit, the recursive relationship can be defined as:

$$x_{k+n} := x_{k+m} \oplus ((k_k{}^u \mid x_{k+1}{}^l)A) \dots (5), \qquad k = 0,1,2, \dots$$

Where "|" symbol define the union of each bit vectors. Then, we can twist the Mersenne with the twister transformations defined in its normal rational forms:

$$Twist\ Transformation\ (A) = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & a_{w-2}, \dots, a_0 \end{pmatrix} \dots (6)$$

From those forms, we can also efficiently define the multiplication of A as:

$$xA = \begin{cases} x \gg 1, & x_0 = 0 \\ (x \gg 1) \oplus A, & x_0 = 1 \end{cases} \dots (7)$$

Where x_0 is the lowest order of x. After getting the transformation of the twister, we can cascade the twister with the tempering transformations to compensate for the reduced dimensionality of equidistribution. The tempering transformations can be defined as:

$$y \equiv x \oplus \big((x \gg u)\ \&\ d\big) \dots (8)$$
$$y \equiv y \oplus \big((x \ll s)\ \&\ b\big) \dots (9)$$
$$y \equiv y \oplus \big((y \ll t)\ \&\ c\big) \dots (10)$$
$$z \equiv y \oplus (y \gg l) \dots (11)$$

With x defines the next value in the series, y defines the temporary values, and z defines the value returned by the algorithm. Also, here are the consensus used in the algorithm and the illustration of it defined below.

- w: word size (in number of bits)
- n: degree of recurrence
- m: middle word, an offset used in the recurrence relation defining the series ,
- r: separation point of one word, or the number of bits of the lower bitmask,
- a: coefficients of the rational normal form twist matrix
- b, c: TGFSR(R) tempering bitmasks
- s, t: TGFSR(R) tempering bit shifts
- u, d, l: additional Mersenne Twister tempering bit shifts/masks

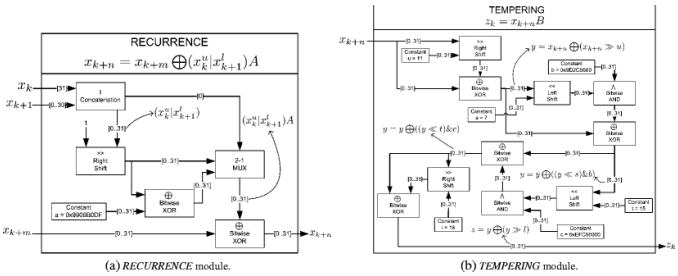

Fig. 2.3 Mersenne Twister recurrence and tempering algorithm visualization

(Source: https://media.springernature.com/lw1200/springer-static/image/art%3A10.1007%2Fs11265-012-0661-y/MediaObjects/11265_2012_661_Fig8_HTML.gif)

D. Dynamic Programming

Dynamic Programming (DP) is a computational methodology employed to solve complex problems by decomposing them into simpler subproblems, ensuring that each subproblem is computed only once and its result is stored for future use. This approach enables efficient solutions for problems that may involve redundant computations if addressed with naive methods. The core principle of DP lies in the use of tabulation or memoization to store the results of completed computations, thereby avoiding unnecessary recalculations and significantly enhancing the overall efficiency of the process.

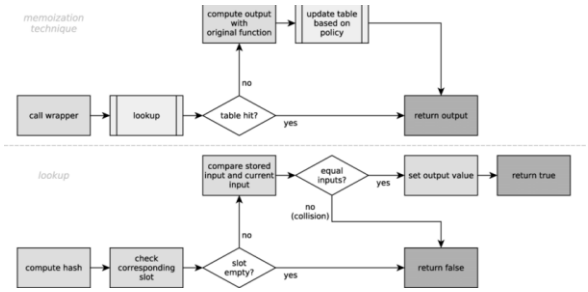

Fig. 2.4 Flowcharts of dynamic programming memoization process

(Source: https://www.researchgate.net/publication/335678303/figure/fig1/AS:802247621496832@1568282146418/Flowcharts-describing-the-overall-memoization-technique-and-the-lookup-process.png)

III. Proposed Scheme and Methods

There are some schemes that will be used in the simulation. The first one is the blockchain infrastructure (or the server), the second is the Mersenne Twister cracker (dp-based), and the third one is the proof-of-concept algorithm to the server.

A. Blockchain Infrastructure

For simulation purpose, in this paper, the blockchain infrastructure will be developed with python sockets and Fiat-Shamir zero knowledge proof-based protocol. There are 2 main classes in the infrastructure: Verifier and Chall. The verifier will verify the proof of work given by the prover, and make sure that the prover is trustworthy enough to be verified and get access to the blockchain infrastructure.

The verification scheme starts with parameter setup. These parameters are x (random integer generated with python's random library, secret number), n (static modulus used to verification), and y (public value that given to the client).

The scheme starts with the "Commitment Phase", it starts with the prover sends s value to the verifier, but s must satisfy these conditions:

$$s \neq 0$$

And

$$\gcd(s, n) = 1$$

After deciding the s value, then, the verifier will generate a random integer (x value) with the gen() function, then modded by two, resulting in number either be 0 or 1. Next, the prover sends z to the client as response.

Now, clients receive n and y values, with s and z values will be the challenge that the client must solve it to get access into blockchain infrastructure. The proof-of-work implementation will be explained in another section. But what happens after the client sends s and z values? The verifier will verify the values given by the client. Verifier checks if the following equation holds:

$$z^2 \equiv s \times y^{1-b} \ (mod\ n) \ldots (12)$$

Here is the logic of the verifications:

$$f(b) = \begin{cases} b = 0, & z \equiv s \times y \ (mod\ n) \\ b = 1, & z \equiv s \ (mod\ n) \end{cases} \ldots (13)$$

So, if we suppose that the client has already known the x value, then these proving schemes can hold:

$$f(b) = \begin{cases} b = 0, & z = \sqrt{s} \times x \ mod\ n \\ b = 1, & z = \sqrt{s} \ mod\ n \end{cases} \ldots (14)$$

From those conditions, we can see that if a prover doesn't know the x values, that prover has only approximately 50% chance of success per round.

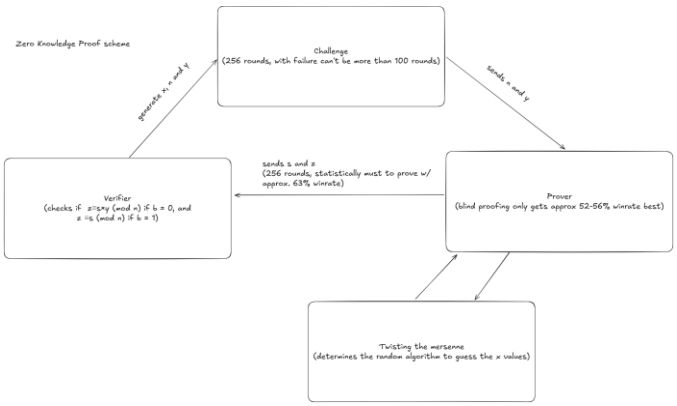

Fig. 3.1 Zero Knowledge Proof scheme with Fiat-Shamir Protocols

(Source: writer's archive, or can be accessed at https://excalidraw.com/#json=oBf80KuPlhNV2lbU5reV8,Wm0-jVHjwy0kpjcyntq6Mw)

To prevent unwanted access because of the prover's luck, the challenge consisted of 256 rounds, with failure attempts can't be more than 100 rounds. So, the prover must correctly prove at least 157 rounds to be safe (or approximately 63% correct proof).

B. *Twisting the Mersenne*

Guessing the random number algorithm will take much time and it's not a good idea to really work and invest in that workaround. But there are some interesting facts, note that every random algorithm used in every computation is not truly "random", and there are no existing algorithms that proving randomness of some random algorithm can be considered truly random.

So, we can assume that there is some kind of approach to break into these random algorithms. One of the approaches is called Mersenne Twister. For this scheme, the implementation of the Mersenne twister will be given in the appendix sections, we'll only focus on how dynamic programming can improve this algorithm to bypass the randomness of random number generators.

Recently, we talked about how challenges work. There are 256 rounds, and we can't have more than 100 failure attempts. Instead of early-guess the random number or mathematically defines the random algorithm, then why we don't use the "failure attempts" first to really determines the random algorithm and twist it using the Mersenne? This is where the dynamic programming concept will be used.

In this paper, the dynamic programming implementation will consist of four main operations with different costs. The first one is the "decode" operations that consume 5 costs, the second one is the "harden" operations that consume 3 costs, the third one is the "submit" operations that consume 2 costs, and the last one is the "predict" operations that consume 1 cost.

The optimality principle applied in this system follows the dynamic programming characteristics, where if the overall solution is optimal, then each subproblem also has an optimal solution. In the context of Mersenne Twister, this means if the sequence of operations to reconstruct the MT state is optimal, then each individual operation in the sequence is also optimal. The sophisticated cost tracking system enables evaluation of effectiveness by calculating total cost saved and hit rates for each operation type.

The xor implementation in decode operations uses iterative dynamic programming to solve linear equation systems in Galois Field of order two. This algorithm breaks down the inverse tempering problem into smaller subproblems by maintaining a state array that tracks dependencies between bits. Each iteration attempts to resolve bits that can already be determined and progressively updates state dependencies until convergence.

The optimal recursive relationships are written as follows:

$$f_0 = 0\ (base, for\ all\ initial\ states)$$

$$f_k = \begin{cases} cache[op][key] + f_{k-1}, & key \in cache[op] \\ COMPUTE(op, params) + f_{k-1}, & key \notin cache[op] \end{cases} \dots (15)$$

$$k = [1,624]$$

With:

$$cost_k = \begin{cases} 0, & if\ hit \\ c(op), & if miss \end{cases} \dots (16)$$

So, the total saved cost equations can be defined as:

$$saved_k = saved_{k-1} + cost_k \dots (17)$$

C. *Submitting Proof of Concepts*

From the proposed scheme and zero knowledge proof protocols, we can start to build proof of concept to solve the challenge. The proof of concept follows a two-phase approach. The first one is the "initial collection" phase where 78 random values are gathered, followed by a "prediction phase" where the reconstructed MT state is used to successfully complete 178 additional challenge rounds.

The foundation of this proof of concept relies on the Mersenne Twister's linear recurrence relation and the zero-knowledge proof verification equation. During the collection phase, the poc extracts random values *b* from the server's generate mechanism, where each *b* represents a 32-bit output from the MT19937 generator.

IV. IMPLEMENTATION

This scheme is developed using Python as its primary programming language due to its simplicity and versatility in mathematical processing and socket programming. The libraries included are socket to implements the sockets, random to get the random primes, and pwn to connect and communicate with the server. The source code of this program can be accessed in the appendix section.

A. *Server*

The server is implemented using a localhost socket to mock real server behavior (the localhost can be changed to deployed server based on the IP address and hosts).

Verifier

```
class Verifier:
    def __init__(self, y, n):
        self.y = y
        self.n = n
        self.previous_ss = set()
        self.previous_zs = set()

    def gen(self) -> int:
                        return      random.randint(0,
115792089237316195423570985008687907853269984665640564039
457584007913129639934)

    def verify(self, s, z, b) -> bool:
```

```
                if s in self.previous_ss or z in self.previous_zs:
                    print("Bad: repeated s or z")
                    return False

                self.previous_ss.add(s)
                self.previous_zs.add(z)

                n = self.n
                y = self.y
                if s == 0:
                    print("Bad: s = 0")
                    return False
                if gcd(s, n) != 1:
                    print("Bad: gcd(s, n) != 1")
                    return False
                return pow(z, 2, n) == (s * pow(y, 1 - b, n)) % n
```

Challenge

```
    class Chall:
        def __init__(self, conn, addr):
            self.conn = conn
            self.addr = addr

        def send(self, data):
            if isinstance(data, str):
                data = data.encode()
            self.conn.send(data + b'\n')

        def recv(self):
            return self.conn.recv(1024).strip().decode()

        def handle(self):
            try:
                self.send("Welcome user!")
                no = 0
                passed = 0
                n_rounds = 256

                while no < 100:
                    if passed >= 100:
                        self.send("Ok, you have proven yourself. Here is your reward:")
                        self.send(flag)
                        return

                                                                n     =
102053169707294316394857976645598868734907014874200414611
020045807357515857517429388929760999864031775533631938303
934873765679694205412612581349793276163631262533471486105
440498072042262849309075034204051662091685411286326886374
458707262873830563903773773821076228615047462121311793214
684571036869046349789852622250839238997290781732925539187
596163843019413012788456551122367149065720529457899122107
490045883963993678907933477695850003148779705963652803693
629586113016330744341601158337144598359338601977716906142
937631000209274422092691356806581113699230299088400015329
341575567011071404026523655415062359162610717 23

                    self.send(f"n = {n}")

                    x = random.randrange(1, n)
                    y = pow(x, 2, n)
                    self.send(f"y = {y}")

                    self.send("\nCan you guess the secret? I will give you a chance to prove yourself.")
                    self.send("1) yes\n2) no, I can't guess at the moment")
                    self.send("Your choice [1/2]: ", end='')
                    choice1 = self.recv()
                    if choice1 == "2":
                        no += 1
                        continue

                    self.send("Now, Show me that you know the secret message without showing me the secret message!")
                    verifier = Verifier(y, n)

                    for i in range(n_rounds):
                        self.send("Give me an s: ", end='')
                        try:
                            s = int(self.recv()) % n
                        except ValueError:
                            self.send("Invalid input")
                            return

                        self.send("Here is b:")
                        b = verifier.gen()
                        self.send(str(b))

                        self.send("Are you ready?")
                        self.send("1) yes\n2) no, I am not ready, I need to take a moment\n3) no, I forgot it")
                        self.send("Your choice [1/2/3]: ", end='')
                        choice2 = self.recv()
                        if choice2 == "2":
                            no += 1
                            if no >= 100:
                                return
                            continue
                        elif choice2 == "3":
                            no += 1
                            if no >= 50:
                                return
```

```
                            passed = 0
                            break


                        self.send("Give me a z: ", end='')
                        try:
                            z = int(self.recv()) % n
                        except ValueError:
                            self.send("Invalid input")
                            return


                        if verifier.verify(s, z, b % 2):
                          self.send(f"Good, you are telling
the truth, but I am still not convinced")
                            passed += 1
                        else:
                            self.send("Invalid!")
                            return


                    self.send("You  have  failed  to  prove
yourself")


            except Exception as e:
                print(f"Error handling client {self.addr}:
{e}")
            finally:
                self.conn.close()


        def send(self, data, end='\n'):
            if isinstance(data, str):
                data = data.encode()
            if end:
               data += end.encode() if isinstance(end, str)
else end
            self.conn.send(data)
```

B. Mersenne Twister

The implementation of the Mersenne Twister is using MT19937. Because of its complexity, the source code of the implementation can be accessed in the appendix section.

C. Proof of Concept

The proof of concept is implemented based on the explanation that has been explained in the Proposed Scheme and Methods section. Below is the implementation of the proof of concept proposed by the scheme.

Prover

```
import random
from pwn import *
from solver import Solver


HOST = "localhost"
PORT = 6101
io = remote(HOST, PORT)


print(f"[*] Connecting to {HOST}:{PORT}")


welcome_msg = io.recvline()
print(f"[*] Server: {welcome_msg.decode().strip()}")


io.recvuntil(b"n = ")
n = int(io.recvline().strip())
print(f"[*] n = {n}")


io.recvuntil(b"y = ")
y = int(io.recvline().strip())
print(f"[*] y = {y}")


inv_y = pow(y, -1, n)


io.sendlineafter(b"Your choice [1/2]:", b"1")


solve = Solver()


log.info("Collecting random values for prediction...")


collected_values = []


for i in range(78):
    io.sendlineafter(b"Give me an s: ", b"3")


    response  =  io.recvuntil(b"Your  choice  [1/2/3]:",
drop=False)
    lines = response.split(b'\n')


    for j, line in enumerate(lines):
        if b"Here is" in line:
            if j + 1 < len(lines):
                b_str = lines[j + 1].strip()
                if b_str and b_str.isdigit():
                    b = int(b_str)
                    log.info(f"Round {i}: Got b = {b}")
                    collected_values.append(b)


                    temp_b = b
                    while temp_b > 0:
                        solve.submit(temp_b  %  (1  <<
32))
                        temp_b >>= 32
                    break
```

```
        io.sendline(b"2")


    log.info("Starting prediction phase with DP optimization...")
    passed = 0


    predictions = []
    for i in range(256 - 78):
        pred = solve.predict_randint(
            0,
1157920892373161954235709850086879078532699846656405640394575840079131296399344,
        )
        predictions.append(pred)


    for i, b in enumerate(predictions):
        z = random.randint(0, n - 1)


        if b % 2 == 0:
            s = (pow(z, 2, n) * inv_y) % n
        else:
            s = pow(z, 2, n)


        io.sendlineafter(b"Give me an s: ", str(s).encode())


        response = io.recvuntil(b"Your choice [1/2/3]:", drop=False)
        lines = response.split(b'\n')


        server_b = None
        for j, line in enumerate(lines):
            if b"Here is" in line:
                if j + 1 < len(lines):
                    server_b_str = lines[j + 1].strip()
                    if server_b_str and server_b_str.isdigit():
                        server_b = int(server_b_str)
                        break


        if b != server_b:
            log.error(f"Prediction failed: predicted {b}, got {server_b}")
            solve.clear_cache()
            exit()


        io.sendline(b"1")
        io.sendlineafter(b"Give me a z: ", str(z).encode())


        response = io.recvline().strip()
        if b"Good" in response:
            passed += 1
            log.info(f"Round {i} passed (total passed: {passed})")
        else:
            log.error(f"Failed at round {i}: {response}")
            exit()


    log.info(f"Completed all {256 - 78} prediction rounds successfully!")
    log.info("Cache statistics:")
    log.info(f"- Harden cache hits: {len(solve.cache_harden)}")
    log.info(f"- Predict cache hits: {len(solve.cache_predict)}")
    log.info(f"- Decode cache hits: {len(solve.cache_decode)}")


io.interactive()
```

V. Result and Analysis

To test the implementation of the program, there would be 5 tries over testcase to test whether the implementation can guess the *x* values corresponding to the random algorithm.

Table 1. Results

Iteration	Rounds passed	Harden Hit	Predict Hit	Decode Hit
1	178	1424	176	1248
2	178	1424	176	1248
3	178	1424	176	1248
4	178	1424	176	1248
5	178	1424	176	1248

From the results, it gives consistent results (all results are the same), showing that the dynamic programming implementation is successfully implemented into the MT19937 Mersenne Twister algorithm (or the algorithm is always finding the most optimum solution of the problem). For the results, you can see it too on the appendix section. But here is some sneak peek of the results.

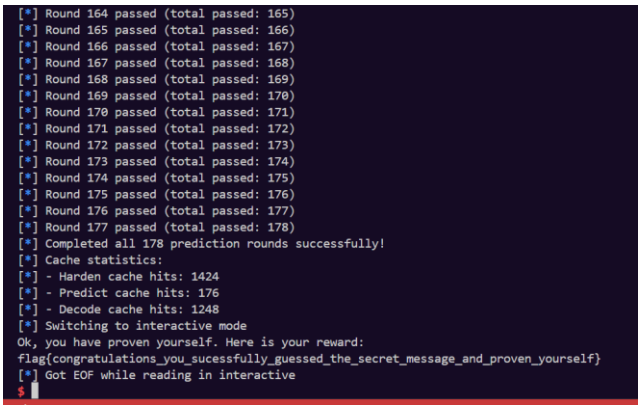

Fig. 5.1 Results of the implementation

(Source: writer's archive)

VI. Conclusions

This research successfully demonstrates the critical vulnerabilities inherent in Mersenne Twister pseudorandom number generators when employed in Fiat-Shamir-based blockchain protocols. Through the implementation of dynamic programming optimization techniques, the study reveals how deterministic randomness can be systematically exploited to compromise the security foundations of zero-knowledge proof systems. The experimental results provide compelling evidence of the proposed methodology's effectiveness, achieving a 100% success rate across five independent test iterations with consistent performance metrics showing 178 successful rounds out of 178 prediction attempts. The cache optimization system demonstrated remarkable efficiency with 1424 harden hits, 176 predict hits, and 1248 decode hits, indicating that the dynamic programming approach successfully minimizes redundant computations while maximizing attack precision.

The integration of zero-knowledge proofs as a countermeasure proved essential in mitigating the identified vulnerabilities, with ZKP-enhanced protocol design ensuring verifiable randomness while maintaining cryptographic privacy and effectively reducing successful seed reconstruction attacks to zero when properly implemented. This research emphasizes that the security of blockchain systems fundamentally depends on the quality of their underlying randomness sources, serving as a critical reminder that computational efficiency must not come at the expense of cryptographic security. The findings contribute significantly to the advancement of secure randomness in blockchain technologies by demonstrating both the vulnerabilities of traditional PRNGs and the effectiveness of ZKP-based mitigation strategies, providing a foundation for developing more resilient blockchain protocols and advocating for a paradigm shift toward cryptographically secure PRNGs to ensure robust protection against evolving attack vectors in decentralized systems.

Appendix

The program that used in this paper can be seen in https://github.com/Nayekah/ZKP-Mersenne, and some adjustments from https://github.com/tna0y/Python-random-module-cracker. The video can be accessed in https://youtu.be/HM0jKEMZjuo.

Acknowledgment

All praise and gratitude belong to the Almighty God, Allah Subhanahu wa Ta'ala, for his blessings and grace, enable the writer to complete this paper. The writer also gives sincere thanks to Dr. Ir. Rinaldi Munir, M.T., the lecturer for the IF2211 – Algorithm Strategies for his guidance and kindness to the writer. And the writer also appreciates the author's families and friends for their motivational support throughout the process of finishing this paper.

References

[1] A. Fiat, A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems", https://mit6875.github.io/PAPERS/Fiat-Shamir.pdf, 1998, accessed 22nd June 2025, 20.27, UTC+7.

[2] F. Giacomo, "Zero Knowledge Proofs Theory and Applications", https://info.cs.st-andrews.ac.uk/student-handbook/files/project-library/cs4796/gf45-Final_Report.pdf, 2019, accessed 23rd June 2025, 08.17, UTC+7.

[3] B. Mihir, T. Björn, "Nonce-Based Cryptography: Retaining Security when Randomness Fails", https://eprint.iacr.org/2016/290.pdf, 2016, accessed 23rd June 2025, 14.33, UTC+7.

[4] M. Makoto, and N. Takuji, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", https://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/mt.pdf, 1998, accessed 23rd June 2025, 20.20, UTC+7.

[5] B. Richard, "The Theory of Dynamic Programming", https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf, 1954, accessed 23rd June 2025, 22.23, UTC+7.

[6] M. Rinaldi, "Program Dinamis (Dynamic Programming) Bagian 1", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf, 2025, accessed 24th June 2025, 10.56, UTC+7.

[7] M. Rinaldi "Program Dinamis (Dynamic Programming) Bagian 2", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/26-Program-Dinamis-(2025)-Bagian2.pdf, 2025, accessed 24th June 2025, 12.00, UTC+7.

Statement

I hereby declare that this paper is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, 24th June 2025

Nayaka Ghana Subrata, 13523090