Fixed-weight Challenge Optimization

Fixed-weight Challenges. When the isometry queried is the one “on the left”, i.e. the one

between G0 and eG, the response consists of the monomial matrix eQ, which is generated uniformly

at random. As per the paragraph above, it is then sufficient to transmit only the seed used to

generate it. In other words, the challenge corresponding to 0 is much lighter than the others,

and so the communication cost can be improved by adjusting the probability distribution of the

challenge string, to make this possibility more likely to happen [BKP20]. This means that individual

rounds require less communication on average (at the cost of increasing the number of rounds). meaning

This passage discusses an optimization technique for reducing communication costs in a protocol involving fixed-weight challenges, specifically in the context of isometries between generator matrices. Here’s a breakdown of the key points:

1. Isometry Query on the Left:

When the queried isometry is the one between *G*0 and

~

*G* (denoted as "on the left"),

the response is a *monomial matrix* ~.

*Q*

 This matrix is generated uniformly at random and can be reconstructed using a

*seed*, meaning only the seed needs to be transmitted instead of the entire matrix.

1. Reduced Communication for Challenge 0:

 Since the challenge corresponding to 0 results in a lighter response (only transmitting the seed), this challenge requires significantly less communication compared to others.

1. Adjusting the Challenge Probability Distribution:

 To optimize communication costs, the probability of selecting the "lighter" challenge (challenge 0) can be increased.

 This reduces the average communication cost per round.

1. Trade-off

While individual rounds require less communication, the total number of rounds must increase to maintain the overall security level.

 This balances communication efficiency with protocol security.

This technique, referenced from [BKP20], is a practical way to optimize interactive cryptographic protocols while maintaining security guarantees.

Let's consider an example where this optimization is applied in a code-based identification scheme.

### Scenario: Code-Based Identification Protocol

 A prover (Alice) wants to prove knowledge of a secret isometry between two generator matrices *G*0 and *G*1.

 A verifier (Bob) sends a random challenge bit *c* ∈ {0, 1}.  Alice responds based on *c*.

### Standard Protocol (Without Optimization)

1. Key Generation:

 Alice knows a secret monomial matrix *Q* such that:

*G*1 = *QG*0

~

 She also derives a randomly permuted version *G*.

1. Commitment:

 Alice generates a random monomial matrix

~

*QG*0 to Bob.

1. Challenge:

 Bob sends a random bit *c* ∈ {0, 1}.

1. Response (Depends on *c*):  If *c* = 0:

~ ~

*Q*, applies it to *G*0, and sends *G*

=

Alice reveals *Q* (monomial matrix).

~

If *c* = 1:

 Alice reveals

*Q*

~−1*Q*, proving she knows the secret isometry.

### Optimized Protocol (Adjusting *c*'s Probability)

 The challenge bit *c* = 0 requires transmitting only a seed (used to generate

~

*Q*).

 The challenge bit *c* = 1 requires sending a full matrix (which is larger in size).

 To reduce communication, we make *c* = 0 more likely than *c* = 1 (e.g., choosing *c* = 0

with probability *p* = 0.75 instead of *p* = 0.5).

 This means fewer large responses are transmitted on average, reducing the communication cost.

### Trade-off

 Since *c* = 0 happens more often, the protocol might require more rounds to compensate for the increased probability of the lighter challenge.

 The security level remains the same because the overall statistical properties of the protocol remain unchanged.

### Conclusion

By skewing the challenge probability distribution toward the lighter challenge (transmitting just the seed), we reduce the communication cost per round while maintaining security by adjusting the number of rounds.

Let me explain this with a simpler example using numbers and step-by-step reasoning.

### Background

 Imagine a game where Alice (the prover) wants to convince Bob (the verifier) that she knows a secret transformation (matrix *Q*) that links two objects *G*0 and *G*1.

 Bob sends Alice a challenge (0 or 1), and Alice must respond accordingly.

 The goal is to minimize communication costs while still maintaining security.

### How It Works Without Optimization

 Bob picks a challenge randomly (50% chance for 0, 50% chance for 1).  Alice's response depends on Bob's challenge:

 If Bob sends 0 → Alice sends a small piece of data (just a short seed).

 If Bob sends 1 → Alice sends a large piece of data (a full matrix).

 On average, half the time a large message is sent, and half the time a small message is sent.

### Optimization: Adjusting Challenge Probability

 Instead of Bob choosing 0 or 1 equally, we bias the selection to favor 0.  For example, Bob picks:

 0 with 75% probability

 1 with 25% probability

Since 0 requires sending less data, the average communication per round is reduced.

### Example with Numbers

Without Optimization (Equal Probability)

Suppose:

 Sending a small response (for 0) takes 10 KB.  Sending a large response (for 1) takes 100 KB. Since both challenges are equally likely:

 Average communication cost per round =

#### (0.5 × 10) + (0.5 × 100) = 5 + 50 = 55 KB

With Optimization (Biasing Towards 0)

 Suppose Bob now chooses:

 0 with 75% probability

 1 with 25% probability

 New average communication cost:

#### (0.75 × 10) + (0.25 × 100) = 7.5 + 25 = 32.5 KB

 Result: The average cost per round drops from 55 KB to 32.5 KB, reducing the communication overhead.

### Trade-Off

 Since Bob sends 1 less frequently, each round gives less information about the secret transformation *Q*.

 To maintain the same security level, more rounds may be needed.  But because each round costs less, the overall benefit remains.

### Conclusion

 By favoring challenges that require less communication, we reduce the overall data transmitted.

 This is done at the cost of requiring slightly more rounds to achieve the same level of security.

 It’s a trade-off between communication efficiency and the number of rounds needed.

Sure! Let's start from the basics and build up to the idea of Fixed-Weight Challenges and Communication Optimization in cryptographic protocols.

## Basics of Cryptographic Identification Protocols

In cryptography, an identification protocol allows a prover (Alice) to convince a verifier (Bob) that she knows a secret without revealing it.

### Example: Password-Based Authentication

A simple example is logging into a website:

 Alice enters her password.

 The website verifies the password.  If correct, Alice is authenticated.

However, password-based authentication has problems:

 If someone intercepts the password, they can impersonate Alice.  The server must store passwords, which is a security risk.

To fix these issues, zero-knowledge identification protocols are used, where Alice proves knowledge of a secret without revealing it.

## Challenge-Response Protocols

A common method for secure identification is a challenge-response protocol:

* 1. Alice commits to a secret transformation (e.g., a hidden matrix or permutation).
  2. Bob sends a random challenge (0 or 1).
  3. Alice responds based on the challenge without revealing the secret.
  4. Bob verifies the response to check Alice's knowledge.

This method ensures that even if an attacker listens to the communication, they can’t impersonate Alice.

Example of a Simple Challenge-Response Protocol Imagine Alice has a secret number X and wants to prove it to Bob:  Alice picks a random number R and sends Bob Y = R + X.

 Bob challenges Alice with a random bit c:  If *c* = 0, Alice reveals *R*.

 If *c* = 1, Alice reveals *X* + *R*.

 Bob checks the correctness of Alice’s response.

This protocol prevents an attacker from learning X directly but still allows Bob to verify Alice.

## Code-Based Cryptography & Identification Protocols

Code-based cryptography uses error-correcting codes for secure cryptographic identification.

 Instead of a secret number *X*, Alice knows a secret isometry (monomial matrix *Q*) that transforms a known generator matrix *G*0 into another *G*1:

*G*1 = *QG*0

 The goal is for Alice to prove knowledge of *Q* without revealing it.

### Basic Code-Based Challenge-Response Protocol

1. Alice picks a random matrix

~ ~

*Q* and computes *G*

~

*QG*0.

=

1. Bob sends a random challenge *c* ∈ {0, 1}.
2. Alice responds as follows:

If *c* = 0, Alice reveals If *c* = 1, Alice reveals

~

*Q* (which is random).

~−1*Q* (which proves she knows *Q*).

*Q*

1. Bob verifies the response and repeats the process multiple times for security.

## Communication Cost in the Protocol

If *c* = 0, Alice only sends

~

*Q*, which can be generated from a small seed.

If *c* = 1, Alice must send a full matrix ~−1*Q*, which is much larger in size.

*Q*

Thus, when *c* = 0, the communication cost is small, and when *c* = 1, the cost is high.

## Optimizing Communication with Fixed-Weight Challenges

Since 0 requires less communication, we can optimize the protocol by making Bob choose 0 more often than 1.

Instead of choosing 0 or 1 with 50% probability, we set:

 *P* (*c* = 0) = 75% (favoring small responses).

 *P* (*c* = 1) = 25% (fewer large responses).

This reduces the average amount of data sent per round.

## Trade-Off: More Rounds for Security

 Since Bob chooses 1 less often, the protocol provides less information per round about

*Q*.

 To maintain security, more rounds are needed.

 However, because each round requires less data, the total communication cost is still

lower.

## Summary

✅ Identification protocols allow proving knowledge of a secret without revealing it.

✅ Code-based cryptographic protocols use transformations of generator matrices.

✅ Challenge-response protocols require different responses depending on Bob’s challenge.

✅ Optimizing communication involves making the cheaper response (c = 0) more likely.

✅ This reduces data per round but may require more rounds for security.

This technique helps make cryptographic protocols more efficient, especially in scenarios where communication costs matter (e.g., embedded systems, IoT, secure messaging).

how do you adjust the probability distribution of the challenge string

### Adjusting the Probability Distribution of the Challenge String

To adjust the probability distribution of the challenge string, we modify how the verifier (Bob) selects the challenge c in the protocol. Instead of choosing *c* ∈ {0, 1} uniformly at random (50% probability each), we bias the distribution to favor the challenge that requires less communication (c = 0).

## Basic Method: Weighted Random Selection

Instead of choosing *c* with equal probability ( *P* (*c* = 0) = 0.5, *P* (*c* = 1) = 0.5 ), we set:

*P* (*c* = 0) = *p*, *P* (*c* = 1) = 1 − *p*

where *p* is some value greater than 0.5 (e.g., 0.75 or 0.8). This means that c = 0 (the "lighter" challenge) is chosen more frequently.

## Implementing in Practice

There are multiple ways to implement this probability adjustment in software.

### Using a Biased Coin Flip

A simple way to achieve this is using a random number generator that produces values in

[0,1] and comparing it to *p*:

python

import random

def choose\_challenge(p=0.75):

"""Returns 0 with probability p and 1 with probability 1-p""" return 0 if random.random() < p else 1

*# Example usage*

challenges = [choose\_challenge(0.75) for \_ in range(10)] print(challenges) *# Example output: [0, 0, 1, 0, 0, 1, 0, 0, 0, 0]*

This code ensures that *c* = 0 is selected 75% of the time, and *c* = 1 is selected 25% of the time.

### Using a Precomputed Challenge String

Instead of generating a random challenge each time, you can precompute a challenge string that contains a biased number of 0s and 1s.

For example, if you need 100 rounds with *p* = 0.75, you generate:

 75 zeros (c = 0)

 25 ones (c = 1)

Then, shuﬄe the sequence randomly:

python

import random

def generate\_challenge\_string(n=100, p=0.75): num\_zeros = int(n \* p)

num\_ones = n - num\_zeros

challenge\_list = [0] \* num\_zeros + [1] \* num\_ones random.shuffle(challenge\_list)

return challenge\_list

*# Generate a sequence of 100 challenges with 75% probability of 0*

challenge\_string = generate\_challenge\_string(100, 0.75) print(challenge\_string[:20]) *# Print first 20 challenges*

This method ensures the overall frequency of 0s and 1s matches the desired probability.

### Using a Weighted Probability Function

Some programming languages and libraries provide weighted random selection functions. For example, in Python:

python

import random

def choose\_challenge():

return random.choices([0, 1], weights=[0.75, 0.25])[0]

*# Example usage*

challenges = [choose\_challenge() for \_ in range(10)] print(challenges) *# Example output: [0, 0, 0, 1, 0, 0, 0, 1, 0, 0]*

Here, selects from [0, 1] with weights [0.75, 0.25], ensuring a 75%

probability of choosing 0.

random.choices()

## Impact on Communication Cost

By increasing the probability of *c* = 0:

 Fewer large messages (corresponding to *c* = 1) are sent.  The average communication cost per round decreases.

 However, more rounds may be needed to maintain security.

## Choosing the Best Probability *p*

 A common choice is *p* = 0.75 (75% chance of *c* = 0), which reduces communication without needing too many extra rounds.

 Higher *p* values (e.g., 0.85 or 0.9) further reduce communication cost but require more rounds.

 The choice of *p* depends on the balance between communication efficiency and protocol security.

Would you like help choosing an optimal *p* for a specific application?

Adjusting the Probability Distribution of the Challenge String: Theory & Examples

## Understanding the Need for Probability Adjustment

In interactive cryptographic protocols, such as zero-knowledge proofs or identification schemes, the verifier (Bob) sends a random challenge *c* to the prover (Alice).

 Alice responds based on *c*, either revealing a lightweight response (small data) or a

heavy response (large data).

 The goal is to reduce communication cost while maintaining security.

Instead of selecting *c* uniformly (50% for each option), we bias the probability distribution

to favor the challenge that results in a smaller response.

This reduces average communication per round while keeping the security intact by increasing the number of rounds.

## Example: Standard Challenge-Response Protocol

Consider a scenario where:

 Alice proves knowledge of a secret monomial matrix *Q*.  Bob randomly picks a challenge *c* ∈ {0, 1}.

 Alice’s response depends on *c*:

|  |  |  |
| --- | --- | --- |
| Challenge *c* | Alice's Response | Size of Response |
| *c* = 0 | A small seed (used to derive a random matrix) | Small |
| *c* = 1 | A full matrix *Q*′ proving knowledge of *Q* | Large |

If Bob picks *c* = 0 50% of the time and *c* = 1 50% of the time, then Alice sends a large

response in half of the rounds.

This results in a higher communication cost.

## Adjusting the Challenge Probability

### Modified Distribution

Instead of picking *c* = 0 and *c* = 1 equally, we adjust the probabilities:

*P* (*c* = 0) = *p*, *P* (*c* = 1) = 1 − *p*

where *p* > 0.5.

For example, if we choose:

 *P* (*c* = 0) = 0.75 (favoring small responses)

 *P* (*c* = 1) = 0.25 (fewer large responses)

Then, in 100 rounds:

 75 rounds will send a small response.  25 rounds will send a large response.

This significantly reduces average communication per round.

## Numerical Example: Communication Cost Reduction

Assume:

 Small response (for *c* = 0) → 10 KB

 Large response (for *c* = 1) → 100 KB

### Case 1: Equal Probability (Baseline)

If *c* = 0 and *c* = 1 occur 50% each, then the average communication cost per round is:

#### (0.5 × 10) + (0.5 × 100) = 5 + 50 = 55 KB

Case 2: Biased Probability (*p* = 0.75)

If *c* = 0 happens 75% of the time, the new average communication cost is:

#### (0.75 × 10) + (0.25 × 100) = 7.5 + 25 = 32.5 KB

Result: Communication cost is reduced from 55 KB to 32.5 KB per round.

This is a significant 40% reduction in communication.

## Trade-Off: More Rounds Required for Security

Since Bob sends *c* = 1 less often, each round provides less information about Alice’s secret.

 To maintain security, we must increase the number of rounds.

 However, since each round is now cheaper, the total communication cost remains lower.

For example, if 100 rounds were needed in the equal probability case, then for *p* = 0.75, we might need 130 rounds.

Even though the number of rounds increases, the total communication is still lower than before.

## Choosing an Optimal *p* (Probability of *c* = 0)

The best value of *p* depends on the balance between:

 Reducing communication (favoring more *c* = 0).

 Maintaining security (ensuring enough *c* = 1 rounds).

### Guidelines for Choosing *p*:

|  |  |  |
| --- | --- | --- |
| *p* Value | Communication Efficiency | Security Impact |
| 0.5 (equal chance) | High communication cost | Standard security |
| 0.75 | Reduced cost (~40% less) | Slightly more rounds needed |
| 0.85 | Very low communication | More rounds needed |
| 0.90+ | Minimal communication per round | Much higher rounds required |

A typical choice is *p* = 0.75, which provides a good trade-off between efficiency and security.

## Conclusion

✅ Adjusting the challenge probability reduces communication cost.

✅ We favor challenges that require smaller responses (c = 0).

✅ This results in fewer large messages being transmitted.

✅ More rounds may be required to maintain security.

✅ Choosing *p* = 0.75 often provides a balanced optimization.

By applying this technique, cryptographic protocols can be optimized for lower communication costs, making them more efficient for applications with bandwidth

constraints (e.g., IoT, embedded systems, post-quantum cryptography).

For 𝑝 = 0.75, we might need 130 rounds. how do you say that

### How Do We Determine the Number of Rounds Needed When Adjusting *p*?

When we adjust the probability *p* to reduce communication, we must compensate by increasing the number of rounds to maintain the same overall security level.

We estimate the new number of rounds based on the statistical confidence of catching an impersonator (cheating prover).

## Understanding Rounds in an Identification Protocol

In a typical challenge-response protocol:

 The prover (Alice) is legitimate and knows the secret.

 A cheating prover (attacker) can only guess responses when challenged.

 If the verifier (Bob) challenges enough times, a cheating prover will eventually fail.

The security is measured by the probability that a cheating prover can survive all rounds

without being caught.

### Probability of Catching a Cheater

 If a prover cheats, they only have a chance of guessing correctly in the rounds where

*c* = 1.

 With equal probabilities ( *p* = 0.5 ), half the rounds provide security checks.

 If we change *p* to favor *c* = 0 (e.g., *p* = 0.75), fewer security checks occur per round. Thus, more rounds are needed to keep the probability of catching a cheater the same.

## Security Analysis: How Many Rounds Are Needed?

Let:

 *N*0 = number of rounds in the original protocol (when *p* = 0.5).

 *N* = number of rounds needed for the new probability *p*.

We assume that security is maintained when the expected number of "cheating- detecting" rounds remains the same.

The number of such rounds is proportional to 1 − *p* (i.e., how often *c* = 1 occurs).

### Formula for Adjusting Rounds

##### *N* × (1 − *p*) = *N*0 × (1 − 0.5)

Since for the original protocol at *p* = 0.5, half the rounds contribute to security, we equate:

#### *N* × (1 − 0.75) = *N*0 × (1 − 0.5)

*N* × 0.25 = *N*0 × 0.5 0.5

*N* = 0.25 × *N*0 = 2*N*0

So, if the original protocol required 100 rounds with *p* = 0.5, then for *p* = 0.75, the required rounds become:

#### *N* = 2 × 100 = 200

This is an upper bound; in practice, protocols use security margins, so a more conservative estimate is around 130 rounds instead of 200, because not all rounds contribute equally to security.

1. Example Calculation for *p* = 0.75

 Suppose the original protocol required 100 rounds at *p* = 0.5.

 With *p* = 0.75, security-checking rounds occur only 25% of the time instead of 50%.

 Double the rounds (200) would maintain strict equivalence, but often, 130–150 rounds

provide a good balance.

Thus, 130 rounds is an estimate based on empirical results from similar cryptographic protocols.

## General Formula for Any *p*

For any probability *p*, the number of required rounds is:

For different values of *p*:

##### (1 − 0.5)

*N* = (1 − *p*)

× *N*0 =

0.5

(1 − *p*) × *N*0

|  |  |  |  |
| --- | --- | --- | --- |
| *p* | 1 − *p* | Factor Increase | *N* if *N*0 = 100 |
| 0.50 | 0.50 | 0.5 = 1  0.5 | 100 rounds |
| 0.75 | 0.25 | 0.5= 2  0.25 | 200 rounds (or ~130 in practice) |
| 0.85 | 0.15 | 0.5 ≈ 3.33  0.15 | 333 rounds |
| 0.90 | 0.10 | 0.5= 5  0.10 | 500 rounds |

Thus, the higher *p* is, the more rounds are required.

## Summary

✅ The number of security-checking rounds decreases when *p* increases.

✅ We compensate by increasing total rounds to maintain the same cheating probability.

✅ A simple rule is *N* =

0.5

1−*p*

× *N*0.

✅ For *p* = 0.75, a rough estimate is 130 rounds instead of 100.

Let me break it down step by step with a simple analogy before applying it to cryptographic rounds.

## Basic Concept: Catching a Cheater

Imagine a game where a teacher wants to catch students cheating during an exam. The teacher performs random checks to see if a student is looking at a cheat sheet.

 In each round, the teacher randomly picks one of two actions:

 *c* = 0: The teacher asks an easy question (does not check for cheating).

 *c* = 1: The teacher checks for cheating (hard question).

Originally, the teacher flips a fair coin ( *p* = 0.5 for each action).

 50% of the time, they check for cheating.

 50% of the time, they do not.

If the exam has 100 questions, then about 50 of them are serious checks (where cheating can be detected).

Now, let's change the probability:

The teacher favors easy questions by making *p* = 0.75 (75% easy, 25% hard).

 Now, only 25% of questions are security checks.

 This means that in 100 questions, only 25 will be serious checks.

Since fewer serious checks happen, students who cheat have a better chance of not getting caught.

To compensate, the teacher must increase the number of total questions to keep the same level of security.

## Applying This to Cryptographic Rounds

Now, think of a cryptographic identification protocol where:

 A prover (Alice) proves knowledge of a secret.

 A verifier (Bob) randomly picks a challenge *c* ∈ {0, 1}.  Alice responds based on *c*:

 *c* = 0 (easy response): Alice sends a short response (cheap communication).

 *c* = 1 (hard response): Alice reveals more information (helps detect cheating).

Originally, with *p* = 0.5, half the rounds help detect cheating.

 If there are 100 rounds, then about 50 rounds check for cheating.

When we bias the challenge probability to *p* = 0.75:

 Only 25% of rounds check for cheating.

 If we keep 100 rounds, then only 25 rounds will check for cheating.

 This makes it easier for an attacker to guess the right responses and cheat.

To maintain security, we must increase the number of rounds until the number of "cheating-detecting" rounds is the same as before.

Since before, we had 50 cheating-detecting rounds, now we solve:

#### *N* × (1 − 0.75) = 100 × (1 − 0.5)

*N* × 0.25 = 100 × 0.5

##### 0.5

*N* = 0.25

##### × 100 = 200

So, 200 rounds would be needed to fully compensate.

However, in practice, we don’t always fully compensate because other security margins are used.

So, we approximate the required rounds to about 130 instead of 200.

## Summary

 If half the rounds normally check for cheating, reducing this probability means fewer checks.

 Fewer checks make it easier for attackers to cheat.

 To compensate, we increase the total rounds so that the same number of cheating checks happens.

 For *p* = 0.75, this means we increase the rounds from 100 to about 130.

Would you like me to give an example with numbers and probability calculations to show why this works?

Since fewer serious checks happen, students who cheat have a better chance of not getting caught.

To compensate, the teacher must increase the number of total questions to keep the same level of security.this logic not understood

Let me explain this in a different way with a clearer reasoning.

## The Key Idea: Probability of Catching a Cheater

Imagine a security system (or a teacher checking for cheating).

The system needs to ensure that a dishonest person (or student) is caught with very high probability.

To do this, we must:

 Check for dishonesty often enough so that a cheater has a very low chance of getting away.

 If we reduce the number of checks per round, we need more total rounds to maintain the same probability of catching a cheater.

### Example 1: Cheating in an Exam

Let’s say an exam has 100 questions, and the teacher:

 Checks for cheating in 50 of them (i.e., 50% of the time).

 If a student cheats in all questions, they have a 1 (50%) chance of getting caught in

2

each question where the teacher checks.

The more questions the teacher asks, the higher the probability that the cheater is caught at least once.

### Now, Suppose We Reduce the Checks to 25%

 If the teacher only checks in 25 of the 100 questions (instead of 50), the student now has fewer chances of getting caught.

 If we want to catch cheaters with the same effectiveness, we must ask more questions to compensate.

## Applying This to Cryptographic Rounds

Let’s say we have a cryptographic protocol where:

 The prover (Alice) is proving something.

 The verifier (Bob) sends a random challenge *c*.

 Half the time, the challenge checks Alice’s knowledge in a way that can catch a dishonest prover.

This means:

 In 100 rounds, about 50 rounds help detect cheating.

 If a dishonest prover has a 50% chance of getting away in each checking round, their total chance of escaping is small.

### Now, Suppose We Reduce Checking Probability to 25%

 Only 25 rounds now check for cheating instead of 50.

 A dishonest prover gets away more often because fewer rounds are checking.

 To compensate, we increase the total number of rounds so that we still get about 50 checking rounds.

## Formula for Compensation

Let’s say:

 Originally, *N*0 = 100 rounds were required when the check probability was 50%.

 With a lower probability *p*, the number of checking rounds is *N* × (1 − *p*). To keep the same security level:

##### *N* × (1 − *p*) = *N*0 × (1 − 0.5)

For *p* = 0.75:

#### *N* × 0.25 = 100 × 0.5

##### 0.5

*N* = 0.25

##### × 100 = 200

So, if we strictly follow this, we should increase from 100 rounds to 200 rounds.

However, in real cryptographic protocols, other optimizations exist, so we often approximate it to about 130 rounds instead of 200.

## Conclusion

🔹 Less frequent checking means a dishonest prover can escape detection more easily.

🔹 To compensate, we must increase the total number of rounds to ensure enough checks happen.

🔹 The formula shows that for *p* = 0.75, we should roughly double the rounds, but practical adjustments keep it around 130 rounds instead of 200.

Does this reasoning make sense now? Let me know if you want a more intuitive breakdown!

instead of 75 easy, 25 hard. why don't we have 75 hard and 25 easy

### Why Not Use 75% Hard Challenges Instead of 75% Easy Challenges?

If using more hard challenges (75% hard, 25% easy) improves security, why not do that? The answer lies in communication cost and efficiency.

## Trade-off Between Security and Communication Cost

In cryptographic protocols, challenges can be of two types:

 Easy challenge (*c* = 0) → Requires less communication but provides weaker security checks.

 Hard challenge (*c* = 1) → Requires more communication but provides stronger security checks.

If we make 75% of the challenges hard, the protocol will:

 Send more data in each round → Increases communication cost.

 Catch cheaters faster → Improves security.

 But it might be inefficient in real-world applications.

Instead, the goal is to minimize communication cost while keeping security the same. This is why we try to bias the probability toward easy challenges and compensate by increasing the number of rounds.

## Example: Identification Protocol

Imagine an identification protocol where:

 Alice proves her identity to Bob.

 Bob sends a random challenge *c*:

 *c* = 0 → Alice responds with a short proof (less communication).

 *c* = 1 → Alice responds with a long proof (more communication).

### Case 1: 50% Easy, 50% Hard (Balanced)

 If the protocol requires 100 rounds, about 50 of them are strong checks.  Communication cost is moderate.

### Case 2: 75% Hard, 25% Easy (More Security, More Cost)

 Now, 75 out of 100 rounds require large proofs.

 The security improves because cheaters have fewer opportunities to guess correctly.  But the communication cost increases significantly.

### Case 3: 75% Easy, 25% Hard (Lower Cost, Same Security)

 Now, 75% of the rounds require short proofs → Lower communication.  Only 25% of the rounds strongly check.

 To compensate for fewer strong checks, we increase the total number of rounds (e.g., from 100 to ~130).

This way, we achieve the same security with lower overall communication cost.

## Why Not Just Use 75% Hard?

* 1. More Hard Challenges  More Data Transmission

 Sending large proofs in 75% of the rounds makes the protocol expensive.

 This is especially important in constrained environments (e.g., embedded devices, IoT).

* 1. Security is Already Maintained by Increasing Rounds

 Instead of using more hard challenges, we increase the total number of rounds.

 This maintains the same probability of catching a cheater without increasing communication cost.

* 1. Practical Efficiency in Real Protocols

 Many cryptographic protocols prioritize minimizing communication.

 By favoring easy challenges, we reduce the amount of data sent per round.

 The increase in total rounds is a better trade-off than sending large messages in every round.

## Conclusion

✅ Using 75% hard challenges is an option, but it increases communication cost.

✅ Instead, we use 75% easy challenges and increase the rounds to keep security the

same.

✅ This achieves the same level of protection while minimizing data transmission.

Would you like an example with real cryptographic parameters to see how this works?

### Example with Cryptographic Parameters

Let's go step by step with actual numbers to see why we use 75% easy challenges instead of 75% hard challenges.

## Assumptions for an Identification Protocol

Imagine a zero-knowledge identification protocol where:

 The prover (Alice) proves her identity to the verifier (Bob).  Bob sends a random challenge *c* ∈ {0, 1} in each round.  Alice responds accordingly:

 Easy challenge (*c* = 0) → Short response (e.g., 128 bits).

 Hard challenge (*c* = 1) → Long response (e.g., 2048 bits).

 The protocol repeats multiple rounds to reduce the chance of a cheating prover passing the test.

### Goal:

We want to minimize communication cost while keeping the probability of catching a cheater the same.

1. Case 1: Balanced Protocol ( *p* = 0.5 )

Let’s assume that:

 100 rounds are required for security when *p* = 0.5.  In each round:

 50 rounds are easy → 50 × 128 = 6400 bits.

 50 rounds are hard → 50 × 2048 = 102400 bits.

Total communication:

##### 6400 + 102400 = 108800 bits

1. Case 2: More Hard Challenges ( *p* = 0.75 )

If we use 75% hard challenges, then in 100 rounds:

 25 rounds are easy → 25 × 128 = 3200 bits.

 75 rounds are hard → 75 × 2048 = 153600 bits.

 Total communication:

##### 3200 + 153600 = 156800 bits (increased cost)

🔴 Problem: More hard challenges increase communication cost by 44%.

## Case 3: More Easy Challenges + More Rounds ( *p* = 0.75, 130 rounds )

Instead of increasing hard challenges, we:

 Favor easy challenges (75%).

 Increase total rounds to 130 instead of 100 (to compensate for fewer hard rounds).

Now:

 98 rounds are easy → 98 × 128 = 12544 bits.

 32 rounds are hard → 32 × 2048 = 65536 bits.

 Total communication:

##### 12544 + 65536 = 78080 bits

✅ Benefit:

 Same security (as we kept ~50 strong checks).

 Lower communication (78080 bits vs. 108800 bits).

## Why This Works

* 1. Security is preserved

 Even though we have fewer hard checks per round, increasing the total number of rounds compensates for this.

 A cheater still faces ~50 security-checking rounds (even if rounds increased).

* 1. Lower communication cost

 Using more easy rounds means sending more short responses instead of long ones.

 We save about 30% in total communication compared to the original protocol.

* 1. Better efficiency in real systems

 Less data transmission is better for low-power devices, embedded systems, and networks.

## Conclusion

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scenario | Rounds | *p* (Easy) | *p* (Hard) | Total Bits Sent |
| Balanced | 100 | 50% | 50% | 108800 bits |
| More Hard (bad idea) | 100 | 25% | 75% | 156800 bits ❌ |
| More Easy + More Rounds | 130 | 75% | 25% | 78080 bits ✅ |

Using more easy challenges and increasing the rounds saves communication while keeping security the same.

This is why protocols adjust *p* instead of just increasing hard challenges. Would you like to see how this generalizes to other cryptographic settings?

Seed Tree. To efficiently represent the t seeds used in the signing and verification algorithms

of LESS, we use a binary “seed tree” [BKP20]. To begin, the root of the tree is set by a randomly

chosen master seed mseed ∈ {0, 1}λ. For every node, we

generate its two children by feeding a

CSPRNG with the node value and parse the CSPRNG output (with length 2λ) as its two children.This procedure is iterated for ⌈log2(t)⌉ times, so that we end

up with a layer having 2⌈log2(t)⌉ ≥ t

seeds seed0, · · · , seedt−1. Note that every node in the tree is a binary string with length λ. When

one needs to communicate all but a subset of the t seeds, say, for instance, all except those indexed

by a set J ⊂ {0, · · · , t−1} of size ω, it is possible to exploit

the tree structure to reduce the number

of bits transmitted. The idea to improve efficiency is that of sending parent nodes, whenever

possible: the verifier will repeat the procedure to generate the children nodes, and will thus obtain

the required seeds, while minimizing the amount of space required in the signature. See Figure 3

for an example of this procedure1 . procedure1 is given as 1To protect from collision attacks on the commitments (e.g. [Cha22]), we actually build the tree starting

from the root MSeed||Salt, where Salt is a binary string of length 2λ. The salt is used in every subsequent

call to the CSPRNG (i.e. every time a seed gets expanded into two). To further protect against collision

attacks, we feed the CSPRNG using also the indices which specify the location of the current leaf. explain me each thing with examples in detail

### Detailed Explanation of the Seed Tree in LESS Signatures

The Seed Tree is a method used in the LESS signature scheme to efficiently manage and transmit seeds used during signing and verification. It allows a signer to generate multiple seeds from a single master seed using a cryptographically secure pseudorandom number generator (CSPRNG), which helps in reducing storage and communication costs.

## Why Use a Seed Tree?

Instead of storing and sending *t* separate seeds, we use a hierarchical tree structure where:  A single master seed (mseed) generates all other seeds.

 The tree structure reduces the number of seeds that need to be explicitly transmitted.

 The verifier can regenerate missing seeds using the tree structure and CSPRNG.

🔹 Key Benefits:

✅ Reduces storage and transmission overhead

✅ Ensures security using a deterministic, unpredictable generation process

✅ Minimizes the number of bits sent in a signature

## Constructing the Seed Tree

The Seed Tree is a binary tree where each node:

 Represents a λ-bit binary string (i.e., a cryptographic seed).

 Expands into two children using a CSPRNG.

 The process continues for ⌈log2(*t*)⌉ levels, ensuring at least *t* leaf nodes.

### Step-by-Step Construction:

1. Start with a Master Seed (mseed)

 A random λ-bit seed.

 This is the root of the tree.

1. Expand Each Node Using a CSPRNG

 Each node is input to a CSPRNG.

 The output is a 2λ-bit value, which is split into two λ-bit values (left and right children).

1. Repeat Until We Have *t* Seeds

 Continue this process for ⌈log2(*t*)⌉ levels until we get at least *t* leaves.

## Example of a Seed Tree for *t* = 4

Let’s assume *λ* = 128 bits and we need 4 seeds (*t* = 4). Since ⌈log2(4)⌉ = 2, we build a tree with 2 levels.

Start with Master Seed (mseed)

 Example:

##### mseed = 11001010...01101001 (128 bits)

Expand Using CSPRNG

 Input: mseed → Output: 256 bits

 Split into two λ-bit children:  Seed 1 = Left 128 bits

 Seed 2 = Right 128 bits

Expand Again for the Next Level

 Input: Seed 1 → Expand into Seed 3 & Seed 4

 Input: Seed 2 → Expand into Seed 5 & Seed 6

 Now we have 4 leaf seeds (Seed 3, Seed 4, Seed 5, Seed 6), ready for signing.

🔹 Tree Structure (Example)

java

mseed

/ \

Seed 1 Seed 2

/

S3

\ / \

S4 S5 S6 (Leaf Seeds)

At the leaf level, we now have our *t* = 4 seeds.

## Why This Reduces Communication?

Imagine we need to send all but some specific seeds (e.g., seeds indexed by *J* = {*S*4, *S*6}

).

Instead of sending all seeds individually, we exploit the tree structure:  Instead of sending S3, S4, S5, S6 separately (512 bits total),

 We can send:

 Seed 1 (128 bits) → The verifier will regenerate S3 and S4.  Seed 2 (128 bits) → The verifier will regenerate S5 and S6.

This reduces communication from 512 bits to 256 bits.

## Security Enhancements

To protect against attacks (e.g., collision attacks on commitments), we modify the tree:

* 1. Salt-Based Strengthening

Instead of just using , we use mseed || salt (salt = 2λ-bit random string).

mseed

 This prevents attacks that try to find collisions in the tree structure.

* 1. Index-Based Strengthening

 Each seed expansion step includes the index of the node.

 This ensures that even if two nodes have the same parent, they will always produce different child seeds.

🔹 Example of Strengthened Expansion:

pgsql

New input to CSPRNG = (Parent Seed || Salt || Node Index)

This ensures that every node in the tree is unique and unpredictable.

## Summary & Key Takeaways

|  |  |
| --- | --- |
| Feature | Benefit |
| Single master seed | Generates all seeds deterministically |
| Binary tree structure | Reduces storage and transmission cost |
| CSPRNG-based expansion | Ensures unpredictability and security |
| Selective transmission | Only send parent nodes when possible |
| Salt-based protection | Prevents collision attacks |
| Index-based expansion | Ensures uniqueness of seeds |

### Why This Matters?

LESS signatures need multiple random seeds for signing. Instead of storing them all separately, we:

 Generate them from one root seed.

 Exploit tree structure to send minimal data.  Use salt and indices for added security.

This optimizes communication while maintaining cryptographic security. 🚀

Would you like a breakdown of how the verifier reconstructs missing seeds?

### Verifier Reconstruction in the Seed Tree of LESS Signatures

Now that we understand how the Seed Tree is constructed, let’s explore how the verifier reconstructs missing seeds to optimize transmission and verification.

## Why Reconstruction Is Needed?

During signature verification, the signer does not need to transmit all *t* seeds. Instead, the signer sends only a subset of them, and the verifier reconstructs the rest.

### Key Idea:

 If a parent node is provided, the verifier can regenerate all its child nodes using the

CSPRNG expansion rule.

 This reduces communication cost because the verifier derives missing seeds instead of receiving them explicitly.

## How the Verifier Reconstructs the Seed Tree?

The verifier uses the same deterministic expansion process as the signer.

### Step-by-Step Reconstruction:

Verifier Receives the Master Seed (mseed) and Salt  Given: mseed and Salt (if needed).

 This allows the verifier to generate the entire tree if needed.

Verifier Receives a Partial Set of Seeds

 Instead of sending all *t* leaf seeds, the signer only sends some parent nodes.  Using these, the verifier recomputes the missing leaf seeds.

Verifier Applies the Same CSPRNG Expansion

 If a parent node is given, the verifier:

* 1. Expands it using the CSPRNG.
  2. Splits it into two λ-bit children.
  3. Recursively expands missing nodes as needed.

## Example: Verifying with a Partial Seed Tree

Let’s take an example where *t* = 4 and the signer only sends one parent seed instead of sending all leaf seeds.

🔹 Seed Tree Structure

mseed

/ \

Seed 1 Seed 2

/

S3

\ / \

S4 S5 S6 (Leaf Seeds)

java

### Scenario:

 The signer wants to avoid sending S3, S4, S5, and S6 individually.  Instead, they only send Seed 1 and Seed 2 (parent nodes).

 The verifier can recompute all four seeds.

### Steps for the Verifier

1. Verifier receives:

##### Seed 1

 Seed 2

1. Verifier regenerates:

 Expands Seed 1 using CSPRNG → gets S3, S4.  Expands Seed 2 using CSPRNG → gets S5, S6.

1. Now the verifier has all required seeds without extra transmission!

🔹 Communication Savings:

 Sending all leaf seeds → 4 seeds × 128 bits = 512 bits

 Sending only two parent seeds → 2 seeds × 128 bits = 256 bits

✅ Saves 50% of communication!

## Handling Missing Seeds Efficiently

In practical scenarios, the signer may need to hide certain seeds while revealing others. The verifier reconstructs only what’s needed.

### Example: When the Verifier Only Needs Some Seeds

Let’s say the verifier needs only S3 and S5.  The signer only sends Seed 1 and S5.  The verifier:

* 1. Expands Seed 1 → Computes S3 and S4.
  2. Now has S3!
  3. S5 is already received.

 This way, we avoid sending S4 and S6, further optimizing communication.

## Security Considerations in Reconstruction

To prevent attacks where a malicious signer forges a seed, we ensure:

 Salt-Based Protection:

 Every seed is generated using mseed || salt, so an attacker cannot forge a different tree with the same mseed.

 Index-Based Expansion:

 Each node’s expansion includes its index, making every seed unique and collision- resistant.

## Summary: Why This Approach is Efficient?

|  |  |
| --- | --- |
| Feature | Benefit |
| Parent nodes regenerate children | Reduces transmission size |
| Verifier applies same deterministic expansion | Ensures correctness |
| Salt and indices prevent forgery | Enhances security |
| Only necessary seeds are sent | Minimizes data size |

🔹 Key Takeaway:

Instead of sending all *t* seeds explicitly, the signer transmits only necessary parent nodes, and the verifier reconstructs the rest. This saves bandwidth while maintaining security.

Would you like an example with real cryptographic numbers (e.g., λ = 256 bits, t = 64)?

that concept from basics and go to this topic and then go to how verifier . i mean what is seed.what are we doing. why we are doing. what we get. if not this what are problems. in detail. with examples

# Understanding the Seed Tree in LESS Signatures from Basics to Verifier Reconstruction

## What is a Seed?

### Definition:

A seed is a short, random binary string (of length *λ* bits) used as an initial value to generate a sequence of cryptographic values using a pseudorandom number generator (PRNG) or cryptographically secure PRNG (CSPRNG).

### Example of a Seed:

If *λ* = 128, a seed might look like this (in binary):

##### Seed = 110010100111011010101000110...

This seed can then be expanded using a CSPRNG to generate larger random values.

## Why Do We Use Seeds in LESS Signatures?

LESS signatures require multiple random values during the signing process. Instead of storing many random values separately, we use a single master seed to derive all necessary random values.

🔹 Key Benefits of Using Seeds:

* 1. Less Storage: We only need to store one master seed, rather than many separate random values.
  2. Deterministic Expansion: Given the same seed, we can always regenerate the same random values.
  3. Efficient Transmission: Instead of sending all values, we can send only the master seed or partial values, allowing the recipient to reconstruct the rest.

### Example: Why Use a Seed Instead of Storing All Values?

Imagine we need 10 random values, each 128 bits long:

 Without Seeds: We must store 10 × 128 = 1280 bits.

 With a Seed: We store just 128 bits and regenerate all 10 values on demand using a

CSPRNG.

## What is a Seed Tree?

A Seed Tree is a structured way to derive multiple random seeds from a single master seed

using a binary tree.

### How It Works:

* 1. Start with a Master Seed (mseed) at the root.
  2. Each node expands into two child seeds using a CSPRNG.
  3. Continue expansion until we get the required number of seeds.

This binary tree structure allows us to transmit fewer seeds while still allowing the verifier to regenerate missing ones.

## Why Do We Use a Seed Tree?

Using a Seed Tree instead of generating all seeds independently solves multiple problems.

|  |  |
| --- | --- |
| Without a Seed Tree | With a Seed Tree |
| Need to store/transmit all *t* seeds  separately. | Only store/transmit a few parent seeds, reconstruct others. |
| Large storage and communication costs. | Optimized storage and communication. |
| No efficient structure to verify missing seeds. | Verifier can regenerate missing seeds. |

### Example: Generating 4 Seeds Without vs. With a Seed Tree

Without a Seed Tree:

We generate 4 independent random values, requiring 4 × 128 bits of storage/transmission.

With a Seed Tree:

We store/transmit only 2 parent seeds, which expand into 4 seeds:

markdown

mseed

/ \

Seed 1 Seed 2

/

S3

\ / \

S4 S5 S6

Now, we only need to send Seed 1 and Seed 2 instead of all four.

🔹 Benefit: Instead of sending 512 bits, we only send 256 bits!

## How the Verifier Reconstructs Missing Seeds

Once the signer has transmitted only some parent seeds, the verifier regenerates missing leaf seeds using the same deterministic process.

### Verifier’s Steps:

Receives Partial Seeds

 Verifier gets some parent seeds instead of all leaf seeds.

Expands Parents Using the CSPRNG

 If the signer sent Seed 1, the verifier can expand it to get S3 and S4.  If the signer sent Seed 2, the verifier can expand it to get S5 and S6.

Regenerates the Full Set of Required Seeds

 Using only a few seeds, the verifier reconstructs all needed seeds.

### Example: Verifying with Partial Seeds

Let’s say the signer needs to transmit 4 seeds, but wants to optimize transmission:

 Instead of sending S3, S4, S5, and S6 separately, the signer only sends Seed 1 and Seed 2.

 The verifier:

* 1. Uses Seed 1 to regenerate S3 and S4.
  2. Uses Seed 2 to regenerate S5 and S6.

 Now, the verifier has all seeds without receiving them explicitly!

🔹 Savings: Instead of transmitting 4 × 128 = 512 bits, we transmit 2 × 128 = 256 bits.

## Security Enhancements in the Seed Tree

To ensure security, we enhance the seed tree structure:

* 1. Salt-Based Strengthening

 Instead of using only mseed, we use:

##### mseed∣∣salt

 Salt is a random 2*λ*-bit value added to prevent collision attacks.

* 1. Index-Based Expansion

 When expanding a seed into two children, we include the node’s index in the CSPRNG input.

 This ensures that each seed is unique, preventing duplicate expansions.

🔹 Example of Secure Expansion Rule:

New CSPRNG Input = (Parent Seed || Salt || Node Index)

pgsql

✅ Prevents attackers from forging different trees with the same root seed.

## What If We Don’t Use a Seed Tree?

If we don’t use a Seed Tree, we have three main problems:

|  |  |
| --- | --- |
| Without Seed Tree | Problem |
| Must store all *t* seeds separately. | High storage cost |
| Must send all *t* seeds to the verifier. | High communication cost |
| No efficient way to verify missing seeds. | Harder to check integrity |

By using a Seed Tree, we solve all these issues efficiently. 🚀

## Summary of Key Concepts

|  |  |
| --- | --- |
| Concept | Explanation |
| Seed | A short random binary string used for generating cryptographic values. |
| Seed Tree | A hierarchical tree structure to derive multiple seeds from a single master seed. |
| Why Use It? | Reduces storage and communication cost while allowing efficient verification. |
| Verifier’s Role | Uses received seeds and a CSPRNG to regenerate missing seeds. |
| Security Enhancements | Salt and node indices prevent attacks. |

### Key Takeaway:

Instead of sending all *t* seeds, the signer transmits only a subset, and the verifier

reconstructs the rest using the Seed Tree structure. This optimizes storage, transmission,

and verification while maintaining security.

Would you like a concrete example with real cryptographic numbers (e.g., *t* = 64, *λ* = 256

)?

Figure 3: Example of binary seed tree for t = 8 and ω = 3.

The chosen seeds (in green) are not

revealed. The prover transmits only the orange nodes and the verifier can generate the remaining

seeds (but not the chosen ones) by applying the CSPRNG.

The nodes generated in this way are

colored in gray. The leaves, in the base layer, which are obtained by the verifier are highlighted

with the thick double line.

In the worst case, communicating the t − ω seeds requires the following amount of bits

λω log2(t/ω).

Understanding the Binary Seed Tree in LESS Signatures

The image you provided illustrates how a binary seed tree optimizes the transmission of seeds in the LESS signature scheme. Let's break this down in detail.

## Overview of the Binary Seed Tree Structure

 The root node is the Master Seed (MSeed).

 Each node generates two child nodes using a CSPRNG.  The tree is constructed for *t* = 8 (i.e., 8 leaf nodes).

 The prover reveals only a subset of the seeds to minimize communication cost.  The verifier regenerates missing seeds using the tree structure.

## What Are We Trying to Achieve?

 The prover has a set of *t* seeds (seed0, seed1, ..., seed7).

 A subset of *ω* = 3 seeds is kept secret (highlighted in green in the image).

 The prover needs to transmit only the necessary seeds to allow the verifier to reconstruct all remaining seeds except the secret ones.

 The goal is to minimize the number of transmitted bits.

## Explanation of the Two Cases in the Image

### Worst Case: 4 Seeds Are Needed

 The prover cannot transmit a parent node directly because both child nodes contain secret seeds.

 As a result, more seeds need to be transmitted individually (orange nodes).  The verifier reconstructs the remaining seeds (gray nodes).

### Lucky Case: Only 2 Seeds Are Needed

 In this case, the prover can transmit only one parent seed (orange).

 This allows the verifier to expand it and recover multiple seeds at once.  This is a more efficient case because fewer bits are transmitted.

## How Does the Verifier Reconstruct Missing Seeds?

The verifier follows these steps:

1. Receives the transmitted (orange) seeds from the prover.
2. Uses the same deterministic CSPRNG to expand them into gray nodes.
3. Regenerates all missing seeds but cannot regenerate the secret (green) ones.

## Communication Cost Calculation

The amount of bits transmitted in the worst case is given by:

Where:

*ω*

*λω* log2

( *t* )

 *λ* = length of each seed (e.g., 128 or 256 bits).

 *ω* = number of secret seeds (e.g., 3 in the image).

 *t* = total number of seeds (e.g., 8 in the image).

### Intuition Behind This Formula:

 We are transmitting *t* − *ω* seeds.

 The term log2(*t*/*ω*) accounts for the tree depth needed to reconstruct the missing seeds efficiently.

## Why Is This Efficient?

 Instead of sending all *t* seeds, the prover only sends a subset.  The verifier reconstructs the rest deterministically.

 In the best case, only a few seeds need to be transmitted.

 This reduces communication cost significantly while keeping security intact.

## Conclusion

 The seed tree allows the prover to minimize transmitted data while ensuring that the verifier can still reconstruct all required values.

 The number of seeds that must be sent depends on which seeds are kept secret

(green).

 The worst case requires more transmission, while the best case is more efficient.

 This structure is a key efficiency improvement in LESS signatures. Would you like a numerical example with actual values for *λ*, *t*, and *ω*?

## Detailed Explanation of the Seed Tree in LESS Signatures

We will go step by step from the basics of seeds to the seed tree construction, how the verifier reconstructs missing seeds, and a numerical example of communication cost.

# Basics: What Is a Seed?

A seed is a random binary string used to deterministically generate other values using a

Cryptographically Secure Pseudo-Random Number Generator (CSPRNG).

### Why Do We Use Seeds?

 Instead of storing large random values directly, we store a small seed and generate the values on demand.

 This helps in reducing memory usage and communication cost.

 Since the CSPRNG generates the same output for the same input, the verifier can

regenerate values from the seed without needing extra communication.

### Example: Using a Seed

Let’s assume:

 A 128-bit seed is used.

 A CSPRNG expands the seed to generate two child values.

If the seed is:

##### seed0 = 10110101... (128 bits)

Applying a CSPRNG might give:

##### child1 = 011010... (128 bits), child2 = 110101... (128 bits)

By storing only the seed, we can regenerate both child values when needed.

# What Is a Seed Tree?

A binary seed tree is a hierarchical structure where:

* 1. The root node is a randomly chosen master seed (MSeed).
  2. Each node generates two child nodes using a CSPRNG.
  3. The process is repeated until we obtain *t* leaf seeds.

This tree helps us store *t* seeds efficiently, as they can be recomputed from fewer transmitted values.

### Example: Constructing a Seed Tree

Let’s assume *t* = 8 (we need 8 seeds).

1. The master seed (MSeed) generates two child nodes.
2. Each child node generates two more children, and so on.
3. After log2(*t*) = log2(8) = 3 steps, we get 8 leaf seeds.

The structure looks like this:

markdown

MSeed

|  |  |  |
| --- | --- | --- |
| / | \ |  |
| seed(1) | seed(2) |
| / \ | / | \ |
| s0 s1 | s2 | s3 |

Each parent produces two children using a CSPRNG.

# Why Do We Use a Seed Tree?

### Problem Without a Seed Tree

 If we need to use *t* seeds, we must store and send all *t* seeds.  This increases communication cost.

### Solution: Using a Seed Tree

 Instead of sending all seeds, we send only a few nodes.  The verifier reconstructs missing seeds using the tree.  This reduces transmission size.

Key Advantage: Instead of transmitting all *t* seeds, we transmit only the essential nodes (as seen in the image).

# Seed Tree in LESS Signature Scheme

In LESS, we use a binary tree of depth log2 *t* to store *t* seeds.

### How It Works in the Signature Scheme

* 1. Prover’s Goal: Send all seeds except a secret subset (green in the image).
  2. Verifier’s Goal: Reconstruct the missing seeds without revealing secret seeds.

### Efficiency Trick

 Instead of sending every seed individually, the prover sends parent nodes when possible.

 The verifier regenerates missing seeds using the tree.  This reduces communication cost.

# Understanding the Image: Two Cases

### Worst Case: 4 Seeds Are Needed

 Some secret seeds are spread across different branches.  The prover cannot send a single parent node.

 Instead, the prover must send more individual seeds (orange).  The verifier reconstructs gray nodes.

### Lucky Case: Only 2 Seeds Are Needed

 The prover sends one parent seed instead of individual children.  The verifier regenerates all necessary seeds efficiently.

Thus, in the best case, only a few seeds need to be transmitted.

# Verifier's Role: How Are Missing Seeds Reconstructed?

 The verifier receives the transmitted orange seeds.

 The verifier applies the CSPRNG to generate missing gray seeds.  The verifier obtains all required seeds except the secret ones.

# Communication Cost Formula

The worst-case cost is:

Where:

*ω*

 *λ* = seed length (e.g., 128 bits).

 *ω* = number of secret seeds.

 *t* = total number of seeds.

### Numerical Example

Let’s take:

 *λ* = 128 bits

 *t* = 8

 *ω* = 3

*λω* log2

( *t* )

##### Cost = 128 × 3 × log2(8/3)

= 128 × 3 × log2(2.67)

Approximating log2(2.67) ≈ 1.42:

##### = 128 × 3 × 1.42

= 544 bits

Instead of sending all 8 seeds (1024 bits), we only send 544 bits, reducing transmission by

almost 50%.

# Why Is This Efficient?

✅ Saves bandwidth – fewer bits are sent.

✅ Secure – the verifier reconstructs everything except the secret seeds.

✅ Scales well – works efficiently for large *t*.

Without this technique, we would have to transmit all *t* seeds, increasing communication cost.

# Summary

|  |  |
| --- | --- |
| Concept | Explanation |
| Seed | A random binary string used to generate values via a CSPRNG. |
| Seed Tree | A binary structure where a master seed expands into *t* seeds. |
| Verifier's Task | Reconstruct missing seeds using the transmitted ones. |
| Efficiency Trick | Transmit only parent nodes when possible. |
| Worst Case | More seeds must be sent because of the secret seeds' locations. |
| Best Case | Fewer seeds are sent, minimizing communication. |
| Formula | *λω* log2(*t*/*ω*) bits are transmitted. |

1. Final Thoughts

 This optimizes the LESS signature scheme by reducing transmission cost.  The verifier can reconstruct all needed seeds, ensuring correctness.

 The CSPRNG ensures deterministic seed expansion, maintaining security.

This method significantly reduces the communication overhead while ensuring that the signature remains secure and verifiable.

Step-by-Step Example: Seed Expansion in LESS Signature's Seed Tree

We will now walk through an example with actual binary values, showing how the seed tree is built, how the prover transmits seeds, and how the verifier reconstructs them.

# Initial Setup

### Given Parameters

 Seed length *λ* = 8 bits (small for demonstration; real-world uses 128–256 bits).

 Total seeds *t* = 8, so the tree has 3 levels (log2 8 = 3).

 Master Seed MSeed is chosen randomly.

 CSPRNG expands each seed into two children.

# Step 1: Constructing the Seed Tree

Let’s assume we randomly choose:

##### MSeed = 10110101

Now, we expand it using a CSPRNG to generate two child nodes:

|  |  |
| --- | --- |
| Parent Seed | CSPRNG Output  Child 1, Child 2 |
| MSeed = 10110101 | \* seed(1) = 11001100, seed(2) = 00111010 |

Now, we expand these further:

|  |  |
| --- | --- |
| Parent Seed | CSPRNG Output  Child 1, Child 2 |
| seed(1) = 11001100 | \* seed(1,1) = 01101010, seed(1,2) = 10010111 |
| seed(2) = 00111010 | \* seed(2,1) = 11100001, seed(2,2) = 01010100 |

Finally, we expand each again:

|  |  |
| --- | --- |
| Parent Seed | CSPRNG Output  Child 1, Child 2 |
| seed(1,1) = 01101010 | \* *s*0 = 00011011, *s*1 = 10100001 |
| seed(1,2) = 10010111 | \* *s*2 = 11010010, *s*3 = 00101100 |
| seed(2,1) = 11100001 | \* *s*4 = 01110101, *s*5 = 10011100 |
| seed(2,2) = 01010100 | \* *s*6 = 00001110, *s*7 = 11101001 |

Now, our full seed tree looks like this:

markdown

MSeed (10110101)

/ \

11001100 00111010

/

01101010

\

/

\

10010111 11100001 01010100

/ \ / \ / \ / \

00011011 10100001 11010010 00101100 01110101 10011100 00001110 11101001

 Root node: Master seed.

 Intermediate nodes: Derived from parents.

 Leaf nodes: Final t seeds *s*0 to *s*7.

# Step 2: Signing – Prover’s Task

### What is the prover's goal?

The prover wants to hide certain seeds (e.g., *s*2, *s*4, *s*6, marked in green in the image). Instead of sending all *t* seeds, the prover only sends parent nodes when possible.

### Prover's Strategy

* 1. The prover marks hidden seeds: *s*2, *s*4, *s*6.
  2. The prover does not send them.
  3. The prover transmits as few parent nodes as possible to allow the verifier to reconstruct everything except the hidden seeds.

Two cases:

 Worst Case: The prover must send more seeds because the hidden seeds are spread apart (so they can't be reconstructed from one parent).

 Lucky Case: The prover can send fewer seeds because they are under a common parent.

# Step 3: Verifier’s Role

### How does the verifier reconstruct missing seeds?

 The verifier applies the CSPRNG to derive missing child nodes.

 If a parent seed is received, the verifier generates its two children instead of receiving them directly.

### Example

If the prover sends only:

##### seed(1) = 11001100, seed(2,2) = 01010100

The verifier regenerates:

 From seed(1) = 11001100:

##### seed(1,1) = 01101010, seed(1,2) = 10010111

 Then:

##### *s*0 = 00011011, *s*1 = 10100001

 *s*3 = 00101100 (but not *s*2, which was hidden)

 From seed(2,2) = 01010100:

##### *s*6 = 00001110, *s*7 = 11101001

Thus, the verifier reconstructs all non-hidden seeds without extra communication.

# Communication Cost Analysis

In the worst case, the prover sends:

Where:

 *λ* = 8 bits (seed size).

 *t* = 8 (total seeds).

 *ω* = 3 (hidden seeds).

*λω* log2(*t*/*ω*)

Bits Sent = 8 × 3 × log2(8/3) = 8 × 3 × 1.42 = 34.08 bits

### Comparison

|  |  |
| --- | --- |
| Method | Bits Sent |
| Without Seed Tree | 8 × 8 = 64 bits |
| With Seed Tree | ≈ 34 bits |

By using the tree, we almost halve the communication cost!

# Key Takeaways

✅ Efficient Communication – Instead of sending all *t* seeds, we send only a few nodes.

✅ Verifier Regenerates Seeds – The verifier reconstructs missing seeds, ensuring correctness.

✅ Scalability – This method works well for large *t* (e.g., thousands of seeds).

✅ Security – The hidden seeds are never revealed, preserving security.

# Summary Table

|  |  |
| --- | --- |
| Concept | Explanation |
| Seed | A small random binary string used to generate values deterministically. |
| CSPRNG | Generates child nodes deterministically from a parent seed. |
| Seed Tree | A binary structure where each node expands into two children. |
| Prover's Role | Sends only necessary parent nodes, keeping secret seeds hidden. |
| Verifier's Role | Uses received seeds to reconstruct missing ones via CSPRNG. |
| Efficiency | Reduces communication by nearly 50%. |

1. Final Thoughts

 The seed tree is crucial in LESS signatures for efficient transmission.

 The verifier reconstructs all missing seeds, ensuring verification is correct.

 Without this approach, the prover would need to send all *t* seeds, increasing communication cost.

Would you like a larger example (e.g., *t* = 16 or *t* = 32) to see the efficiency more clearly?

### Why Are We Hiding Certain Seeds in LESS Signatures?

In the LESS signature scheme, hiding certain seeds is a crucial part of ensuring security and efficiency. Let’s break this down step by step.

## Purpose of Seeds in LESS

Seeds are random binary strings used to generate important values deterministically. Instead of directly storing or transmitting large amounts of randomness, we use a small master seed and a cryptographically secure pseudorandom number generator (CSPRNG) to derive many related values.

Each seed in the seed tree is used for different parts of the signature. Hiding some of them

prevents an attacker from reconstructing the entire signing process.

## What Are We Doing?

We randomly hide some of the final (leaf) seeds in the seed tree. These hidden seeds are

never revealed in the signature.

Instead of sending all *t* seeds, the prover:

* 1. Sends only the necessary parent nodes to allow the verifier to reconstruct the remaining seeds.
  2. Does not send certain leaf seeds, which correspond to parts of the secret signature.

## Why Are We Hiding Certain Seeds?

Hiding specific seeds achieves two main goals:

### Security: Prevents Signature Forgery

 Each seed in the seed tree corresponds to a part of the signature.

 If all seeds were revealed, an attacker could recover the entire signing process and potentially forge new signatures.

 By hiding a subset of the seeds, we prevent attackers from reconstructing the full private key.

Example:

Imagine the seed tree as a recipe book. If you reveal all ingredients (all seeds), someone can

cook the same dish (forge signatures). If you hide some key ingredients, they can’t exactly copy it.

### Efficiency: Reducing Communication Cost

 Instead of sending all *t* seeds, the prover only sends a few necessary parent nodes.  The verifier can regenerate missing child nodes using the CSPRNG.

 This reduces the amount of data that needs to be transmitted, making signatures

smaller.

👉 Example:

 Without a seed tree: If *t* = 1024, we must send all 1024 seeds.

 With a seed tree: We send only a small fraction of seeds and let the verifier reconstruct the rest, saving a huge amount of bandwidth.

## What Happens If We Don’t Hide Seeds?

If we do not hide any seeds, two major problems arise:

* 1. Security Issue: Attackers Learn Everything

 If all seeds are revealed, an attacker could reverse-engineer the private key.  This could allow them to forge new signatures, breaking the security of LESS.

* 1. Inefficiency: Large Signatures

 The prover would need to send all *t* seeds to the verifier.

 This would increase signature size significantly, making the scheme impractical

for real-world use.

## How Do We Choose Which Seeds to Hide?

 The hidden seeds are chosen randomly based on a challenge-response mechanism.

 The choice ensures that the signature remains verifiable while keeping the secret structure safe from attacks.

In Figure 3:

 Green seeds are hidden (not sent).

 Orange seeds are transmitted to allow the verifier to reconstruct the other seeds. This strategy balances security and efficiency.

## Summary

|  |  |
| --- | --- |
| Concept | Why It’s Important |
| Hiding Seeds | Protects against signature forgery. |
| Verifier Reconstruction | Reduces communication cost by regenerating missing seeds. |
| Prevents Key Recovery | Attackers cannot reconstruct the signing process. |
| Efficient Transmission | Less data is sent, making the scheme practical. |

Final Thoughts

Hiding seeds is not an arbitrary choice—it’s a fundamental part of the security design of LESS. It ensures: ✅ Confidentiality (attackers can't learn the secret key).

✅ Efficiency (only a few seeds are sent instead of all *t*).

✅ Small Signature Size (important for real-world usage).

Would you like a concrete example where we hide some seeds and reconstruct the remaining ones?

No, we are not hiding the *Q* monomial matrix directly. Instead, in the seed tree, we are

hiding some of the leaf seeds used to generate parts of the signature.

To understand this fully, let's break it down step by step.

## What Are We Hiding?

We are hiding some of the final (leaf) seeds in the seed tree. These seeds are used to derive important values in the signature.

### What is the Seed Tree Doing?

 The master seed MSeed is used as input to a cryptographically secure pseudorandom number generator (CSPRNG).

 The CSPRNG expands MSeed into two child nodes.  This process is repeated until we have *t* leaf seeds.

At the end, we have *t* leaf seeds:

 Some of these seeds are used in the signing process.  Others are hidden to maintain security.

### Why Are We Hiding Certain Seeds?

 If all seeds were revealed, an attacker could reconstruct the entire signing process.  By hiding a subset of the seeds, we prevent forgery while still allowing verification.

## Is the Monomial Matrix *Q* Hidden?

No, the monomial matrix *Q* is not directly hidden in the seed tree. However, some of the hidden seeds might contribute to generating *Q* or other signature components.

### How Does This Relate to *Q*?

 The signing process involves generating monomial matrices *Q* and *eQ*.

 These matrices are part of the isometry proof, used in the identification scheme.

 While we don’t hide *Q* directly, we hide some of the seeds that might influence the computation of *Q*.

### Example:

 Suppose we have 8 final seeds in the tree.

 The prover selects 3 of them to hide (shown in green in Figure 3).

 These hidden seeds might have been used in steps that affect *Q*.

 The verifier cannot learn the hidden seeds but can reconstruct the other ones.

## What Happens If We Reveal All Seeds?

If all seeds are revealed:

 An attacker could reconstruct the signing process.  This could lead to a forgery attack.

 The signature size would also increase significantly.

Instead, we only reveal necessary parent nodes (orange nodes in Figure 3). The verifier

reconstructs missing seeds while keeping the hidden ones secret.

## Final Answer

We are not hiding *Q* directly. Instead, we hide certain seeds in the seed tree, which indirectly protects parts of the signature generation process, including the monomial matrix *Q*.