Single Secret Leader Election

Dan Boneh   Saba Eskandarian   Lucjan Hanzlik   Nicola Greco

What is Single Secret Leader Election?

A group of participants want to randomly choose *exactly one* leader, such that:

1. Identity of the leader is known only to the leader and nobody else
2. Leader can later publicly prove that she is the leader

Should work even if many registered participants don’t send messages.

What is Single Secret Leader Election?

A group of participants want to randomly choose *exactly one* leader, such that:

1. Identity of the leader is known only to the leader and nobody else
2. Leader can later publicly prove that she is the leader

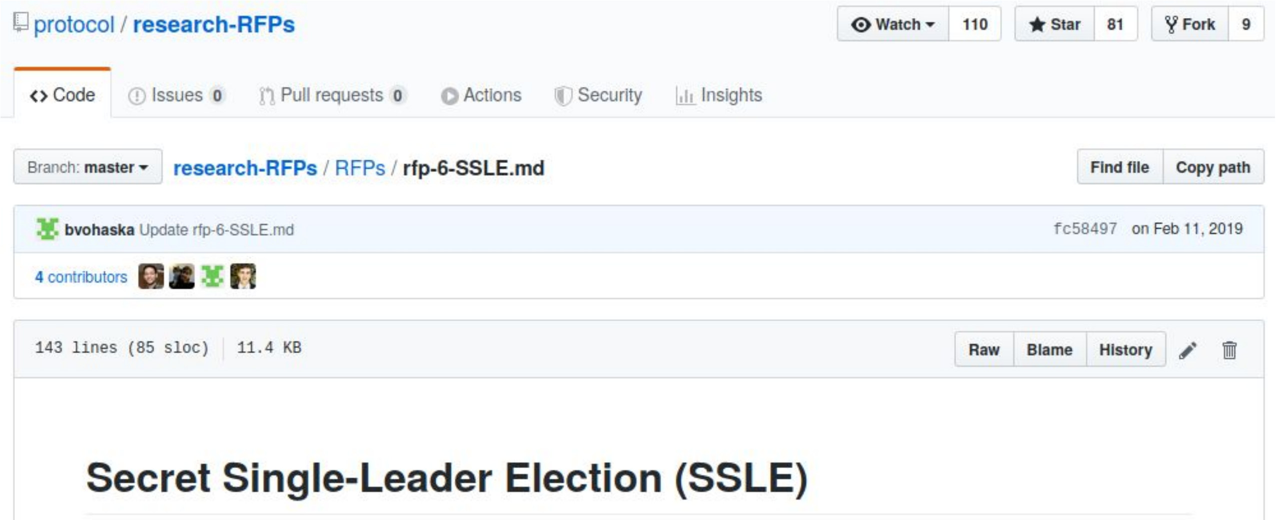
Should work even if many registered participants don’t send messages.

Applications of SSLE - PoS Blockchains

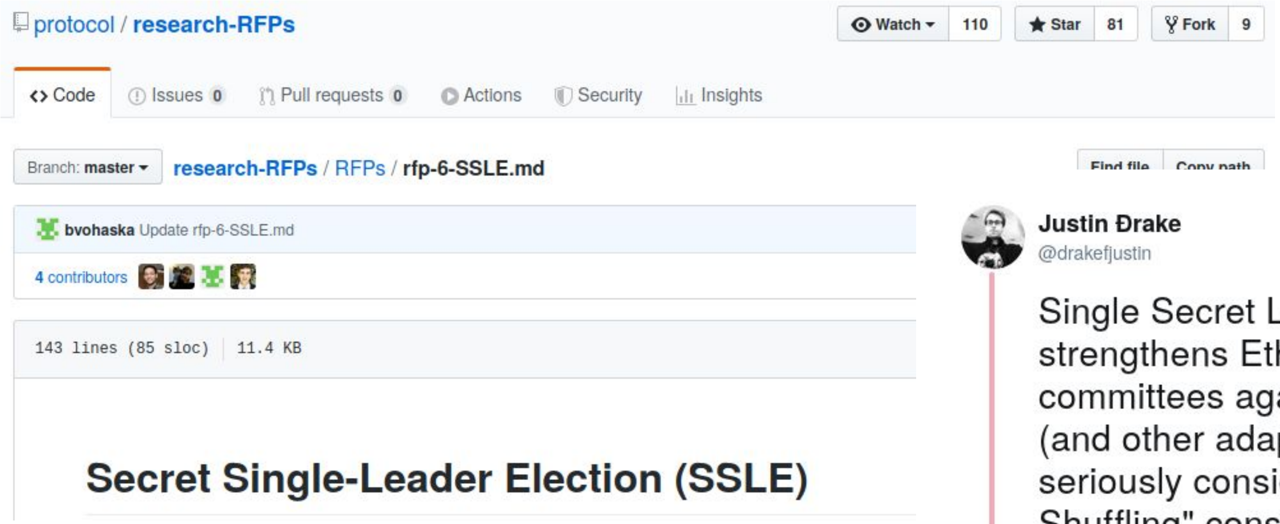
Need leader to submit blocks

Publicizing leader ahead of time makes the whole protocol vulnerable

Applications of SSLE - PoS Blockchains



Applications of SSLE - PoS Blockchains



A Non-Example

Common approach:

1. Everyone picks a random point on number line

A Non-Example

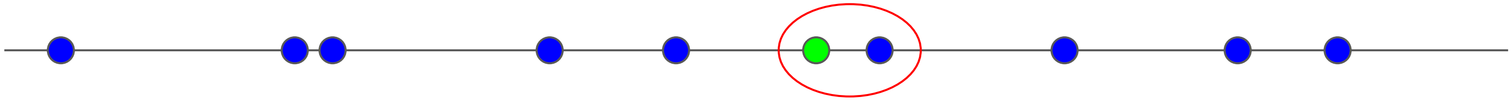
Common approach:

1. Everyone picks a random point on number line
2. Randomness beacon picks a random point on number line

A Non-Example

Common approach:

1. Everyone picks a random point on number line
2. Randomness beacon picks a random point on number line
3. Whoever is closest to the beacon wins



A Non-Example

Setup:

1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

A Non-Example

Setup:

1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)

A Non-Example

Setup:

1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)
2. Beacon produces $R \in F_p$

A Non-Example

Setup:

1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)
2. Beacon produces $R \in F_p$
3. Any participant with $|R - v_i| < 10 * 2^{\lambda} / N$ decommits to v_i

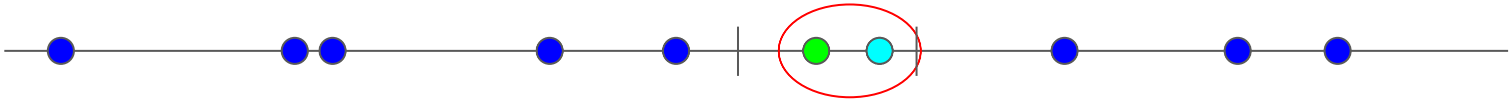
A Non-Example

Setup:

1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)
2. Beacon produces $R \in F_p$
3. Any participant with $|R - v_i| < 10 * 2^{\lambda} / N$ decommits to v_i
4. Winner is participant with minimum $|R - v_i|$



A Non-Example

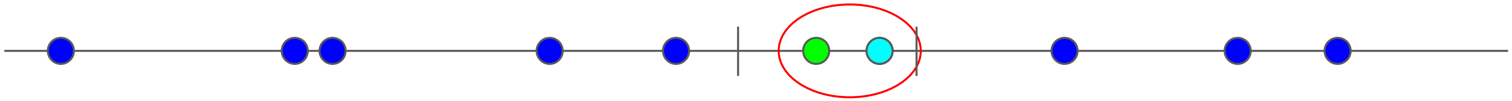
Setup:

1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)
2. Beacon produces $R \in F_p$
3. Any participant with $|R - v_i| < 10 * 2^\lambda / N$ decommits to v_i
4. Winner is participant with minimum $|R - v_i|$

This is *almost* what we want.



A Non-Example

Setup:

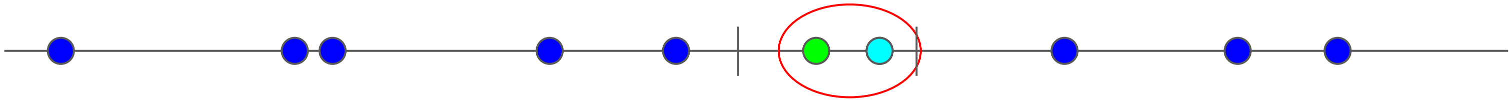
1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)
2. Beacon produces $R \in F_p$
3. Any participant with $|R - v_i| < 10 * 2^\lambda / N$ decommits to v_i
4. Winner is participant with minimum $|R - v_i|$

This is *almost* what we want.

Only the single leader publishes v_i *in expectation*



A Non-Example

Setup:

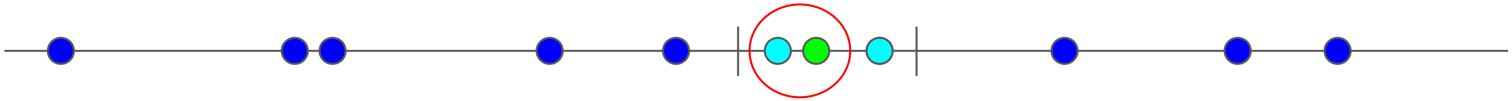
1. Choose λ-bit prime p
2. Randomness beacon that outputs $R \in F_p$

Election:

1. Each participant i picks a secret v_i, produces commitment com(v_i)
2. Beacon produces $R \in F_p$
3. Any participant with $|R - v_i| < 10 * 2^{\lambda} / N$ decommits to v_i
4. Winner is participant with minimum $|R - v_i|$

This is *almost* what we want.

Only the single leader publishes v_i *in expectation*



Why Single Secret Leader Election?

Having multiple potential leaders wastes effort and impedes consensus

From Protocol Labs RFC:

- Fork grinding
- Faster convergence
- Simpler protocol

Cost: requires a registration step

What Makes SSLE Challenging?

Want to minimize long-term storage

What Makes SSLE Challenging?

Want to minimize long-term storage

Want to minimize communication

What Makes SSLE Challenging?

Want to minimize long-term storage

Want to minimize communication

Want to minimize computation

What Makes SSLE Challenging?

Want to minimize long-term storage

Want to minimize communication

Want to minimize computation

Can’t expect every participant to send messages

What Makes SSLE Challenging?

Want to minimize long-term storage

Want to minimize communication

Want to minimize computation

Can’t expect every participant to send messages

Can’t expect every participant to stay online between rounds

Outline

Introduction

Formalizing SSLE

3 SSLE Constructions:

- From DDH & Shuffling
- From obfuscation
- From tFHE

SSLE Requirements

Three security properties:

1. Uniqueness: only one leader is chosen by the election
2. Unpredictability: non-winners cannot guess who the winner is
3. Fairness: each user has 1/N chance of becoming the leader

Goal: *robust* election where DoS of c/N users disrupts election with probability c/N

SSLE Requirements

Three security properties:

1. Uniqueness: only one leader is chosen by the election
2. Unpredictability: non-winners cannot guess who the winner is
3. Fairness: each user has 1/N chance of becoming the leader

Goal: *robust* election where DoS of c/N users disrupts election with probability c/N

Our focus will be on the elections, not on using them to build blockchains.

SSLE Syntax

All algorithms assume access to public state *st*

Elections have access to randomness beacon output *R*

SSLE Syntax

All algorithms assume access to public state *st*

Elections have access to randomness beacon output *R*

<u>SSLE Algorithms</u>

1. Setup
2. Registration
3. Registration verification
4. Election
5. Election verification

Formalizing Definitions

<u>Adversary</u> | **Setup** | <u>Challenger</u>

Run setup$\rightarrow$*pp*, st_0, sk_1, …, sk_N (if applicable)

Choose set $M \subseteq [N]$, $|M|=c$ $\longrightarrow$

$\longleftarrow$ pp, st_0, $\{sk_i\}_{i \in M}$

Formalizing Definitions

Adversary

Challenger

Setup

Choose set $M \subseteq [N]$, $|M|=c$ →

Run setup→$pp, st_0, sk_1, \ldots, sk_N$ (if applicable)

← $pp, st_0, \{sk_i\}_{i \in M}$

Elections

Register any users →

Run registration verification for each uncorrupted user. Output 0 if any fails.

Formalizing Definitions

Adversary

Setup

Challenger

Choose set $M \subseteq [N]$, $|M|=c$ →

Run setup→pp, st_0, sk_1, …, sk_N (if applicable)

← pp, st_0, $\{sk_i\}_{i \in M}$

Elections

Register any users →

Run registration verification for each uncorrupted user. Output 0 if any fails.

Run an election →

(if uncorrupted winner)
Winner index i, proof π_i ←

Formalizing Definitions

Adversary | Challenger

Setup

Choose set $M \subseteq [N]$, $|M|=c$ →

Challenger: Run setup→pp, st_0, sk_1, …, sk_N (if applicable)

← pp, st_0, $\{sk_i\}_{i \in M}$

Elections

Register any users →

Run an election →

← (if uncorrupted winner) Winner index i, proof π_i

Challenger: Run registration verification for each uncorrupted user. Output 0 if any fails.

Formalizing Definitions

Adversary

Challenger

Setup

Choose set $M \subseteq [N]$, $|M|=c$ →

Run setup→pp, $st_0, sk_1, \ldots, sk_N$ (if applicable)

← pp, st_0, $\{sk_i\}_{i \in M}$

Elections

Register any users →

Run an election →

Run registration verification for each uncorrupted user. Output 0 if any fails.

← (if uncorrupted winner)
Winner index i, proof π_i

Challenge

Formalizing Definitions

Adversary		Challenger
	Setup	
	Choose set $M \subseteq [N]$, $\|M\|=c$ →	Run setup→$pp, st_0, sk_1, \ldots, sk_N$ (if applicable)
	← $pp, st_0, \{sk_i\}_{i \in M}$	
	Elections	
	Register any users →	Run registration verification for each uncorrupted user. Output 0 if any fails.
	Run an election →	
	(if uncorrupted winner) Winner index i, proof π_i ←	
Uniqueness	**Challenge**	
	(j, π_j) for $j \in M$, for each election in election phase →	Output 1 if for *any* election, there is more than one tuple (k, π_j) for which election verification accepts.

Formalizing Definitions

<u>Adversary</u> | <u>Challenger</u>

Setup

Choose set $M \subseteq [N]$, $|M|=c$ →

Run setup→$pp, st_0, sk_1, \ldots, sk_N$ (if applicable)

← $pp, st_0, \{sk_i\}_{i \in M}$

Elections

Register any users →

Run an election →

Run registration verification for each uncorrupted user. Output 0 if any fails.

← (if uncorrupted winner)
Winner index i, proof π_i

Unpredictability

Challenge

Run one last election →

Guess winner is user $i \in [N]$ →

If winner is not in $[N] \backslash M$, output 0.

Otherwise, if winner is user i, output 1.

Secure if challenger never outputs 1 with probability greater than $1/(N-c)$.

Formalizing Definitions

Adversary

Challenger

Setup

Choose set $M \subseteq [N]$, $|M|=c$ →

Run setup→pp, $st_0, sk_1, \ldots, sk_N$ (if applicable)

← pp, st_0, $\{sk_i\}_{i \in M}$

Elections

Register any users →

Run registration verification for each uncorrupted user. Output 0 if any fails.

Run an election →

← (if uncorrupted winner)
Winner index i, proof π_i

Fairness

Challenge

Run one last election →

If winner is not in $[N]\backslash M$, output 1.

Secure if challenger never outputs 1 with probability greater than c/N.

Three Constructions of SSLE

Obfuscation

Ideal solution, but uses theoretical tools

tFHE

Closer to realistic, only gives a threshold version of security

DDH

“Compromise” solution -- $\sqrt{N}$ communication per election, $1/(\sqrt{N}-c)$ unpredictability

Should be suitable for practical use cases

Three Constructions of SSLE

<u>DDH</u>

“Compromise” solution -- $\sqrt{N}$ communication per election, $1/(\sqrt{N}-c)$ unpredictability

Should be suitable for practical use cases

<u>Obfuscation</u>

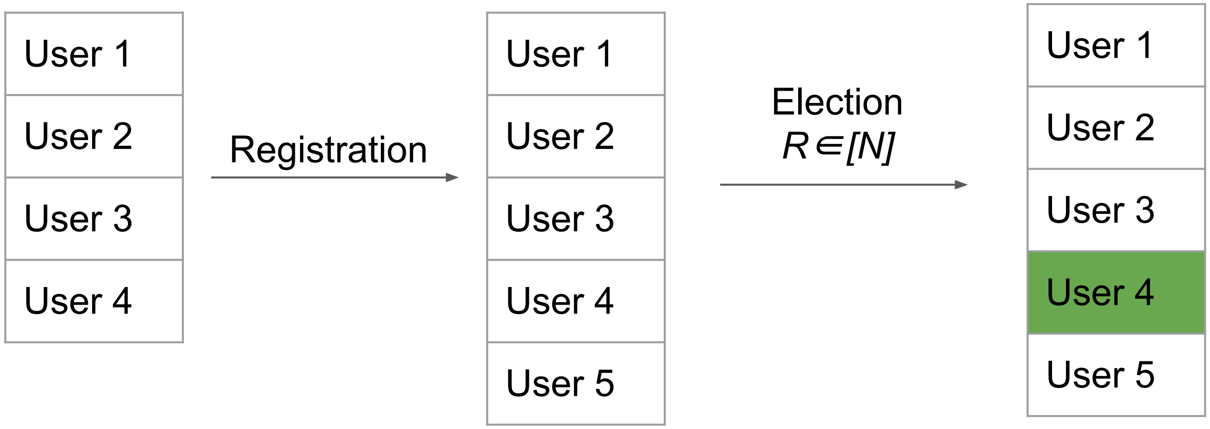
Ideal solution, but uses theoretical tools

<u>tFHE</u>

Closer to realistic, only gives a threshold version of security

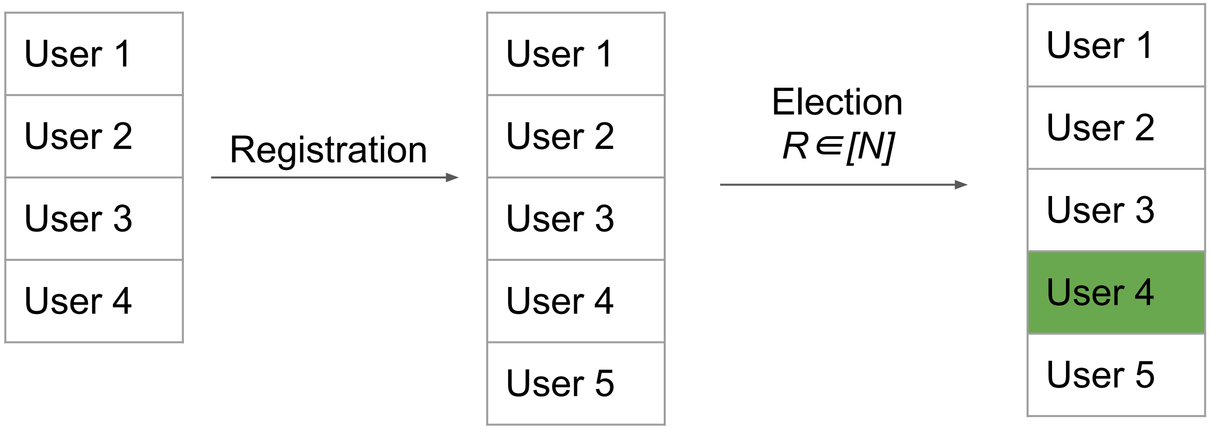
SSLE from DDH

The easiest single non-secret leader election



SSLE from DDH

The easiest single non-secret leader election



✓ Uniqueness

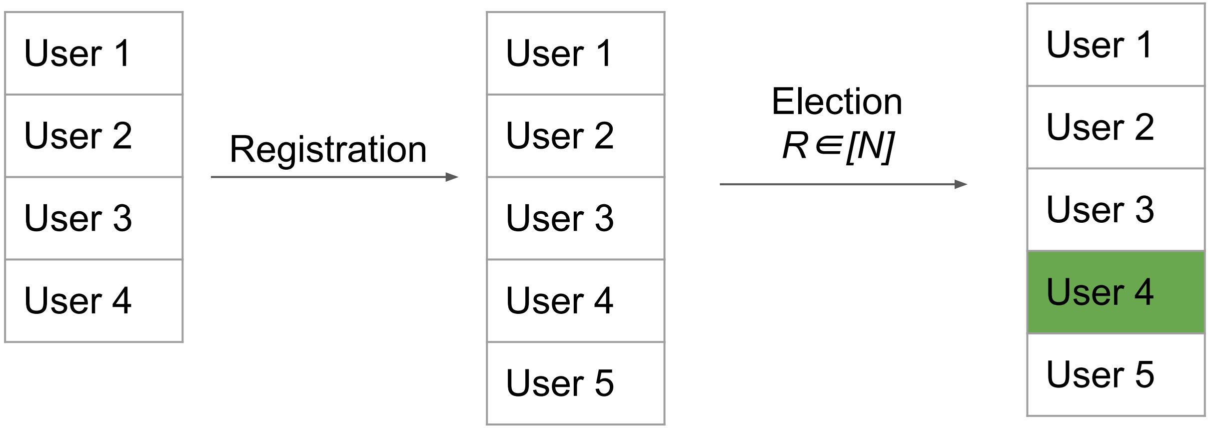
✓ Fairness

☒ Unpredictability

How to hide the leader?

SSLE from DDH

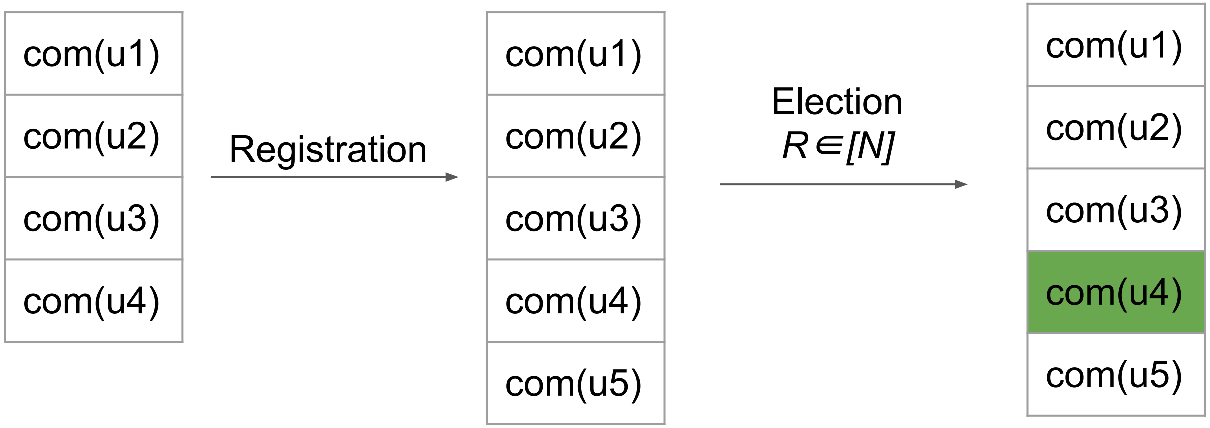
1. Commitments



- ✓ Uniqueness
- ✓ Fairness
- ☒ Unpredictability

SSLE from DDH

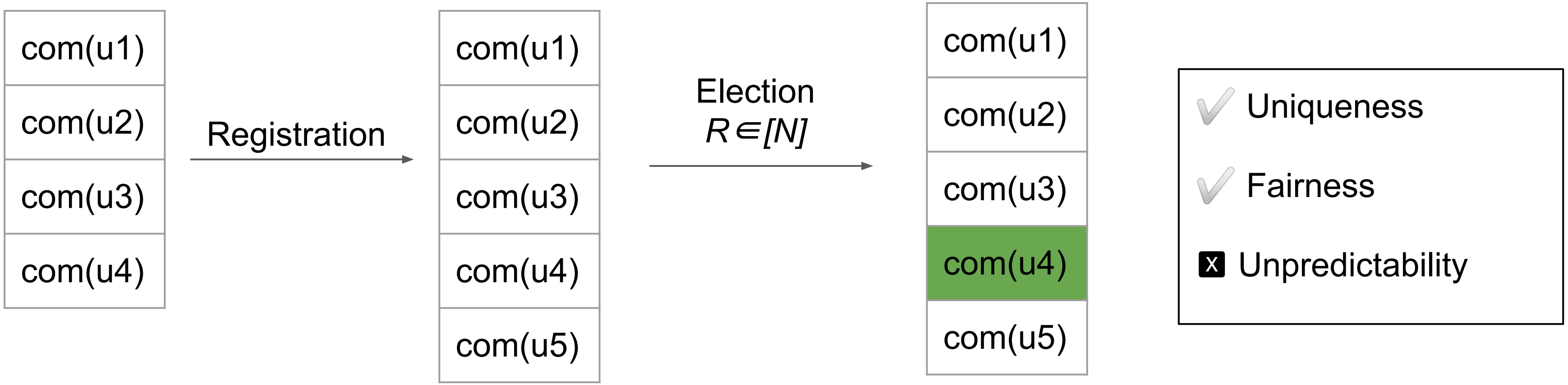
1. Commitments



- ✓ Uniqueness
- ✓ Fairness
- ☒ Unpredictability

SSLE from DDH

1. Commitments
2. Shuffling



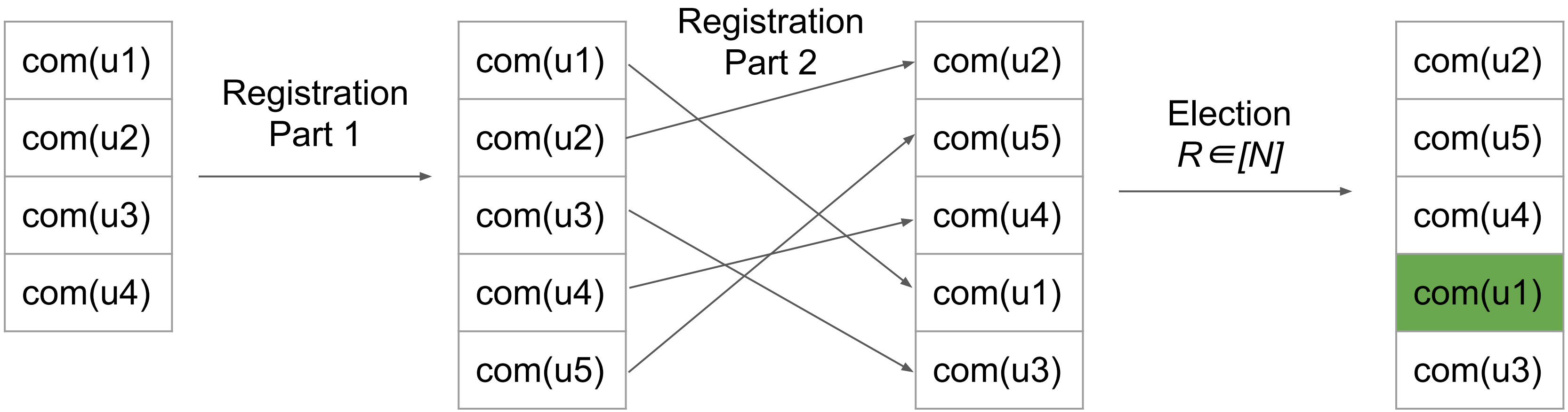
SSLE from DDH

1. Commitments
2. Shuffling

✓ Uniqueness

✓ Fairness

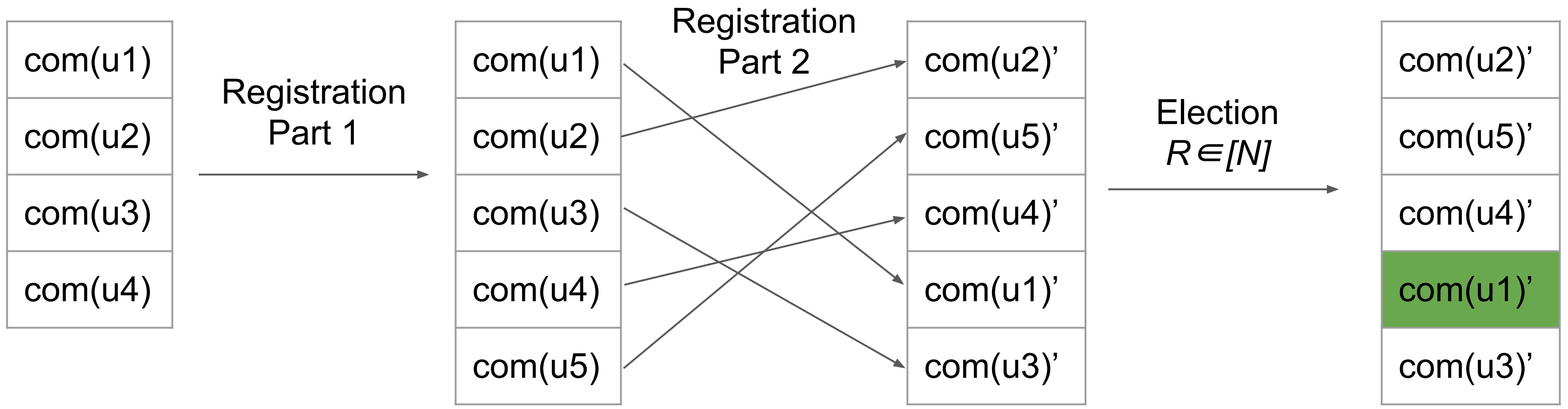
☒ Unpredictability



SSLE from DDH

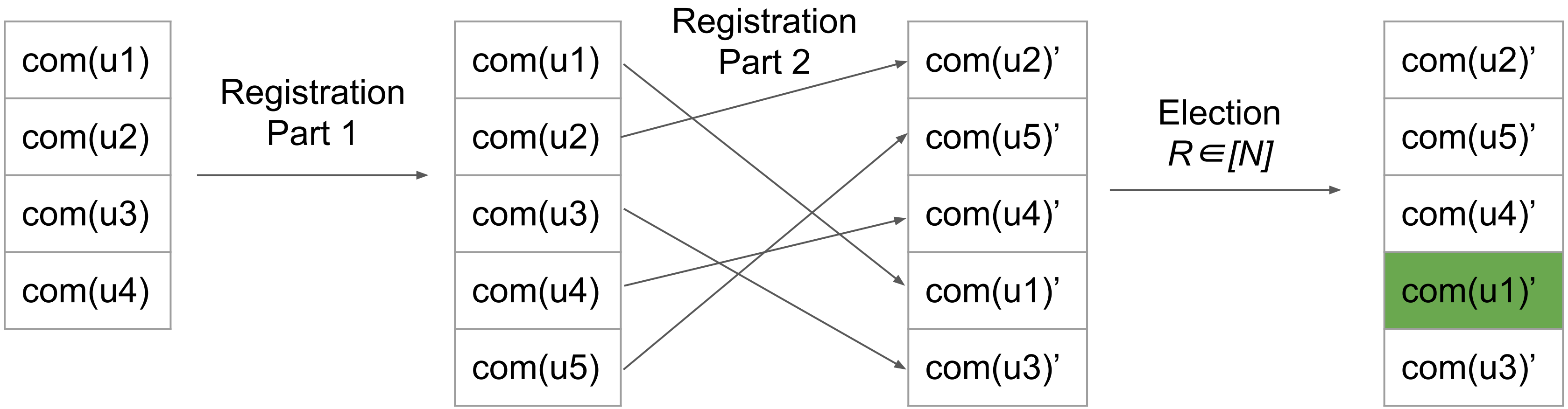
1. Commitments
2. Shuffling
3. Rerandomization

- [x] Uniqueness
- [x] Fairness
- [x] Unpredictability



SSLE from DDH

1. Commitments
2. Shuffling
3. Rerandomization & Reidentification



A Rerandomizable & Reidentifiable Commitment

Let $g \in G$, G is a group where DDH is hard

$\text{Com}(k, r) \rightarrow (g^r, g^{rk})$

A Rerandomizable & Reidentifiable Commitment

Let $g \in G$, G is a group where DDH is hard

Com(k, r) $\rightarrow$ (g^r, g^{rk})

Rerandomization: $(g^r, g^{rk}) \rightarrow (g^{rr'}, g^{rr'k})$

Reidentification: given (u,v), check if $u^k = v$

A Rerandomizable & Reidentifiable Commitment

Let $g \in G$, G is a group where DDH is hard

Com(k, r) $\rightarrow$ (g^r, g^{rk})

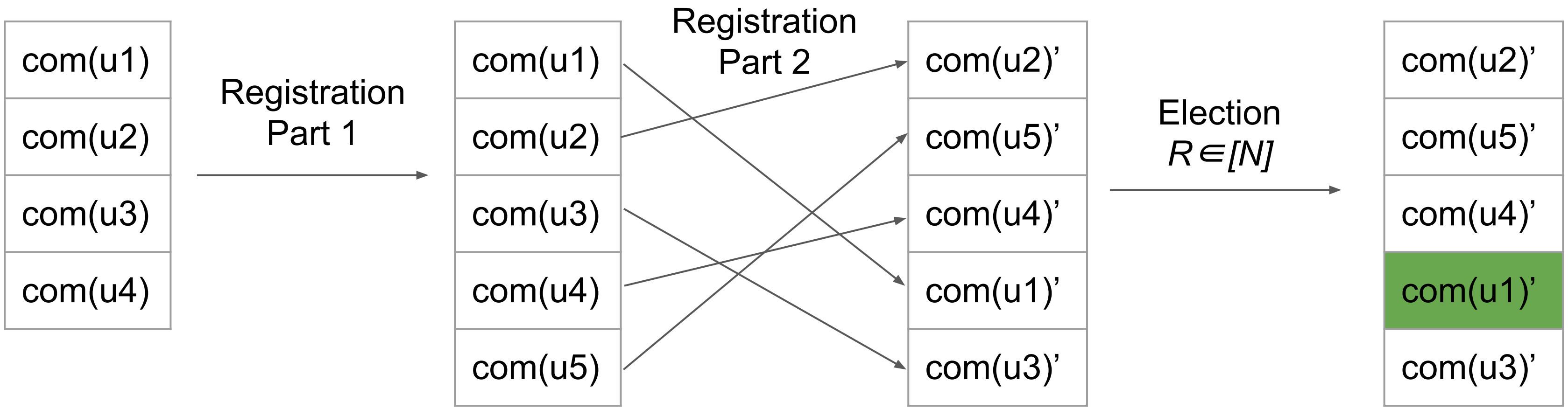
Rerandomization: $(g^r, g^{rk}) \rightarrow (g^{rr'}, g^{rr'k})$

Reidentification: given (u,v), check if $u^k = v$

Security follows from DDH: $(g^r, g^{rk}, g^{rr'}, g^{rr'k})$ vs $(g^r, g^{rk}, g^{rr'}, g^{rz})$

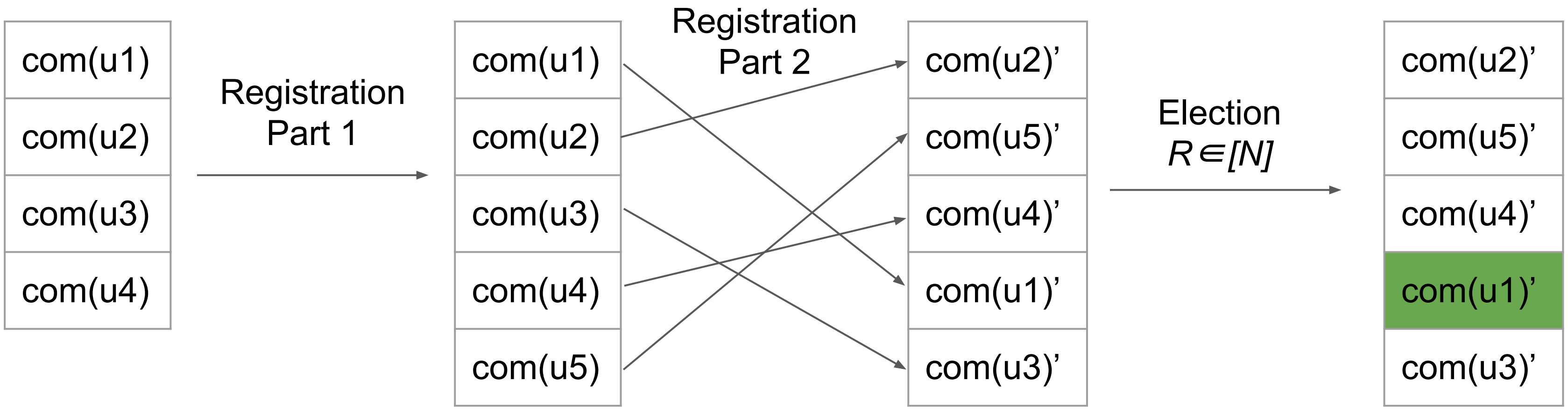
SSLE from DDH

1. Commitments
2. Shuffling
3. Rerandomization & Reidentification



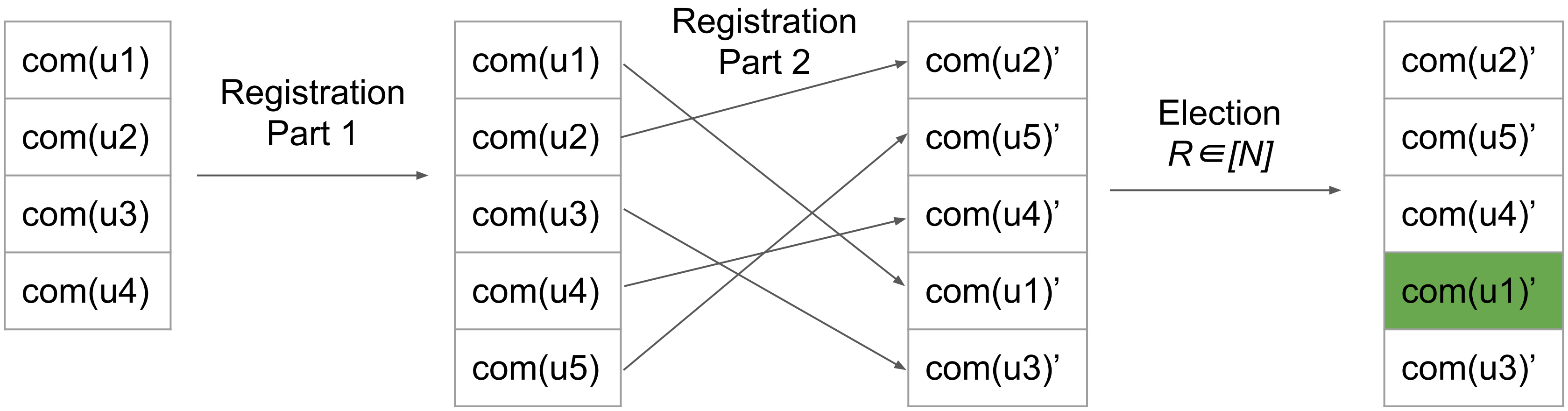
SSLE from DDH

1. Commitments
2. Shuffling
3. Rerandomization & Reidentification
4. Verification of shuffle



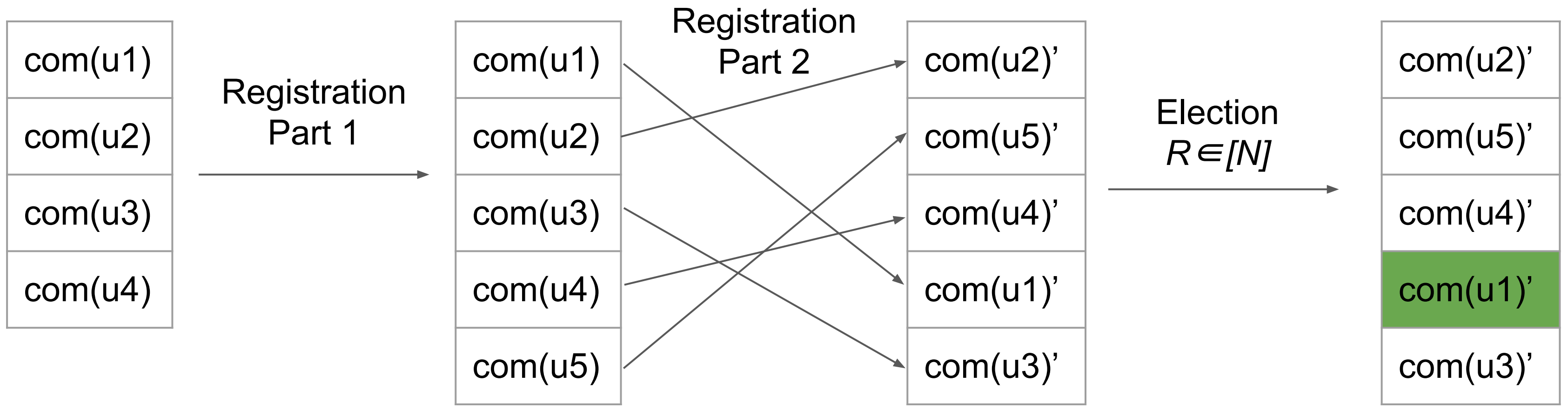
SSLE from DDH

1. Commitments
2. Shuffling
3. Rerandomization & Reidentification
4. Verification of shuffle -- NIZK or other users check

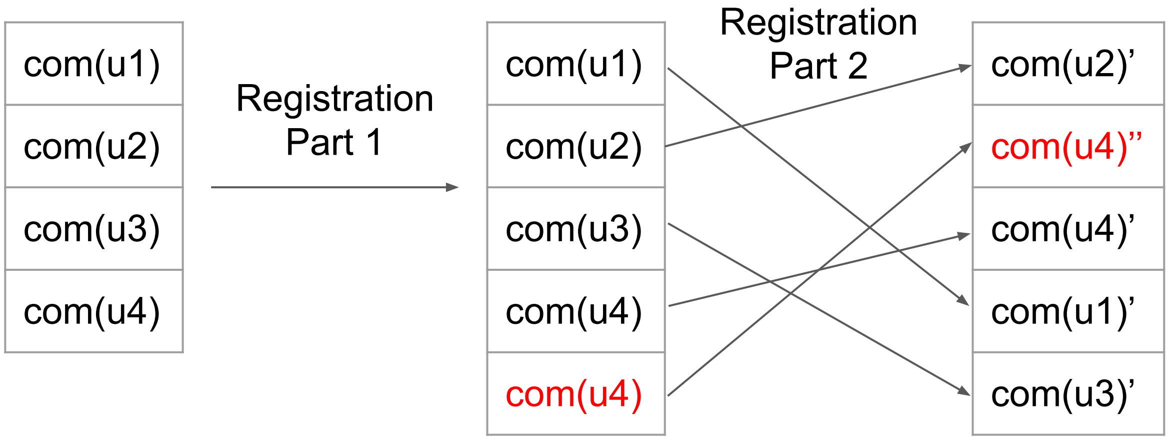


SSLE from DDH

1. Commitments
2. Shuffling
3. Rerandomization & Reidentification
4. Verification of shuffle -- NIZK or other users check
5. Defend against duplication attacks



Duplication Attack



Duplication attack makes it possible for 2 different users to register with a commitment to the same value

Breaks uniqueness and unpredictability

Preventing Duplication Attacks

How to ensure that users never commit to the same value?

Idea: Derive a secret commitment value and a tag from a master secret

Sample random k

$H(k) \rightarrow k_L, k_R$

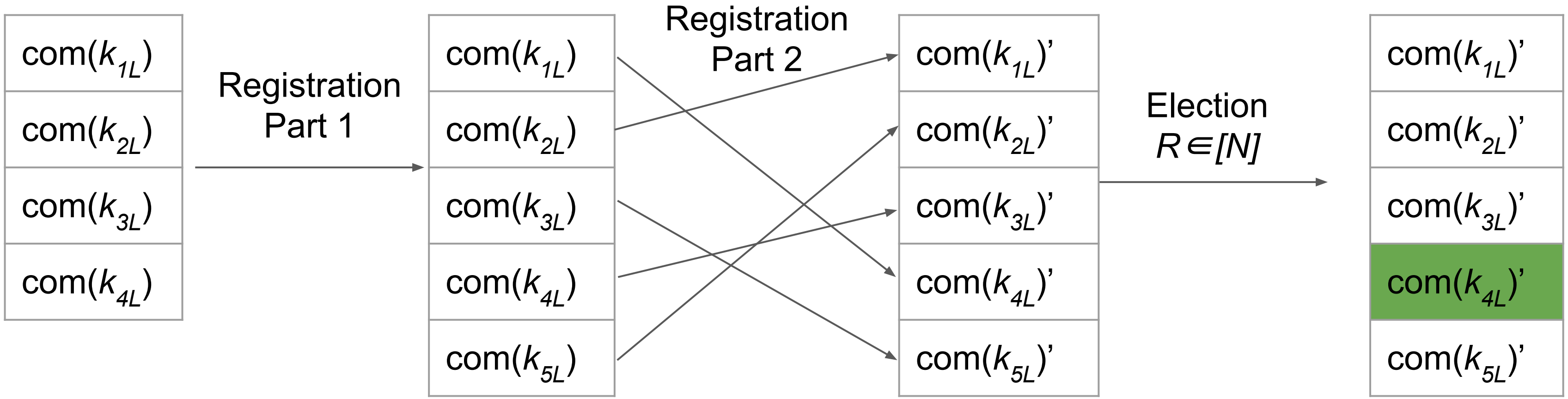
Post com(k_L) and k_R

Registrations to the same secret detected by duplicate k_R

(H modeled as random oracle)

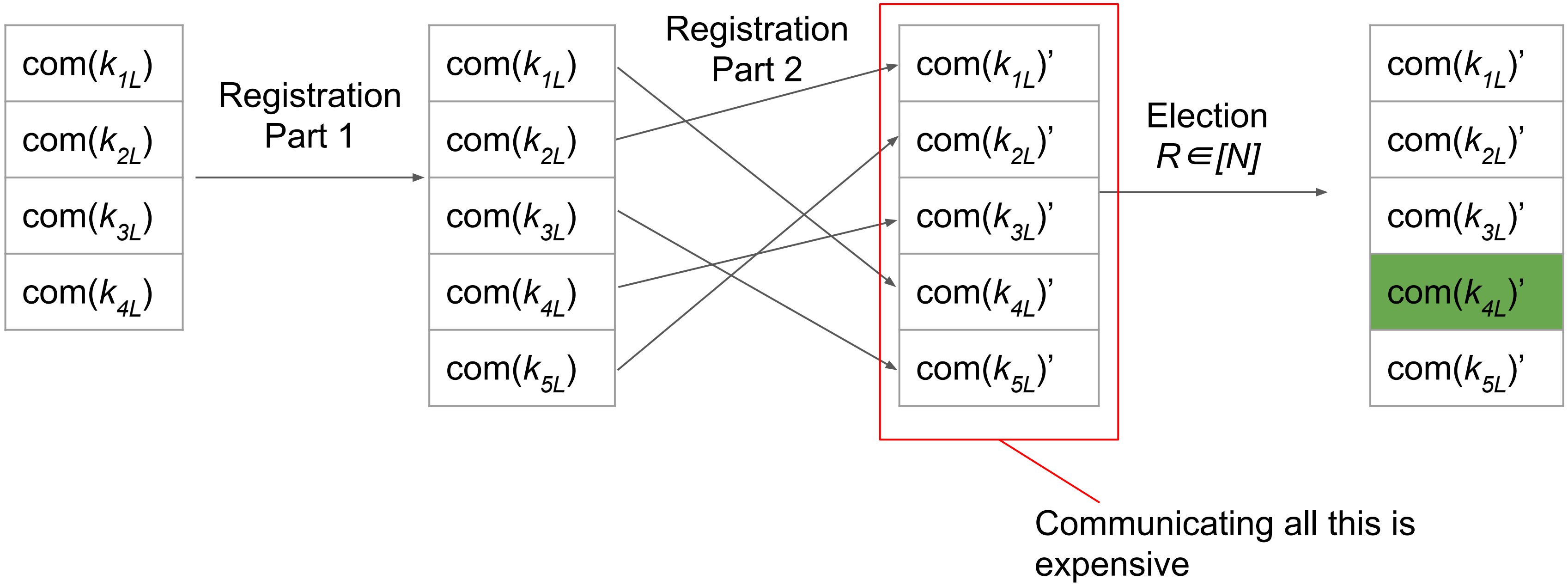
Saving Communication

Protocol thus far has required linear communication for each registration



Saving Communication

Protocol thus far has required linear communication for each registration

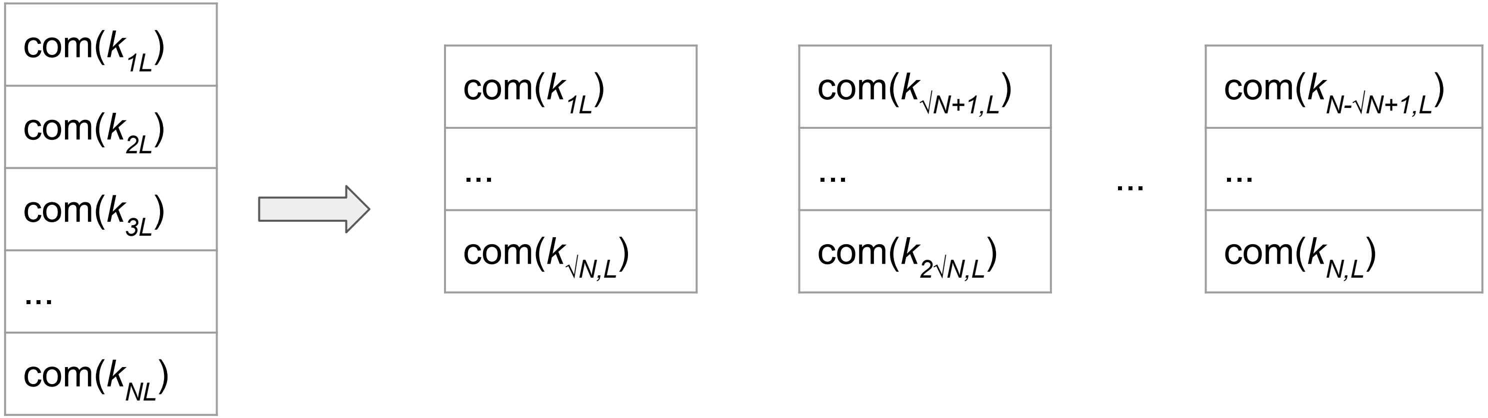


Saving Communication

Communication/Security tradeoff: instead of shuffling new entry into the whole list, split the list into a number of buckets and only shuffle into one bucket.

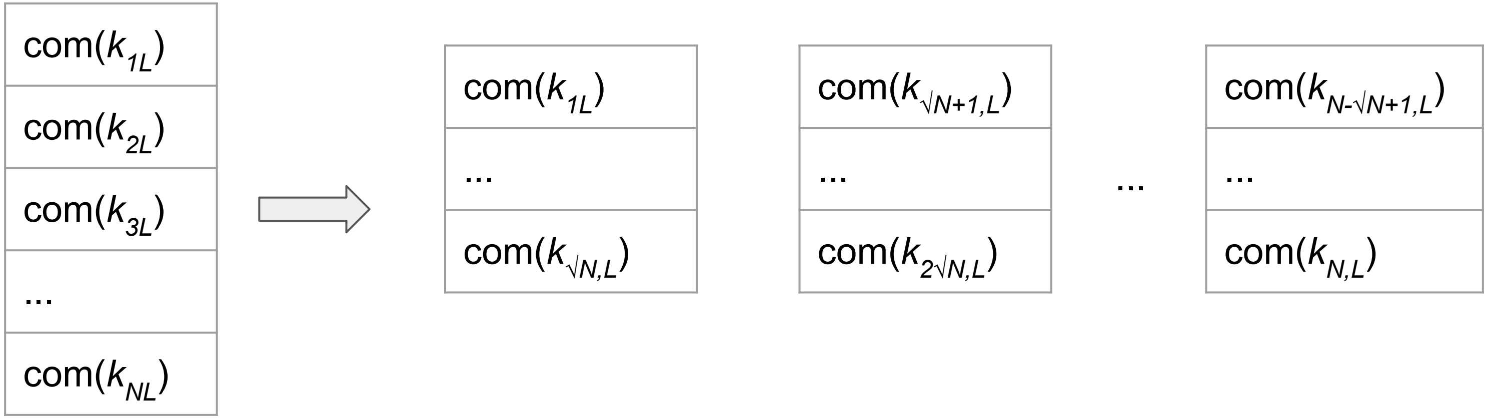
Saving Communication

Communication/Security tradeoff: instead of shuffling new entry into the whole list, split the list into a number of buckets and only shuffle into one bucket.



Saving Communication

Communication/Security tradeoff: instead of shuffling new entry into the whole list, split the list into a number of buckets and only shuffle into one bucket.



Larger buckets mean more unpredictability but also more communication

√N sized buckets seems like a good tradeoff

Security

With a deterministic choice of buckets, we get the following theorem:

Theorem 19. *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE is a unique, fair, and $\frac{1}{\sqrt{N-c}}$-unpredictable SSLE scheme in the random oracle model.*

Security

With a deterministic choice of buckets, we get the following theorem:

Theorem 19. *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE is a unique, fair, and $\frac{1}{\sqrt{N-c}}$-unpredictable SSLE scheme in the random oracle model.*

We can do better by randomizing the choice of buckets, so an adversary needs to corrupt O(N) users to guess winner with constant probability

Security

With a deterministic choice of buckets, we get the following theorem:

Theorem 19. *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE is a unique, fair, and $\frac{1}{\sqrt{N-c}}$-unpredictable SSLE scheme in the random oracle model.*

We can do better by randomizing the choice of buckets, so an adversary needs to corrupt O(N) users to guess winner with constant probability

Theorem 20 (Informal). *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE modified to assign buckets randomly at user registration time is a unique, fair, and $\frac{1}{\frac{N-c}{\sqrt{N}} - \sqrt{\frac{2\lambda(N-c)}{\sqrt{N}}}}$-unpredictable SSLE scheme in the random oracle model.*

Security

With a deterministic choice of buckets, we get the following theorem:

Theorem 19. *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE is a unique, fair, and $\frac{1}{\sqrt{N-c}}$-unpredictable SSLE scheme in the random oracle model.*

We can do better by randomizing the choice of buckets, so an adversary needs to corrupt O(N) users to guess winner with constant probability

Theorem 20 (Informal). *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE modified to assign buckets randomly at user registration time is a unique, fair, and $\frac{1}{\frac{N-c}{\sqrt{N}} - \sqrt{\frac{2\lambda(N-c)}{\sqrt{N}}}}$-unpredictable SSLE scheme in the random oracle model.*

Open problem: we believe we can do better with a more clever shuffling/bucketing algorithm, e.g. by using something like a square shuffle [Hastad06]

Security

With a deterministic choice of buckets, we get the following theorem:

Theorem 19. *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE is a unique, fair, and $\frac{1}{\sqrt{N-c}}$-unpredictable SSLE scheme in the random oracle model.*

We can do better by randomizing the choice of buckets, so an adversary needs to corrupt O(N) users to guess winner with constant probability

Theorem 20 (Informal). *Assuming that $\mathbb{G}$ is a group in which the DDH problem is hard, then for any adversary $\mathcal{A}$, SSSLE modified to assign buckets randomly at user registration time is a unique, fair, and $\frac{1}{\frac{N-c}{\sqrt{N}} - \sqrt{\frac{2\lambda(N-c)}{\sqrt{N}}}}$-unpredictable SSLE scheme in the random oracle model.*

Open problem: we believe we can do better with a more clever shuffling/bucketing algorithm, e.g. by using something like a square shuffle [Hastad06]

Open problem: constant communication per election (in a practical scheme)

SSLE from Obfuscation

Obfuscation [BGI+01, GGH+13]

Obfuscator $iO(C)$ produces a new circuit C' such that:

1. C and C' have the exact same behavior.
2. For any two circuits C_0, C_1 that have the exact same behavior, no adversary can distinguish between $iO(C_0)$ and $iO(C_1)$.

SSLE from Obfuscation

Obfuscation [BGI+01, GGH+13]

Obfuscator *iO(C)* produces a new circuit *C'* such that:

1. *C* and *C'* have the exact same behavior
2. For any two circuits C_0, C_1 that have the exact same behavior, no adversary can distinguish between $iO(C_0)$ and $iO(C_1)$

Puncturable PRF [BW13, BGI14, KPTZ13]

PRF where you can generate a *punctured* key that allows you to evaluate the PRF everywhere except at that point.

Given the punctured key, the value of the PRF at the punctured point is still pseudorandom.

SSLE from Obfuscation

Plan:

1. Write a program that picks leader using secret key embedded in the program
2. Obfuscate program during trusted setup and distribute to everyone
3. Any participant just needs to post a public key to register for elections
4. Obfuscated program output should allow leader to prove she won

SSLE from Obfuscation

Program to obfuscate, first attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

SSLE from Obfuscation

Program to obfuscate, first attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $w \leftarrow F(k,s)$

SSLE from Obfuscation

Program to obfuscate, first attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $w \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4. *Output* b

SSLE from Obfuscation

Program to obfuscate, first attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $w \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4. *Output b*

✔️ Elects one leader randomly based on secret key

❎ Anyone can learn the leader by trying each value of *i*

SSLE from Obfuscation

Program to obfuscate, second attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $w \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4.

SSLE from Obfuscation

Program to obfuscate, second attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w, r) \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4.

SSLE from Obfuscation

Program to obfuscate, second attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w, r) \leftarrow F(k,s)$
3. $b \leftarrow 1$ if $i = w \bmod n$, $b \leftarrow 0$ otherwise
4. $ct \leftarrow Encrypt(pk_i, b; r)$
5. Output ct

SSLE from Obfuscation

Program to obfuscate, second attempt

P((pk$_0$, …, pk$_{N-1}$), i, N, R):

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w, r) \leftarrow F(k,s)$
3. $b \leftarrow 1$ if $i = w \bmod n$, $b \leftarrow 0$ otherwise
4. $ct \leftarrow Encrypt(pk_i, b; r)$
5. Output ct

✓ Elects one leader randomly based on secret key

✓ Only user i can decrypt b_i

☒ Not clear how winner can prove that she won the election

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w, r) \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4.

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4.

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ if $i = w \bmod n$, $b \leftarrow 0$ otherwise
4. $c \leftarrow com(b; r)$

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4. $c \leftarrow com(b; r)$
5. $ct \leftarrow Encrypt(pk_i, r; r')$
6. Output c, ct

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4. $c \leftarrow com(b; r)$
5. $ct \leftarrow Encrypt(pk_i, r; r')$
6. Output c, ct

✓ Elects one leader randomly based on secret key

✓ Only user i can decrypt b_i

✓ Prove leadership by revealing r

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4. $c \leftarrow com(b; r)$
5. $ct \leftarrow Encrypt(pk_i, r; r')$
6. Output c, ct

Why not encrypt?

- ✓ Elects one leader randomly based on secret key
- ✓ Only user i can decrypt b_i
- ✓ Prove leadership by revealing r

SSLE from Obfuscation

Program to obfuscate, final attempt

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ if $i = w \bmod n$, $b \leftarrow 0$ otherwise
4. $c \leftarrow com(b; r)$
5. $ct \leftarrow Encrypt(pk_i, r; r')$
6. Output c, ct

✓ Elects one leader randomly based on secret key

✓ Only user i can decrypt b_i

✓ Prove leadership by revealing r

Why not encrypt?
If the encryption does not commit, adversary could potentially find bad randomness that allows a non-winning ciphertext to decrypt to 1

SSLE from Obfuscation

<u>Program to obfuscate, final attempt</u>

$P((pk_0, \ldots, pk_{N-1}), i, N, R)$:

1. $s \leftarrow R, pk_0, \ldots, pk_{N-1}$
2. $(w,r,r') \leftarrow F(k,s)$
3. $b \leftarrow 1$ *if* $i = w \bmod n$, $b \leftarrow 0$ *otherwise*
4. $c \leftarrow com(b; r)$
5. $ct \leftarrow Encrypt(pk_i, r; r')$
6. Output c, ct

✓ Elects one leader randomly based on secret key

✓ Only user i can decrypt b_i

✓ Prove leadership by revealing r

See paper for proofs of uniqueness, selective fairness, selective unpredictability

SSLE from tFHE

Reminder: why can't we use a generic MPC protocol for SSLE?

Easy DoS opportunity if everyone has to come back for a second round

SSLE from tFHE

Reminder: why can't we use a generic MPC protocol for SSLE?

Easy DoS opportunity if everyone has to come back for a second round

What if only a few people have to come back *<u>and it doesn't matter which ones</u>*?

SSLE from tFHE

Reminder: why can’t we use a generic MPC protocol for SSLE?

Easy DoS opportunity if everyone has to come back for a second round

What if only a few people have to come back *and it doesn’t matter which ones*?

Tools from threshold crypto can enable this!

SSLE from tFHE

<u>Threshold Encryption</u>:
Standard public-key encryption, but instead of one secret key, many users have *shares* of a secret key that produce *partial decryptions*, with *t* partial decryptions needed to produce a plaintext.

SSLE from tFHE

Threshold Encryption:
Standard public-key encryption, but instead of one secret key, many users have *shares* of a secret key that produce *partial decryptions*, with *t* partial decryptions needed to produce a plaintext.

Fully Homomorphic Encryption (FHE):
Standard public-key encryption, but ciphertexts can be added together and multiplied. Expensive operation is multiplication, high multiplicative depth is especially costly.

SSLE from tFHE

Threshold Encryption:
Standard public-key encryption, but instead of one secret key, many users have *shares* of a secret key that produce *partial decryptions*, with t partial decryptions needed to produce a plaintext.

Fully Homomorphic Encryption (FHE):
Standard public-key encryption, but ciphertexts can be added together and multiplied. Expensive operation is multiplication, high multiplicative depth is especially costly.

Threshold FHE (tFHE):
Combine the two tools above.

SSLE from tFHE

Threshold Encryption:
Standard public-key encryption, but instead of one secret key, many users have *shares* of a secret key that produce *partial decryptions*, with *t* partial decryptions needed to produce a plaintext.

Fully Homomorphic Encryption (FHE):
Standard public-key encryption, but ciphertexts can be added together and multiplied. Expensive operation is multiplication, high multiplicative depth is especially costly.

Threshold FHE (tFHE):
Combine the two tools above.

Using these tools, we can only really hope for *threshold* unpredictability and fairness

SSLE from tFHE

Plan:

1. All participants get a tFHE decryption key
2. Define a computation that picks the leader
3. Evaluate computation under tFHE
4. Some subset of *t* users post partial decryptions
5. Output of computation somehow secretly determines winner

SSLE from tFHE

Plan:

1. All participants get a tFHE decryption key
2. Define a computation that picks the leader
3. Evaluate computation under tFHE
4. Some subset of *t* users post partial decryptions
5. Output of computation somehow secretly determines winner

Unlike the obfuscation case, everyone gets the *same* output.

SSLE from tFHE

Idea:

Each participant registers with a secret k

Output of computation is the secret of a randomly chosen participant

The participant knows her secret, but nobody else knows who owns it

SSLE from tFHE

Idea:

Each participant registers with a secret k

Output of computation is the secret of a randomly chosen participant

The participant knows her secret, but nobody else knows who owns it

<u>Main remaining problems to solve:</u>

1. Efficiently generating randomness inside the tFHE
2. Efficiently using the randomness to pick someone's secret

SSLE from tFHE

Idea:

Each participant registers with a secret k

Output of computation is the secret of a randomly chosen participant

The participant knows her secret, but nobody else knows who owns it

Main remaining problems to solve:

1. Efficiently generating randomness inside the tFHE
2. Efficiently using the randomness to pick someone's secret

See paper for other details

SSLE from tFHE

k_1
k_2
k_3
k_4
k_5

SSLE from tFHE

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix} \bullet \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = k_4$$

Random weight-1 vector

SSLE from tFHE

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix} \bullet \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = k_4$$

Random weight-1 vector

How can we efficiently generate a random weight-1 vector given some random bits inside the tFHE?

"efficiently" = low multiplicative depth

SSLE from tFHE

1. Start with *logN* random bits

0	1	1	0

SSLE from tFHE

1. Start with *logN* random bits
2. Split bits into length-2 vectors where $b \rightarrow (b, 1\text{-}b)$:
   a. $0 \rightarrow (0,1)$
   b. $1 \rightarrow (1,0)$

0	1	1	0

(0,1)	(1,0)	(1,0)	(0,1)

SSLE from tFHE

1. Start with *logN* random bits
2. Split bits into length-2 vectors where *b* → *(b, 1-b)*:
   a. 0 → (0,1)
   b. 1 → (1,0)
3. Take outer product of adjacent vectors and flatten
   a. E.g. (0,1) □ (1,0) = (0,0,1,0)

0	1	1	0

(0,1)	(1,0)	(1,0)	(0,1)

(0,0,1,0)	(0,0,0,1)

SSLE from tFHE

1. Start with *logN* random bits
2. Split bits into length-2 vectors where *b* → *(b, 1-b)*:
    a. 0 → (0,1)
    b. 1 → (1,0)
3. Take outer product of adjacent vectors and flatten
    a. E.g. (0,1) □ (1,0) = (0,0,1,0)
4. Repeat step 3 until only a single length-*N* vector remains

0	1	1	0

(0,1)	(1,0)	(1,0)	(0,1)

(0,0,1,0)	(0,0,0,1)

(0,0,0,0, 0,0,0,0, 0,0,0,1, 0,0,0,0)

SSLE from tFHE

1. Start with *logN* random bits
2. Split bits into length-2 vectors where *b* → *(b, 1-b)*:
    a. 0 → (0,1)
    b. 1 → (1,0)
3. Take outer product of adjacent vectors and flatten
    a. E.g. (0,1) □ (1,0) = (0,0,1,0)
4. Repeat step 3 until only a single length-*N* vector remains

Multiplicative depth: *loglogN*

0	1	1	0

(0,1)	(1,0)	(1,0)	(0,1)

(0,0,1,0)	(0,0,0,1)

(0,0,0,0, 0,0,0,0, 0,0,0,1, 0,0,0,0)

Single Secret Leader Election

Elect exactly 1 leader such that only the leader learns who she is and can prove it

Our contributions:

Formalization of SSLE requirements and security definitions

Three constructions: from DDH, tFHE, and obfuscation

Single Secret Leader Election

Elect exactly 1 leader such that only the leader learns who she is and can prove it

Our contributions:

Formalization of SSLE requirements and security definitions

Three constructions: from DDH, tFHE, and obfuscation

Paper: `https://eprint.iacr.org/2020/025.pdf`

Contact: `saba@cs.stanford.edu`