Stacking Sigma
A Framework to Compose $\Sigma$–Protocols for Disjunctions

Aarushi Goel         Matthew Green          Mathias Hall-Andersen         Gabriel Kaptchuk

Johns Hopkins University  Aarhus University  Boston University
Zero Knowledge Proofs

$L \in NP$

Prover

$\ldots$

$\ldots$

$\ldots$

Verifier

$x \in L$

Okay, I believe you

Prove it!
Zero Knowledge Proofs

Let $L \in NP$

$x \in L$

Prove it!

Okay, I believe you

Verifier

Prover

Cheating prover should not be able to convince the verifier if $x \notin L$
Zero Knowledge Proofs

\[ L \in \text{NP} \]

\( x \in L \)  
Prover

..  
..  
..  

Prove it!  
Verifier

Okay, I believe you

Soundness
Cheating prover should not be able to convince the verifier if \( x \notin L \)

Zero knowledge
Verifier should not learn anything other than the validity of the statement
Sigma Protocols

$L \in NP$

Prover

Verifier

$\alpha$

$\gamma$

$\beta$

Public coin proofs

Honest verifier zero-knowledge

Can be made non-interactive in the random oracle model
Disjunctive Statements: Interesting Class of Languages

\[ x_1 \in L_1 \text{ or } x_2 \in L_2 \text{ or } \ldots \text{ or } x_n \in L_n \]

Where each \( L_i \in NP \)
Disjunctive Statements: Interesting Class of Languages

\[ x_1 \in L_1 \quad or \quad x_2 \in L_2 \quad or \quad \ldots \quad or \quad x_n \in L_n \]

Where each \( L_i \in NP \)

Applications:

Set-Membership Proofs – Ring signatures, confidential transactions
Disjunctive Statements: Interesting Class of Languages

\[ x_1 \in L_1 \quad \text{or} \quad x_2 \in L_2 \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L_n \]

Where each \( L_i \in NP \)

**Applications:**

Set-Membership Proofs – Ring signatures, confidential transactions

Proving existence of bugs in codebase
Disjunctive Statements: Interesting Class of Languages

\[ x_1 \in L_1 \quad \text{or} \quad x_2 \in L_2 \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L_n \]

Where each \( L_i \in NP \)

Applications:

- Set-Membership Proofs – Ring signatures, confidential transactions
- Proving existence of bugs in codebase
- Proving correct execution of a processor

\ldots\ldots
# Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For Σ-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
</tbody>
</table>

---
# Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For $\Sigma$-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>Stacked Garbling [HK20]</td>
<td>NO</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
</tr>
</tbody>
</table>

- Classical [CDS94, AOS02]: For $\Sigma$-Protocols, all languages have linear proof size and linear prover time. Non-interactive proofs are provided in the Random Oracle Model.
- Stacked Garbling [HK20]: NO support for non-interactive proofs. Linear in one branch for proof size and linear in all branches for prover time.
# Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For Σ-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
<td></td>
</tr>
<tr>
<td>Stacked Garbling [HK20]</td>
<td>NO</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Mac n’ Cheese [BMRS20]</td>
<td>For special kind of interactive proofs</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branch</td>
<td>NO</td>
<td></td>
</tr>
</tbody>
</table>
### Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For $\Sigma$-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>Stacked Garbling [HK20]</td>
<td>NO</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
</tr>
<tr>
<td>Mac n’ Cheese [BMRS20]</td>
<td>For special kind of interactive proofs</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branch</td>
<td>NO</td>
</tr>
<tr>
<td>Bullet-Proofs/Compressed-Protocols [BCC+16,BBB+18, ACF20..]</td>
<td>NO</td>
<td>Restricted Class</td>
<td>Sublinear</td>
<td>Linear in all the branch</td>
<td>Random Oracle Model</td>
</tr>
</tbody>
</table>
# Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For $\Sigma$-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>Stacked Garbling [HK20]</td>
<td>NO</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
</tr>
<tr>
<td>Mac n’ Cheese [BMRS20]</td>
<td>For special kind of interactive proofs</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branch</td>
<td>NO</td>
</tr>
<tr>
<td>Bullet-Proofs/Compressed-Protocols [BCC+16,BBB+18, ACF20..]</td>
<td>NO</td>
<td>Restricted Class</td>
<td>Sublinear</td>
<td>Linear in all the branch</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>SNARKs</td>
<td></td>
<td>All</td>
<td>Sublinear</td>
<td>Super-linear in all the branches</td>
<td>CRS/Random Oracle Model</td>
</tr>
</tbody>
</table>
# Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For $\Sigma$-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>Stacked Garbling [HK20]</td>
<td>NO</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
</tr>
<tr>
<td>Mac n’ Cheese [BMRS20]</td>
<td>For special kind of interactive proofs</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branch</td>
<td>NO</td>
</tr>
<tr>
<td>Bullet-Proofs/Compressed-Protocols [BCC+16,BBB+18, ACF20..]</td>
<td>NO</td>
<td>Restricted Class</td>
<td>Sublinear</td>
<td>Linear in all the branch</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>SNARKs</td>
<td></td>
<td>All</td>
<td>Sublinear</td>
<td>Super-linear in all the branches</td>
<td>CRS/Random Oracle Model</td>
</tr>
</tbody>
</table>
# Zero-Knowledge Proofs for Disjunctive Statements

<table>
<thead>
<tr>
<th>Result</th>
<th>General Compiler</th>
<th>Languages</th>
<th>Proof Size</th>
<th>Prover Time</th>
<th>Non-interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical [CDS94, AOS02]</td>
<td>For Σ-Protocols</td>
<td>All</td>
<td>Linear in all the branches</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>Stacked Garbling [HK20]</td>
<td>NO</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
</tr>
<tr>
<td>Mac n’ Cheese [BMRS20]</td>
<td>For special kind of interactive proofs</td>
<td>Circuit SAT</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>NO</td>
</tr>
<tr>
<td>Bullet-Proofs/Compressed-Protocols [BCC+16,BBB+18, ACF20..]</td>
<td>NO</td>
<td>Restricted Class</td>
<td>Sublinear</td>
<td>Linear in all the branch</td>
<td>Random Oracle Model</td>
</tr>
<tr>
<td>SNARKs</td>
<td></td>
<td>All</td>
<td>Sublinear</td>
<td>Super-linear in all the branches</td>
<td>CRS/Random Oracle Model</td>
</tr>
<tr>
<td><strong>Our Work</strong></td>
<td>For Σ-Protocols</td>
<td>All</td>
<td>Linear in one branch</td>
<td>Linear in all the branches</td>
<td>Random Oracle Model</td>
</tr>
</tbody>
</table>
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L_1$ or $x_2 \in L_2$ or $\ldots$ or $x_n \in L_n$
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L_1$ \text{ or } $x_2 \in L_2$ \text{ or } $\ldots$ \text{ or } $x_n \in L_n$
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L_1 \text{ or } x_2 \in L_2 \text{ or } \ldots \text{ or } x_n \in L_n$
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L_1 \text{ or } x_2 \in L_2 \text{ or } \ldots \text{ or } x_n \in L_n$

$\Sigma_1 \quad \Sigma_2 \quad \Sigma_n$

Prover

Verifier

$a_1 \quad a_2 \quad \ldots \quad a_n$

$c_1 \quad c_2 \quad \ldots \quad c_n$

$z_1 \quad z_2 \quad \ldots \quad z_n$
Stacking $\Sigma$-Protocols for Disjunctions

Can we generically compose these $\Sigma$-Protocols for disjunctions?
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L_1 \lor x_2 \in L_2 \lor \ldots \lor x_n \in L_n$

$a_1 \quad a_2 \quad \ldots \quad a_n$

$c_1 \quad c_2 \quad \ldots \quad c_n$

$z_1 \quad z_2 \quad \ldots \quad z_n$

Can we generically compose these $\Sigma$-Protocols for disjunctions?

Without modifying the underlying $\Sigma$-Protocols
Stacking $\Sigma$-Protocols for Disjunctions

Can we generically compose these $\Sigma$-Protocols for disjunctions?

- Without modifying the underlying $\Sigma$-Protocols
- Communication is that same as that of proving a single branch
Applications of such Stacking Compilers

Reduces manual effort of modifying existing techniques
Applications of such Stacking Compilers

- Reduces manual effort of modifying existing techniques
- Newly developed Σ-protocols can also be used to produce stacked proofs immediately
Applications of such Stacking Compilers

- Reduces manual effort of modifying existing techniques
- Newly developed Σ-protocols can also be used to produce stacked proofs immediately
- Empowering protocol designers to choose appropriate Σ-protocols based on their application
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L$ or $x_2 \in L$ or ..... or $x_n \in L$

$a_1$ ..... $a_n$

c_1 ..... c_n

z_1 ..... z_n

$\Sigma$ $\Sigma$ $\Sigma$
Stacking \( \Sigma \)-Protocols for Disjunctions

\[
x_1 \in L \quad \text{or} \quad x_2 \in L \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L
\]

\[
a_1 \quad a_2 \quad \ldots \quad a_n
\]

\[
c_1 \quad c_2 \quad \ldots \quad c_n
\]

\[
z_1 \quad z_2 \quad \ldots \quad z_n
\]
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L \quad or \quad x_2 \in L \quad or \quad \cdots \quad or \quad x_n \in L$

Prover

$a_1 \quad a_2 \quad \cdots \quad a_n$

$c$

$z_1 \quad z_2 \quad \cdots \quad z_n$

Verifier
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L \text{ or } x_2 \in L \text{ or } \ldots \text{ or } x_n \in L$
Stacking \( \Sigma \)-Protocols for Disjunctions

\[ x_1 \in L \text{ or } x_2 \in L \text{ or } \ldots \text{ or } x_n \in L \]

Prover has \( w_2 \)

Prover can compute these messages
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L$ or $x_2 \in L$ or $\ldots$ or $x_n \in L$

Prover

$w_2$

Verifier

Prover has $w_2$
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L$  or  $x_2 \in L$  or  ......  or  $x_n \in L$

Prover

Cheat?

Verifier

Cheat?

Prover has $w_2$
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L$ or $x_2 \in L$ or ..... or $x_n \in L$

Prover has $w_2$

Simulate

Verifier
Stacking $\Sigma$-Protocols for Disjunctions

$$x_1 \in L \quad \text{or} \quad x_2 \in L \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L$$

Common simulation strategy in $\Sigma$-protocols:

- **Step 1:** Sample $c$
- **Step 2:** Compute $a$, $z$

Prover

Verifier

Prover has $w_2$

Simulate

Simulate
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L$ or $x_2 \in L$ or ..... or $x_n \in L$

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$a_2$</th>
<th>.....</th>
<th>$a_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z_1$</td>
<td>$z_2$</td>
<td>.....</td>
<td>$z_n$</td>
</tr>
</tbody>
</table>

Prover has $w_2$

Simulate

 verifier

Simulate

Common simulation strategy in $\Sigma$-protocols:

Step 1: Sample $c$

Step 2: Compute $a, z$
Stacking $\Sigma$-Protocols for Disjunctions

$x_1 \in L$ or $x_2 \in L$ or $\ldots$ or $x_n \in L$

Common simulation strategy in $\Sigma$-protocols:

Step 1: Sample $c$

Step 2: Compute $a, z$
Stacking $\Sigma$-Protocols for Disjunctions

- $x_1 \in L$ or $x_2 \in L$ or ..... or $x_n \in L$

Common simulation strategy in $\Sigma$-protocols:

- Step 1: Sample $c$
- Step 2: Compute $a, z$

Prover

Verifier

Com($c$)

Prover has $w_2$
Stacking $\Sigma$-Protocols for Disjunctions

Common simulation strategy in $\Sigma$-protocols:

Step 1: Sample $c$

Step 2: Compute $a, z$

Simulate $(a_1, z_1), (a_3, z_3), \ldots, (a_n, z_n)$

$op = \text{Equivocate } com \text{ to } [a_1, a_2, \ldots, a_n]$
Stacking $\Sigma$-Protocols for Disjunctions

Common simulation strategy in $\Sigma$-protocols:

Step 1: Sample $c$

Step 2: Compute $a, z$

Prover

$Com(\text{garbage, } a_2, \ldots, \text{garbage})$

Verifier

Simulate $(a_1, z_1), (a_3, z_3) \ldots, (a_n, z_n)$

$op = \text{Equivocate } com \text{ to } [a_1, a_2, \ldots, a_n]$
Stacking $\Sigma$-Protocols for Disjunctions

$\begin{array}{c}
x_1 \in L \quad \text{or} \quad x_2 \in L \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L
\end{array}$

Common simulation strategy in $\Sigma$-protocols:

- **Step 1:** Sample $c$
- **Step 2:** Compute $a, z$

Prover

$Com(\text{garbage, } a_2, \ldots, \text{garbage})$

Verifier

$\text{Prover has } w_2$

$z_1, z_2, \ldots, z_n, op$

Properties of these commitment schemes?
Partially Binding Vector Commitments

$t$-out-of-$n$ positions are binding. Rest can be equivocated.

Binding positions are fixed at the time of commitment.

Binding positions remain hidden from the receiver.

We propose a construction using Discrete Log
Stacking $\Sigma$-Protocol for Disjunctions

$x_1 \in L \ or \ x_2 \in L \ or \ \ldots \ or \ x_n \in L$

$Com(\text{garbage}, a_2, \ldots, \text{garbage})$

$\text{Prover}$

$\text{Verifier}$

Prover has $w_2$

Compute $z_2$ honestly

Simulate $(a_1, z_1), (a_3, z_3), \ldots, (a_n, z_n)$ using $c$

$op = \text{Equivocate com to } [a_1, a_2, \ldots, a_n]$

This is a valid $\Sigma$-protocol for disjunctions. But we haven’t really saved any communication?
Bulkiest Part of a $\Sigma$-Protocol
Bulkiest Part of a $\Sigma$-Protocol

$\text{Prover}$

$\text{Verifier}$

$\text{Hash}(a)$

$c$

$z, \text{open}$

W.l.o.g., Third round messages are the longest!
Stacking $\Sigma$-Protocol for Disjunctions

$\exists x_1 \in L \lor x_2 \in L \lor \ldots \lor x_n \in L$

$Com(\text{garbage, } a_2, \text{garbage})$

$\text{Prover has } w_2$

Compute $z_2$ honestly

Simulate $(a_1, z_1), (a_3, z_3), \ldots, (a_n, z_n)$ using $c$

$op = \text{Equivocate } com \text{ to } [a_1, a_2, \ldots, a_n]$

$z_1 \quad z_2 \quad \ldots \quad z_n, \quad op$

Can we re-use the third-round message of the active branch?
Stacking $\Sigma$-Protocol for Disjunctions

Prover has $w_2$

$Com(\text{garbage, } a_2, \ldots, \text{garbage})$

Compute $z_2$ honestly

Simulate $(a_1, z_1), (a_3, z_3), \ldots, (a_n, z_n)$ using $c, z_2$

$op = \text{Equivocate com to } [a_1, a_2, \ldots, a_n]$

$z_2, op$

$x_1 \in L \lor x_2 \in L \lor \ldots \lor x_n \in L$
Stacking $\Sigma$-Protocol for Disjunctions

$x_1 \in L \quad \text{or} \quad x_2 \in L \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L$

Prover has $w_2$

Prover

Verifier

Compute $z_2$ honestly

Simulate $(a_1, z_1), (a_3, z_3), \ldots, (a_n, z_n)$ using $c, z_2$

$op = \text{Equivocate } com \text{ to } [a_1, a_2, \ldots, a_n]$

Doesn’t work generically. The underlying $\Sigma$-Protocols, must satisfy some properties
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge

Simulation: For any instance $x$ and challenge $c$, first compute a third-round message, then simulate the corresponding first round message.
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge

Simulation: For any instance $x$ and challenge $c$, first compute a third-round message, then simulate the corresponding first round message.

$$\left\{ (a, c, z) \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p) \right\} \approx \left\{ (a, c, z) \mid z \leftarrow D^{(z)}_x, a \leftarrow S^\text{EHVZK}(1^\lambda, x, c, z) \right\}$$
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge

**Simulation:** For any instance $x$ and challenge $c$, first compute a third-round message, then simulate the corresponding first round message.

$$\left\{(a, c, z) \mid r^p \overset{\$}{\leftarrow} \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p)\right\} \approx \left\{(a, c, z) \mid z \overset{\$}{\leftarrow} D_{x,c}^{(z)}; a \leftarrow S^{EHVZK}(1^\lambda, x, c, z)\right\}$$

Property 2: Recyclable Third Round Messages
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge

**Simulation:** For any instance $x$ and challenge $c$, first compute a third-round message, then simulate the corresponding first round message.

\[
\left\{ (a, c, z) \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p) \right\} \approx \left\{ (a, c, z) \mid z \leftarrow D(x, c; a) \leftarrow S^{EHVZK}(1^\lambda, x, c, z) \right\}
\]

Property 2: Recyclable Third Round Messages

Given a fixed challenge, the distribution of possible third round messages for any pair of statements in the language are indistinguishable from each other.
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge

Simulation: For any instance $x$ and challenge $c$, first compute a third-round message, then simulate the corresponding first round message.

$$\left\{(a, c, z) \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p)\right\} \approx \left\{(a, c, z) \mid z \leftarrow \mathcal{D}_{x, c}^{(z)}, a \leftarrow S^{EHVZK}(1^\lambda, x, c, z)\right\}$$

Property 2: Recyclable Third Round Messages

Given a fixed challenge, the distribution of possible third round messages for any pair of statements in the language are indistinguishable from each other.

$$\mathcal{D}_{c}^{(z)} \approx \{z \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p)\}$$
Stackable Properties

Property 1: Extended Honest Verifier Zero-knowledge

\[
\left\{ (a, c, z) \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p) \right\} \approx \left\{ (a, c, z) \mid z \leftarrow D_{x,c}^{z}; a \leftarrow S^{EHVZK}(1^\lambda, x, c, z) \right\}
\]

Property 2: Recyclable Third Round Messages

\[
D_c^{(z)} \approx \left\{ z \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p) \right\}
\]

\[
\left\{ (a, z) \mid r^p \leftarrow \{0, 1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p) \right\} \approx \left\{ (a, z) \mid z \leftarrow D_c^{(z)}; a \leftarrow S^{EHVZK}(1^\lambda, x, c, z) \right\}
\]
Stacked $\Sigma$-Protocol for Disjunctions

Prover has $w_2$

$Com(\text{garbage}, a_2, \ldots, \text{garbage})$

Communication = proof size for proving a single branch + size of commitment + size of opening

Can be short

At least linear in the length of the vector
Recursive Stacking

\[ x_1 \in L \quad \text{or} \quad x_2 \in L \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L \]

1 out of 2 disjunction

\[ \Sigma_2 = \text{Stack } \Sigma \text{ and } \Sigma \]

Communication = | \Sigma | + Commitment + 1
Recursive Stacking

\[ x_1 \in L \quad or \quad x_2 \in L \quad or \quad \ldots \quad or \quad x_n \in L \]

| 1 out of 2 disjunction | \( \Sigma_2 = \text{Stack} \Sigma \text{ and } \Sigma \) | Communication = \(| \Sigma | + \text{Commitment} + 1 \) |
|-------------------------|------------------------------------------------|--------------------------------------------------|
| 1 out of 4 disjunction  | \( \Sigma_4 = \text{Stack} \Sigma_2 \text{ and } \Sigma_2 \) | Communication = \(| \Sigma | + 2 \times \text{Commitment} + 1 + 1 \) |
## Recursive Stacking

\[ x_1 \in L \quad or \quad x_2 \in L \quad or \quad \ldots \quad or \quad x_n \in L \]

<table>
<thead>
<tr>
<th>Disjunction Type</th>
<th>Stack Operation</th>
<th>Communication Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 out of 2</td>
<td>( \Sigma_2 = \text{Stack } \Sigma \text{ and } \Sigma )</td>
<td>(</td>
</tr>
<tr>
<td>1 out of 4</td>
<td>( \Sigma_4 = \text{Stack } \Sigma_2 \text{ and } \Sigma_2 )</td>
<td>(</td>
</tr>
<tr>
<td>1 out of 8</td>
<td>( \Sigma_8 = \text{Stack } \Sigma_4 \text{ and } \Sigma_4 )</td>
<td>(</td>
</tr>
</tbody>
</table>
**Recursive Stacking**

\[ x_1 \in L \quad \text{or} \quad x_2 \in L \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L \]

<table>
<thead>
<tr>
<th>Disjunction</th>
<th>( \Sigma_n = \text{Stack } \Sigma_{n/2} \text{ and } \Sigma_{n/2} )</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 out of 2 disjunction</td>
<td>( \Sigma_2 = \text{Stack } \Sigma \text{ and } \Sigma )</td>
<td></td>
</tr>
<tr>
<td>1 out of 4 disjunction</td>
<td>( \Sigma_4 = \text{Stack } \Sigma_2 \text{ and } \Sigma_2 )</td>
<td></td>
</tr>
<tr>
<td>1 out of 8 disjunction</td>
<td>( \Sigma_8 = \text{Stack } \Sigma_4 \text{ and } \Sigma_4 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disjunction</th>
<th>( \Sigma_n = \text{Stack } \Sigma_{n/2} \text{ and } \Sigma_{n/2} )</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 out of n disjunction</td>
<td>( \Sigma_n = \text{Stack } \Sigma_{n/2} \text{ and } \Sigma_{n/2} )</td>
<td></td>
</tr>
</tbody>
</table>
Examples of Stackable $\Sigma$-Protocols

Many natural sigma protocols are stackable
Examples of Stackable $\Sigma$-Protocols

Many natural sigma protocols are stackable

Example 1: Schnorr’s $\Sigma$-Protocol

\[ R(x, w): x =? = g^x \]
Examples of Stackable $\Sigma$-Protocols

Many natural sigma protocols are stackable

Example 1: Schnorr’s $\Sigma$-Protocol

$R(x, w): x = ? = g^x$

$z = cw + r$
Examples of Stackable $\Sigma$-Protocols

Many natural sigma protocols are stackable

Example 1: Schnorr’s $\Sigma$-Protocol

$R(x, w): x \equiv g^x$

Prover

\[ a = g^r \]

Verifier

\[ c \]

\[ z = cw + r \]

Simulation Strategy: Sample random $z$. Compute $a = g^z x^{-c}$
Examples of Stackable $\Sigma$-Protocols

Many natural sigma protocols are stackable

Example 1: Schnorr’s $\Sigma$-Protocol

Example 2: Graph 3-coloring

Is a graph $G = (V, E)$, 3-colorable?
Examples of Stackable $\Sigma$-Protocols

Many natural sigma protocols are stackable

Example 1: Schnorr’s $\Sigma$-Protocol

Example 2: Graph 3-coloring

Is a graph $G = (V, E)$, 3-colorable?

$\begin{align*}
    a &= \text{commitment of permuted 3-coloring} \\
    c &= \text{random edge in graph} \\
    z &= \text{open colors of the edge}
\end{align*}$
Examples of Stackable \( \Sigma \)-Protocols

Many natural sigma protocols are stackable

Example 1: Schnorr’s \( \Sigma \)-Protocol

Example 2: Graph 3-coloring

Is a graph \( G = (V, E) \), 3-colorable?

\( a = \) commitment of permuted 3-coloring

\( c = \) random edge in graph

\( z = \) open colors of the edge

Prover

Verifier

Independent of instance
**Examples** of Stackable Σ-Protocols

Many natural sigma protocols are stackable

| Example 1: | Schnorr’s Σ-Protocol |
| Example 2: | Graph 3-coloring |
| Example 2: | MPC-in-the-head [IKOS] |
[IKOS07] is Stackable?

For function $R(x, \cdot)$, that takes $w$ as input

Run MPC in the head, commit to views of all parties
[IKOS07] is Stackable?

For function $R(x,.)$, that takes $w$ as input

Run MPC in the head, commit to views of all parties

Choose a random subset of parties
[IKOS07] is Stackable?

Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Open views of the chosen parties

For function $R(x,.)$, that takes $w$ as input
[IKOS07] is Stackable?

For function $R(x,.),$ that takes $w$ as input

- Run MPC in the head, commit to views of all parties
- Choose a random subset of parties
- Open views of the chosen parties
- Simulator
- Choose a random subset of parties
[IKOS07] is Stackable?

Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Open views of the chosen parties

For function $R(x, \cdot)$, that takes $w$ as input

Simulator

Choose a random subset of parties

Simulate the views of these parties' using simulator of the underlying MPC protocol
[IKOS07] is Stackable?

For function $R(x, .)$, that takes $w$ as input

Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Open views of the chosen parties

Simulator

Choose a random subset of parties

Simulate the views of these parties' using simulator of the underlying MPC protocol

Honestly commit to these views and garbage for the remaining parties' views
[IKOS07] is Stackable?

For function $R(x,. )$, that takes $w$ as input

Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Open views of the chosen parties

Simulate the views of these parties' using simulator of the underlying MPC protocol

Honestly commit to these views and garbage for the remaining parties’ views

It is naturally EHVZK. What about recyclable third round messages?
$F$-Universally Simulatable MPC
Adversary’s view in many MPC protocols can be condensed and decoupled from the structure of the functionality being evaluated
**F-Universally Simulatable MPC**

Adversary’s view in many MPC protocols can be condensed and decoupled from the structure of the functionality being evaluated.

Example: Many secret sharing-based MPC (e.g. [BGW88])
$F$-Universally Simulatable MPC

Adversary’s view in many MPC protocols can be condensed and decoupled from the structure of the functionality being evaluated

Example: Many secret sharing-based MPC (e.g. [BGW88])

Simulator simulates random shares for the adversary for each of these gates
F-Universally Simulatable MPC

Adversary’s view in many MPC protocols can be condensed and decoupled from the structure of the functionality being evaluated.

Example: Many secret sharing-based MPC (e.g. [BGW88])

- Simulator simulates random shares for the adversary for each of these gates.
- Given previously simulated shares and the output, simulate the final message.
$F$-Universally Simulatable MPC

Adversary’s view in many MPC protocols can be condensed and decoupled from the structure of the functionality being evaluated.

Example: Many secret sharing-based MPC (e.g. [BGW88])

- Given previously simulated shares and the output, simulate the final message
- Simulator simulates random shares for the adversary for each of these gates
  
  Independent of the function/circuit!

Deterministic computation
**F-Universally Simulatable MPC**

Adversary’s view in many MPC protocols can be condensed and decoupled from the structure of the functionality being evaluated

Example: Many secret sharing-based MPC (e.g. [BGW88])

Given previously simulated shares and the output, simulate the final message

Simulator simulates random shares for the adversary for each of these gates

Deterministic computation

Independent of the function/circuit!
Modified [IKOS07] for $F$-Universally Simulatable MPC

For function $R(x,.)$, that takes $w$ as input

Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Condensed views of the chosen parties and randomness used in corresponding commitments
Modified [IKOS07] for $F$-Universally Simulatable MPC

For function $R(x, .)$, that takes $w$ as input

Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Condensed views of the chosen parties and randomness used in corresponding commitments

Verifier can expand condensed views assuming output is 1, check if commitments are valid and perform all other consistency checks
Run MPC in the head, commit to views of all parties

Choose a random subset of parties

Condensed views of the chosen parties and randomness used in corresponding commitments

For function $R(x,.)$, that takes $w$ as input

Verifier can expand condensed views assuming output is 1, check if commitments are valid and perform all other consistency checks

Since condensed views are independent of the functionality, this protocol now has recyclable third-round message

Modified [IKOS07] for $F$-Universally Simulatable MPC
Disjunctions with Different Languages

\[ x_1 \in L_1 \quad \text{or} \quad x_2 \in L_2 \quad \text{or} \quad \ldots \quad \text{or} \quad x_n \in L_n \]
Disjunctions with Different Languages

\[ x_1 \in L_1 \quad or \quad x_2 \in L_2 \quad or \quad \ldots \quad or \quad x_n \in L_n \]
Disjunctions with Different Languages

\[ x_1 \in L_1 \; \text{or} \; x_2 \in L_2 \; \text{or} \; \ldots \; \text{or} \; x_n \in L_n \]

\[ \Sigma_1 \quad \Sigma_2 \quad \Sigma_n \]

Sometimes same protocol works for different languages
Disjunctions with Different Languages

\[ x_1 \in L_1 \quad or 
\quad x_2 \in L_2 \quad or 
\quad \ldots \quad or 
\quad x_n \in L_n \]

\[ \Sigma_1 \quad \Sigma_2 \quad \Sigma_n \]

Sometimes same protocol works for different languages

If third round messages are over different fields/rings – represent as bits and see what parts can be re-used
Thank You!