Composable Non-interactive Zero-knowledge Proofs in the Random Oracle Model

Yashvanth Kondi

Based on joint work with abhi shelat (Asiacrypt ’22)
In this talk...

- Zero-knowledge proofs (of knowledge)
  - Understand and use their security guarantees

- A taste for how they are designed and analysed
  - Provably secure composition
  - Random Oracle Model

- [Ks 22] Uncover a gap in the literature that was glossed over as folklore—turns out to permit a new kind of attack
  Briefly discuss on how we fix it
Quick Disclaimer

- **What will be covered:**
  Intuitive abstract idea of how to construct composition-safe ZK, how our attack works

- **What won’t be touched:**
  Formalism of definitions, concrete instantiations, efficiency (this is to help understanding, not to hand-wave; please ask if something is unclear!)
Composable
Non-interactive
Zero-knowledge Proofs
in the Random Oracle Model
Zero-knowledge Proofs

- Very powerful cryptographic primitive, introduced by [Goldwasser Micali Rackoff 85]

- Intuition: Prover convinces a Verifier of a statement, without revealing “why” it’s true.
  - Prover typically needs to use some secret information
  - Verifier obtains no useful information about Prover’s secrets
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Zero-knowledge Proofs

- Simple application: proof of possession (key ownership)
Defining Zero-knowledge Proofs

- ZK is intuitive: No information about the key should be leaked by the proof.
- But what does it mean to “know” something?
- “Proof of Knowledge” is formalized by an “extractor” $\text{Ext}$. 
Defining Zero-knowledge Proofs

- ZK is intuitive: No information about the key should be leaked by the proof
- But what does it mean to “know” something?
- “Proof of Knowledge” is formalized by an “extractor” Ext

Zero-knowledge Proof: “I know 🧭 that unlocks 🗝️”
Defining Zero-knowledge Proofs

- ZK is intuitive: No information about the key should be leaked by the proof.
- But what does it mean to “know” something?
- “Proof of Knowledge” is formalized by an “extractor” Ext.

Zero-knowledge Proof: “I know 🔑 that unlocks 🗝️.”

Ext
Defining Zero-knowledge Proofs

- ZK is intuitive: No information about the key should be leaked by the proof

- But what does it mean to “know” something?

- “Proof of Knowledge” is formalized by an “extractor” Ext
Why is Ext special?

- Clearly, Ext must not be an algorithm that just anybody can run.
- Ext has carefully chosen special privileges:
  - Powerful enough to accomplish extraction
  - Still meaningful as a security claim
- We will look at a certain type of ZK proof to build intuition.
Σ Protocols

[Damgård 02]
\[ \Sigma \text{ Protocols} \quad [\text{Damgård 02}] \]

\[ X = \text{key} \]

[Image of a person with Bitcoin glasses, a key, and a building]
Σ Protocols

[Damgård 02]

\[ X = \text{Lock} = w \]
$\Sigma$ Protocols

[Damgård 02]

$X = \text{lock}$

$\text{key} = w$

$P(X, w)$
Σ Protocols

[Damgård 02]

\[ X = \begin{array}{c}
\text{lock} \\
\text{key} = w
\end{array} \]

\[ P(X, w) \quad V(X) \]
\[ \Sigma \text{ Protocols} \]

[Damgård 02]

\[ X = V(X) \]
$\Sigma$ Protocols

[Damgård 02]

$X = V(X)$
Σ Protocols
[Damgård 02]

\[ P(X, w) \]

\[ X = \] (encrypted image)

\[ V(X) \]
\[ \Sigma \text{ Protocols} \]

\[ [\text{Damgård 02}] \]

\[ X = \text{[Lock]} \]

\[ V(X) \]

\[ P(X, w) \]
$\Sigma$ Protocols

[Damgård 02]

$X = \{ \text{lock} \}$

$V(X)$
\( \Sigma \) Protocols

[Damgård 02]

\[ X = V(X) \]

\[ P(X, w) \]

\[ a \]

\[ e \]
Σ Protocols
[Damgård 02]

\[ X = V(X) \]

\[ P(X, w) \]
\[ \Sigma \text{ Protocols} \]

\[ X = \text{Commitment} \]

\[ V(X) \]

\[ P(X, w) \]

[Damgård 02]
\[ \Sigma \text{ Protocols} \]

[Damgård 02]

\[ X = a \]

\[ V(X) \]

\[ P(X, w) \]

Commitment

\[ e \]

Challenge

\[ z \]
**Σ Protocols**  
[Damgård 02]  

\[ X = e^{a} \]  

Commitment \[ a \]  
Challenge \[ e \]  

Response \[ z \]
\[ \Sigma \text{ Protocols} \]

\[ [\text{Damgård 02}] \]

\[ P(X, w) \]

\[ X = \]

\[ a \]

\[ e \]

\[ z \]

\[ \text{Ext} \]
\[ \Sigma \text{ Protocols} \]

[Damgård 02]

\[ X = \]
$\Sigma$ Protocols

$P(X, w)$

$X = \text{Ext}$

$e, z$
$P(X, w)$

$\Sigma$ Protocols

[Damgård 02]

$X = \text{Ext}$

$a$

$e'$

$z'$

$e$  $z$
\[ P(X, w) \]

\[ X = \text{Ext} \]

\[
\begin{align*}
X &= a \\
e & z \\
e' & z'
\end{align*}
\]
Σ Protocols

\[ P(X, w) \]

\[ X = \underline{\text{Ext}} \]

\[ e \quad z \]

\[ e' \quad z' \]
Σ Protocols

[Damgård 02]

\[ X = \]

\[ P(X, w) \]

\[ a \]

\[ \text{Toy example} \]

\[ z = we + f(a) \]

\[ z' = we' + f(a) \]

solve for \( w \)
Σ Protocols

[Damgård 02]

\[ P(X, w) \]

\[ X = \text{Lock} \]

\[ a \]

\[ \text{Ext} \]

\[ e \quad z \]

\[ e' \quad z' \]

**Toy example**

\[ z = we + f(a) \]

\[ z' = we' + f(a) \]

solve for \( w \)

This is a useful protocol feature to keep in mind
Composable
Non-interactive
Zero-knowledge Proofs
in the Random Oracle Model
Composable?
Composable?
Composable?

Ext
Composable?
Composable?

Ext

Ext
Composable?
Composable?
Composable?
Composable?
Composable?

[Diagram showing keys, lock, and Ext blocks]
Composable?
Rewinding extraction strategies are bad for concurrent composition
Straight-line Extraction

• What special privileges can we grant Ext that compose nicely?

• One option is a “Common Reference String”
  - i.e. system parameter for which Ext has a backdoor
  - Well studied, theoretically sound
  - Unsatisfying in practice; trusted generator needed
Composable
Non-interactive
Zero-knowledge Proofs
in the Random Oracle Model
Random Oracle Model

\[ H : \{0,1\}^* \mapsto \{0,1\}^\ell \]
Random Oracle Model

\[ H : \{0,1\}^* \mapsto \{0,1\}^\ell \]
Random Oracle Model

• Began as a heuristic to analyze protocols that use cryptographic hash functions

• Developed as a methodology to design efficient protocols with meaningful provable guarantees

• Intuition:
  - Cryptographic hashes are complex and highly unstructured
  - Unless you evaluate $H(x)$ from scratch, it looks random
Random Oracles as Ext Privilege

\[ H \]
Random Oracles as Ext Privilege

\[ H \]

\[ \text{Ext} \]
Random Oracles as Ext Privilege

$q_i \rightarrow H \rightarrow \text{Ext}$
Random Oracles as Ext Privilege
Random Oracles as Ext Privilege

\[
Q_i \rightarrow H \rightarrow Q_i \\
\]

Ext
Random Oracles as Ext Privilege
Random Oracles as Ext Privilege
Random Oracles as Ext Privilege

- Bob “knows” all of the \( \{Q_i\} \) values queried to \( H \)
- Ext could obtain useful information from \( \{Q_i\} \)
- \( \{Q_i\} \) can be obtained without rewinding
Composable
Non-interactive
Zero-knowledge Proofs
in the Random Oracle Model
Non-interactive

- As the name suggests, a non-interactive proof is a single message protocol

- Useful communication pattern for many applications

- Common methodology: compile $\Sigma$ protocol

- [Pass 03] gave a simple straight-line extractable compiler in the random oracle model
Fischlin’s Compiler

- [Fischlin 05] gave a much more efficient compiler in the same model as [Pass 03]

- More interesting to analyze, and has remained the state of the art for $\Sigma \mapsto \text{NIZK}$ compilers
Fischlin’s Compiler

- [Fischlin 05] gave a much more efficient compiler in the same model as [Pass 03]

- More interesting to analyze, and has remained the state of the art for $\Sigma \mapsto \text{NIZK compilers}$
Fischlin’s Compiler

- [Fischlin 05] gave a much more efficient compiler in the same model as [Pass 03]

- More interesting to analyze, and has remained the state of the art for \( \Sigma \mapsto \text{NIZK} \) compilers
Fischlin’s Compiler

• [Fischlin 05] gave a much more efficient compiler in the same model as [Pass 03]

• More interesting to analyze, and has remained the state of the art for $\Sigma \mapsto \text{NIZK}$ compilers

\[ H(a, e, z) = 0 \]
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample Σ-protocol first message ‘a’

$(a,0,z_0)$
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

$$ (a,0,z_0) $$

$\times$ 0010101

$\times$ $H$
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

\[
\begin{align*}
(a,0,z_0) & \quad \text{0010101} \\
(a, i, z_i) & \quad 1001001
\end{align*}
\]
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

$$(a,0,z_0)$$

$$(a,i,z_i)\quad\times\quad 0010101\quad\times\quad 1001001\quad\times\quad 1001001$$

$$(a,e,z)\quad\checkmark\quad 0000000$$
Fischlin’s Transformation

- Let $\mathcal{H} : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

- $(a,0,z_0)$
- $0010101$
- $(a,i,z_i)$
- $1001001$
- $(a,e,z)$
- $0000000$

Output
Fischlin’s Transformation

- Let $H: \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

Soundness: Except with $\Pr=2^{-\ell}$, $P$ is forced to query more than one accepting transcript to $H$

Completeness: $P$ terminates in poly time when $\ell$ is small, i.e. $O(\log \kappa)$
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^{\ell}$ be a random oracle

Sample $\Sigma$-protocol first message ‘a’

<table>
<thead>
<tr>
<th>Message</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a,0,z_0)$</td>
<td>0010101</td>
</tr>
<tr>
<td>$(a,i,z_i)$</td>
<td>100101</td>
</tr>
<tr>
<td>$(a,e,z)$</td>
<td>0000000</td>
</tr>
</tbody>
</table>

This gives Ext the values $(e, z)$ and $(e', z')$ as needed, by looking at queries made to $H$

**Soundness:** Except with $\Pr=2^{-\ell}$, $P$ is forced to query more than one accepting transcript to $H$

**Completeness:** $P$ terminates in poly time when $\ell$ is small, i.e. $O(\log \kappa)$
Fischlin’s Transformation

- Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

- $H(a,0,z_0)$
- $0010101$
- $H(a,i,z_i)$
- $1001001$
- $H(a,e,z)$
- $0000000$

Output

This gives Ext the values $(e,z)$ and $(e',z')$ as needed, by looking at queries made to $H$

**Soundness**: Except with $\Pr = 2^{-\ell}$, $P$ is forced to query more than one accepting transcript to $H$

**Completeness**: $P$ terminates in poly time when $\ell$ is small, i.e. $O(\log \kappa)$
Fischlin’s Transformation

• Let $H : \{0,1\}^* \mapsto \{0,1\}^\ell$ be a random oracle

Sample $\Sigma$-protocol first message ‘$a$’

$(a,0,z_0)$

0010101

×

$(a,i,z_i)$

1001001

×

$(a,e,z)$

0000000

Output

This gives Ext the values $(e, z)$ and $(e', z')$ as needed, by looking at queries made to $H$

**Soundness**: Except with $\Pr = 2^{-\ell}$, $P$ is forced to query more than one accepting transcript to $H$

**Completeness**: $P$ terminates in poly time when $\ell$ is small, i.e. $O(\log \kappa)$

**Problem!**

**Full Soundness**: Repeat $r$ times
Fischlin vs Pass: Qualitative

- Pass’ compiler works for any Sigma protocol
- Fischlin’s compiler works for a restricted class of Sigma protocols with ‘quasi-unique responses’
- Supported by many standard Sigma protocols (eg. DLog), but many may not—especially if a statement can have multiple witnesses (eg. Pedersen Commitment opening, 1-of-2 witnesses, etc.)
Quasi-unique Responses [Fischlin 05]

**Hard**: \((a, e, z, z') \leftarrow A(pp)\) such that \(V(a, e, z) = V(a, e, z') = 1\)

Fixing \((a, e)\) fixes \(z\)
Quasi-unique Responses [Fischlin 05]

**Hard:** \((a, e, z, z') \leftarrow \mathcal{A}(\text{pp})\) such that
\[ V(a, e, z) = V(a, e, z') = 1 \]

Fixing \((a, e)\) fixes \(z\)

\[
\begin{align*}
(a,0,z_0) & \quad \times \\
(a, i, z_i) & \quad \times \\
(a, e, z) & \quad \checkmark
\end{align*}
\]
Quasi-unique Responses [Fischlin 05]

**Hard**: \((a, e, z, z') \leftarrow \mathcal{A}(pp)\) such that
\[
V(a, e, z) = V(a, e, z') = 1
\]

Fixing \((a, e)\) fixes \(z\)

Easy to see how this ties into soundness of Fischlin’s compiler
Quasi-unique Responses [Fischlin 05]

**Hard**: $(a, e, z, z') \leftarrow \mathcal{A}(\text{pp})$ such that $V(a, e, z) = V(a, e, z') = 1$

Fixing $(a, e)$ fixes $z$

Easy to see how this ties into soundness of Fischlin’s compiler

Prover can produce a proof without ever having to try more than one challenge
Quasi-unique Responses [Fischlin 05]

**Hard:** \((a, e, z, z') \leftarrow \mathcal{A}(\text{pp})\) such that
\[ V(a, e, z) = V(a, e, z') = 1 \]

Fixing \((a, e)\) fixes \(z\)

Easy to see how this ties into soundness of Fischlin’s compiler

Extractor needs transcripts with different challenges

Prover can produce a proof without ever having to try more than one challenge

Recall:
Is it *really* necessary, though?

- **Folklore**: breaking Sigma protocol abstraction, and simply ‘adjusting syntax’ of the extractor is usually sufficient to preserve Proof of Knowledge

- This is demonstrated by the Sigma protocol to prove knowledge of one-out-of-two witnesses [Cramer Damgård Schoenmakers 94]

- In [K shelat 22] we formalize this folklore
What about Zero-knowledge?

- Interestingly, Fischlin’s proof of Zero-knowledge also depends on quasi-unique responses.
- Unlike extraction, it is not intuitive as to why (or whether it’s even necessary).
- [K shelat 22]: In the absence of unique responses, an explicit attack on Witness Indistinguishability (WI).
Witness Indistinguishability

- The following kind of statement finds many applications:

I know either 🗝️ OR 🔓
Witness Indistinguishability

- The following kind of statement finds many applications:

  Zero-knowledge Proof: “I know \(\) that unlocks \(\) OR \(\)

  OR

  "I know \(\) that unlocks \(\) OR \(\) that unlocks \(\)"
Witness Indistinguishability

- The following kind of statement finds many applications:

  
  Zero-knowledge Proof: "I know that unlocks OR that unlocks"

  Witness Indistinguishable: No information about which key Bob actually has (Implied by ZK)
Witness Indistinguishability

• The following kind of statement finds many applications:

Zero-knowledge Proof:
“I know that unlocks OR that unlocks”

Witness Indistinguishable: No information about which key Bob actually has (Implied by ZK)

Important note: This holds even if both keys are actually known to bank (like known plaintext security)
Useful Fact

- Some $\Sigma$ protocols have the following property:
  (including some multi-witness ones)
Useful Fact

- Some $\Sigma$ protocols have the following property: (including some multi-witness ones)

Taken in isolation, no information about which key Bob has
Useful Fact

• Some $\Sigma$ protocols have the following property: (including some multi-witness ones)

$\Sigma \triangleleft a \triangleleft e \triangleleft \left\langle \begin{array}{c} \text{OR} \\ \text{Bank} \end{array} \right\rangle \triangleleft \left\langle \begin{array}{c} \text{Bob} \\ \text{Key} \end{array} \right\rangle \triangleleft z$

Taken in isolation, no information about which key Bob has
Useful Fact

- Some $\Sigma$ protocols have the following property:
  (including some multi-witness ones)
Useful Fact

- Some $\Sigma$ protocols have the following property:
  (including some multi-witness ones)

(Before Bob’s response) compute $z'$ and $z^*$
Useful Fact

- Some $\Sigma$ protocols have the following property: (including some multi-witness ones)

(Before Bob’s response) compute $z'$ and $z^*$
Useful Fact

- Some $\Sigma$ protocols have the following property:
  (including some multi-witness ones)

(Before Bob’s response)
compute $z'$ and $z^*$
Useful Fact

- Some $\Sigma$ protocols have the following property:
  (including some multi-witness ones)

\[ a \xrightarrow{e'} z' \xrightarrow{e'} a e z \]

(Before Bob’s response) compute $z'$ and $z^*$
Useful Fact

- Some $\Sigma$ protocols have the following property: (including some multi-witness ones)

\[ a e z \]

\[ a' e' z' \]

(Before Bob’s response) compute $z'$ and $z^*$
Useful Fact

• Some $\Sigma$ protocols have the following property: (including some multi-witness ones)

\[
\begin{align*}
    e' & \\
    a & \\
    a e z & \\
    \text{(Before Bob’s response) compute } z' \text{ and } z^* 
\end{align*}
\]
Attack Strategy

\[ H(a, e, z) = 0 \]
Reveals nothing about Bob’s key in isolation

\[ H(a, e, z) = 0 \]
Reveals nothing about Bob’s key in isolation

$H(a, e, z) = 0$

- Imagine we could ask Bob to answer challenge $e'$
  ...his answer ($z'$ or $z^*$) would determine which key he has

- Turns out we can achieve this effect by probing $H$
  (with no special privileges)
Probing Strategy

If both possibilities “agree” at $e$, then they “disagree” at any $e' \neq e$
Probing Strategy

If both possibilities “agree” at $e$, then they “disagree” at any $e' \neq e$
Probing Strategy

Common $a$

If both possibilities “agree” at $e$, then they “disagree” at any $e' \neq e$
Probing Strategy

If both possibilities “agree” at \( e \), then they “disagree” at any \( e' \neq e \).
Probing Strategy

If both possibilities “agree” at \( e \), then they “disagree” at any \( e' \neq e \)

<table>
<thead>
<tr>
<th>Common ( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key: ((0, z_0') (1, z_1') \ldots (e, z))</td>
</tr>
</tbody>
</table>

C...
Probing Strategy

Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”.
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”

\[
(0, z'_0) \rightarrow (1, z'_1) \rightarrow \cdots \\
(0, z_0^*) \rightarrow (1, z_1^*) \rightarrow \cdots \rightarrow (e, z) \\
\]

W.h.p., only one path—induced by one of the two keys—is plausible
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”

\[
(0, z_0') 
\rightarrow 
(1, z_1') 
\rightarrow 
\vdots 
\rightarrow 
H \
\rightarrow 
\cdots
\]

W.h.p., only one path—induced by one of the two keys—is plausible
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”

Probing Strategy

W.h.p., only one path—induced by one of the two keys—is plausible
Given $(a, e, z)$ produced by Fischlin’s compiler, we can test which path is “plausible”

This path induces fresh queries to $H$

W.h.p., only one path—induced by one of the two keys—is plausible
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”.

This path induces \textbf{fresh} queries to \(H\).

W.h.p., only one path—induced by one of the two keys—is plausible.
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible.”

This path induces \textit{fresh} queries to \(H\).

W.h.p., only one path—induced by one of the two keys—is plausible.
Given \((a, e, z)\) produced by Fischlin’s compiler, we can test which path is “plausible”.

This path induces fresh queries to \(H\).

Would have terminated here.

W.h.p., only one path—induced by one of the two keys—is plausible.
How to Fix it? [Ks 22]

• The probing strategy very strongly depends on being able to “re-trace” the Prover’s steps
  - This is enabled by the deterministic nature of Fischlin’s compiler

• We showed that randomizing the order in which the Prover tries challenges will fix the problem

• We strengthen Fischlin’s technique to be good enough to apply to most useful Sigma protocols
In Summary

- We saw what non-interactive zero-knowledge proofs of knowledge are, how they can be used.

- We got a taste for how they are designed and analysed, and how to understand security guarantees like concurrent composition and ROM.

- We uncovered a gap in the literature that was glossed over as folklore, and saw how it turned out to be a vulnerability (and briefly discussed how it’s now fixed).

Questions?

eprint.iacr.org/2022/393

Thanks Eysa Lee for
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \quad X = g^x \quad V(X) \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]

\[ r \leftarrow \mathbb{Z}_q \]

\[ X = g^x \]

\[ V(X) \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]
\[ r \leftarrow \mathbb{Z}_q \]

\[ X = g^x \]

\[ a = g^r \]

\[ V(X) \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]

\[ r \leftarrow \mathbb{Z}_q \]

\[ X = g^x \]

\[ a = g^r \]

\[ e \in \mathbb{Z}_q \]

\[ V(X) \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]

\[ r \leftarrow \mathbb{Z}_q \]

\[ a = g^r \]

\[ e \in \mathbb{Z}_q \]

\[ z = xe + r \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]
\[ r \leftarrow \mathbb{Z}_q \]

\[ X = g^x \]
\[ a = g^r \]
\[ e \in \mathbb{Z}_q \]
\[ z = xe + r \]
\[ g^z \overset{?}{=} X^e \cdot a \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]

\[ r \leftarrow \mathbb{Z}_q \]

\[ X = g^x \]

\[ a = g^r \]

\[ e \in \mathbb{Z}_q \]

\[ z = xe + r \]

\[ g^z \overset{?}{=} X^e \cdot a \]
Example: Schnorr PoK of Discrete Logarithm

\[ P(X, x) \]

\[ r \leftarrow \mathbb{Z}_q \]

\[ X = g^x \]

\[ a = g^r \]

\[ e \in \mathbb{Z}_q \]

\[ z = xe + r \]

\[ g^z \overset{?}{=} X^e \cdot a \]

**Ext(a, (e, z), (e', z')):**
\[ x = (z' - z) / (e' - e) \]
Output \( x \)

**HVZK \( S(e):**
\[ z \leftarrow \mathbb{Z}_q \]
\[ a = g^z / X^e \]
Output \((a, z)\)
The Fiat-Shamir Transform

- [Fiat Shamir 87] provides a simple method to compile any public-coin protocol to a non-interactive proof, given a suitably chosen hash function
The Fiat-Shamir Transform

- [Fiat Shamir 87] provides a simple method to compile any public-coin protocol to a non-interactive proof, given a suitably chosen hash function $P(X, w) = V(X)$.

\[
\begin{align*}
\text{Verify}(a, e, z) & \quad \text{Verify}(a, e, z) \\
\end{align*}
\]
The Fiat-Shamir Transform

- [Fiat Shamir 87] provides a simple method to compile any public-coin protocol to a non-interactive proof, given a suitably chosen hash function

\[ P(X, w) \]

\[ V(X) \]

\[ a \]

\[ e = H(X, a) \]

\[ z \]

\[ \text{Verify}(a, e, z) \]
The Fiat-Shamir Transform

- [Fiat Shamir 87] provides a simple method to compile any public-coin protocol to a non-interactive proof, given a suitably chosen hash function

\[ P(X, w) \quad V(X) \]

\[ a, z \quad e = H(X, a) \]

Verify\((a, e, z)\)
Fiat-Shamir: Security

- “Forking” extraction strategy in Random Oracle Model [Pointcheval Stern 96]:

\[ p^* \]

\[ \begin{align*}
  a_0 & \quad a_i & \quad a_m \\
  e_0 & \quad e_i & \quad e_m \\
  \vdots & \quad \vdots & \quad \vdots \\
\end{align*} \]

Output \((a_i, e_i, z_i)\)
Fiat-Shamir: Security

- “Forking” extraction strategy in Random Oracle Model [Pointcheval Stern 96]:

Output \((a_i, e_i, z_i)\)
Fiat-Shamir: Security

• “Forking” extraction strategy in Random Oracle Model [Pointcheval Stern 96]:

\[
\begin{align*}
P^* & \quad a_0 \quad e_0 \\
& \quad \vdots \\
& \quad a_i \quad e_i \\
& \quad \vdots \\
& \quad a_m \quad e_m
\end{align*}
\]

\[
\begin{align*}
P^* & \quad a_0 \quad e_0 \\
& \quad \vdots \\
& \quad a_i \quad e_i \\
& \quad \vdots \\
& \quad a_m \quad e_m
\end{align*}
\]

Output \((a_i, e_i, z_i)\)
Fiat-Shamir: Security

• “Forking” extraction strategy in Random Oracle Model [Pointcheval Stern 96]:

\[
\begin{align*}
&\mathcal{P}^* \\
&\begin{array}{c}
a_0 \\
e_0 \\
\vdots \\
a_i \\
e_i \\
\vdots \\
a_m \\
e_m
\end{array} \\
&H \\
&\text{Output } (a_i, e_i, z_i)
\end{align*}
\]

\[
\begin{align*}
&\mathcal{P}^* \\
&\begin{array}{c}
a_0 \\
e_0 \\
\vdots \\
a_i \\
e_i \\
\vdots \\
a_m \\
e_m
\end{array} \\
&H^* \\
&\text{Output } (a_i, e_i^*, z_i^*)
\end{align*}
\]
Fiat-Shamir: Security

- “Forking” extraction strategy in Random Oracle Model [Pointcheval Stern 96]:

Output \((a_i, e_i, z_i)\)

Output \((a_i, e^*_i, z^*_i)\)

\[
\begin{align*}
P^* & \quad a_0 \quad e_0 \\
& \quad \vdots \\
& \quad a_i \quad e_i \\
& \quad \vdots \\
& \quad a_m \quad e_m
\end{align*}
\]

\[
\begin{align*}
P^* \quad a_0 \quad e_0 \\
& \quad \vdots \\
& \quad a_i \quad e_i \\
& \quad a^*_m \quad e^*_m
\end{align*}
\]

\[
\text{Outputs witness } w
\]

\[
\text{Ext}\left(\left(\begin{array}{c}
(a_i, e_i) \\
(a_i, e_i^*)
\end{array}\right),
(z_i, z_i^*)\right)
\]

Fiat-Shamir: Security

- “Forking” extraction strategy in Random Oracle Model [Pointcheval Stern 96]:

\[
\begin{align*}
\mathbb{P}^* & \quad a_0 \quad \vdots \quad a_i \quad \vdots \quad a_m \\
& \quad e_0 \quad \vdots \quad e_i \quad \vdots \quad e_m \\
& \quad \quad H \\
& \text{Output } (a_i, e_i, z_i) \\
& \quad p
\end{align*}
\]

\[
\begin{align*}
\mathbb{P}^* & \quad a_0 \quad \vdots \quad a_i \quad \vdots \quad a_m \\
& \quad e_0 \quad \vdots \quad e_i \quad \vdots \quad e_m \\
& \quad \quad H^* \\
& \text{Output } (a_i, e_i^*, z_i^*) \\
& \quad p \\
& \text{Ext} \left( (a_i, e_i) (a_i, e_i) \right) z_i, z_i^* \\
& \text{Outputs witness } w \\
& \approx p^2
\end{align*}
\]
Fiat-Shamir Compilation

- Advantages:
  - Simple to describe/implement
  - Very efficient; proving, verification cost exactly the same as input Σ-protocol

- Downsides:
  - Forking strategy does not compose; unclear how to prove concurrent security
  - Quadratic security loss
Straight-line Extraction

- Formalized by [Pass 03] in the Random Oracle Model:

\[ P^* H Q_0 r_0 Q_i r_i Q_m r_m \]
\[ \vdots \]

\[ \mathcal{P} (Q_0, r_0, \ldots, Q_m, r_m) \]

Outputs witness \( w \)

\[ p^* \approx p \]
Straight-line Extraction

- Formalized by [Pass 03] in the Random Oracle Model:

\[
P^* \rightarrow Q_0, r_0 \\
\vdots \\
Q_i, r_i \\
\vdots \\
Q_m, r_m \\
\]

Supports concurrent composition [Pass 03]

\[\{Q_0, r_0\}, \ldots, (Q_m, r_m)\]

Outputs witness \(w\)

Probability of success:

\[p \approx p\]
Logical OR-Composition of Σ Protocols

[Cramer Damgård Schoenmakers 94]

\[ P_{\text{OR}}(w_b) \quad x_0, x_1 \quad V \]
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$$\Sigma(w_b) \quad P_{\Sigma}(w_b) \quad P_{\text{OR}}(w_b) \quad x_0, x_1 \quad V$$
Logical OR-Composition of $\Sigma$ Protocols

$P_{\Sigma}(w_b)$ $\xrightarrow{a_b} P_{\text{OR}}(w_b)$ $x_0, x_1$ $V$

[Cramer Damgård Schoenmakers 94]
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$P_\Sigma(w_b) \xrightarrow{a_b} P_{OR}(w_b) \xrightarrow{(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow Sim(x_{1-b})} x_0, x_1 \xrightarrow{V}$
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

\[ P_{\Sigma}(w_b) \quad \xrightarrow{a_b} \quad P_{\text{OR}}(w_b) \]

\[ (a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b}) \]

\[ \rightarrow a_0, a_1 \]

\[ \rightarrow \quad \rightarrow \]

\[ x_0, x_1 \quad V \]
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

\[ P_{\Sigma}(w_b) \xrightarrow{a_b} P_{\text{OR}}(w_b) \xrightarrow{(a_{1-b}, e_{1-b}, z_{1-b})} \xleftarrow{\text{Sim}(x_{1-b})} x_0, x_1 \xrightarrow{V} \]

\[ \xrightarrow{a_0, a_1} \xleftarrow{e} \]
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$P_\Sigma(w_b) \xrightarrow{a_b} P_{OR}(w_b)$

$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$

$a_0, a_1 \xrightarrow{e} x_0, x_1 \xrightarrow{V} e_b = e - e_{1-b}$
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$P_{\Sigma}(w_b)$

$P_{\text{OR}}(w_b)$

$x_0, x_1$

$V$

$a_b$

$e_b = e - e_{1-b}$

$a_0, a_1$

$e$

$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$
Logical OR-Composition of $\Sigma$ Protocols

$P_\Sigma(w_b)$  $\rightarrow$  $P_{\text{OR}}(w_b)$  $\rightarrow$  $x_0, x_1$  $\rightarrow$  $V$

$\rightarrow$  (a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$

$\rightarrow$  $a_0, a_1$

$\rightarrow$  $e_b = e - e_{1-b}$

$\leftarrow$  $e$

$\leftarrow$  $e$

Recall: violates unique responses

[Cramer Damgård Schoenmakers 94]
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$P_\Sigma(w_b)$

$P_{\text{OR}}(w_b)$

$x_0, x_1$

$V$

$a_b$

$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$

$a_0, a_1$

$e$

$e = e - e_{1-b}$

$(e_0, z_0), (e_1, z_1)$

$P_\Sigma(w_b) \rightarrow a_b$

$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$

$a_0, a_1$

$e$

$e = e - e_{1-b}$

$(e_0, z_0), (e_1, z_1)$

$P_{\text{OR}}(w_b) \leftarrow e$
Logical OR-Composition of $\Sigma$ Protocols

$P_\Sigma(w_b)$

$P_{OR}(w_b)$

$x_0, x_1$

$V$

$P_\Sigma(w_b) \xrightarrow{a_b} (a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$

$e_b = e - e_{1-b}$

$e_b = e - e_{1-b}$

$(e_0, z_0), (e_1, z_1)$

Both are accepting
Logical OR-Composition of $\Sigma$ Protocols

[Recall: violates unique responses]

\[ P_{\Sigma}(w_b) \]

\[ a_b \]

\[ (a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b}) \]

\[ e_b = e - e_{1-b} \]

\[ e \]

\[ (e_0, z_0), (e_1, z_1) \]

Both are accepting
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$$P_{OR}(w_b) \quad x_0, x_1 \quad V$$

$$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$$

$$e_b = e - e_{1-b}$$

Recall: $$(a, e, z, z') \leftarrow A(pp)$$ violates unique responses
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

\[ P_{OR}(w_b) \]

\[ (a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b}) \]

\[ e_b = e - e_{1-b} \]

\[ a_0, a_1 \]

\[ e \]

\[ (e_0, z_0), (e_1, z_1) \]

\[ V \]

Recall: \((a, e, z, z') \leftarrow \mathcal{A}(pp)\) violates unique responses

... but what does \((a, e, z, z')\) look like here?
Logical OR-Composition of $\Sigma$ Protocols

[Recall: $(a, e, z, z') \leftarrow \mathcal{A}(\text{pp})$ violates unique responses]

... but what does $(a, e, z, z')$ look like here?

$P_{OR}(w_b)$

$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$

$e_b = e - e_{1-b}$

$x_0, x_1$

$V$

$\leftarrow (e_0, z_0), (e_1, z_1)$
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

\[ P_{OR}(w_b) \]

\[(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b}) \]

\[ e_b = e - e_{1-b} \]

\[ \begin{array}{c}
    a_0, a_1 \\
    \hline
    e
\end{array} \]

Recall: \((a, e, z, z') \leftarrow A(pp)\) violates unique responses

... but what does \((a, e, z, z')\) look like here?

\[(e_0, z_0), (e_1, z_1)\]
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

$$P_{OR}(w_b)$$

$$(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b})$$

$$e_b = e - e_{1-b}$$

$$x_0, x_1$$

$$V$$

Recall: $$(a, e, z, z') \leftarrow A(pp)$$ violates unique responses

... but what does $$(a, e, z, z')$$ look like here?

$$(e'_0, z'_0), (e'_1, z'_1)$$
Logical OR-Composition of $\Sigma$ Protocols

[Cramer Damgård Schoenmakers 94]

\[ P_{\text{OR}}(w_b) \]

\[(a_{1-b}, e_{1-b}, z_{1-b}) \leftarrow \text{Sim}(x_{1-b}) \]

\[ e_b = e - e_{1-b} \]

\[ x_0, x_1 \quad V \]

Recall: \((a, e, z, z') \leftarrow A(pp)\) violates unique responses

... but what does \((a, e, z, z')\) look like here?

\[ (e'_0, z'_0), (e'_1, z'_1) \]

Either \(e_0 \neq e'_0\), or \(e_1 \neq e'_1\)
Logical OR-Composition of Σ Protocols

Recall: $(a, e, z, z') \leftarrow A(pp)$ violates unique responses
... but what does $(a, e, z, z')$ look like here?

Either $e_0 \neq e'_0$, or $e_1 \neq e'_1$

$w_b \leftarrow \text{Ext}(a_b, (e_b, z_b), (e'_b, z'_b))$
Logical OR-Composition of $\Sigma$ Protocols

$P_{OR}$

$z_{1-b} \leftarrow \text{Sim}(x_{1-b})$

$a_0, a_1$

$e$

$e - e_{1-b}$

Recall: $(a, e, z, z') \leftarrow \mathcal{A}(pp)$ violates unique responses

... but what does $(a, e, z, z')$ look like here?

Either $e_0 \neq e'_0$, or $e_1 \neq e'_1$

$w_b \leftarrow \text{Ext}(a_b, (e_b, z_b), (e'_b, z'_b))$

Quasi-unique responses not strictly necessary for extraction (folklore)

[Cramer Damgård Schoenmakers 94]
Tightening Conditions for Extraction

\[ P(X, w) \]

\[ V(X) \]

2-special soundness:

\[ w \leftarrow \text{Ext}(X, a, (e_1, z_1), (e_2, z_2)) \text{ such that } R(X, w) = 1 \]
Tightening Conditions for Extraction

\[ P(X, w) \] \quad \text{such that} \quad w \leftarrow \text{Ext}(X, a, (e_1, z_1), (e_2, z_2)) \text{ such that } R(X, w) = 1

\[ V(X) \]
Tightening Conditions for Extraction

\[ P(X, w) \] \[ \rightarrow \] \[ a \] \[ \rightarrow \] \[ e \] \[ \leftarrow \] \[ z \] \[ \rightarrow \] \[ V(X) \]

Strong 2-special soundness:

\[ w \leftarrow \operatorname{Ext}(X, a, (e_1, z_1), (e_2, z_2)) \text{ such that } R(X, w) = 1 \]

\[ e_1 \neq e_2 \text{ OR } z_1 \neq z_2 \]
Tightening Conditions for Extraction

\[ P(X, w) \]

\[ V(X) \]

\[
\begin{align*}
  a & \quad \quad \\
  e & \quad \quad \\
  z & \quad \quad \\
\end{align*}
\]

Strong 2-special soundness:

\[
 w \leftarrow \text{Ext}(X, a, (e_1, z_1), (e_2, z_2)) \text{ such that } R(X, w) = 1 \\
 e_1 \neq e_2 \text{ OR } z_1 \neq z_2
\]

...are we done?