Adaptively Secure Succinct Garbled RAM with Persistent Memory

Ran Canetti, Yilei Chen, Justin Holmgren, Mariana Raykova

DIMACS workshop
MIT Media Lab
June 8~10, 2016

: June 11, 2016, Boston, heavy snow.

: June 11, 2016, Boston, heavy snow. Alice finds a quasi-polynomial time algorithm for factoring.

: June 11, 2016, Boston, heavy snow. Alice finds a quasi-polynomial time algorithm for factoring.

Alice

: June 11, 2016, Boston, heavy snow. Alice finds a quasi-polynomial time algorithm for factoring.

: Instead of submitting to STOC, she thinks it’s cool to write a program and show off to her friends.

\> Factoring.hs RSA2048

> Factoring.hs RSA2048

Running time 7 hrs 34 mins

25195908475…20720357

= 83990...4079279 x 3091701...723883

Next question

: It is slow on her laptop (quasi-polynomial time, you know) … cannot fit into a party.

: It is slow on her laptop (quasi-polynomial time, you know) … cannot fit into a party.

: So she turns to cloud, but clouds are big brothers

: It is slow on her laptop (quasi-polynomial time, you know) … cannot fit into a party.

: So she turns to cloud, but clouds are big brothers

: She heard that one can delegate the computation in a way that the server learns only the output of the computation but nothing else

“My friends and NSA will be shocked by the runtime without learning anything other than the output”

“The algorithm has huge preprocessing, stores lots of non-zero points on the Zeta function ...”

“My friends and NSA will be shocked by the runtime without learning anything other than the output”

"The algorithm has huge preprocessing, stores lots of non-zero points on the Zeta function ..."

"My friends and NSA will be shocked by the runtime without learning anything other than the output"

"Wait ... the audiences already know too much."

> sudo apt-get install   FHE

```
> sudo apt-get install   FHE
> FHE   Factoring.hs
```

> sudo apt-get install   FHE

> FHE   Factoring.hs

Turning the program into circuits ...

```
> sudo apt-get install   FHE
> FHE   Factoring.hs
Turning the program into circuits ...
^C
```

```
> sudo apt-get install   FHE
> FHE   Factoring.hs
Turning the program into circuits ...
^C
>
> sudo apt-get install  Yao
> Yao   Factoring.hs
```

> sudo apt-get install FHE

> FHE Factoring.hs

Turning the program into circuits ...

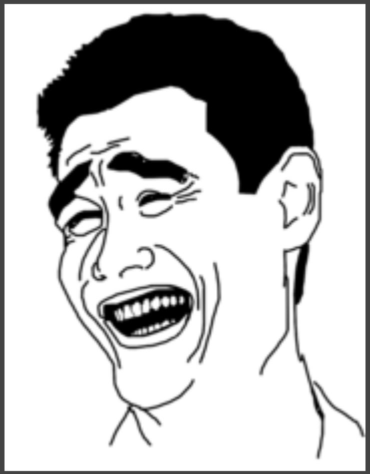
^C

>

> sudo apt-get install Yao

> Yao Factoring.hs

Still turning the program into circuits ...



#Yao

```
> sudo apt-get install   FHE
> FHE   Factoring.hs
Turning the program into circuits ...
^C
>
> sudo apt-get install  Yao
> Yao   Factoring.hs
Still turning the program into circuits ...
^C^C^C^C^C^C^C
>
```

```
> sudo apt-get install  GRAM_Lu_Ostrovsky
> GRAM_Lu_Ostrovsky  Factoring.hs
```

> sudo apt-get install GRAM_Lu_Ostrovsky
> GRAM_Lu_Ostrovsky Factoring.hs

Warning: Program size as big as the running time, continue (y) or not (n)

```
> sudo apt-get install  GRAM_Lu_Ostrovsky
> GRAM_Lu_Ostrovsky  Factoring.hs
Warning: Program size as big as the running time,
continue (y) or not (n)
n
>
```

> sudo apt-get install PRAM

```
> sudo apt-get install  PRAM
> PRAM  Factoring.hs
```

> sudo apt-get install PRAM
> PRAM Factoring.hs
Done -> PRAM_Factoring

```
> sudo apt-get install  PRAM
> PRAM  Factoring.hs
Done -> PRAM_Factoring
> PRAM_Factoring   RSA2048
```

> sudo apt-get install PRAM

> PRAM Factoring.hs

Done -> PRAM_Factoring

> PRAM_Factoring   RSA2048

Warning: cannot adaptively choose functions or inputs, security at user’s own risk, continue (y) or not (n)

> sudo apt-get install PRAM

> PRAM Factoring.hs

Done -> PRAM_Factoring

> PRAM_Factoring   RSA2048

Warning: cannot adaptively choose functions or inputs, security at user's own risk, continue (y) or not (n)

n

“Huge amount of preprocessed data reusable”

“Adaptively pick integers”

“Don’t turn into circuits, don’t blow up too much”

Garbling/randomized encoding for RAM with persistent memory

Garbling/randomized encoding for RAM with persistent memory

Gen => msk

Garbling/randomized encoding for RAM with persistent memory

Gen => msk

msk + D_0 => $G(D_0)$

Garbling/randomized encoding for RAM with persistent memory

Gen => msk

msk + D_0 => $G(D_0)$

msk + P_1 => $G(P_1)$

Garbling/randomized encoding for RAM with persistent memory

Gen => msk

msk + D_0 => $G(D_0)$

msk + P_1 => $G(P_1)$

Eval $G(D_0)$ $G(P_1)$ => $P_1(D_0)$

Garbling/randomized encoding for RAM with persistent memory

Persistency

Gen => msk

msk + D_0 => $G(D_0)$

msk + P_1 => $G(P_1)$

Eval $G(D_0)$ $G(P_1)$ => $P_1(D_0)$ $G(D_1)$

Garbling/randomized encoding for RAM with persistent memory

Persistency

Gen => msk

msk + D_0 => $G(D_0)$

msk + P_1 => $G(P_1)$

Eval $G(D_0)$ $G(P_1)$ => $P_1(D_0)$ $G(D_1)$

msk + P_2 => $G(P_2)$

Garbling/randomized encoding for RAM with persistent memory

Persistency

Gen => msk

msk + D_0 => $G(D_0)$

msk + P_1 => $G(P_1)$

Eval $G(D_0)$ $G(P_1)$ => $P_1(D_0)$ $G(D_1)$

msk + P_2 => $G(P_2)$

Eval $G(D_1)$ $G(P_2)$ => $P_2(D_1)$ $G(D_2)$

...

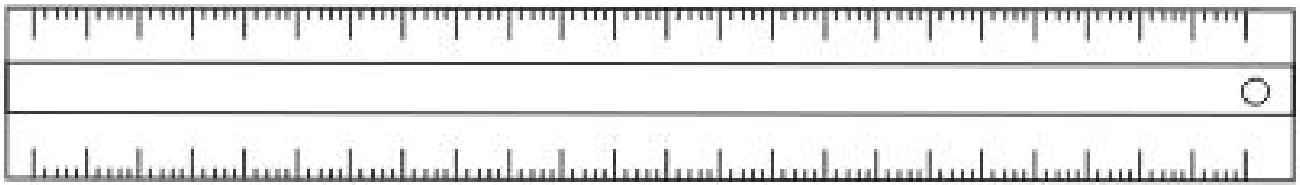
Garbling/randomized encoding for RAM with persistent memory

Succinct

D_0

$G(D_0)$

P_1



$G(P_1)$

Garbling/randomized encoding for RAM with persistent memory

Adaptively simulation secure

?

?

$P_1(D_0)$

Garbling/randomized encoding for RAM with persistent memory

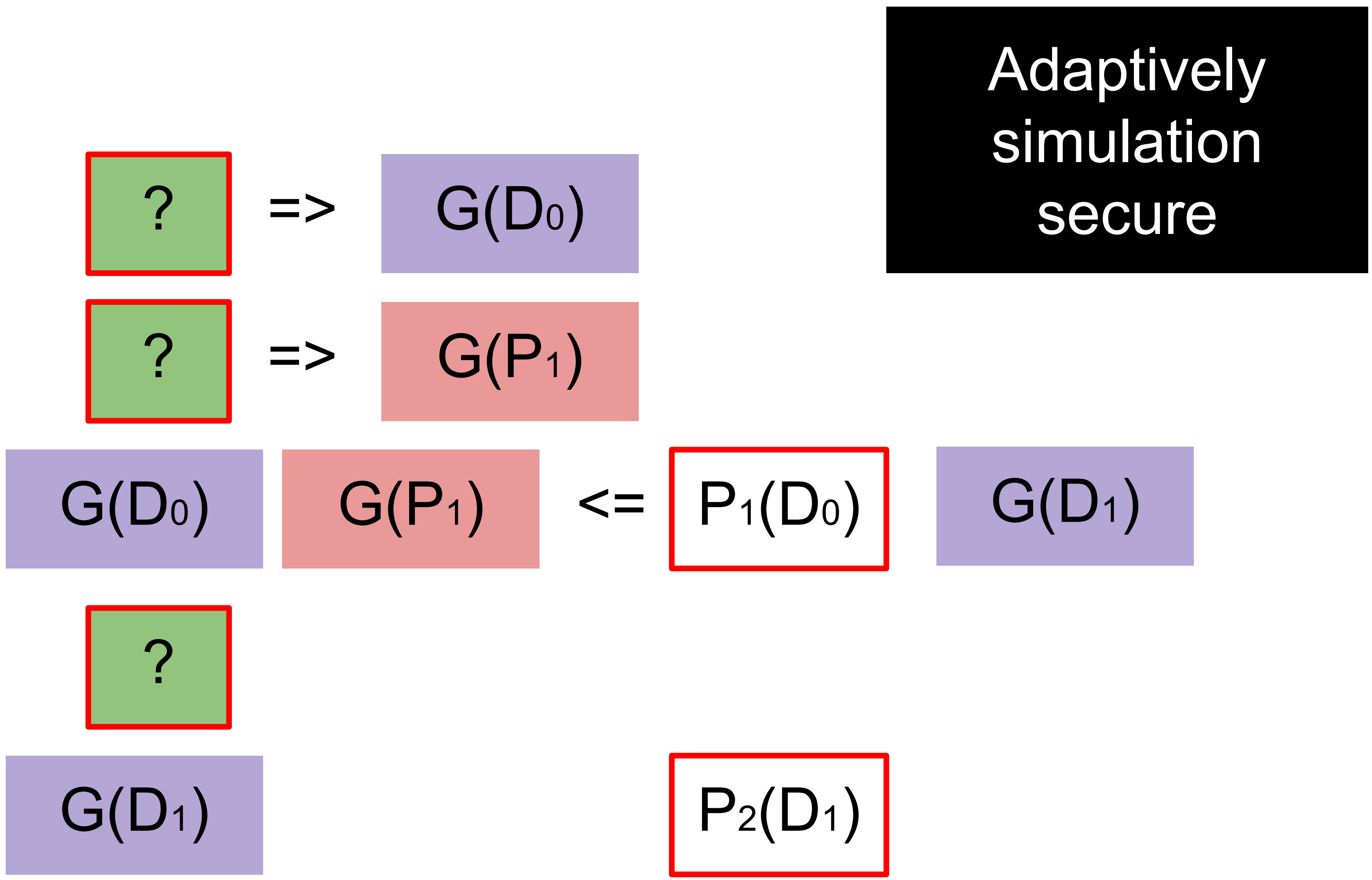
Adaptively simulation secure

? => $G(D_0)$

? => $G(P_1)$

$G(D_0)$ $G(P_1)$ <= $P_1(D_0)$ $G(D_1)$

Garbling/randomized encoding for RAM with persistent memory



Garbling/randomized encoding for RAM with persistent memory

Adaptively simulation secure

? => $G(D_0)$

? => $G(P_1)$

$G(D_0)$ $G(P_1)$ <= $P_1(D_0)$ $G(D_1)$

? => $G(P_2)$

$G(D_1)$ $G(P_2)$ <= $P_2(D_1)$ $G(D_2)$

Theorem

[Main Theorem]

Adaptively secure succinct garbled RAM with persistent memory from indistinguishability obfuscation for circuits, and poly-to-1 collision-resistant hash function.

Starring

Indistinguishability Obfuscator

Indistinguishability Obfuscator for circuits

Defined by [Barak-Goldreich-Impagliazzo-Rudich-Sahai-Vadhan-Yang ‘01]

Security:

$$iO[\, F_0 \,] \approx iO[\, F_1 \,]$$

if F_0 and F_1 have identical functionality

Candidate constructions:

[Garg-Gentry-Halevi-Raykova-Sahai-Waters ‘13], [Barak-Garg-Kalai-Paneth-Sahai ‘14], [Brakerski-Rothblum ‘14], [Pass-Seth-Telang ‘14], [Zimmerman ‘15], [Applebaum-Brakerski ‘15], [Ananth-Jain ‘15], [Bitansky-Vaikuntanathan ‘15], [Gentry-Gorbunov-Halevi ‘15], [Lin ‘16], …

Cryptanalyses:

[Cheon-Han-Lee-Ryu-Stehle ‘15], [Coron et al ‘15], [Miles-Sahai-Zhandry ‘16], ...

Poly-to-one Collision Resistant Hash function

Poly-to-one collision resistant hash functions

H is collision resistant + each image has at most poly preimages.

[Thm] Exists for constant c, assuming Factoring or Discrete-log is hard.

The rest of the talk:

1. The main idea of the construction.
2. The technical heart: adaptively-enforceable accumulator.
3. Wrap up, and the easiest ways to think of our scheme.

Starting point: Canetti-Holmgren's selective secure scheme.

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

You never know how hard it is to use iO before actually play with it.

⟦ said Justin Holmgren, June 22, 2015, sunny ⟧

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

Canetti-Holmgren scheme details:
Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

Canetti-Holmgren scheme details:
Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same.
[KLW-technique]

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique]

q can be different [encrypt the state]

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Hide access pattern. [oram]

Starting point: Canetti-Holmgren's selective secure scheme.

Garble the CPU-step circuit, encrypt and authenticate the intermediate states, memories.

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Hide access pattern. [oram]

Canetti-Holmgren (ITCS16)

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:

Koppula-Lewko-Waters (STOC15)
(iO-friendly) Iterator
(iO-friendly) Accumulator
(iO-friendly) Splittable signature

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:

Koppula-Lewko-Waters (STOC15)
(iO-friendly) Iterator
(iO-friendly) Accumulator
(iO-friendly) Splittable signature

Accumulator
iO-friendly Merkle-tree

$\mathsf{Setup}(1^\lambda, S)$ samples $\mathsf{Acc.PP} \leftarrow \mathsf{Acc.Setup}(1^\lambda, S)$ and samples a PPRF F.

$\mathsf{GbMem}(SK, s) \to \tilde{s}$ computes an accumulator ac_s corresponding to s, generates $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(0,0))$ and computes $\sigma_s \leftarrow \mathsf{Spl.Sign}(\mathsf{sk}, (\perp, \perp, \mathsf{ac}_s, \mathsf{ReadWrite}(0 \mapsto 0)))$. $\tilde{s}$ is then defined as a memory configuration which contains both $(\mathsf{ac}_s, \sigma_s)$ and store_0.

$\mathsf{GbPrg}(SK, M_i, T_i, i) \to \tilde{M}_i$ first transforms M_i so that its initial state is $\perp$. Note this can be done without loss of generality by hard-coding the "real" initial state in the transition function. GbPrg then computes $\tilde{C}_i \leftarrow i\mathcal{O}(C_i)$, where C_i is described in Algorithm 1. Finally, we define $\tilde{M}_i$ not by its transition function, but by pseudocode, as the RAM machine which:

1. Reads $(\mathsf{ac}_0, \sigma_0)$ from memory (recall these were inserted under the names $(\mathsf{ac}_s, \sigma_s)$). Define $\mathsf{op}_0 = \mathsf{ReadWrite}(0 \mapsto 0)$, $q_0 = \perp$, and $\mathsf{itr}_0 = \perp$.
2. For $i = 0, 1, 2, \ldots$:
   (a) Compute $\mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Acc.Update}(\mathsf{Acc.PP}, \mathsf{store}_i, \mathsf{op}_i)$.
   (b) Compute $\mathsf{out}_i \leftarrow \tilde{C}_i(i, q_i, \mathsf{itr}_i, \mathsf{ac}_i, \mathsf{op}_i, \sigma_i, v_i, \mathsf{ac}_{i+1}, \pi_i)$.
   (c) If out_i parses as (y, σ), then write $(\mathsf{ac}_{i+1}, \sigma)$ to memory, output y, and terminate.
   (d) Otherwise, out_i must parse as $(q_{i+1}, \mathsf{itr}_{i+1}, \mathsf{ac}_{i+1}, \mathsf{op}_{i+1}), \sigma_{i+1}$.

We note that $\tilde{M}_i$ can be compiled from $\tilde{C}_i$ and Acc.PP. This means that later, when we prove security, it will suffice to analyze a game in which the adversary receives $\tilde{C}_i$ instead of $\tilde{M}_i$. This also justifies our relatively informal description of $\tilde{M}_i$.

Input: Time t, state q, iterator itr, accumulator ac, operation op, signature σ, memory value v, new accumulator ac', proof π
Data: Puncturable PRF F, RAM machine M_i with transition function δ_i, Accumulator verification key $\mathsf{vk}_{\mathsf{Acc}}$, index i, iterator public parameters Itr.PP, time bound T_i

$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(i, t))$;
if $t > T_i$ *or* $\mathsf{Spl.Verify}(\mathsf{vk}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}), \sigma) = 0$ *or* $\mathsf{Acc.Verify}(\mathsf{vk}_{\mathsf{Acc}}, \mathsf{ac}, \mathsf{op}, \mathsf{ac}', v, \pi) = 0$ **then return** $\perp$;
$\mathsf{out} \leftarrow \delta_i(q, v)$;
if $\mathsf{out} \in Y$ **then**
  $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(i+1, 0))$;
  return $\mathsf{out}, \mathsf{Sign}(\mathsf{sk}', (\perp, \perp, \mathsf{ac}', \mathsf{ReadWrite}(0 \mapsto 0))$
else
  Parse out as (q', op');
  $\mathsf{itr}' \leftarrow \mathsf{Itr.Iterate}(\mathsf{Itr.PP}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}))$;
  $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(i, t+1))$;
  return $(q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'), \mathsf{Sign}(\mathsf{sk}', (q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'))$

Algorithm 1: Transition function for M_i, with memory verified by a signed accumulator.

What is written in eprint 2015/1074

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:

Koppula-Lewko-Waters (STOC15)
(iO-friendly) Iterator
(iO-friendly) Accumulator
(iO-friendly) Splittable signature

G(D_0)

initialize

Accumulator
iO-friendly Merkle-tree

Setup($1^\lambda, S$) samples Acc.PP $\leftarrow$ Acc.Setup($1^\lambda, S$) and samples a PPRF F.

…ccumulator ac_s corresponding to s, generates $(\mathsf{sk}, \mathsf{vk}) \leftarrow$ Spl.Setup($1^\lambda; F(0,0)$) … sk, $(\perp, \perp, \mathsf{ac}_s$, ReadWrite$(0 \mapsto 0)))$. $\tilde{s}$ is then defined as a memory config- … $(\mathsf{ac}_s, \sigma_s)$ and store_0.

…nsforms M_i so that its initial state is $\perp$. Note this can be done without … ng the "real" initial state in the transition function. GbPrg then computes … ribed in Algorithm 1. Finally, we define $\tilde{M}_i$ not by its transition function, but by pseudocode, as the RAM machine which:

1. Reads $(\mathsf{ac}_0, \sigma_0)$ from memory (recall these were inserted under the names $(\mathsf{ac}_s, \sigma_s)$). Define $\mathsf{op}_0 =$ ReadWrite$(0 \mapsto 0)$, $q_0 = \perp$, and $\mathsf{itr}_0 = \perp$.
2. For $i = 0, 1, 2, \ldots$:
   (a) Compute $\mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow$ Acc.Update(Acc.PP, $\mathsf{store}_i, \mathsf{op}_i$).
   (b) Compute $\mathsf{out}_i \leftarrow \tilde{C}_i(i, q_i, \mathsf{itr}_i, \mathsf{ac}_i, \mathsf{op}_i, \sigma_i, v_i, \mathsf{ac}_{i+1}, \pi_i)$.
   (c) If out_i parses as (y, σ), then write $(\mathsf{ac}_{i+1}, \sigma)$ to memory, output y, and terminate.
   (d) Otherwise, out_i must parse as $(q_{i+1}, \mathsf{itr}_{i+1}, \mathsf{ac}_{i+1}, \mathsf{op}_{i+1}), \sigma_{i+1}$.

We note that $\tilde{M}_i$ can be compiled from $\tilde{C}_i$ and Acc.PP. This means that later, when we prove security, it will suffice to analyze a game in which the adversary receives $\tilde{C}_i$ instead of $\tilde{M}_i$. This also justifies our relatively informal description of $\tilde{M}_i$.

Input: Time t, state q, iterator itr, accumulator ac, operation op, signature σ, memory value v, new accumulator ac$'$, proof π
Data: Puncturable PRF F, RAM machine M_i with transition function δ_i, Accumulator verification key $\mathsf{vk}_{\mathsf{Acc}}$, index i, iterator public parameters Itr.PP, time bound T_i

```
(sk, vk) ← Spl.Setup(1^λ; F(i, t));
if t > T_i or Spl.Verify(vk, (q, itr, ac, op), σ) = 0 or Acc.Verify(vk_Acc, ac, op, ac', v, π) = 0 then return ⊥;
out ← δ_i(q, v);
if out ∈ Y then
  | (sk', vk') ← Spl.Setup(1^λ; F(i + 1, 0));
  | return out, Sign(sk', (⊥, ⊥, ac', ReadWrite(0 ↦ 0))
else
  | Parse out as (q', op');
  | itr' ← Itr.Iterate(Itr.PP, (q, itr, ac, op));
  | (sk', vk') ← Spl.Setup(1^λ; F(i, t + 1));
  | return (q', itr', ac', op'), Sign(sk', (q', itr', ac', op'))
```

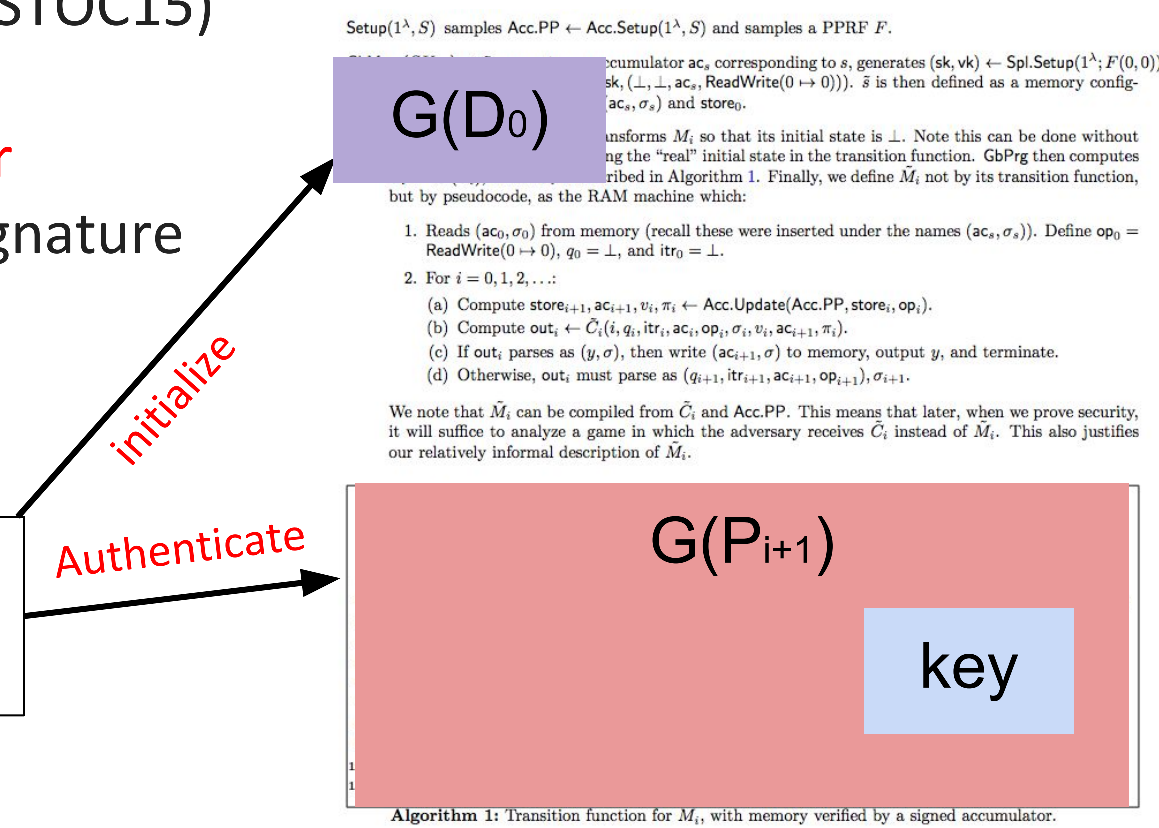
Algorithm 1: Transition function for M_i, with memory verified by a signed accumulator.

What is written in eprint 2015/1074

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:

Koppula-Lewko-Waters (STOC15)
(iO-friendly) Iterator
(iO-friendly) Accumulator
(iO-friendly) Splittable signature

Accumulator
iO-friendly Merkle-tree



Setup($1^\lambda, S$) samples Acc.PP $\leftarrow$ Acc.Setup($1^\lambda, S$) and samples a PPRF F.

…ccumulator ac_s corresponding to s, generates $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(0,0))$ … $\mathsf{sk}, (\perp, \perp, \mathsf{ac}_s, \mathsf{ReadWrite}(0 \mapsto 0)))$. $\tilde{s}$ is then defined as a memory config- … $(\mathsf{ac}_s, \sigma_s)$ and store_0.

…ansforms M_i so that its initial state is $\perp$. Note this can be done without …ng the "real" initial state in the transition function. GbPrg then computes …ribed in Algorithm 1. Finally, we define $\tilde{M}_i$ not by its transition function, but by pseudocode, as the RAM machine which:

1. Reads $(\mathsf{ac}_0, \sigma_0)$ from memory (recall these were inserted under the names $(\mathsf{ac}_s, \sigma_s)$). Define $\mathsf{op}_0 = \mathsf{ReadWrite}(0 \mapsto 0)$, $q_0 = \perp$, and $\mathsf{itr}_0 = \perp$.
2. For $i = 0, 1, 2, \ldots$:
   (a) Compute $\mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Acc.Update}(\mathsf{Acc.PP}, \mathsf{store}_i, \mathsf{op}_i)$.
   (b) Compute $\mathsf{out}_i \leftarrow \tilde{C}_i(i, q_i, \mathsf{itr}_i, \mathsf{ac}_i, \mathsf{op}_i, \sigma_i, v_i, \mathsf{ac}_{i+1}, \pi_i)$.
   (c) If out_i parses as (y, σ), then write $(\mathsf{ac}_{i+1}, \sigma)$ to memory, output y, and terminate.
   (d) Otherwise, out_i must parse as $(q_{i+1}, \mathsf{itr}_{i+1}, \mathsf{ac}_{i+1}, \mathsf{op}_{i+1}), \sigma_{i+1}$.

We note that $\tilde{M}_i$ can be compiled from $\tilde{C}_i$ and Acc.PP. This means that later, when we prove security, it will suffice to analyze a game in which the adversary receives $\tilde{C}_i$ instead of $\tilde{M}_i$. This also justifies our relatively informal description of $\tilde{M}_i$.

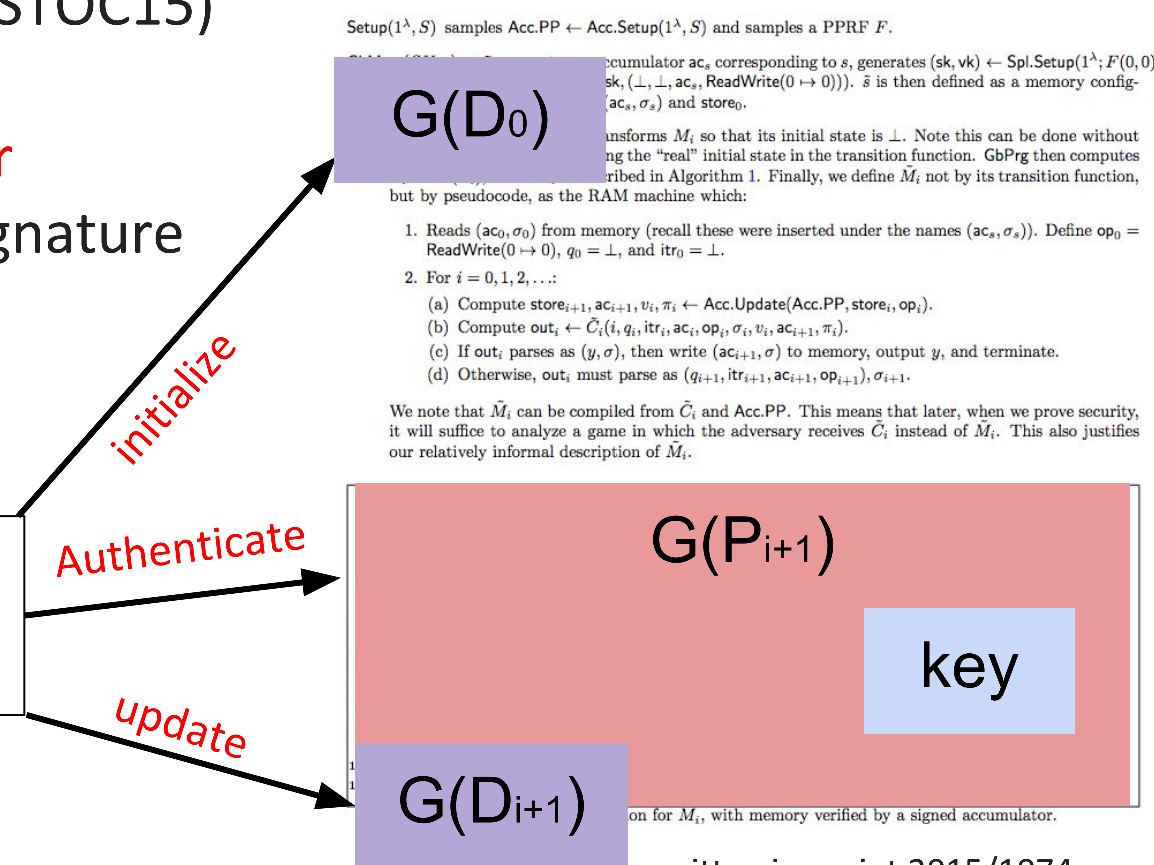
Algorithm 1: Transition function for M_i, with memory verified by a signed accumulator.

What is written in eprint 2015/1074

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:

Koppula-Lewko-Waters (STOC15)
(iO-friendly) Iterator
(iO-friendly) Accumulator
(iO-friendly) Splittable signature

Accumulator
iO-friendly Merkle-tree



What is written in eprint 2015/1074

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Properties needed for the Accumulator
- Normal property like a Merkle-tree.

#Merkletree

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

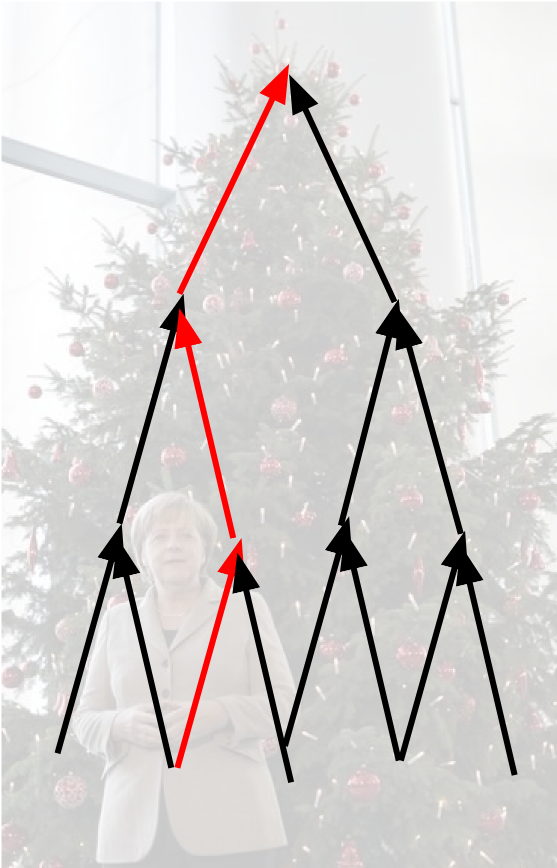
Properties needed for the Accumulator

- Normal property like a Merkle-tree.

#Merkletree

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Properties needed for the Accumulator
- Normal property like a Merkle-tree.

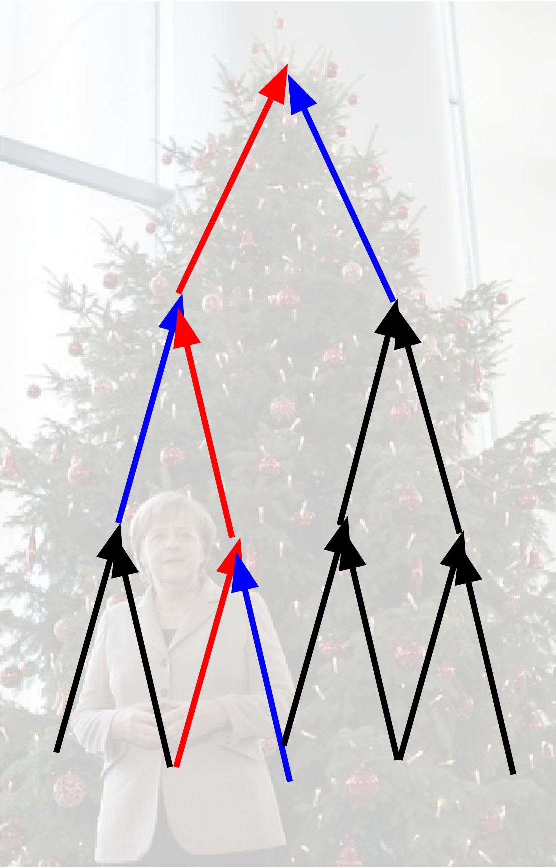


#Merkletree

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Properties needed for the Accumulator

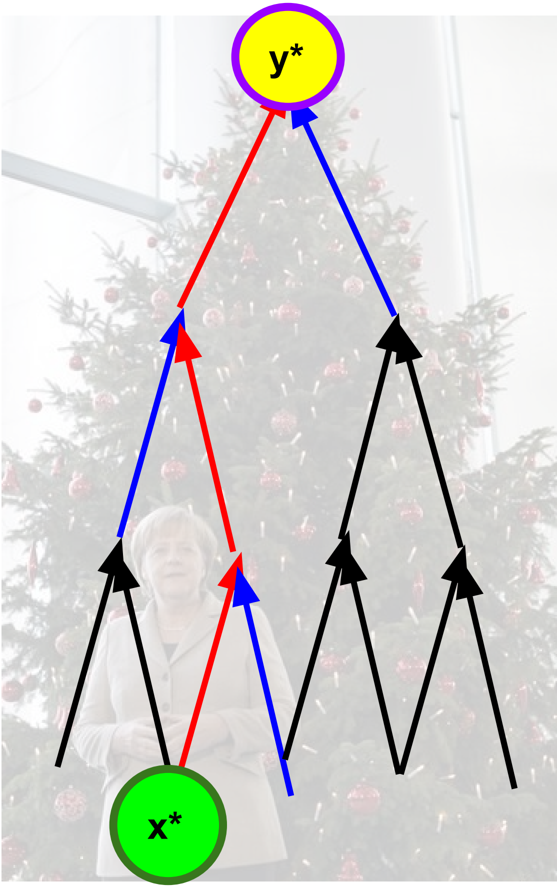
- Normal property like a Merkle-tree.



#Merkletree

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Properties needed for the Accumulator
- Normal property like a Merkle-tree.
- Enforcement (iO-friendly property): there's only one preimage x* of the current root value y*.



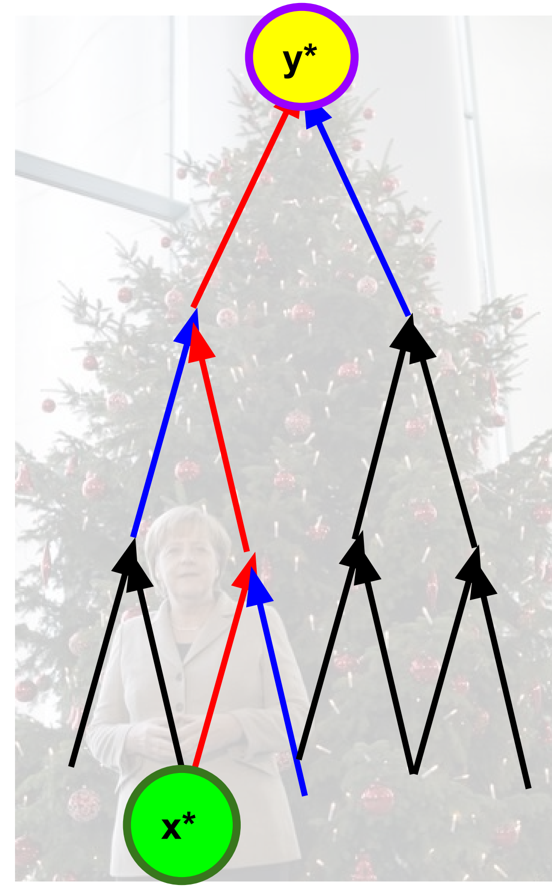
#Merkletree

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Properties needed for the Accumulator

- Normal property like a Merkle-tree.
- Enforcement (iO-friendly property): there's only one preimage x* of the current root value y*.

Impossible information theoretically.



#Merkletree

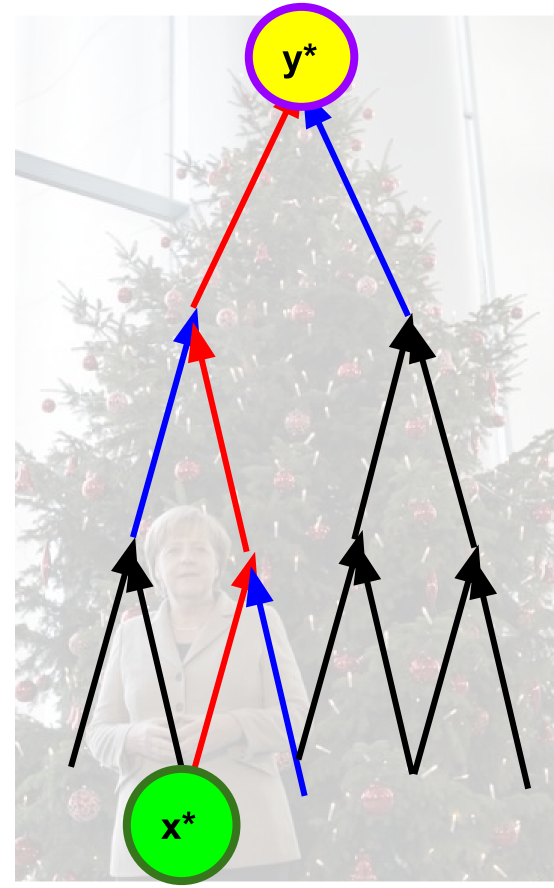
Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

Properties needed for the Accumulator

- Normal property like a Merkle-tree.
- Enforcement (iO-friendly property): there's only one preimage x* of the current root value y*.

Impossible information theoretically.

KLW's computational enforcement:
Normal.Gen(   )->H
Enforce.Gen(x*, y*)->H*,   H ≈ H*



#Merkletree

Canetti-Holmgren (ITCS16)
+ Zoom-in the core step:
++ Zoom-in the accumulator

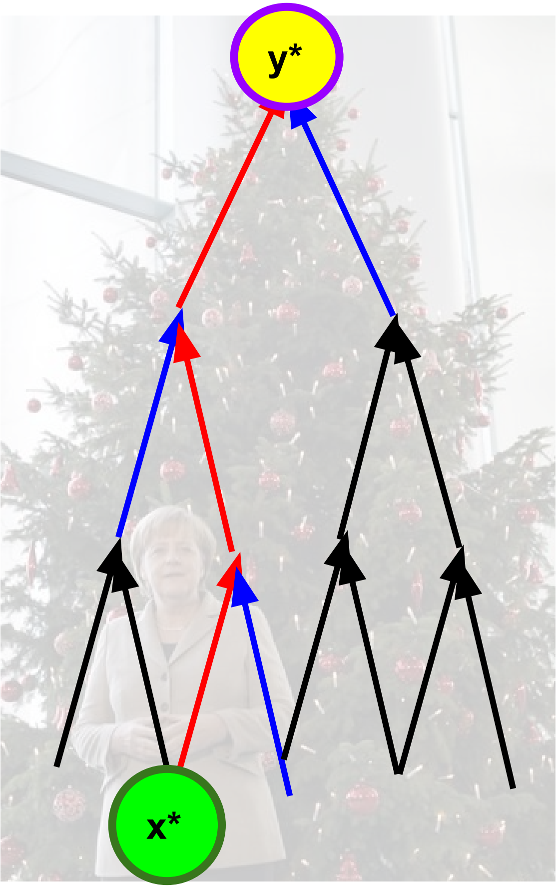
Properties needed for the Accumulator

- Normal property like a Merkle-tree.
- Enforcement (iO-friendly property): there's only one preimage x* of the current root value y*.

Impossible information theoretically.

KLW's computational enforcement:
Normal.Gen(   )->H
Enforce.Gen(x*, y*)->H*,   H ≈ H*

Alternatively: SSB hashing => [Ananth-Chen-Chung-Lin-Lin]



#Merkletree

Selective Enforcing

Adaptive Enforcing

Selective Enforcing

Adaptive Enforcing

x* <= Adversary

Selective Enforcing

Adaptive Enforcing

x* <= Adversary

Gen( ) => H

Enforcing(x*, y*) => H*

Selective Enforcing

x* <= Adversary

Gen( ) => H

Enforcing(x*, y*) => H*

Adaptive Enforcing

Gen( ) => H

Selective Enforcing

x* <= Adversary

Gen( ) => H

Enforcing(x*, y*) => H*

Adaptive Enforcing

Gen( ) => H

x* <= Adversary(H)

Selective Enforcing

x* <= Adversary

Gen( ) => H

Enforcing(x*, y*) => H*

Adaptive Enforcing

Gen( ) => H

x* <= Adversary(H)

Enforcing(x*, y*) => H*

Selective Enforcing

x* <= Adversary

Gen( ) => H

Enforcing(x*, y*) => H*

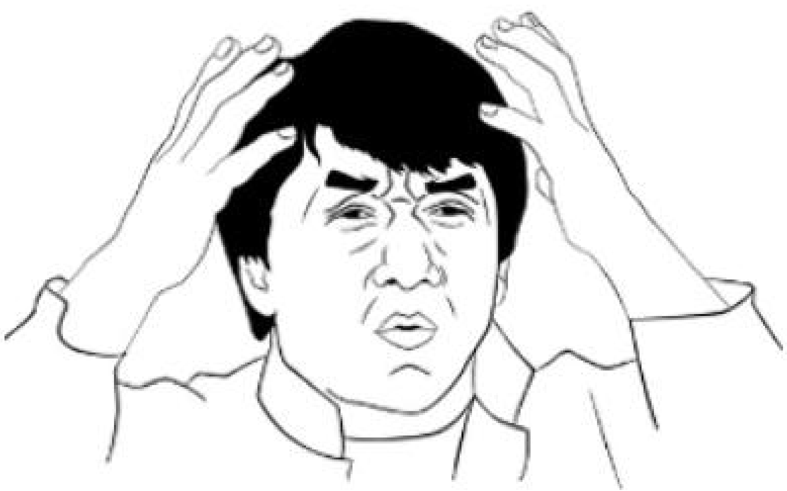
Adaptive Enforcing

Gen( ) => H

x* <= Adversary(H)

Enforcing(x*, y*) => H*

(… wait, what?)

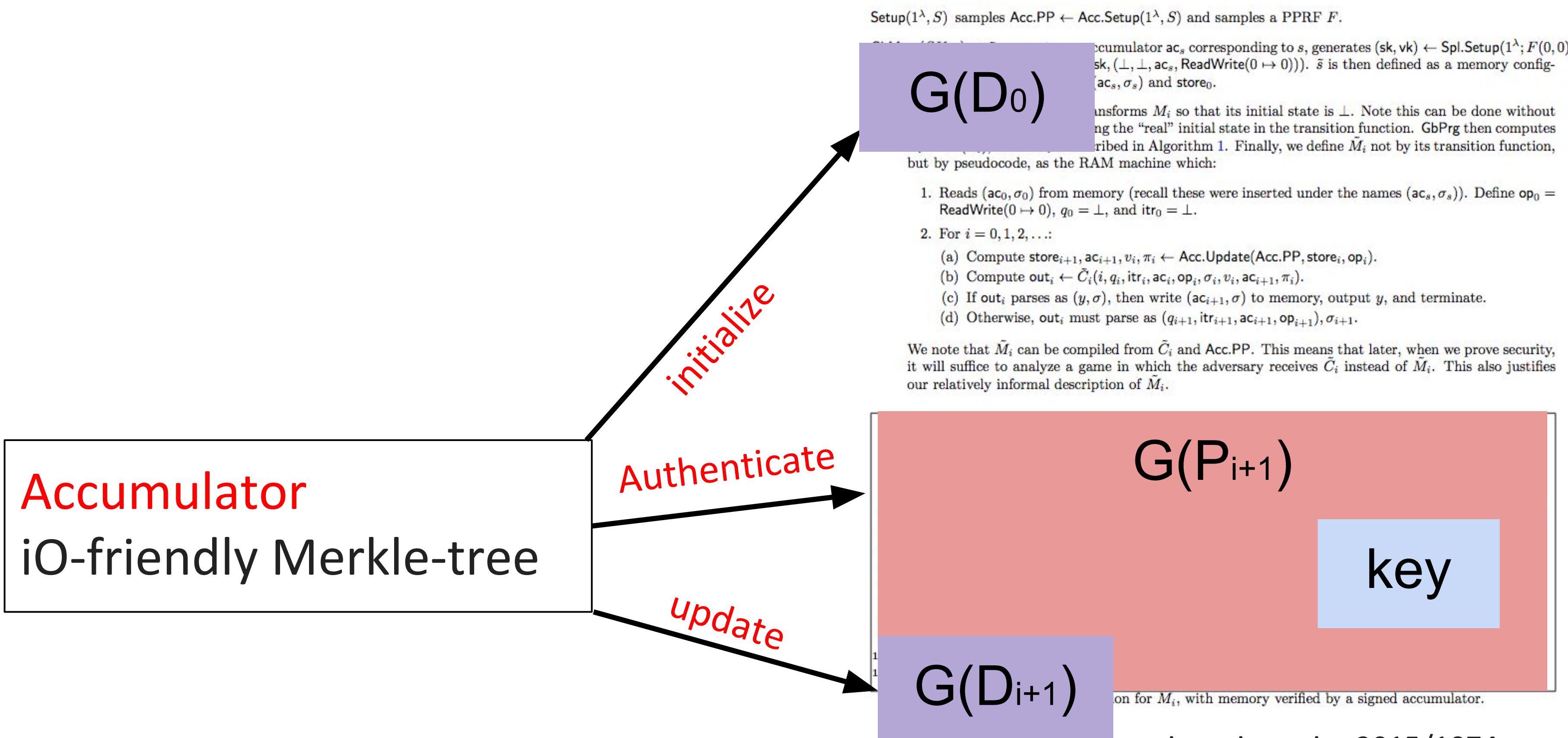


#Mindblowing

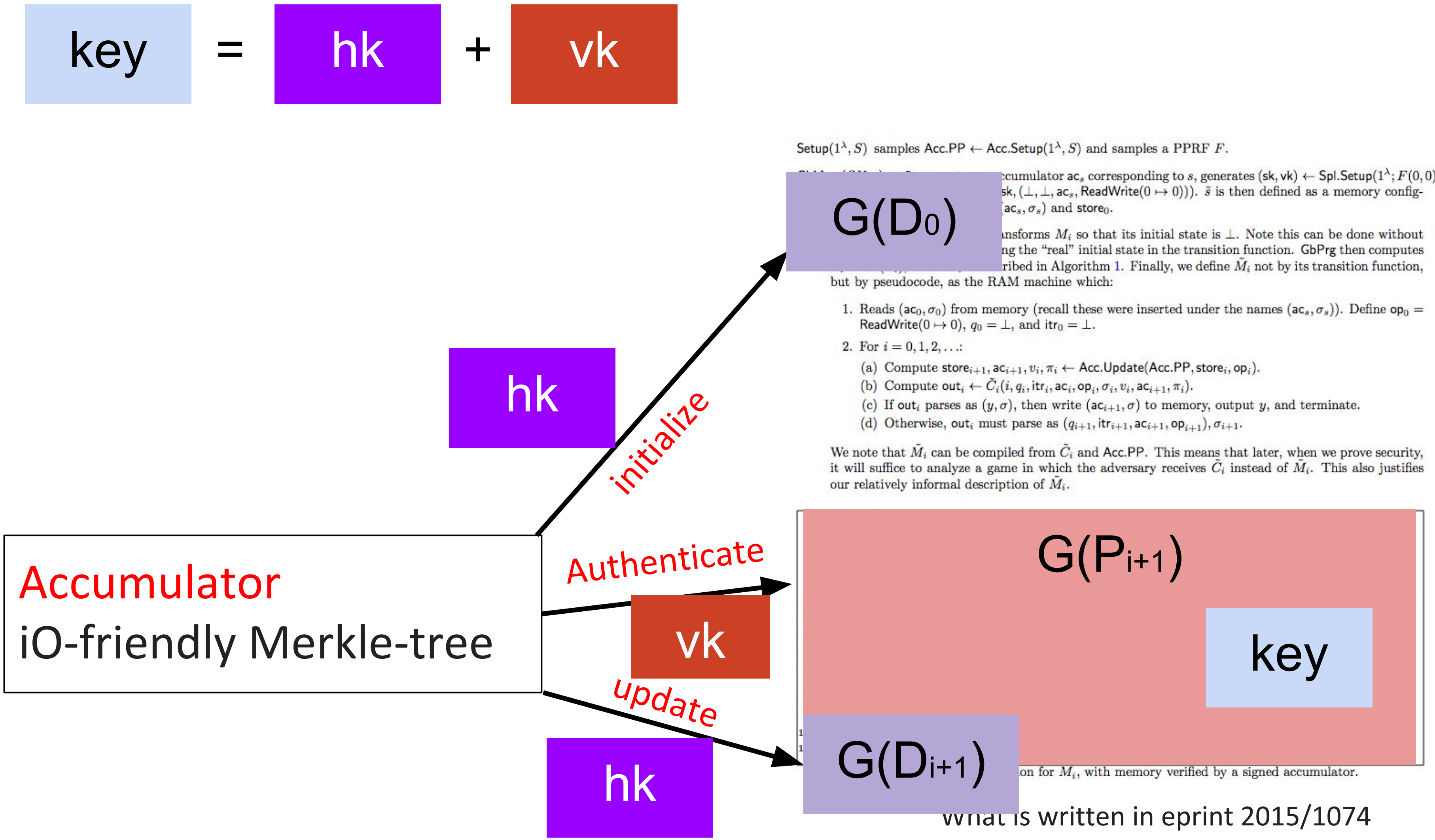
Fact I

Can separate the key

key = hk + vk



What is written in eprint 2015/1074


key
=
hk
+
vk
Setup$(1^\lambda, S)$ samples Acc.PP $\leftarrow$ Acc.Setup$(1^\lambda, S)$ and samples a PPRF F.
ccumulator ac_s corresponding to s, generates (sk, vk) $\leftarrow$ Spl.Setup$(1^\lambda; F(0,0))$
sk, $(\perp, \perp, ac_s$, ReadWrite$(0 \mapsto 0)))$. $\tilde{s}$ is then defined as a memory config-
(ac_s, σ_s) and store$_0$.
ansforms M_i so that its initial state is $\perp$. Note this can be done without
ng the "real" initial state in the transition function. GbPrg then computes
ribed in Algorithm 1. Finally, we define $\tilde{M}_i$ not by its transition function,
but by pseudocode, as the RAM machine which:
1. Reads (ac_0, σ_0) from memory (recall these were inserted under the names (ac_s, σ_s)). Define op_0 = ReadWrite$(0 \mapsto 0)$, $q_0 = \perp$, and $itr_0 = \perp$.
2. For $i = 0, 1, 2, \ldots$:
(a) Compute $store_{i+1}, ac_{i+1}, v_i, \pi_i \leftarrow$ Acc.Update(Acc.PP, $store_i, op_i$).
(b) Compute $out_i \leftarrow \tilde{C}_i(i, q_i, itr_i, ac_i, op_i, \sigma_i, v_i, ac_{i+1}, \pi_i)$.
(c) If out_i parses as (y, σ), then write (ac_{i+1}, σ) to memory, output y, and terminate.
(d) Otherwise, out_i must parse as $(q_{i+1}, itr_{i+1}, ac_{i+1}, op_{i+1}), \sigma_{i+1}$.
We note that $\tilde{M}_i$ can be compiled from $\tilde{C}_i$ and Acc.PP. This means that later, when we prove security, it will suffice to analyze a game in which the adversary receives $\tilde{C}_i$ instead of $\tilde{M}_i$. This also justifies our relatively informal description of $\tilde{M}_i$.
G(D0)
hk
initialize
Accumulator
iO-friendly Merkle-tree
Authenticate
vk
G(Pi+1)
key
update
hk
G(Di+1)
on for M_i, with memory verified by a signed accumulator.
What is written in eprint 2015/1074

Adaptive Enforcing

hk

Adaptive Enforcing

hk

x* <= Adversary(hk)

Adaptive Enforcing

hk

x* <= Adversary(hk)

vk ≈ vk*(x*)

Fact II
If you believe diO ...

key = hk + vk

Adaptive Enforcing

key = hk + vk

Adaptive Enforcing

hk

always_hk_Gen( ) -> hk := CRHF key h

key = hk + vk

Adaptive Enforcing

hk

x* <= Adversary(H)

always_hk_Gen( ) -> hk := CRHF key h

key = hk + vk

Adaptive Enforcing

hk

x* <= Adversary(H)

vk

always_hk_Gen( ) -> hk := CRHF key h

normal_vk_Gen( ) -> vk

vk(x,y) = diO(if h(x)=y, output 1; else: output 0)

key = hk + vk

Adaptive Enforcing

hk

x* <= Adversary(H)

vk ≈ vk*(x*)

always_hk_Gen( ) -> hk := CRHF key h

normal_vk_Gen( ) -> vk

vk(x,y) = diO(if h(x)=y, output 1; else: output 0)

enforce_vk_Gen(x*, y*) -> vk*

vk*(x,y) = diO(if y!=y* and h(x)=y, output 1;

<u>Elseif y=y* and x=x*, output 1</u>;

Else: output 0)

Fact III:
If you don’t believe diO,
can still do this with iO.

From iO + preimage-bounded CRHF:

c-to-1 CRHF can be constructed from discrete-log or factoring

From iO + preimage-bounded CRHF:

c-to-1 CRHF can be constructed from discrete-log or factoring

enforce_vk(x*, y*) -> vk*

vk*(x,y) = ~~d~~iO(if y!=y* and h(x)=y, output 1;
Elseif y=y* and x=x*, output 1;
Else: output 0)

From iO + preimage-bounded CRHF:

c-to-1 CRHF can be constructed from discrete-log or factoring

enforce_vk(x*, y*) -> vk*

vk*(x,y) = ~~d~~iO(if y!=y* and h(x)=y, output 1;
Elseif y=y* and x=x*, output 1;
Else: output 0)

By diO-iO equivalence lemma [Boyle-Chung-Pass '14]:
" If f1 and f2 differ only on polynomially many input-output values, and they are hard to find, then

$$iO(f1) \approx iO(f2)$$ "

From iO + preimage-bounded CRHF:

c-to-1 CRHF can be constructed from discrete-log or factoring

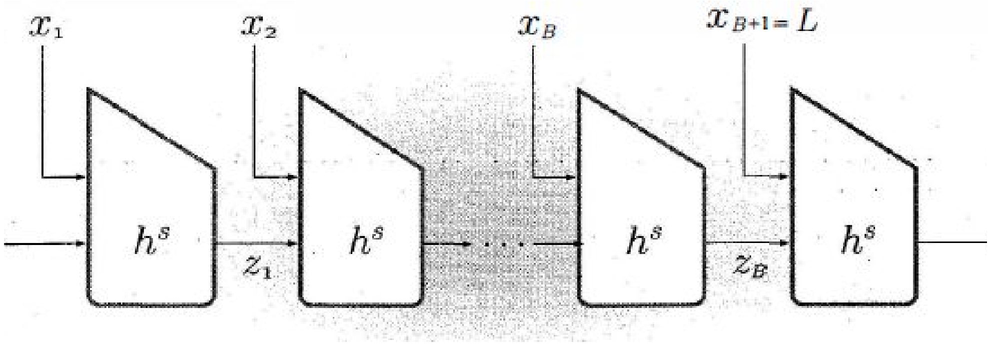
enforce_vk(x*, y*) -> vk*

vk*(x,y) = ~~d~~iO(if y!=y* and h(x)=y, output 1;

Elseif y=y* and x=x*, output 1;

Else: output 0)

From shrinking 1 bit to length-halving: Merkle-Damgaard.



Fact IV:
Adaptive Enforceable Accumulator done

Rest of the upgrades:

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique. Assume iO]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Hide access pattern. [oram]

Rest of the upgrades:

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique. Assume iO]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Hide access pattern. [oram]

+ adaptively enforceable accumulator
[from iO+dlog or factoring]

Rest of the upgrades:

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique. Assume iO]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Hide access pattern. [oram]

+ adaptively enforceable accumulator
[from iO+dlog or factoring]

Need a special property of the ORAM
"Strong local randomness", satisfied by Chung-Pass ORAM.
With this property, can "guess" polynomially many addresses.

Rest of the upgrades:

Canetti-Holmgren scheme details:

Fixed-transcript => Fixed-access => Fixed-address => Fully secure

Indistinguishable as long as transc = (q, op) are the same. [KLW-technique. Assume iO]

q can be different [encrypt the state]

Memory content can be different [encrypt the data]

Hide access pattern. [oram]

+ adaptively enforceable accumulator
[from iO+dlog or factoring]

SSB hash
[Hubacek-Wichs]
[OPWW]

Need a special property of the ORAM
"Strong local randomness", satisfied by Chung-Pass ORAM.
With this property, can "guess" polynomially many addresses.

[Ananth-Chen-Chung-Lin-Lin, eprint 2015/1082] can be viewed as accomplishing this for all the steps.

Summary

1. Adaptively secure garbled RAM with persistent memory.
2. Everything is succinct.
3. Upgrading to delegation with verifiability is almost for free.
4. “Reusability” is natural.
5. New iO-friendly tool: adaptively-enforceable accumulator (from iO+Preimage-bounded-CRHF)

Scenes

The Directors Cut

> sudo apt-get install GRAM_Canetti_Holmgren

> sudo apt-get install GRAM_Canetti_Holmgren

package indistinguishability_obfuscation not an accepted assumption, security at user's own risk, continue (y) or not (n)

> sudo apt-get install GRAM_Canetti_Holmgren
package indistinguishability_obfuscation not an accepted assumption, security at user's own risk, continue (y) or not (n)
y

> sudo apt-get install GRAM_Canetti_Holmgren
package indistinguishability_obfuscation not an accepted assumption, security at user’s own risk, continue (y) or not (n)
y
> upgrade GRAM_CCHR
Done

> sudo apt-get install GRAM_Canetti_Holmgren

package indistinguishability_obfuscation not an accepted assumption, security at user’s own risk, continue (y) or not (n)

y

> upgrade GRAM_CCHR

Done

> NSAcloud: GRAM_CCHR_Factoring RSA2048

> sudo apt-get install GRAM_Canetti_Holmgren
package indistinguishability_obfuscation not an accepted assumption, security at user's own risk, continue (y) or not (n)
y
> upgrade GRAM_CCHR
Done
> NSAcloud: GRAM_CCHR_Factoring RSA2048
Running time 1.0s
25195908475…20720357
= 83990...4079279 x 3091701...723883

Next question