Succinct Persistent Adaptive Garbled RAM
or
How To Delegate Your Database

Ran Canetti
TAU and BU

Based on joint works with
Justin Holmgren, Yilei Chen, Mariana Raykova
ePrint reports 2015/388 and 2015/1074
Delegating Computation

“query” = program + input

Verifiability + Privacy + Efficiency

Efficiency
- Bandwidth
- Server
- Client
Delegating Computation

“Old-fashioned” Setting: Small input + Big Computations

• Verifiable Computation Protocols [Blum-Kannan89, Blum-Luby-Rubinfeld90, Kilian92, Micali00, Ergun-Kumar-Rubinfeld99, Goldwasser-Kalai-Rothblum08, Gennaro-Gentry-Parno10...]

• Fully Homomorphic Encryption [Gentry09 ... ...

→ Client work + Bandwidth proportional to input size

Today: Big Data + Small Computations
Delegating Computation

### Database

<table>
<thead>
<tr>
<th>ID</th>
<th>age</th>
<th>M/F</th>
<th>salary</th>
</tr>
</thead>
</table>

- Database

- Server
- "query" = circuit/program
- Client

- update
- answer

- Verifiability + Privacy + Efficiency

- Bandwidth
- Server
- Client
Requirement 1: Verifiability

Is $a = C(\text{current-DB})$ ?

- **DB**
- **Server**
- **User**

"pay-per-bit" channel
Requirement 1: Verifiability

Is \( a = C(\text{current-DB}) \)?
Requirement 2: Privacy

Learn nothing!

Server

DB

C, answer a

C

"pay-per-bit" channel

User
Requirement 3: Query delegation

DB

\( a = C(DB) \)

Server

learns only
\( c(DB) \)

\( C \)

\( C \)
Putting it all together:
Remote Database ideal functionality

- Obtain DB from owner, reveal size to adv
- Receive (Query, Recipient) from owner:
  - Run Query(DB)
    (potentially updating DB, disclose size & runtime to adv)
- Output answer to Recipient, disclose size to adv
- If Recipient corrupted, Adversary learns (only!) the answer
Requirement 4: efficiency & size

Want:

• Size of query & answer proportional to that of “plaintext query and answer”
• All clients are efficient in size of answer
• Database size is comparable to plaintext
• Server runtime proportional to original
A scheme that UC-realizes the above functionality and has the above efficiency requirements is called a secure database delegation scheme.
Existing solutions

Verifiability:

- Memory delegation [Chung-Kalai-Vadhan]
- SNARKS & Proof Carrying Data [Chiesa-Tromer, Bitansky-C-Chiesa-Tromer,...]
- Accumulators & set computations [Tamassia, Triandopoulos, Papadopoulos,...]
- General RAM computations with persistent memory
  [Kalai-Paneth,Brakerski-Holmgren-Kalai]

But: no privacy...
Existing solutions

Verifiability:
• Memory delegation [Chung-Kalai-Vadhan]
• SNARKS & Proof Carrying Data [Chiesa-Tromer, Bitansky-C-Chiesa-Tromer,...]
• Accumulators & set computations [Tamassia, Triandopoulos, Papadopoulos,...]
• General RAM computations with persistent memory [Kalai-Paneth, Brakerski-Holmgren-Kalai]

But: no privacy...

Privacy:
• Homomorphic encryption... but requires $\Omega(DB)$ work!
• Searchable encryption (order preserving, token based, CryptDB,...)

But: no verifiability...
Main result:
Assuming circuit IO and const-to-1 CRHF's, there exist a secure database delegation scheme.
Main result:
Assuming circuit IO and const-to-1 CRHF\textsc{s}, there exist a secure database delegation scheme.

Proof:

Use Succinct Persistent Adaptive Garbled RAM...
Main result:
Assuming circuit IO and const-to-1 CRHFs, there exist a secure database delegation scheme.

Proof:
Use Succinct Persistent Adaptive Garbled RAM...

[concurrently by Ananth-Chen-Chung-Lin-Lin]
Garbling / Randomized Encoding
[Yao, Ishai-Kushilevitz, Bellare-Hoang-Rogaway]

- Algorithm \textit{Garble}

- \(\tilde{f}, \tilde{x} \leftarrow \text{Garble}(f, x)\):
  - Correctness: \(f(x) = \tilde{f}(\tilde{x})\)
  - Security: If \(f(x) = f'(x')\), then
    \[\text{Garble}(f, x) \approx \text{Garble}(f', x')\]
  - Efficiency: Computing \(\tilde{f}(\tilde{x})\) is as easy as computing \(f(x)\)
  - Succinctness: sizes of \(\tilde{f}, \tilde{x}\) are proportional to the size of \(f, x\)
Garbling / Randomized Encoding
[Yao, Ishai-Kushilevitz, Bellare-Hoang-Rogaway]

- Algorithm \textit{Garble} \((K\text{gen}, \text{Fgarble}, \text{Igarble})\)

\[
\tilde{f}, \tilde{x} \leftarrow \text{Garble}(f, x): \quad k \leftarrow K\text{gen}(), \quad \tilde{f} \leftarrow \text{Fgarble}(k, f), \quad \tilde{x} \leftarrow \text{Igarble}(k, x)
\]

- Correctness: \(f(x) = \tilde{f}(\tilde{x})\)

- Security: If \(f(x) = f'(x)\), then
  \[
  \text{Garble}(f, x) \approx \text{Garble}(f', x')
  \]

- Efficiency: Computing \(\tilde{f}(\tilde{x})\) is as easy as \(f(x)\)

- Succinctness: sizes of \(\tilde{f}, \tilde{x}\) are prop. to \(x, f(x)\)

- Adaptivity: \(\text{Adv}\) can choose \(f\) as a function of \(\tilde{x}, \) and \(x\) as a function of \(\tilde{f}\).
Brief History (partial)

• [Yao]: circuit garbling. No succinctness
• ...
• [Goldwasser-Kalai-Poppa-Vinod-Zeldovich]: TM garbling. Size Proportional to input size
• [Lu-Ostrovsky, Gentry-Halevi-Raykova-Wichs,...]: RAM machine garbling. Size proportional to runtime.
• [Bellare-Hoang-Rogaway]: adaptive circuit garbling, in ROM
• [Koppula-Lewko-Waters]: TM garbling, fully succinct.
• [C-Holmgren,Chen-Chow-Chung-Lai-Lin]: Fully succinct RAM garbling.
Garbling with persistent memory
[Gentry-Halevi-Raykova-Wichs]

- Algorithm $Garble = (Kgen,Fgarble,Igarble)$
- $k \leftarrow Kgen()$, $\tilde{x} \leftarrow Igarble(k, x)$ $\tilde{f}_i \leftarrow Fgarble(k, f_i)$, $i=1,2,...$
- Correctness: $f_i(x_i) = \tilde{f}_i(\tilde{x}_i)$ for all $i$
- Security: If $f_i(x_i) = f_i'(x_i')$, for all $i$, then $\tilde{x}, \tilde{f}_1, ... \tilde{f}_i \approx \tilde{x}', \tilde{f}'_1, ... \tilde{f}'_i$
- Efficiency: Computing $\tilde{f}(\tilde{x})$ is as easy as $f(x)$
- Succinctness: sizes of $\tilde{f}_i, \tilde{x}_i$ prop. to size of $x_i, f_i(x_i)$
- Adaptivity: Adv can choose $f_i$ after seeing $\tilde{x}, \tilde{f}_1, ... \tilde{f}_{i-1}$
From Succinct Persistent Adaptive Garbled RAM (SPAGRAM) to database delegation

• To delegate database x: Garble x, send to server. Choose keys (sig, ver) for a signature scheme. Post ver.

• To query program C, garble the program: “Output C(x), sign using key sig.” Send to server (or to third party)
From
Succinct Persistent Adaptive Garbled RAM (SPAGRAM) to database delegation

• To delegate database x: Garble x, send to server.
  Choose keys (sig, ver) for a signature scheme. Post ver.

• To query program C, garble the program:
  “Output C(x), sign using key sig.”
  Send to server (or to third party)

Note: Adaptivity is key!
RAM Garbling with persistent memory: constructions

[GHRW]: Efficient, non-succinct, non-adaptive, assuming “special purpose public-coins DIO”.

[C-Holmgren, Chung etal]: Succinct, non-adaptive, from IO+OWFs

[CCHR, ACCLL]: Adaptive (from IO+const-2-1 CRHF$	ext{s}$/DDH)
Indistinguishability Obfuscation (IO)
[Barak-Goldreich-Impagliazzo-Sahai-Rudich-Vadhan-Yang 01, Goldwasser-Rothblum 07]

Several candidate constructions
[Garg-Gentry-Halevi-Raykova-Sahai-Waters 13... ... ... Lin 16]
The age of IO

• Amazing concept:
  • Extremely powerful, versatile
  • A whole set of new techniques
  • Elusive... “too good to be true”

• Does it exist? Under what assumptions?
• Can we show impossibility?
• Can we make it more efficient / realistic?
• How to use it?
• Relaxed/stronger notions?
Towards making IO more realistic
(Towards impossibility of IO?)

We Have

Circuit Obfuscation

Real World

Can we obfuscate more realistic computations?
Trivial “Solution”

```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4    mid = [(low + high)/2]
5    if x ≤ T[mid]
6      high = mid
7    else low = mid + 1
8  return high
```

$\log n$  $n \log n$
What We’d Like

• Indistinguishability Obfuscation for a RAM program $M$ directly

• $iO(M)$ should itself a RAM program, with almost the same complexity parameters as $M$.

• If $M(x) = M'(x)$ for all inputs $x$, then $iO(M) \approx iO(M')$
Progress So Far

• Turing Machine & RAM obfuscation from non-standard “knowledge assumptions” (DIO and variants) [BCP14, ABGSZ14, GHRW14, IPS14]

• “semi-succinct” TM & RAM obfuscation from subexp-IO and IOWFs: size depends on space of computation. [Bitansky-Garg-Lin-Pass-Telang, C-Holmgren-Jain-Vinod]

• Fully succinct Turing Machine obfuscation from subexp IO and IOWFs [Koppula-Lewko-Waters 14]

• Fully succinct RAM obfuscation from subexp IO and IOWFs [C-H, Chung etal]

• Extension to PRAM [Chung etal]

⇒ All recent works obtain succinct garbling as a first step.
Our Techniques
A Naïve Attempt at RAM garbling

Memory

CPU

Address 93 please

\(x\) \(\rightarrow x'\)

\(x\) \(\rightarrow x'\)

\(x'\) \(\rightarrow x'\)
A Naïve Attempt at RAM garbling

Answer: 42
Naïve Attempt at RAM garbling
What’s wrong? Everything

• Doesn’t prevent adversary from giving circuit illegal inputs
• Doesn’t hide any intermediate state
• Doesn’t hide memory addresses accessed

We’ll address these challenges one by one.
Goal: Succinct Garbling
2-step approach

1. Construct a weaker notion of garbling
2. Compile a weak garbler into a full garbler
Roadmap: How to compile a stronger garbler

Weaken conditions for indistinguishability:
What needs to be the same?

<table>
<thead>
<tr>
<th></th>
<th>Final Output</th>
<th>Addresses</th>
<th>Memory Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same-Trace</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Same-Address</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Full</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

What’s missing?
• Internal RAM state
• Circuit behavior on illegal inputs
Same-Trace Garbling

$Tr(M, x) \overset{\text{def}}{=} \begin{array}{|c|c|c|c|}
\hline
\text{Time} & \text{Address} & \text{Value Written} & \text{Answer} \\
\hline
1 & a_1 & s_1 & \bot \\
\vdots & \vdots & \vdots & \vdots \\
T - 1 & a_{T-1} & s_{T-1} & \bot \\
T & \bot & \bot & y \\
\hline
\end{array}$

**Theorem:** There is an algorithm STGarble such that:

If $Tr(M, x) = Tr(M', x')$, then

$$\text{STGarble}(M, x) \approx \text{STGarble}(M', x')$$
Same-Trace Garbler Construction

• Obfuscate CPU; to ensure integrity of computation use:
  • signature schemes
  • positional accumulators
  • iterators.

(Essentially follows [KLGW14]'s “Message-hiding encoding”)

Same-Address Garbling

**Goal:** If \((M, x)\) and \((M', x')\) access same addresses, then

\[
\text{SAGarble}(M, x) \approx \text{SAGarble}(M', x')
\]

**Simple Case:** Addresses are locally computable.

**Strategy:** Encrypt memory words and apply Same-Trace Garbler
## Same-Address Garbling (General Case)

- What if addresses *not* locally computable?

<table>
<thead>
<tr>
<th>Time</th>
<th>Address</th>
<th>Value Written</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_1$</td>
<td>$c_1$</td>
<td>⊥</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>$T - j - 1$</td>
<td>$a_{T-j-1}$</td>
<td>$c_{T-j-1}$</td>
<td>⊥</td>
</tr>
<tr>
<td>$T - j$</td>
<td>$a_{T-j}$</td>
<td>$z_{T-j}$</td>
<td>⊥</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>$T - 1$</td>
<td>$a_{T-1}$</td>
<td>$z_{T-1}$</td>
<td>⊥</td>
</tr>
<tr>
<td>$T$</td>
<td>⊥</td>
<td>⊥</td>
<td>$y$</td>
</tr>
</tbody>
</table>

How to access $a_{T-j}, \ldots, a_{T-1}$?
Same-Address Garbling (General Case)

• What if addresses not locally computable?
• Solution: double-execution

<table>
<thead>
<tr>
<th>Time</th>
<th>Address</th>
<th>Value Written</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_1$</td>
<td>$c_1</td>
<td></td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>$T - 1$</td>
<td>$a_{T-1}$</td>
<td>$c_{T-1}</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>⊥</td>
<td>⊥</td>
<td>$y$</td>
</tr>
</tbody>
</table>

$c_i = (i, F(i || a_i) \oplus s_i)$

$d_i = (i, G(i || a_i) \oplus s_i)$

$F$ and $G$ are puncturable PRFs
(Full) Garbling

RAM machines \( M, M' \); Inputs \( x, x' \)

**Want:** If \( M(x) = M'(x') \), then

\[
\text{Garble}(M, x) \approx \text{Garble}(M', x')
\]

**Difficulty:** Hiding memory addresses accessed

**Tools:**
- Oblivious RAM with “Randomness Locality”
- Same Address Garbler (\( SAGarble \))
Oblivious RAM

- Transform RAM machine to have a (distributionally) fixed memory access pattern

\[
\text{Addresses } a_1, \ldots, a_t \quad \approx \quad \text{Addresses } a'_1, \ldots, a'_t
\]
Localized Randomness ORAM

- The vectors of accessed addresses depend (as a function) on small, disjoint subsets of the random bits.

Randomness:

Mem. Accesses: $a_1, a_2, \ldots, a_t$

\[\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_t\]
Localized Randomness ORAM

• The vectors of accessed addresses depend (as a function) on small, disjoint subsets of the random bits

• Each $\tilde{A}_i$ can be efficiently sampled as $OSample(i)$
Localized Randomness ORAM

- The vectors of accessed addresses depend on small, disjoint subsets of the random bits
- Each $\tilde{A}_i$ can be efficiently sampled as $OSample(i)$

Satisfied by Chung-Pass ORAM

Randomness:

Mem. Accesses $a_1, a_2, \ldots, a_t$ → $\tilde{A}_1, \tilde{A}_2, \ldots, \tilde{A}_t$
Full Garbling Construction

\[ \text{Garble}(M, x) \triangleq \]

**RAM Machine**

1. Read initial ORAM state \( q_{\text{ORAM}} \) from memory
2. Run \( M \) obliviously with randomness \( F(1), \ldots, F(T) \)
3. Output \( M \)'s answer

**Memory**

| \( q_{\text{ORAM}} \) | ORAM encoding \( x \) |
Persistent Memory

• Same construction, except:
• In initial memory garbling, add "step 0"
• Augment the i-th machine to look for "step i-1" in memory, and overwrite with "step i".
  (all machines use the same parameters for signature, accumulator, iterator, encryption, oram)
• Simulation strategy the same.
Adaptivity

First issue:
Positional accumulator is a static object:
  Guarantees unconditional binding at a single point.
  But point needs to be set ahead of time...
Recall: Positional accumulator
[Hubacek-Wichs, KLV, Okamoto-Pietrzak-Waters-Wichs]

- Geygen -> pk
- Accumulate \((pk, S, i, x) \rightarrow S'\)
- Verify \((pk, S, i, x) \rightarrow yes | no\)
- Fgen \((i, x) = pk_{i,x}\)

Properties:
- Computational binding
- Forced binding
- Indistinguishability of forced keys: \(pk \sim pk_{i,x}\)

➤ Forced locations need to be fixed in advance
Solutions

• First attempt: Reduction guesses location

Doesn’t work... Pos. Acc. not strong enough [doesn’t guarantee consistency with writes]

[ACCLL]: Fix the notion and guess...
Adaptive Positional accumulator

- Geygen -> \( ak, vk \)
- Accumulate \((ak, S, i, x) \rightarrow S'\)
- Verify \((vk, S, i, x) \rightarrow yes | no\)
- Fgen \((ak, i, x) = vk_{i,x}\)

Properties:
- Computational binding
- Forced binding
- Indistinguishability of forced keys: \( vk \sim vk_{i,x} \)

⇒ Forced locations can be chosen adaptively...
Adaptive Positional accumulator

Construction:

• Define “AP-hash”: same properties as “APA” but for hash function Use IO
• From AP-hash to APA: Use Merkle paradigm
• Construct AP-hash:
  \( \text{vk}: \text{IO}[“Check that the input x is consistent with hash value y”} \]
  \( \text{fvk}_{i,x,y}: \text{IO}[“if input is i’,x’,y and either i <> i’ or x <> x’ then reject, else run normal check”} \]
Adaptivity: ORAM

Second issue:

• ORAM + PPRF is a static object:
  Guarantees unconditional secrecy for a single location.
  But location needs to be set ahead of time...

• Solution: Reduction guesses location...
Questions:

IO with persistent memory?

IO with unbounded input?

Succinct garbling without IO?