Succinct non-interactive zero knowledge for a von Neumann architecture

Eli Ben-Sasson
Alessandro Chiesa
Eran Tromer
Madars Virza
I’m a manager at a hospital

I’m a researcher.
I’m a manager at a hospital

What’s $A(x, DB)$?
Privacy.
Privacy.

Verifiability.

\[ A(x, DB) = z \]
Privacy.

There is a $\pi$ s.t. $A(x, DB) = z$.

Verifiability.

$A(x, DB) = z$
Privacy. Verifiability.

There is a \( h = H(DB) \) s.t. \( A(x, DB) = z \).
There is a $w'$ s.t. $A(x, w') = z$.
SNARKs require setup

Once: initial trusted step

Many times

Generator

CRS pk vk

Prover

Verifier

A, x

W

z, π
SNARKs require setup

Once: initial trusted step

Many times

G

P

A, x

V

w

z, π

G, P, V

CRS

pk, vk
Prior work on SNARKs

(i) Theoretical constructions

[Kilian 92] [Micali 94] [Di Crescenzo Lipmaa 08] [Groth 10]
[Goldwasser Lin Rubinstein 11] [Lipmaa 12] [Bitansky Chiesa 12]
[Bitansky Canetti Chiesa Tromer 12] [Damgård Faust Hazay 12]
[Gennaro Gentry Parno Raykova 13] [Bitansky Chiesa Ishai Ostrovsky Paneth 13]

(ii) Working prototypes

[Parno Gentry Howell Raykova 13]
[Ben-Sasson Chiesa Genkin Tromer Virza 13]
[Braun Feldman Ren Setty Blumberg Walfish 13]

SNARKs are **practically feasible** for certain applications!

Application example: **Zerocash** relies on a SNARK...
... for a tailored program that only has to check SHA256 computations and integer arithmetic.

[Ben-Sasson Chiesa Garman Green Miers Tromer Virza 2014]
Program-specific SNARKs

**Inconvenient:** must repeat trusted setup for each $A$

Currently: all prior implementations are program-specific

Ideally: one-time trusted setup that works for any $A$
Goal: SNARKs with universal trusted setup

Naive approach:

Fix $A = \text{interpreter for Python!}$
Goal: SNARKs with universal trusted setup

Naive approach:
**Goal:** SNARKs with universal trusted setup

**Our results:** Achieve the above goal. *

Performance for $T$-step programs:

<table>
<thead>
<tr>
<th>Time(Prover)</th>
<th>Space(Prover)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50T$ ms</td>
<td>$3T$ MB</td>
</tr>
</tbody>
</table>

We do so via two independent contributions:

1. Universal circuit generator

2. SNARK implementation for circuits

Both are significantly more efficient than prior work.

(* whenever the computation is suitably bounded)
The high-level picture

Best theoretical constructions achieve SNARKs for circuits.

Contribution 1:

So we build a highly optimized SNARK for circuits ...

... and create small circuit that can verify any program.

Contribution 2:
Contribution 1:
Universal circuit generator
program size bound $L$
input size bound $N$
time bound $T$

\[ b = 1 \text{ iff } A(x, w) = z \]

\* = “provided that: $|A| \leq L$, $|x| \leq N$, Time($A$) $\leq T$”

- **Universal.**
- **Efficient.**
  Asymptotically: $|C_{L,N,T}| = O((L + N + T) \log(L + N + T))$
  Concretely: $|C_{L,N,T}| / T \approx 2000$ for reasonable bounds
- **Expressive.**
  32-bit RISC von Neumann machine
  (can even support self-modifying code and JIT!)
Our circuit generator
(based on [BCGTV13])

Circuit checks trace of a RAM computation, as follows:

① Check ALU instructions in the straightforward way

Table:

<table>
<thead>
<tr>
<th>t</th>
<th>registers</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

[BCGTV13]: use multiplexer

Cost: $O(L \cdot T)$!
Our circuit generator

(based on [BCGTV13])

Circuit checks trace of a RAM computation, as follows:

① Check ALU instructions in the straightforward way
② Perform instruction fetch using non-deterministic routing
③ Check read/write instructions using same routing network

<table>
<thead>
<tr>
<th>t</th>
<th>registers</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Cost: $2T \log(T + L)$
Comparison with [BCGTV13], L = 1000

[BCGTV13] vs This work: Comparison of computational costs over time (T).

- **Instruction fetch checks** (blue)
- **Routing checks** (green)
- **Memory checks** (red)
- **Instruction execution checks** (yellow)

**Improvement** shows the relative decrease in cost over time.
Comparison with [BCGTV13], $L = 5000$

[BCGTV13]

This work

Improvement

Instruction fetch checks
Routing checks
Memory checks
Instruction execution checks
Comparison with [BCGTV13], $L = 10000$

- **Instruction fetch checks**
- **Routing checks**
- **Memory checks**
- **Instruction execution checks**

### Graphs
- **[BCGTV13]**
- **This work**
- **Improvement**
Comparison with program analysis

Most circuit generators use program analysis:

- Fairplay [Malkhi Nisan Pinkas Sella 04]
- FairplayMP [Ben-David Nisan Pinkas 08]
- Ginger [Setty Vu Panpalia Braun Blumberg Walfish 12]
- Zaatar [Setty Braun Vu Blumberg Parno Walfish 12]
- Pinocchio [Parno Gentry Howell Raykova 13]
- Pantry [Braun Feldman Ren Setty Blumberg Walfish 13]

(a) dissect program $\mathcal{A}$ into components (loops, branches, ...)
(b) separately translate each component into a subcircuit

How to compare?

- Not universal (must know $\mathcal{A}$ in advance)
- Efficiency varies from $\mathcal{A}$ to $\mathcal{A}$... hard to characterize.

Roughly, very efficient for “circuit-like” programs; less efficient for memory-intensive programs.
Contribution 2:
SNARK implementation for circuits
SNARK implementation for circuits

- Theoretical protocol based on [PGHR13]
- Tailored choices at every level of stack: Finite fields, elliptic curves, pairings, multi-exponentiation, interpolation.

For $10^6$-gate circuit and 1000-bit input:

- Generator: $1.1 \times$
- Prover: $5.3 \times$
- Verifier: $1.8 \times$
**libsnark: our SNARK implementation**

- **Open source**
  
  ![GitHub](https://github.com/scipr-lab/libsnark)

  libsnark: a C++ library for zkSNARK proofs

  [GitHub Source](https://github.com/scipr-lab/libsnark)

- **Practical impact**

  Time to create ZeroCash transaction:

  - 3 min
  - 10 min
  - 15 min

  ![Diagram](https://via.placeholder.com/150)
Universality:
One-time trusted setup to verify any computation in zero-knowledge.

Efficiency:
Fastest zk-SNARK for circuits.

Applications:
Look at libsnark and come talk to us!
Thank you!
ePrint 2013/879
Case study: memcpy with JIT compilation

memcpy(void *dest, const void *src, size_t n)

Trivial as a circuit only for **fixed** src, dest and n.

... _copy:
  load.b reg, [src]
  store.b [dest], reg
  add src, src, 1
  add dest, dest, 1
  sub n, n, 1
  cunjmp _copy
...

Can be eliminated when unrolling

With JIT:

- Use Newton’s method to calculate \( k = \sqrt{2n/7} \)
- Unroll regular code \( k \) times
- Jump to the generated code

**Total cost:** \( \approx 6n \)

**Total cost:** \( \approx 4n + 11.48 \sqrt{n} \)