Program Representations (1)

Bar-Ilan Winter School on Verifiable Computation
Class 7
January 5, 2016

Michael Walfish
Dept. of Computer Science, Courant Institute, NYU
Let’s clarify a few things from class 5:

• What are the constraints C’ versus the constraints C?

• How does the assignment \( z \) (satisfying or not) affect V’s checks?

• How and why do QAPs dramatically improve the picture?
Attempt 3: Use long PCPs interactively (summary)

[IKO07, SMBW12]

Achieves simplicity, with good constants …

… but pre-processing is required (because $|q_i| = |v|$)

… and prover’s work is quadratic; address that shortly
Attempt 4: Use long PCPs non-interactively

[BCIOP13]

Query process now happens “in the exponent”

… pre-processing still required (again because \(|q_i| = |v|\))

… prover’s work still quadratic; addressing that soon
Recap

<table>
<thead>
<tr>
<th>who</th>
<th>efficient (short) PCPs</th>
<th>arguments, CS proofs</th>
<th>arguments w/ preprocessing</th>
<th>SNARGs w/ preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMSS92, AS92, BGSHV, Dinur, ...</td>
<td>Kilian92, Micali94</td>
<td>Kilian92, Micali94</td>
<td>IKO07, SMBW12, SVPBBW12</td>
<td>Groth10, GGPR12, BCIOP13, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>what</th>
<th>classical PCP</th>
<th>commit to PCP by hashing</th>
<th>commit to long PCP using linearity</th>
<th>encrypt queries to a long PCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>security</td>
<td>unconditional</td>
<td>CRHFs</td>
<td>linearly HE</td>
<td>knowledge-of-exponent</td>
</tr>
<tr>
<td>why/why not</td>
<td>not efficient for V</td>
<td>constants are unfavorable</td>
<td>simple</td>
<td>simple, non-interactive</td>
</tr>
</tbody>
</table>

(Thanks to Rafael Pass.)
Final attempt: apply linear query structure to GGPR’s QAPs
[Groth10, Lipmaa12, GGPR12]

Addresses the issue of quadratic costs.

PCP structure implicit in GGPR. Made explicit in [BCIOP13, SBVBBW13].
Summary of published argument implementations

"Zaatar" [SBVBBBW13]

interactive argument [IKO07]

"Pinocchio," "libsnaprk" [PGHR13, BCTV14b]

SNARG, zk-SNARK with preprocessing [Groth10, BCCT12, GGPR12]

preprocessing lowered to (high) constant [BCCT13, BCTV14a, CTV15]

linear PCP via QAPs [GGPR13]

plaintext queries

queries in exponent

- standard assumptions
- amortize over batch
- interactive

- non-falsifiable assumptions
- amortize indefinitely
- non-interactive, ZK, ...

QAPs play the same role (but much, much better!) as "Q(z) plus the \([z, z \otimes z]\) encoding" (which is from [ALMSS92]; see [SMBW12, Apdx A] for a self-contained listing). This works because QAPs have a linear query structure, meaning that the query is a vector and the response is the dot product with a fixed vector).
Onto the front-end
This session: front-end techniques

- **Key ideas:** arithmetization, the convenience of non-determinism, data-dependent control flow, the price of generality, amortization

Recall the technical role of the front-end: given computation $f$, produce constraints $C$, where $C$ is degree-2 constraints over $\mathbb{F}$ and variables $(X, Y, Z)$ s.t.

$$\forall x, y: \exists w \text{ s.t. } y = f(x, w) \iff C(X = x, Y = y) \text{ is satisfiable}$$
applicable computations

<table>
<thead>
<tr>
<th>concrete costs</th>
<th>“regular”</th>
<th>straight line</th>
<th>pure</th>
<th>stateful</th>
<th>general loops</th>
<th>function pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowest</td>
<td>CMT++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thaler13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CMT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CMT12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Allspice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VSBW13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>args</td>
<td>Pepper</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMBW12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ginger</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVPBBW12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>highest</td>
<td>Zaatar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SBVBPW13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pinocchio</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PGHR13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Geppetto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CFH..PZ15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pantry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BFRSBW13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Buffet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WSRBW15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

better

structured ASIC

arbitrary ASIC

lowest

CMT++
Thaler13
CMT
CMT12
Allspice
VSBW13
Pepper
SMBW12
Ginger
SVPBBW12
Zaatar
SBVBPW13
Pinocchio
PGHR13
Geppetto
CFH..PZ15
Pantry
BFRSBW13
Buffet
WSRBW15

higher

PCD & bootstrapping
BCTV14a, CTV15

BCTV
BCTV14b
BCGTV
BCGTV13

CPU
This session: front-end techniques

- Key ideas: arithmetization, the convenience of non-determinism, data-dependent control flow, the price of generality, amortization

- Focus on “non-deterministic ASICs”; provides intuition for the rest
Rest of this session

(1) Arithmetization: from programs to constraints

(2) Enhancing expressiveness: data-dependent control flow

(3) Costs and comparisons
We will walk through the process of transforming a program into equivalent constraints (arithmetization):

- How program structures translate.
- How the translation is automated by a C compiler.
- How the translation targets the format required by the back-end.

A lot of this is folklore (not many references, but see Braun’s thesis [Braun12] and the appendices of Ginger [SVPBBW12]).

We will work over the field $\mathbb{F}_p$ (the integers mod a prime, $p$). Let’s begin with a warmup . . .

Assignment allocates a fresh constraint variable (circuit wire):

\[
\begin{align*}
    a &= 4; \\
    a &= a + 3; \\
\end{align*}
\]
Boolean functions turn into arithmetic:

// assume x1 and x2 are 0-1 valued
\[ y = x_1 \ \text{AND} \ x_2; \quad \implies \]
\[ y = x_1 \ \text{OR} \ x_2; \quad \implies \]

**Exercise:** Fill in the equivalent constraints for the functions below:

\[ y = \neg x_1; \quad \implies \]
\[ y = x_1 \ \text{NAND} \ x_2; \quad \implies \]
\[ y = x_1 \ \text{NOR} \ x_2; \quad \implies \]
\[ y = x_1 \ \text{XOR} \ x_2; \quad \implies \]
Equality checks are efficient:

// x1 and x2 need not be Boolean
z3 = (z1 != z2) ? 1 : 0; =⇒

Observe: the constraints exploit “non-determinism” … even though the computation is deterministic.

EXERCISE: Fill in the equivalent constraints for the function below:

y = (x1 == x2) ? 1 : 0; =⇒
Conditionals require constraints (or gates) for each branch:

```plaintext
if (x1)
    y = x2;
else
    y = x3;
⇒
```

**EXERCISE:** Fill in the equivalent constraints for the excerpt below:

```plaintext
if (z1 == 3)
    z2 = 10;
else if (z1 == 5)
    z2 = 20;
else
    z2 = 30;
⇒
```

**EXERCISE:** Fill in the equivalent constraints for the excerpt below:

```plaintext
// assume z1, z2 are already defined
if (z3 == 9)
    z1 = z1 + 6;
else
    z2 = z2 + 10;
⇒
```
Loops are unrolled:

\[
\begin{align*}
i &= 0; \\
\text{for } (j=0; j<10; j++) \{ &
i++;
\}
\end{align*}
\]

\[
Z = 0, \\
Z_0 = Z + 1, \\
Z_1 = Z_0 + 1, \\
\vdots \\
Z_9 = Z_8 + 1
\]

Loop bounds must be static (for now).
EXERCISE (primitive load): (a) Write a program in pseudocode that takes two inputs: an array of some fixed size (which you can represent as a vector of variables) and an index in the array. Return the value at the specified index in the array. (b) Translate your program into constraints. (c) What’s the most efficient set (smallest number) of constraints that you can produce for this program?

EXERCISE (Challenge!): Your solution to the previous exercise probably had $O(m)$ constraints, where $m$ is the size of the input array. Can you lower the number of constraints to $O(\log m)$? (This will also require changing the input specification.)
Negative numbers require care. ($\mathbb{F}_p$ has no notion of “less than zero”).

What about order comparisons (such as $x_1 < x_2$)?

```plaintext
if (x1 < x2)
    y = 3;
else
    y = 4;
```

\[
M\{C_<\},
M(Y - 3) = 0,
(1 - M)\{C_{\geq}\},
(1 - M)(Y - 4) = 0
\]

\[
C_< = \begin{cases}
B_0(1 - B_0) & = 0, \\
B_1(2 - B_1) & = 0, \\
\ldots
\end{cases}
\]

Cost: $O(w)$, where $w$ is bit width of variables.
**EXERCISE:** Write down constraints for $\leq$ and $>$.  

**EXERCISE:** Write down constraints for $z_3 = z_2 \mid z_1$, where $\mid$ is bitwise or.  

**EXERCISE (Challenge!):** So far, we have presumed that the original computation was working over the integers; we then mapped integer operations into $\mathbb{F}_p$, and from there to constraints. Extend this model to rational numbers: let the program work (in principle) over $\mathbb{Q}$, identify a suitable finite field for the constraints, and describe how to translate operations to constraints.

Hint: Show that there is a choice of $p$ for which a computation over $\mathbb{Q}/p$ (the quotient field of $\mathbb{F}_p$) is isomorphic to a computation over $\mathbb{Q}$. How will you handle the order comparisons ($<$, etc.)?
The foregoing process is automated. A compiler for (a subset of) C:

- Transforms the input program to *single assignment*
- Uses “pseudoconstraints” for some of the assignments
- Outputs constraints and annotations (hints for the prover)

By tracking the sizes of intermediate values, the compiler:

- Infers lower bound on prime \( p \).
  
  - Example: for matrix multiplication, compiler is told that inputs are signed \( N \) bits. Compiler can infer that \( p \) must be at least \( m \cdot 2^{2N} \).

- Produces only necessary bitwise constraints.

For more about the mechanics of compilation, see Braun’s thesis [Braun12]; a summary is in Pantry [BFRSBW13; §2, §7]. See also Ginger [SVPBBW12] and Pinocchio [PGHR13].
The compiler must obey the constraint format required by the back-end:

- Degree-2, and possibly also:
- Quadratic form, meaning $p_A \cdot p_B = p_C$, where each $p$ is a degree-1 polynomial. This is needed for QAP-based back-ends [GGPR13].

**Exercise:** Assuming $C$ consists of degree-2 constraints, describe a (straightforward) reduction from $M\{C\}$ to a set of degree-2 constraints. What is the cost of the reduction, in terms of extra variables and constraints introduced?

**Exercise:** Consider the constraint \{3 \cdot Z_1Z_2 + 2 \cdot Z_3Z_4 + Z_5 - Z_6 = 0\}. Replace this with three constraints in quadratic form.

**Exercise:** What is the cost, in terms of the number of extra variables and constraints, of transforming a set of degree-2 constraints $C$ to a set $C'$ in quadratic form? What is the worst case? Do “usual” computations experience the worst case?
The compiler must obey the constraint format required by the back-end:

- Degree-2, and possibly also:
- Quadratic form, meaning \( p_A \cdot p_B = p_C \), where each \( p \) is a degree-1 polynomial. This is needed for QAP-based back-ends [GGPR13].

("Quadratic Form" = "R1CS")

Question: what are the R1CS constraints for matrix multiplication?
Digression: What is Freivalds algorithm for matrix multiplication?
(1) Arithmetization: from programs to constraints

(2) Enhancing expressiveness: data-dependent control flow

(3) Costs and comparisons
What happens when loops are nested?

```plaintext
i=0;
for (j=0; j<10; j++) {
    i++;
    for (k=0; k<2; k++) {
        i=i*2;
    }
}
```

Inner loop unrolls into every iteration of outer loop.

| i=0; |
| for (j=0; j<10; j++) { |
|   i++; |
|   for (k=0; k<2; k++) { |
|     i=i*2; |
|   } |
| } |

| \[ Z = 0, \] |
| \[ Z_0 = Z + 1, \quad \text{// } j = 0 \] |
| \[ Z_1 = Z_0 \cdot 2, \quad \text{// } k = 0 \] |
| \[ Z_2 = Z_1 \cdot 2, \quad \text{// } k = 1 \] |
| \[ Z_3 = Z_2 + 1, \quad \text{// } j = 1 \] |
| \[ Z_4 = Z_3 \cdot 2, \quad \text{// } k = 0 \] |
| \[ Z_5 = Z_4 \cdot 2, \quad \text{// } k = 1 \] |
| \ldots |
What happens when loops are nested?

```plaintext
i=0;
for (j=0; j<10; j++) {
    i++;
    for (k=0; k<2; k++) {
        i=i*2;
    }
}
```

Inner loop unrolls into every iteration of outer loop.

What if the loop bounds were data-dependent?
Consider a decoder for a run-length encoded string with output size \( \text{OUTLENGTH} \). Compiling this requires bounding both loops.

“a5b2” ⇒ “aaaaabb”

\[
i = j = 0;
\]
\[
\text{while} \ (j < \text{OUTLENGTH}) \ \{ \\
\text{inchar} = \text{input}[i++]; \\
\text{length} = \text{input}[i++]; \\
\text{do } \{ \\
\text{output}[j++] = \text{inchar}; \\
\text{length}--; \\
\} \ \text{while} \ (\text{length} > 0);
\}
\]

1. Read (inchar, length) pair.
2. Emit inchar, length times.
Consider a decoder for a run-length encoded string with output size \texttt{OUTLENGTH}. Compiling this requires bounding both loops.

```
“a5b2” ⇒ “aaaaabb”
```

\[
i = j = 0;
\text{while} \ (j < \texttt{OUTLENGTH}) \{ \text{inchar} = \texttt{input}[i++]; \text{length} = \texttt{input}[i++];
\text{do} \{ \text{output}[j++] = inchar; \text{length}--; \} \text{while} \ (\text{length} > 0); \}
\]

At one extreme, a single character’s run length could be \texttt{OUTLENGTH}.

Consider a decoder for a run-length encoded string with output size \( \text{OUTLENGTH} \). Compiling this requires bounding both loops.

\[
\text{``a5b2'' } \Rightarrow \text{``aaaaabb''}
\]

```java
i = j = 0;
while (j < \text{OUTLENGTH}) { /* bound=\text{OUTLENGTH} */
    inchar = input[i++];
    length = input[i++];

    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

At the other extreme, every character’s run length could be 1, and the outer loop would iterate \( \text{OUTLENGTH} \) times.
Consider a decoder for a run-length encoded string with output size \textit{OUTLENGTH}. Compiling this requires bounding both loops.

“a5b2” ⇒ “aaaaaabb”

i = j = 0;
while (j < \textit{OUTLENGTH}) { /* bound=\textit{OUTLENGTH} */
    inchar = input[i++];
    length = input[i++];

do {
    output[j++] = inchar;
    length--;
} while (length > 0);
}

Thus, the compiler must unroll the inner loop to \textit{OUTLENGTH}^2 iterations, even though the computation is linear in \textit{OUTLENGTH}. 

Observations:

1. Loop nests are equivalent to finite state machines (FSMs) . . .
2. . . . but FSMs are more efficiently represented in constraints

Idea: transform loop nests into FSMs.
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];
    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}

How can a compiler perform such a transformation systematically?
Step 1: build a control flow graph:

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];
    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```
Step 1: build a control flow graph:

i = j = 0;
while (j < OUTLENGTH) {
inchar = input[i++];
length = input[i++];
   do {
      output[j++] = inchar;
      length--;
   } while (length > 0);
}

- Identify vertices: straight line code segments.
- Identify edges: control flow between segments. 1 transitions to 2 unconditionally. 2 self-transitions when length > 0. 2 transitions to 1 when length <= 0.
Step 1: build a control flow graph:

```c
i = j = 0;
while (j < OUTLENGTH) {
inchar = input[i++];
length = input[i++];
do {
    output[j++] = inchar;
    length--;
} while (length > 0);
}
```

- Identify vertices: straight line code segments.
- Identify edges: control flow between segments.
  1 transitions to 2 unconditionally.
Step 1: build a control flow graph:

```c
i = j = 0;
while (j < OUTLENGTH) {
  inchar = input[i++];
  length = input[i++];
  do {
    output[j++] = inchar;
    length--;
  } while (length > 0);
}
```

- Identify vertices: straight line code segments.
- Identify edges: control flow between segments.
  - 1 transitions to 2 unconditionally.
  - 2 self-transitions when `length > 0`.
Step 1: build a control flow graph:

```
i = j = 0;
while (j < OUTLENGTH) {
inchar = input[i++];
length = input[i++];
do {
   output[j++] = inchar;
   length--;
} while (length > 0);
}
```

- Identify vertices: straight line code segments.
- Identify edges: control flow between segments.
  1 transitions to 2 unconditionally.
  2 self-transitions when length > 0.
  2 transitions to 1 when length <= 0.
Step 2: from the control flow graph

```
i = j = 0;
while (j < OUTLENGTH) {
inchar = input[i++];
length = input[i++];
do {
    output[j++] = inchar;
    length--;
} while (length > 0);
}
i = j = 0;
state = 1;
while (j < OUTLENGTH) {
    if (state == 1) {
        inchar = input[i++];
        length = input[i++];
        state = 2;
    }
    if (state == 2) {
        output[j++] = inchar;
        length--;
        if (length <= 0) {
            state = 1;
        }
    }
}
```
Step 2: from the control flow graph, output the finite state machine.

```
i = j = 0;
state = 1;
while (j < OUTLENGTH) {
    if (state == 1) {
        inchar = input[i++];
        length = input[i++];
        state = 2;
    }
    if (state == 2) {
        output[j++] = inchar;
        length--;
        if (length <= 0) {
            state = 1;
        }
    }
}
```
Step 2: from the control flow graph, output the finite state machine.

\[ i = j = 0; \]

\[ \text{while} \ (j < \text{OUTLENGTH}) \{ \]
\[ \quad \text{inchar} = \text{input}[i++]; \]
\[ \quad \text{length} = \text{input}[i++]; \]
\[ \quad \text{do} \{ \]
\[ \quad \quad \text{output}[j++] = \text{inchar}; \]
\[ \quad \quad \text{length}--; \]
\[ \quad \} \text{while} \ (\text{length} > 0); \]
\[ \} \]

\[ \text{while} \ (j < \text{OUTLENGTH}) \{ \]
\[ \quad \text{if} \ (\text{state} == 1) \{ \]
\[ \quad \quad \text{inchar} = \text{input}[i++]; \]
\[ \quad \quad \text{length} = \text{input}[i++]; \]
\[ \quad \quad \text{state} = 2; \]
\[ \quad \} \]
\[ \quad \text{if} \ (\text{state} == 2) \{ \]
\[ \quad \quad \text{output}[j++] = \text{inchar}; \]
\[ \quad \quad \text{length}--; \]
\[ \quad \quad \text{if} \ (\text{length} \leq 0) \{ \]
\[ \quad \quad \quad \text{state} = 1; \]
\[ \quad \quad \} \]
\[ \} \]
\[ \} \]
The technique generalizes to break, continue, arbitrary nesting, sequential loops, etc.

The whole thing works by source-to-source translation: from a program with tested loops to one in FSM form, and from there into constraints.

The technique is detailed in Buffet [WSRBW15]; it is inspired by, and extends, loop flattening from the parallel compilers literature [GF95, KNP05, YCFVEEGH08, Knijnenburg98, Polychron87].

Caveats:

• Programmer must tell compiler # of steps to unroll the FSM.
• No “program memory” ⇒ no function pointers.
EXERCISE: Transform the code below to a FSM. Assume that a bound is known on the total number of iterations that your FSM will take.

```java
// assume k is initialized earlier
// assume x is user-supplied input
while (j < MAX1) {
    k = k + 1;
    for (i = 0; i < x; i++) {
        if (i + j == k) {
            break;
        }
        j = j + 1;
    }
    j = j + 2;
}
```
A more general solution to data-dependent loop bounds

[BCTV14b, BCGTV13]

The state variable in the FSM is like a coarse program counter …
… what if the constraints modeled a program counter, registers, etc.?

Great programmability: handles all of C (but still requires bounded execution, because programmer selects # of CPU steps.)
An important question, when considering expressiveness, is how one represents **RAM computations** inside the circuit or constraint formalism. There are multiple approaches to this problem; time permitting, we may cover this topic.

For now, note that [BCTV14b] has an innovative solution, based on permutation networks, and assuming the “CPU approach”. Buffet [WSRBW15] borrows this solution and adapts it to the “ASIC approach”.

A self-contained, short description of [BCTV14b]’s solution is in section 2.3 of [WSRBW15].
(1) Arithmetization: from programs to constraints

(2) Enhancing expressiveness: data-dependent control flow

(3) Costs and comparisons
Costs arise from the front-end, the back-end, and their interaction

Goals:

- Understand concrete costs
- Understand the different amortization regimes
- Understand current trade-offs

Plan:

- Compare front-ends, by holding back-end constant
- Compare back-ends on two different circuits
- Examine various metrics (mostly running times)
- Examine the amortization regimes
Front-end comparison

Back-end: libsnark, which is BCTV’s [BCTV14b] optimized implementation of Pinocchio/GGPR [PGHR13, GGPR13].

Front-ends: implementations or re-implementations of

- Zaatar (ASIC) [SBVBPW13]
- BCTV (CPU) [BCTV14b]
- Buffet (ASIC) [WSRHBW15]
<table>
<thead>
<tr>
<th>concrete costs</th>
<th>applicable computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowest</td>
<td></td>
</tr>
<tr>
<td>CMT++ Thaler13</td>
<td></td>
</tr>
<tr>
<td>CMT CMT12</td>
<td>Allspice vsbw13</td>
</tr>
<tr>
<td></td>
<td>Pepper SMBW12</td>
</tr>
<tr>
<td></td>
<td>Ginger SVPBBW12</td>
</tr>
<tr>
<td></td>
<td>Zaatar SBVPBW13</td>
</tr>
<tr>
<td></td>
<td>Pinocchio PGHR13</td>
</tr>
<tr>
<td></td>
<td>Geppetto CFH..PZ15</td>
</tr>
<tr>
<td></td>
<td>Pantry BFRSBW13</td>
</tr>
<tr>
<td></td>
<td>Buffet WSRBW15</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>highest</td>
<td></td>
</tr>
<tr>
<td>PCD &amp; bootstrapping BCTV14a, CTV15</td>
<td></td>
</tr>
</tbody>
</table>
Front-end comparison

Back-end: libsnark, which is BCTV’s [BCTV14b] optimized implementation of Pinocchio/GGPR [PGHR13, GGPR13].

Front-ends: implementations or re-implementations of

- Zaatar (ASIC) [SBVBPW13]
- BCTV (CPU) [BCTV14b]
- Buffet (ASIC) [WSRHBW15]

Evaluation platform: cluster at Texas Advanced Computing Center (TACC)
Each machine runs Linux on an Intel Xeon 2.7 GHz with 32GB of RAM.
What are the verifier’s costs?

What are the prover’s costs?

| Proof length | 288 bytes |
| V per-instance | $6 \text{ ms} + (|x| + |y|) \cdot 3 \mu\text{s}$ |
| V pre-processing | $|C| \cdot 180 \mu\text{s}$ |
| P per-instance | $|C| \cdot 60 \mu\text{s} + |C| \log |C| \cdot 0.9 \mu\text{s}$ |
| P’s memory requirements | $O(|C| \log |C|)$ |

How do the front-ends compare to each other?

Are the constants good or bad?
How does the prover’s cost vary with the choice of front-end?

Extrapolated prover execution time, normalized to Buffet
All of the front-ends have terrible concrete performance
The maximum input size is far too small to be called practical.

<table>
<thead>
<tr>
<th>approach</th>
<th>Zaatar</th>
<th>BCTV</th>
<th>Buffet</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \times m$ mat. mult</td>
<td>215</td>
<td>7</td>
<td>215</td>
</tr>
<tr>
<td>merge sort $m$ elements</td>
<td>256</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>KMP str len: $m$</td>
<td>m=320, k=32</td>
<td>m=160, k=16</td>
<td>m=2900, k=256</td>
</tr>
</tbody>
</table>

The data reflect a “gate budget” of $\approx 10^7$ gates.

Pre-processing costs 10-30 minutes; proving costs 8-13 minutes.
Back-end comparison

- Data are from our re-implementations and match or exceed published results.

- All experiments are run on the same machines (2.7Ghz, 32GB RAM). Average 3 runs (experimental variation is minor).
  - For a few systems, we extrapolate from detailed microbenchmarks

- Benchmarks: $128 \times 128$ matrix multiplication and clustering algorithm
1. **What is the per-instance verification cost?**

2. **What are the cross-over points?**

   ![Graph showing computation and verification costs versus number of instances]

3. **What is the server’s overhead versus native execution?**
Verification cost sometimes beats (unoptimized) native execution.

Ishai et al. (PCP-based efficient argument)

128×128 matrix multiplication

Baseline 2 (103 ms)

Baseline 1 (3.5 ms)
local (slope: 103 ms/inst)
Zaatar (slope: 26 ms/inst)
CMT (slope: 36 ms/inst)
Allspice (slope: 35 ms/inst)
Ginger (slope: 14 ms/inst)
Pinocchio (slope: 10 ms/inst)
Thaler (slope: 12 ms/inst)

verification cost (minutes of CPU time)

number of instances
The prover’s costs are rather high.
Amortization comparison (of built systems)

Systems \([\text{CMT12, VSBW13, Thaler13}]\) derived from \([\text{GKR08}]\) require little or no amortization (but have some expressivity limitations)

Of the schemes that handle arbitrary circuits (that is, those based on arguments), preprocessing costs amortize differently. Ordered best to worst:

1. **Bootstrapped GGPR-based SNARKs** \([\text{BCTV14a, CTV15}]\)
   - Constant preprocessing; amortize over all computations (but concrete costs to prover are extremely high).

2. **BCTV** \([\text{BCTV14b}]\): “CPU” front-end + non-interactive GGPR back-end
   - Amortize over all future computations of the same length

3. **Pinocchio** \([\text{PGHR13}]\): “ASIC” front-end + non-interactive GGPR back-end
   - Amortize over all future uses of a given computation

4. **Zaatar** \([\text{SBVBPW13}]\): “ASIC” front-end + interactive GGPR/IKO back-end
   - Amortize over a batch of instances of a given computation
Summary of concrete performance

- Front-end: generality brings a concrete price (but better in theory)

- Verifier’s “variable costs”: genuinely inexpensive
- Verifier’s “pre-processing”: depends on setting
- Prover’s computational costs: mostly disastrous

- Memory: creates scaling limit for verifier and prover

Performance is plausibly acceptable in certain settings …

- It must be “regular” (to avoid setup costs), or there must be many identical instances (to amortize setup costs)
- The given computation needs to be small

… But none of the systems is at true practicality
Summary of front-ends

**“CPU”**
- Verbose (costly)
- Good amortization
- Great programmability

**“ASIC”**
- Concise
- Amortization worse
- Programmability not bad

C prog $\rightarrow$ MIPS .exe

Circuit is unrolled CPU execution
[BCGTV13, BCTV14a, BCTV14b, CTV15]

Each line translates to gates/constraints
[SVPBBW12, SBVBPW13, VSBW13, PGHR13, BFRSBW13, BCGGMTV14, BBFR14, FL14, KPPSST14, WSRBW15, CFHKKNPZ15]
Summary of key concepts and points

1. **Arithmetization**: how to translate programs to equations
   - Non-deterministic circuit/constraint models make this easier
   - The process can be automated

2. **Data-dependent control flow** can be provided naturally in either a “CPU” front-end or an “ASIC” front-end
   - Likewise for RAM operations

3. **There are trade-offs among expressiveness, amortization behavior, and performance**
   - None of the implementations have achieved genuine practicality
References


