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:

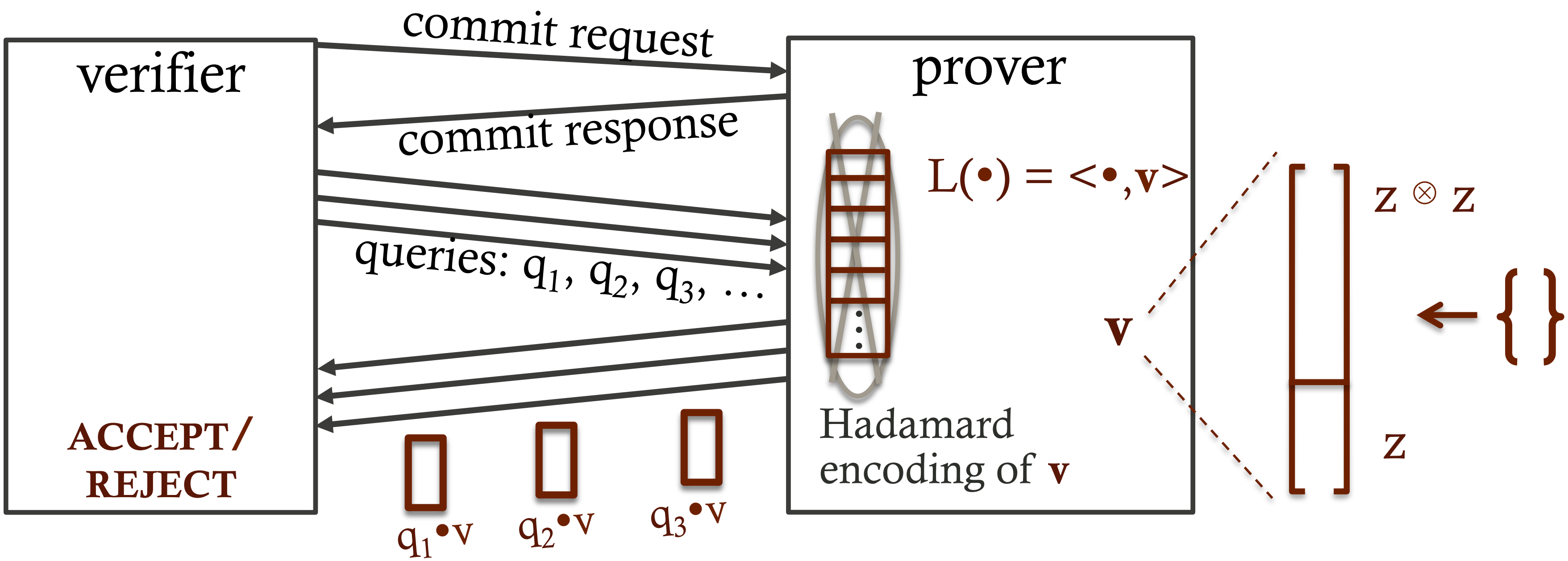
- What are the constraints C’ versus the constraints C?

- 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 …

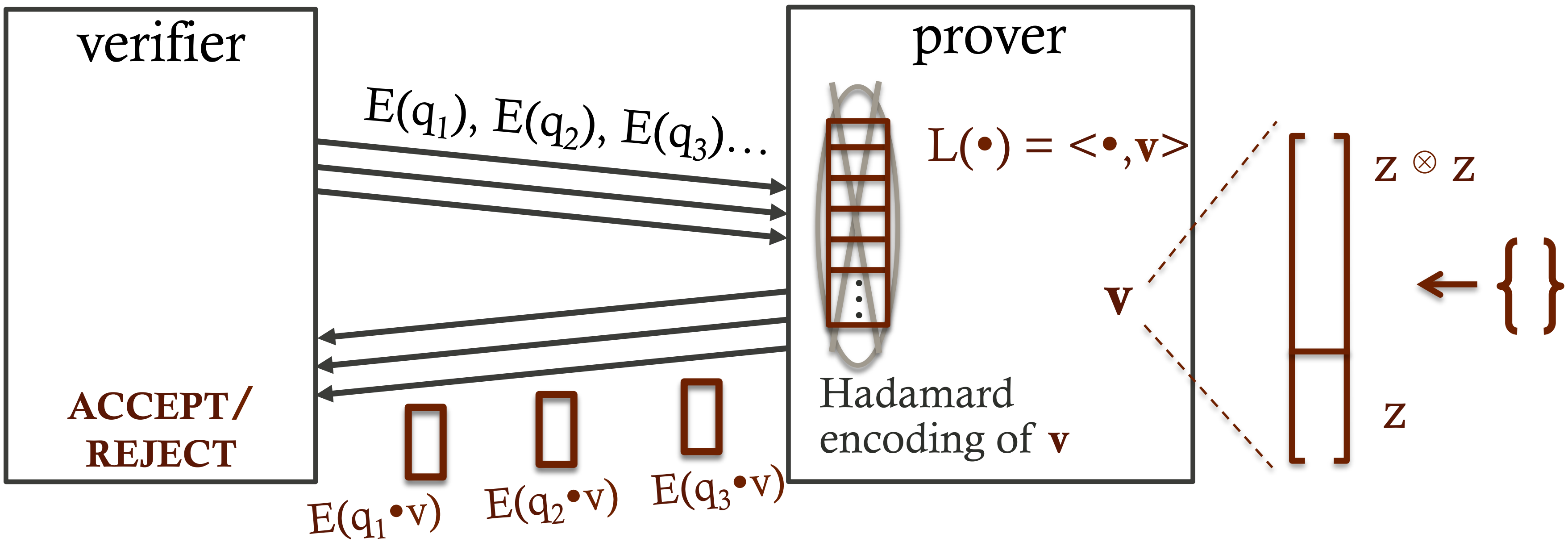
… but pre-processing is required (because $|q_i| = |v|$)

… and prover's work is quadratic; address that shortly

Attempt 4: Use long PCPs non-interactively

[BCIOP13]

preprocessing SNARG



Query process now happens "in the exponent"

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

… prover's work still quadratic; addressing that soon

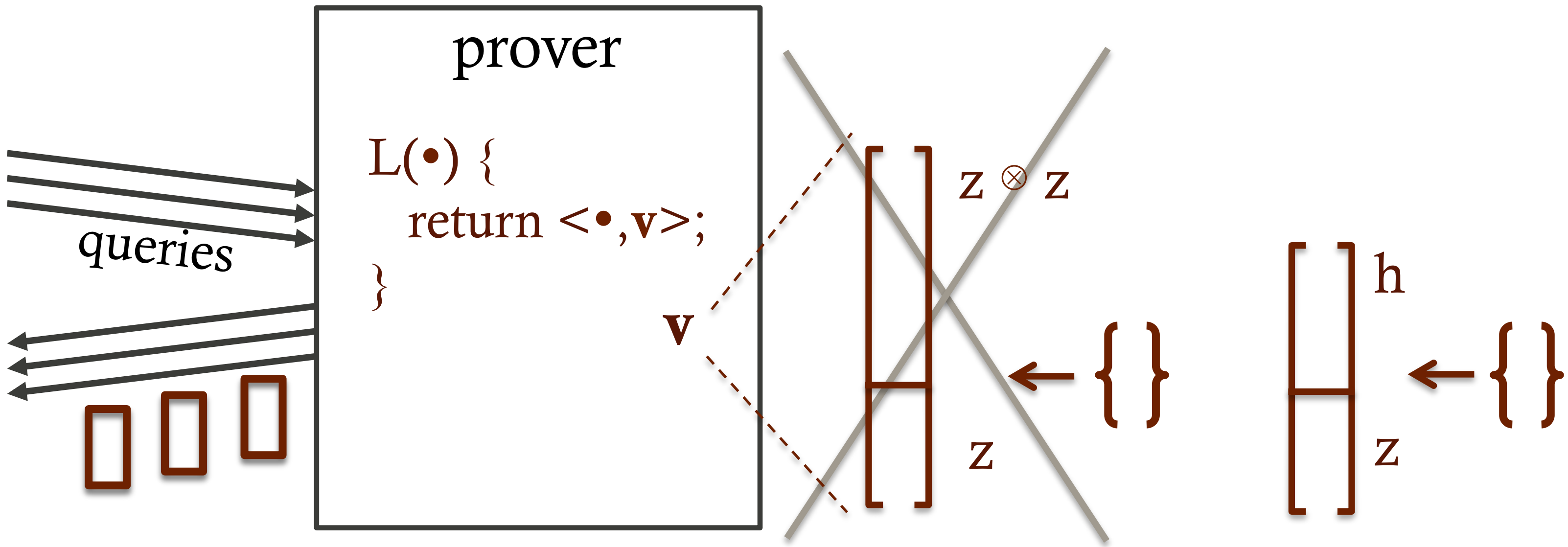
Recap

	efficient (short) PCPs	arguments, CS proofs	arguments w/ preprocessing	SNARGs w/ preprocessing
who	ALMSS92, AS92, BGSHV, Dinur, …	Kilian92, Micali94	IKO07, SMBW12, SVPBBW12	Groth10, GGPR12, BCIOP13, …
what	classical PCP	commit to PCP by hashing	commit to long PCP using linearity	encrypt queries to a long PCP
security	unconditional	CRHFs	linearly HE	knowledge-of-exponent
why/why not	not efficient for V	constants are unfavorable	simple	simple, non-interactive

(Thanks to Rafael Pass.)

Final attempt: apply linear query structure to GGPR's QAPs

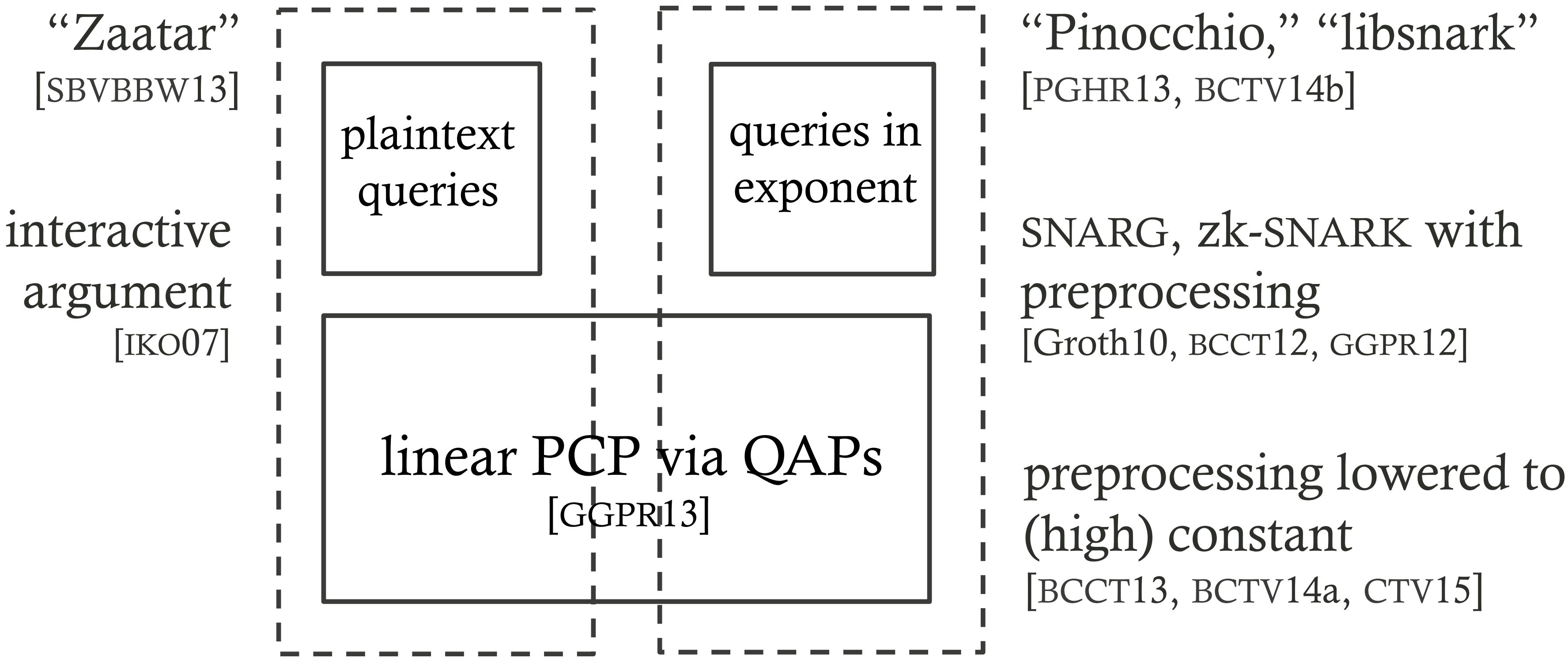
[Groth10, Lipmaa12, GGPR12]



Addresses the issue of quadratic costs.

PCP structure implicit in GGPR. Made explicit in [BCIOP13, SBVBBW13].

Summary of published argument implementations

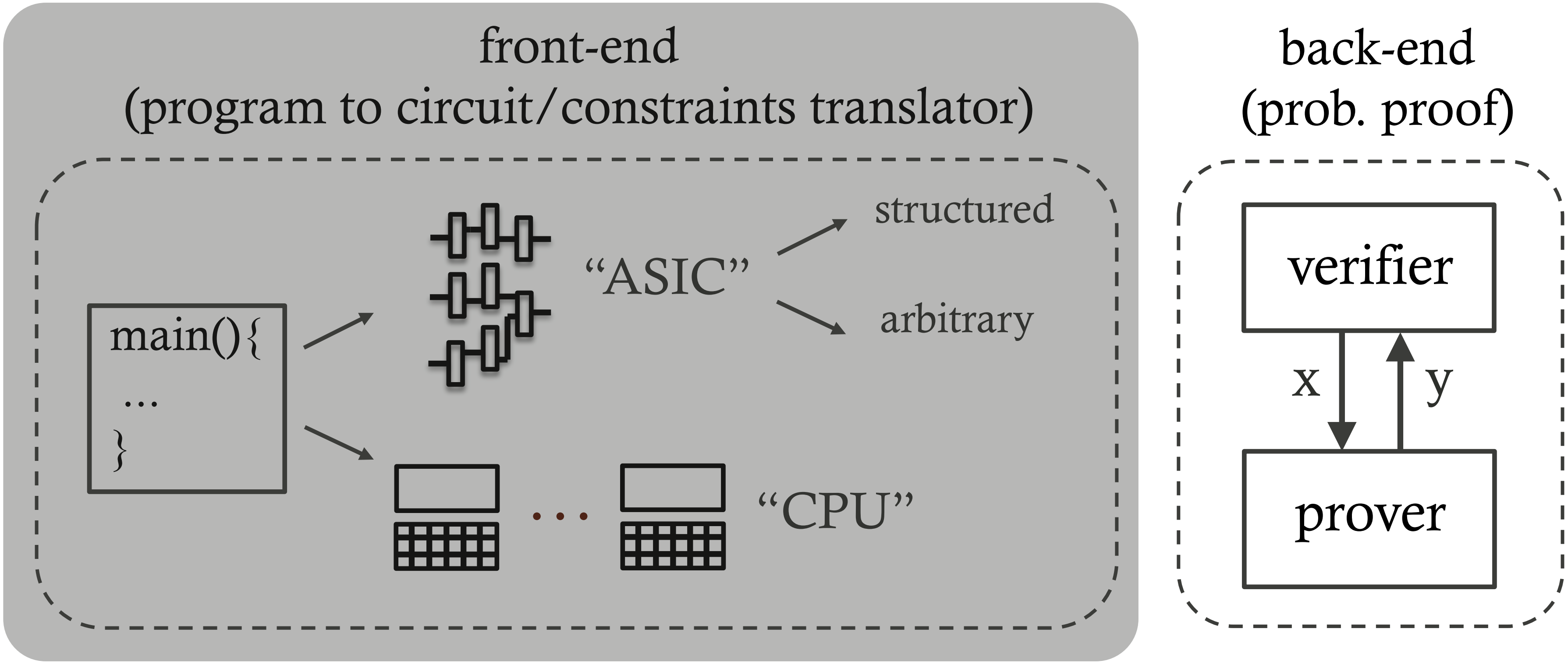


- standard assumptions
- amortize over batch
- interactive

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

QAPs play the same role (but much, much better!) as "Q(z) plus the [z, z ⊗ 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).

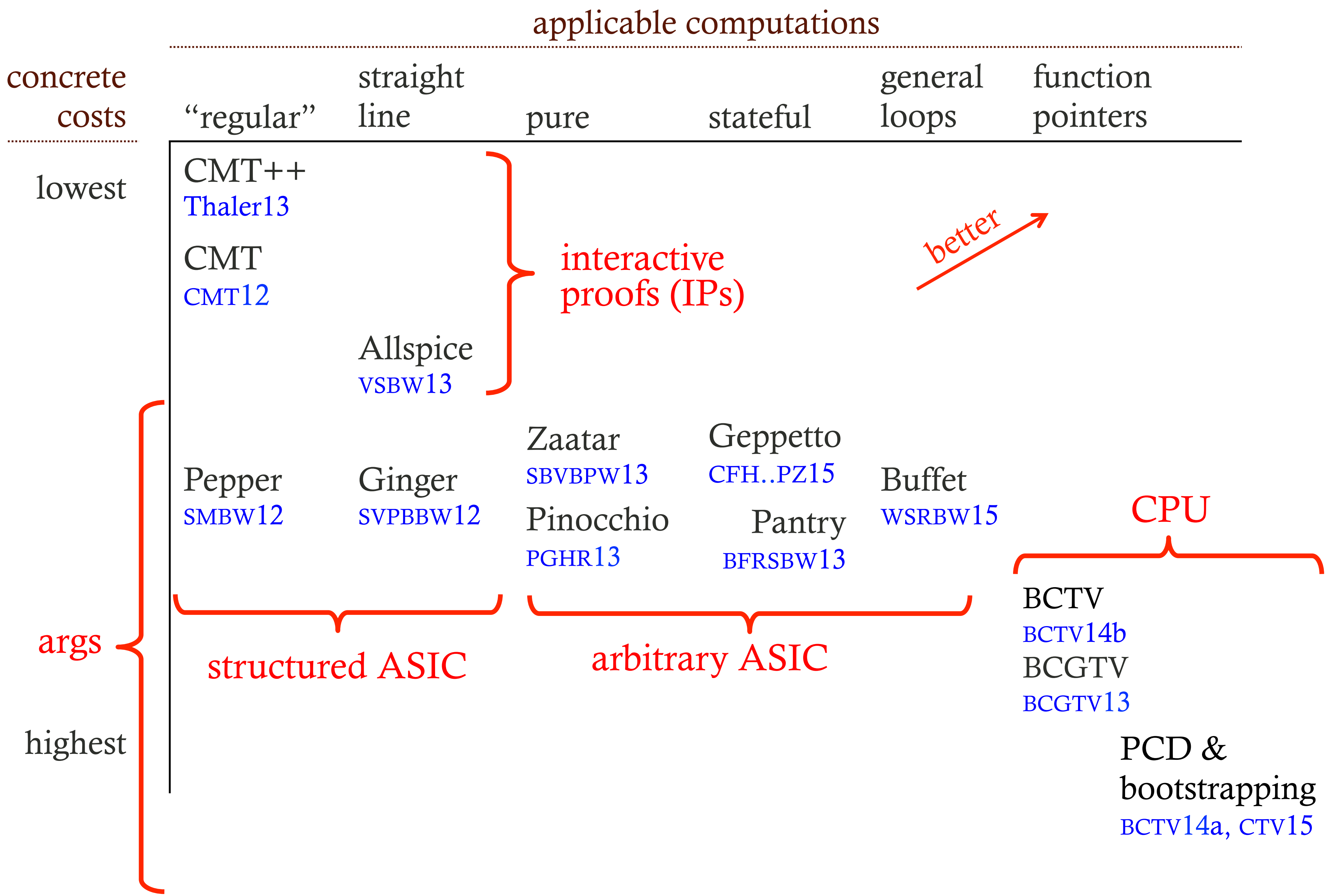
Onto the front-end…….

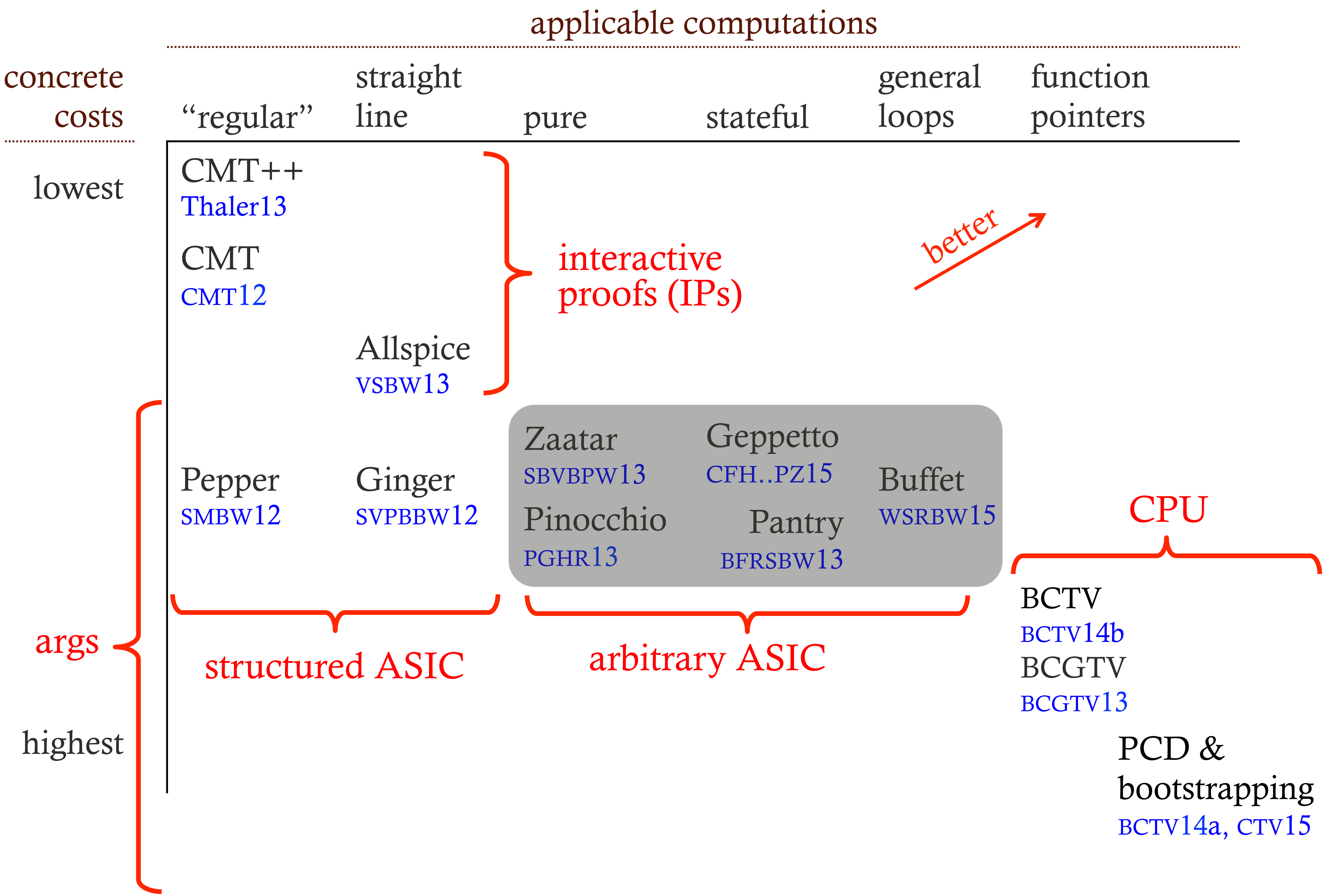


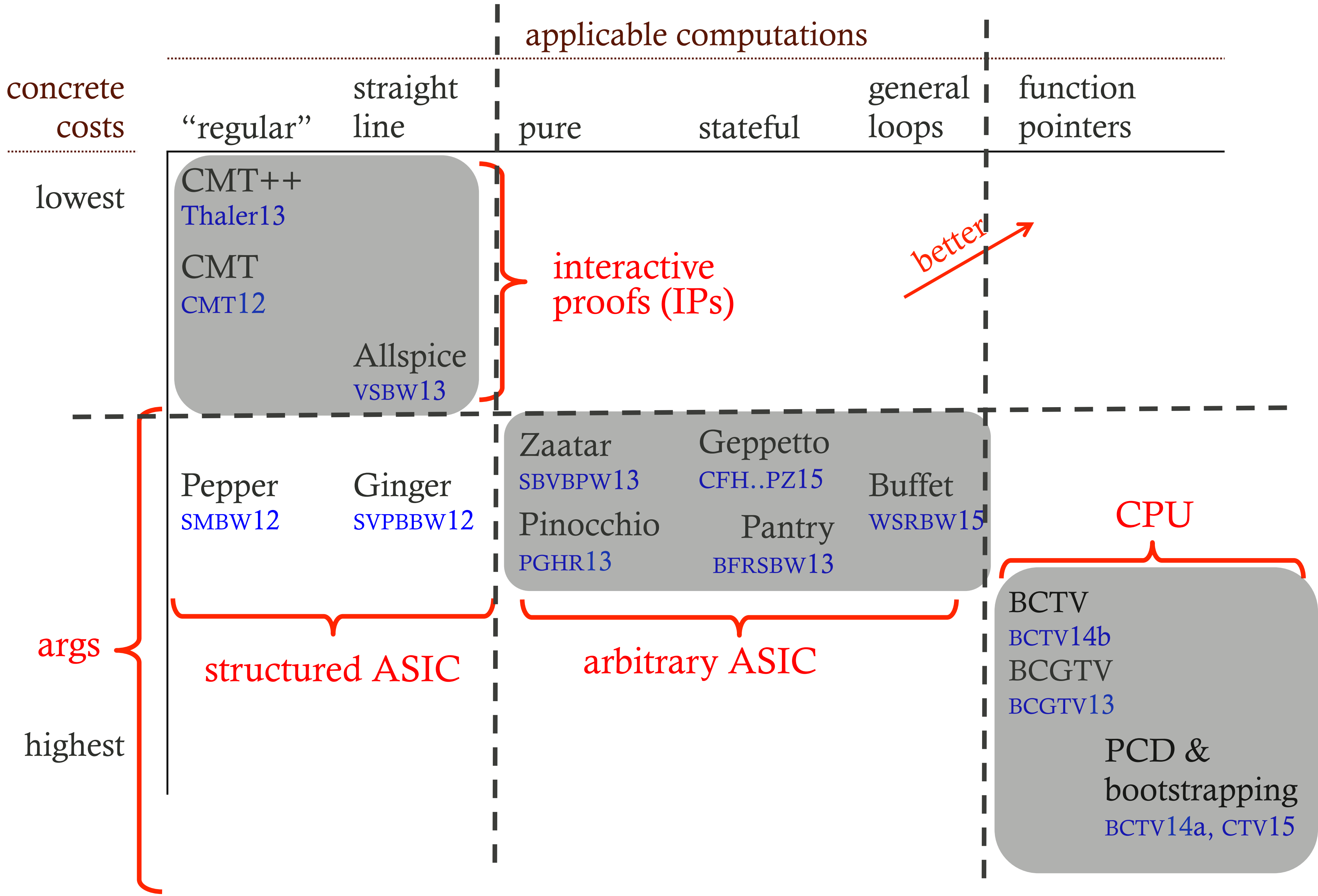
This session: front-end techniques

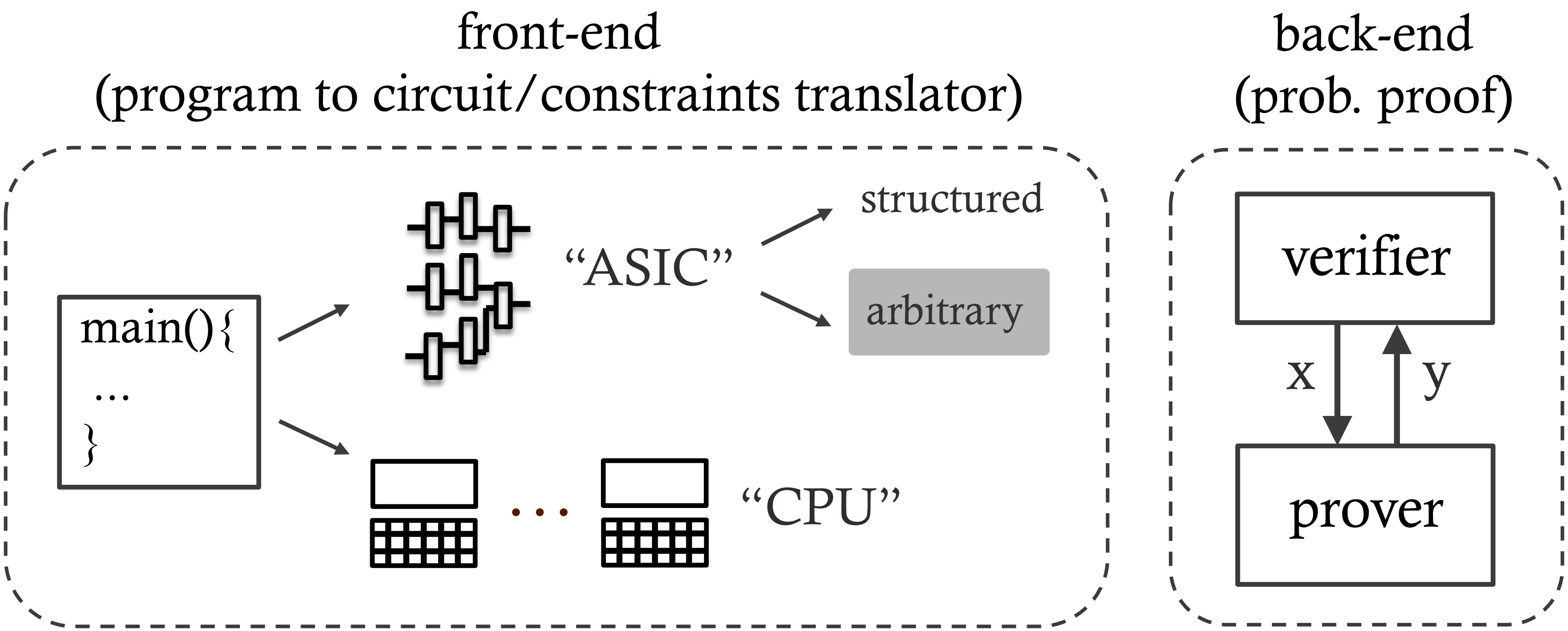
- Key ideas: arithmetization, the convenience of non-determinism, data-dependent control flow, the price of generality, amortization

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 s.t. y=f(x,w) $\Leftrightarrow$ C(X=x,Y=y) is satisfiable


applicable computations
concrete costs
"regular"
straight line
pure
stateful
general loops
function pointers
lowest
highest
CMT++
Thaler13
CMT
CMT12
Allspice
VSBW13
interactive proofs (IPs)
better
Pepper
SMBW12
Ginger
SVPBBW12
Zaatar
SBVBPW13
Pinocchio
PGHR13
Geppetto
CFH..PZ15
Pantry
BFRSBW13
Buffet
WSRBW15
CPU
BCTV
BCTV14b
BCGTV
BCGTV13
PCD & bootstrapping
BCTV14a, CTV15
args
structured ASIC
arbitrary ASIC


applicable computations
concrete costs
"regular"
straight line
pure
stateful
general loops
function pointers
lowest
highest
CMT++
Thaler13
CMT
CMT12
Allspice
VSBW13
interactive proofs (IPs)
better
args
Pepper
SMBW12
Ginger
SVPBBW12
Zaatar
SBVBPW13
Geppetto
CFH..PZ15
Pinocchio
PGHR13
Pantry
BFRSBW13
Buffet
WSRBW15
CPU
structured ASIC
arbitrary ASIC
BCTV
BCTV14b
BCGTV
BCGTV13
PCD &
bootstrapping
BCTV14a, CTV15


applicable computations
concrete costs
"regular"
straight line
pure
stateful
general loops
function pointers
lowest
CMT++
Thaler13
CMT
CMT12
Allspice
VSBW13
interactive proofs (IPs)
better
Zaatar
SBVBPW13
Geppetto
CFH..PZ15
Buffet
WSRBW15
Pepper
SMBW12
Ginger
SVPBBW12
Pinocchio
PGHR13
Pantry
BFRSBW13
CPU
BCTV
BCTV14b
BCGTV
BCGTV13
PCD &
bootstrapping
BCTV14a, CTV15
args
structured ASIC
arbitrary ASIC
highest



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):

```
a = 4;
a = a + 3;
```

$\Longrightarrow$

Boolean functions turn into arithmetic:

```
// assume x1 and x2 are 0-1 valued
y = x1 AND x2;          ⟹
y = x1 OR x2;           ⟹
```

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

```
y = NOT x1;             ⟹
y = x1 NAND x2;         ⟹
y = x1 NOR x2;          ⟹
y = x1 XOR x2;          ⟹
```

Equality checks are efficient:

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

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;
```
$\Longrightarrow$

Conditionals require constraints (or gates) for each branch:

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

$\Longrightarrow$

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

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

$\Longrightarrow$

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

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

$\Longrightarrow$

Loops are unrolled:

```
i=0;
for (j=0; j<10; j++) {
    i++;
}
```

$$\Longrightarrow \left\{ \begin{array}{l} Z = 0, \\ Z_0 = Z + 1, \\ Z_1 = Z_0 + 1, \\ \quad\vdots \\ Z_9 = Z_8 + 1 \end{array} \right\}$$

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 `x1 < x2`)?

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

$$\Longrightarrow \left\{ \begin{array}{l} M\{\mathcal{C}_<\}, \\ M(Y-3)=0, \\ (1-M)\{\mathcal{C}_{>=}\}, \\ (1-M)(Y-4)=0 \end{array} \right\}$$

$$\mathcal{C}_< = \left\{ \begin{array}{ll} B_0(1-B_0) & =0, \\ B_1(2-B_1) & =0, \\ \cdots & \\ & \\ & \\ & \end{array} \right\}$$

Cost: $O(w)$, where w is bit width of variables.

EXERCISE: Write down constraints for <= and >.

EXERCISE: Write down constraints for `z3 = z2 | z1`, where `|` 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 $\mathcal{C}$ consists of degree-2 constraints, describe a (straightforward) reduction from $M\{\mathcal{C}\}$ 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 $\{3 \cdot Z_1 Z_2 + 2 \cdot Z_3 Z_4 + Z_5 - Z_6 = 0\}$. 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 $\mathcal{C}$ to a set $\mathcal{C}$' 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?

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

$$\Longrightarrow \left\{ \begin{array}{ll} Z = 0, & \\ Z_0 = Z + 1, & \texttt{// j == 0} \\ Z_1 = Z_0 \cdot 2, & \texttt{// k == 0} \\ Z_2 = Z_1 \cdot 2, & \texttt{// k == 1} \\ Z_3 = Z_2 + 1, & \texttt{// j == 1} \\ Z_4 = Z_3 \cdot 2, & \texttt{// k == 0} \\ Z_5 = Z_4 \cdot 2, & \texttt{// k == 1} \\ \ldots & \end{array} \right\}$$

Inner loop unrolls into every iteration of outer loop.

What happens when loops are nested?

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

$$\Longrightarrow \left\{ \begin{array}{ll} Z = 0, & \\ Z_0 = Z + 1, & \texttt{// j == 0} \\ Z_1 = Z_0 \cdot 2, & \texttt{// k == 0} \\ Z_2 = Z_1 \cdot 2, & \texttt{// k == 1} \\ Z_3 = Z_2 + 1, & \texttt{// j == 1} \\ Z_4 = Z_3 \cdot 2, & \texttt{// k == 0} \\ Z_5 = Z_4 \cdot 2, & \texttt{// k == 1} \\ \ldots & \end{array} \right\}$$

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 `OUTLENGTH`. Compiling this requires bounding both loops.

“`a5b2`” ⇒ “`aaaaabb`”

```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];      1

    do {
        output[j++] = inchar;
        length--;             2
    } while (length > 0);
}
```

1. Read (`inchar`,`length`) pair.
2. Emit `inchar`, `length` times.

Consider a decoder for a run-length encoded string with output size `OUTLENGTH`. Compiling this requires bounding both loops.

“`a5b2`” ⇒ “`aaaaabb`”

```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

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

At one extreme, a single character’s run length could be `OUTLENGTH`.

Consider a decoder for a run-length encoded string with output size `OUTLENGTH`. Compiling this requires bounding both loops.

“`a5b2`” ⇒ “`aaaaabb`”

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

    do {                         /* bound=OUTLENGTH */
        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 `OUTLENGTH` times.

Consider a decoder for a run-length encoded string with output size `OUTLENGTH`. Compiling this requires bounding both loops.

"`a5b2`" $\Rightarrow$ "`aaaaabb`"

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

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

Thus, the compiler must unroll the inner loop to $\texttt{OUTLENGTH}^2$ iterations, even though the computation is linear in `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);
}
```

```
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;
        }
    }
}
```

How can a compiler perform such a transformation systematically?

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);
}
```

Step 1: build a control flow graph:

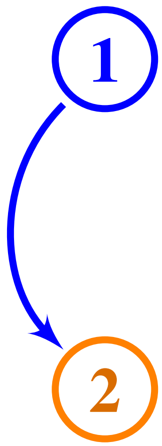
1

2

```
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.

Step 1: build a control flow graph:

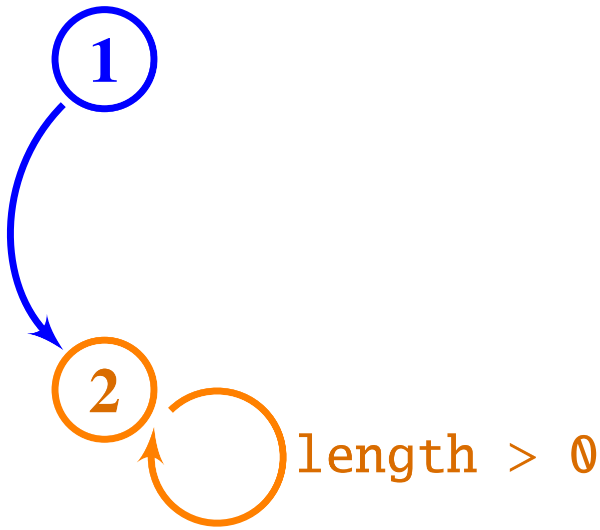



```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    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.

Step 1: build a control flow graph:



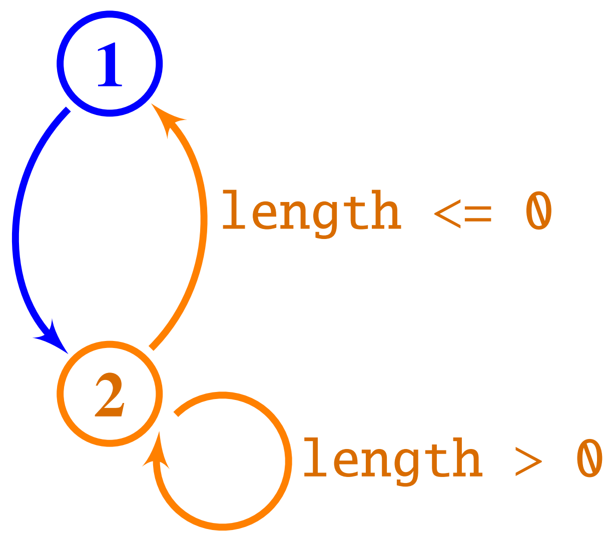

```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    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`.

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);
}
```

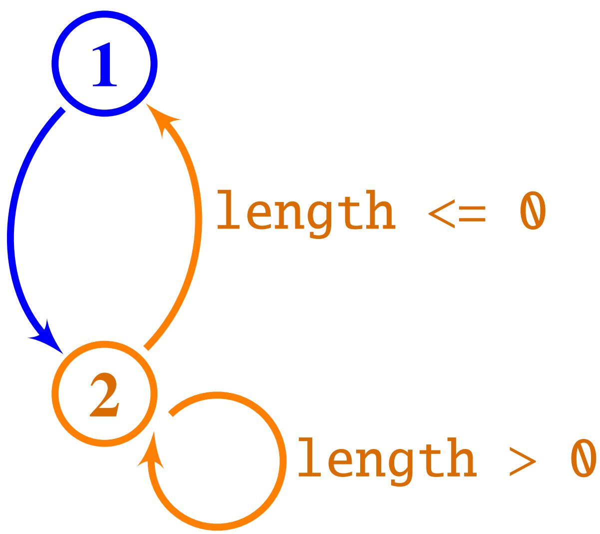
- Identify vertices: straight line code segments.
- Identify edges: control flow between segments.

  1 transitions to 2 unconditionally.

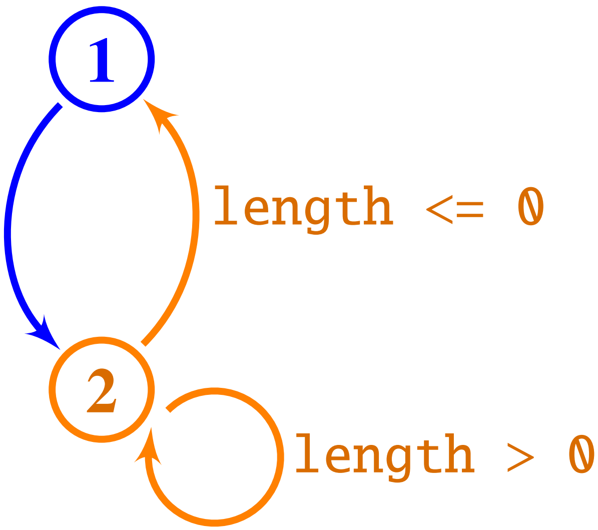
  2 self-transitions when `length > 0`.

  2 transitions to 1 when `length <= 0`.

Step 2: from the control flow graph



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;

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;
        }
    }
}
```

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" $\Rightarrow$ no function pointers.

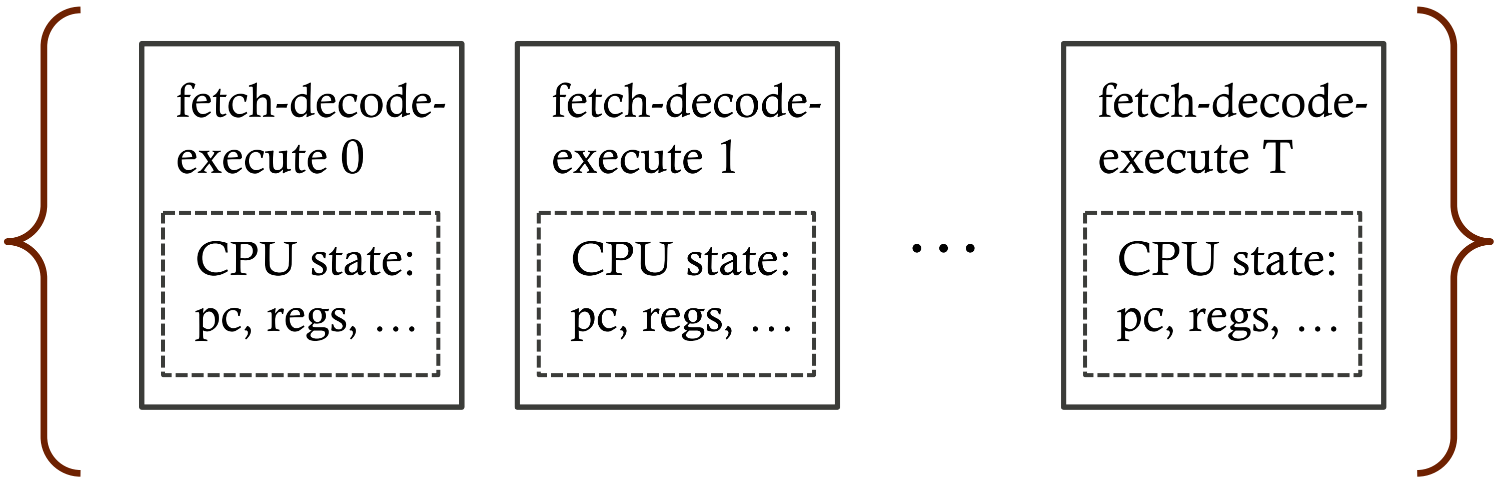
EXERCISE: Transform the code below to a FSM. Assume that a bound is known on the total number of iterations that your FSM will take.

```
// 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]

preview

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]

applicable computations

concrete costs	“regular”	straight line	pure	stateful	general loops	function pointers
lowest	CMT++ Thaler13					
	CMT CMT12					
		Allspice VSBW13				
	Pepper SMBW12	Ginger SVPBBW12	Zaatar SBVBPW13 Pinocchio PGHR13	Geppetto CFH..PZ15 Pantry BFRSBW13	Buffet WSRBW15	
						BCTV BCTV14b BCGTV BCGTV13
highest						PCD & bootstrapping BCTV14a, CTV15

better

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.

(1) What are the verifier's costs?

(2) What are the prover's costs?

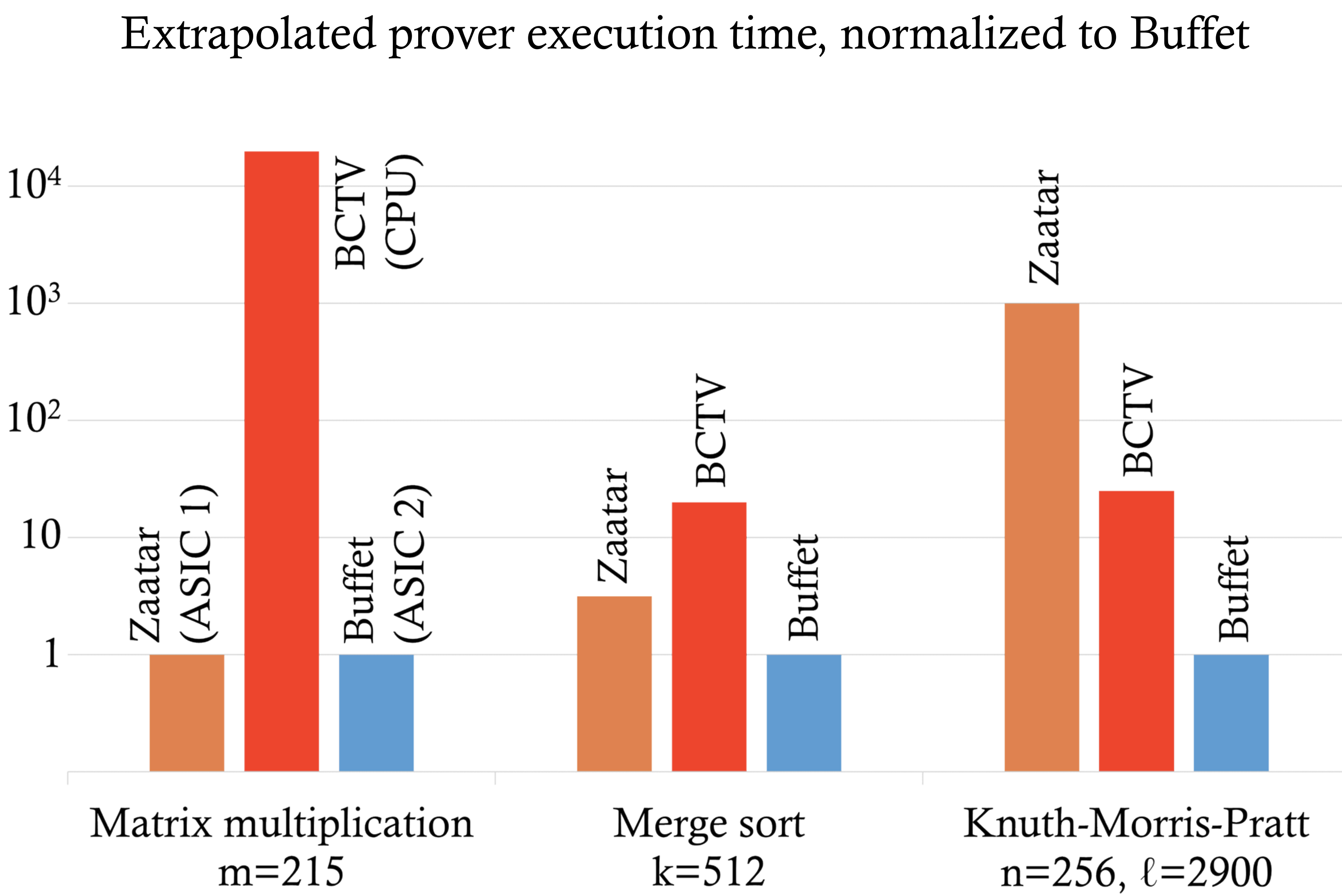
Proof length	288 bytes
V per-instance	6 ms + $(\lvert x\rvert + \lvert y\rvert)\cdot 3$ µs
V pre-processing	$\lvert C\rvert \cdot 180$ µs
P per-instance	$\lvert C\rvert \cdot 60$ µs $+ \lvert C\rvert \log \lvert C\rvert \cdot 0.9$µs
P's memory requirements	$O(\lvert C\rvert \log \lvert C\rvert)$

($\lvert C\rvert$: circuit size)

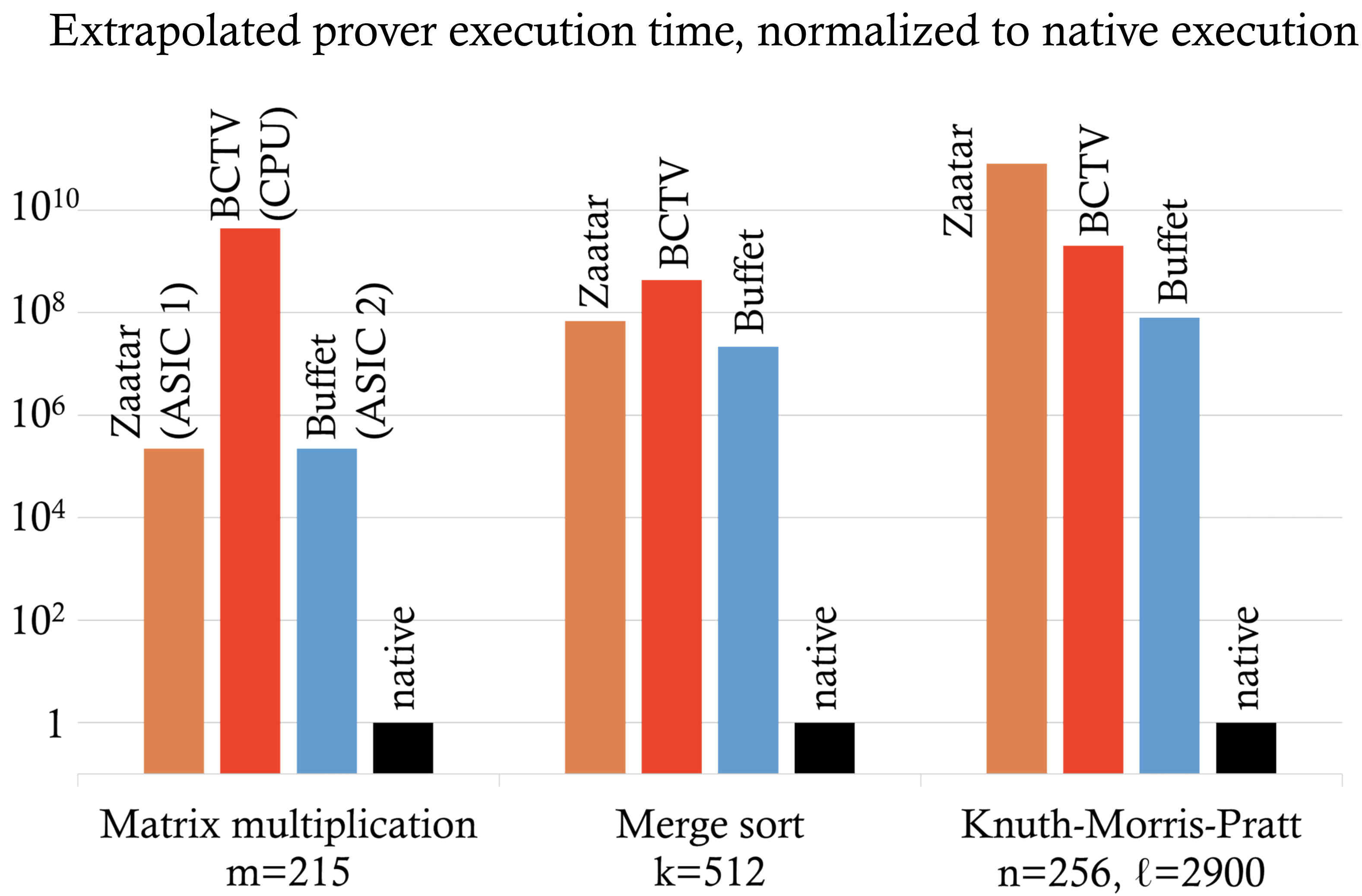
(3) How do the front-ends compare to each other?

(4) Are the constants good or bad?

How does the prover's cost vary with the choice of front-end?



All of the front-ends have terrible concrete performance



The maximum input size is far too small to be called practical

	Zaatar	BCTV	Buffet
approach	ASIC	CPU	ASIC
$m \times m$ mat. mult	215	7	215
merge sort m elements	256	32	512
KMP str len: m substr len: k	m=320, k=32	m=160, k=16	m=2900, k=256

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

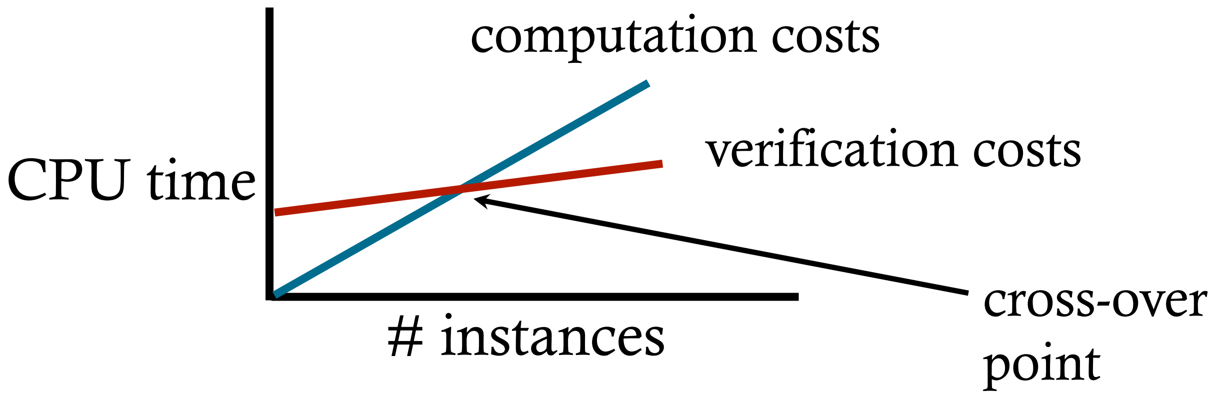
Pre-processing costs 10-30 minutes; proving costs 8-13 minutes

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: 128×128 matrix multiplication and clustering algorithm

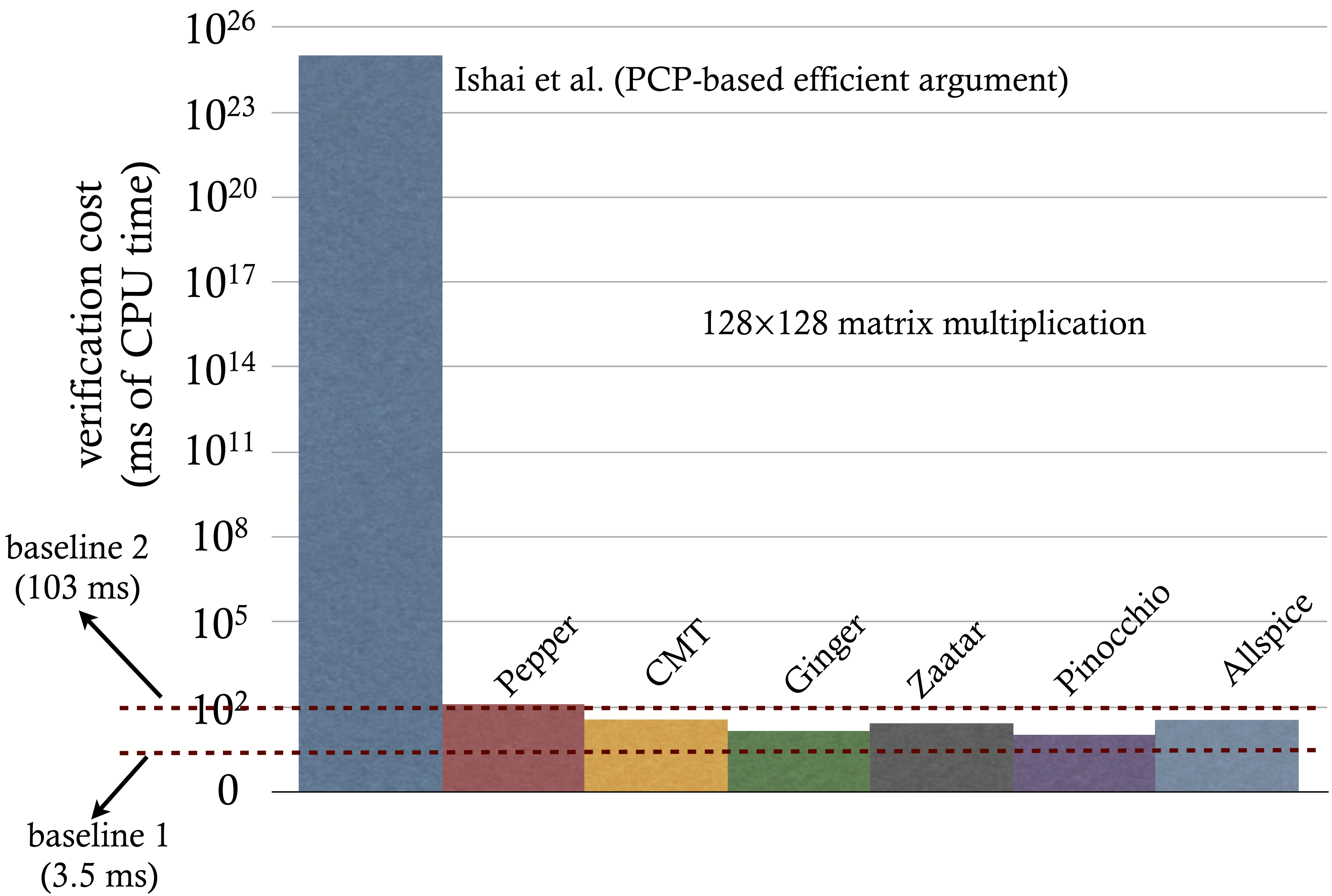
1. What is the per-instance verification cost?

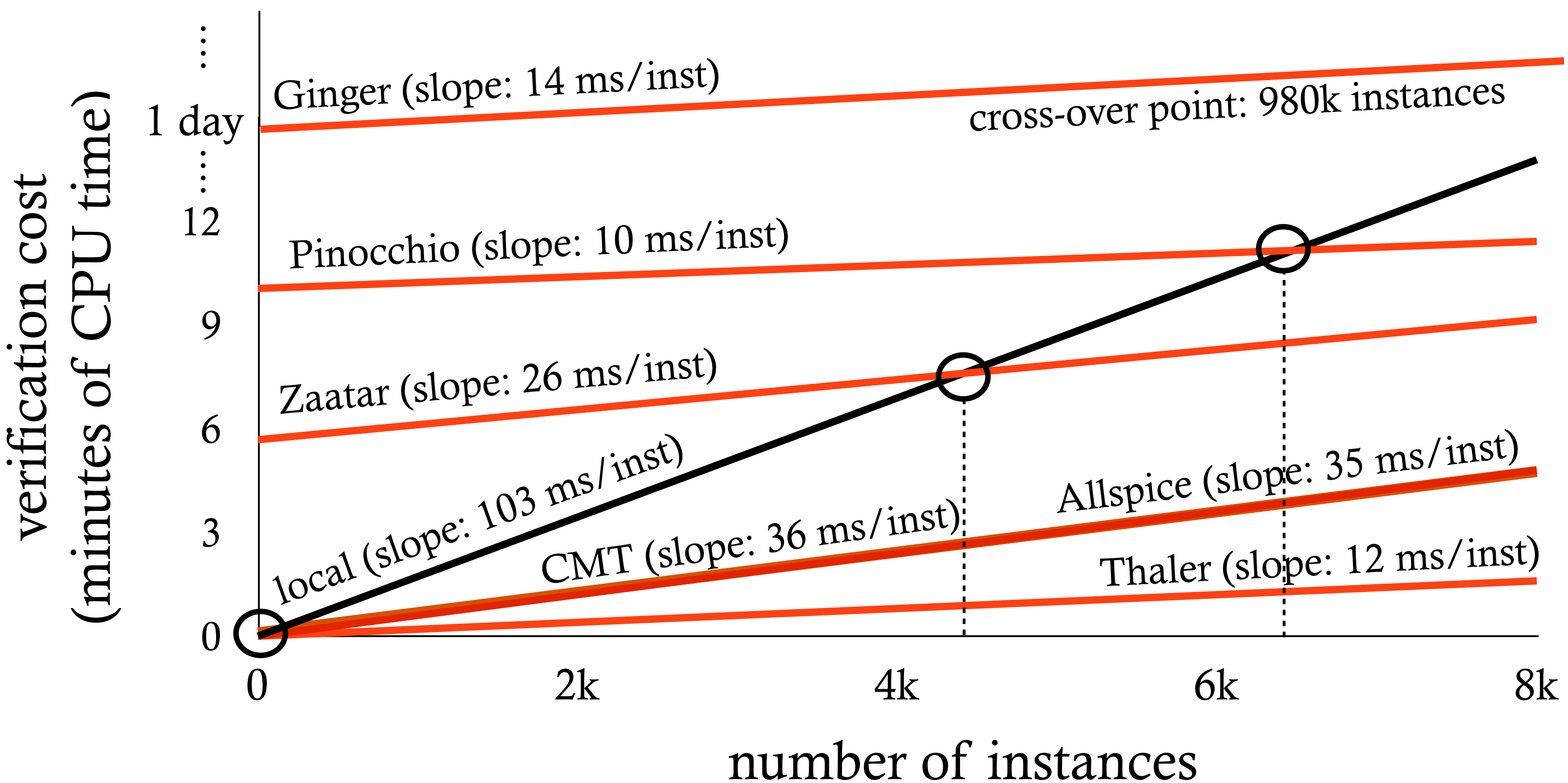
2. What are the cross-over points?



3. What is the server's overhead versus native execution?

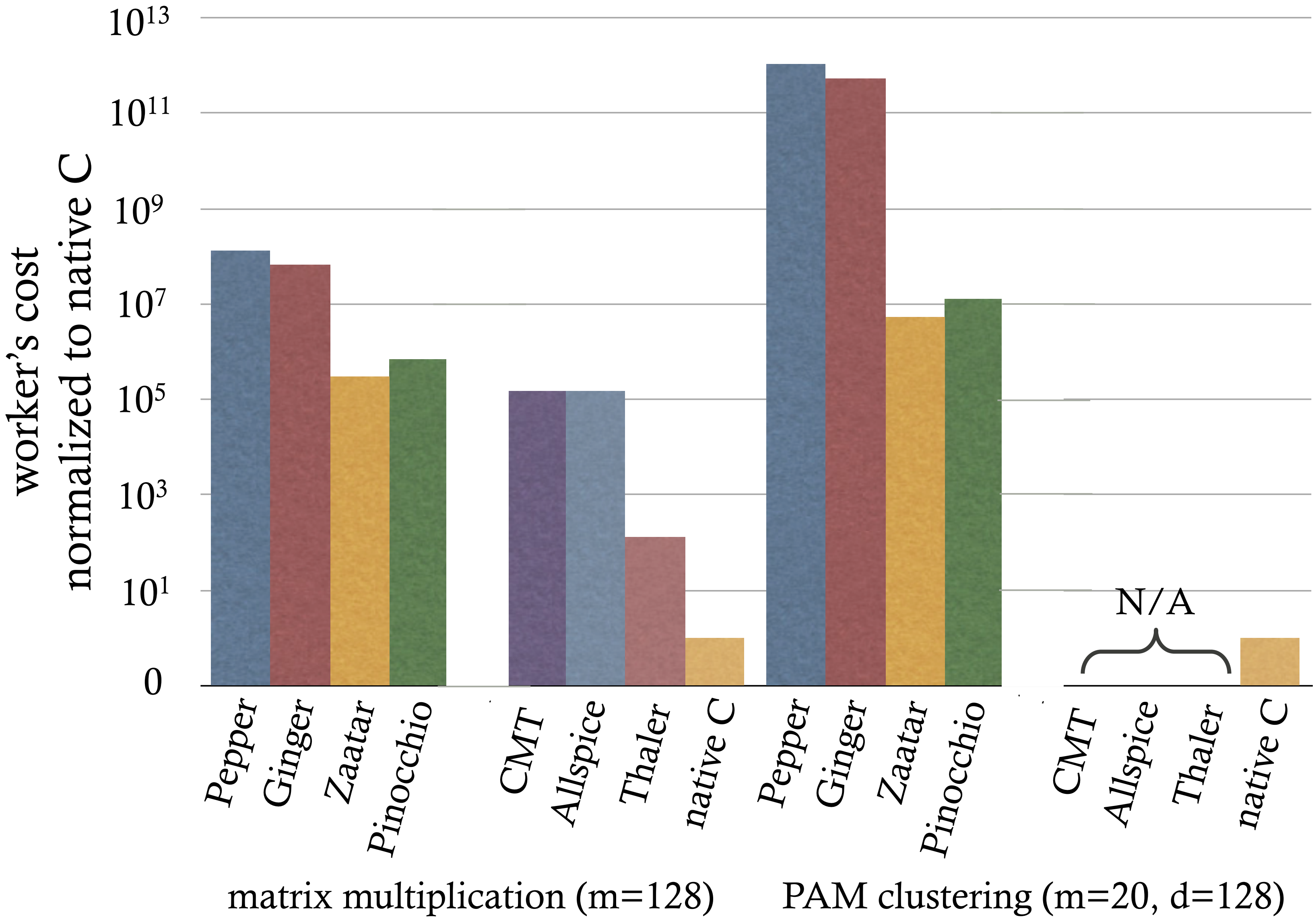
Verification cost sometimes beats (unoptimized) native execution.





Ginger (slope: 14 ms/inst)
cross-over point: 980k instances
Pinocchio (slope: 10 ms/inst)
Zaatar (slope: 26 ms/inst)
local (slope: 103 ms/inst)
CMT (slope: 36 ms/inst)
Allspice (slope: 35 ms/inst)
Thaler (slope: 12 ms/inst)
1 day
12
9
6
3
0
0
2k
4k
6k
8k
verification cost
(minutes of CPU time)
number of instances

The prover's costs are rather high.



Amortization comparison (of built systems)

Systems [CMT12, VSBW13, Thaler13] derived from [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 [BCTV14a, CTV15]
   - Constant preprocessing; amortize over all computations (but concrete costs to prover are extremely high).
2. BCTV [BCTV14b]: "CPU" front-end + non-interactive GGPR back-end
   - Amortize over all future computations of the same length
3. Pinocchio [PGHR13]: "ASIC" front-end + non-interactive GGPR back-end
   - Amortize over all future uses of a given computation
4. Zaatar [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

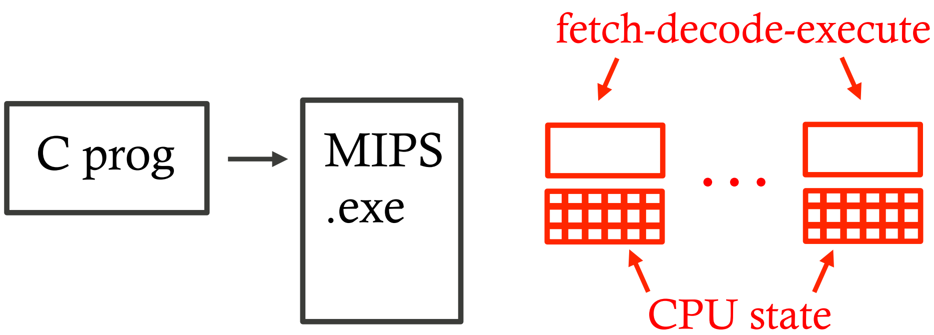
Performance is plausibly acceptable in certain settings …

- 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

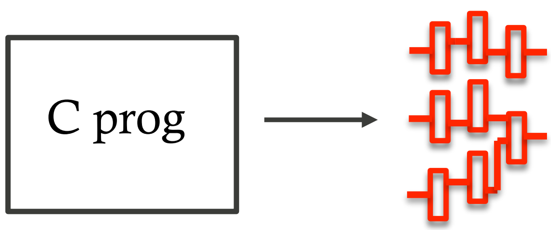


circuit is unrolled CPU execution

[BCGTV13, BCTV14a, BCTV14b, CTV15]

"CPU"

- Verbose (costly)
- Good amortization
- Great programmability



each line translates to gates/constraints

[SVPBBW12, SBVBPW13, VSBW13, PGHR13, BFRSBW13, BCGGMTV14, BBFR14, FL14, KPPSST14, WSRBW15, CFHKKNPZ15]

"ASIC"

- Concise
- Amortization worse
- Programmability not bad

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

[AB09] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[ALMSS92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998. Prelim. version FOCS 1992.

[AS92] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *Journal of the ACM*, 45(1):70–122, January 1998. Prelim. version FOCS 1992.

[Babai85] L. Babai. Trading group theory for randomness. In *STOC*, May 1985.

[BBFR15] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In *IEEE Symposium on Security and Privacy*, May 2015.

[BCC86] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and Systems Sciences*, 37(2):156–189, October 1988. Prelim. versions: several papers in CRYPTO and FOCS 1986.

[BCCGLRT14] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer. The hunting of the SNARK. Cryptology ePrint Archive, Report 2014/580. 2014.

[BCCT12] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, January 2012.

[BCCT13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC*, June 2013.

[BCGGMTV14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, May 2014.

[BCGT13] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. In *STOC*, June 2013.

[BCGTV13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, August 2013.

[BCHKS96] M. Bellare, D. Coppersmith, J. Håstad, M. Kiwi, and M. Sudan. Linearity testing in characteristic two. *IEEE transactions on information theory*, 42(6):1781–1795, November 1996.

[BCIOP13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *IACR TCC*, March 2013.

[BCTV14a] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, August 2014.

[BCTV14b] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security Symposium*, August 2014.

[BEGKN91] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, October 1991.

[BF11] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *Eurocrypt*, May 2011, pages 149–168.

[BFLS91] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, May 1991.

[BFR13] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM CCS*, November 2013.

[BFRSBW13] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, November 2013.

[BG02] B. Barak and O. Goldreich. Universal arguments and their applications. *SIAM Journal on Computing*, 38(5):1661–1694, 2008. Prelim. version CCC 2002.

[BGHSV05] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Short PCPs verifiable in polylogarithmic time. In *Conference on Computational Complexity (CCC)*, 2005.

[BGHSV06] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. *SIAM Journal on Computing*, 36(4):889–974, December 2006.

[BGLR93] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximations. In *STOC*, 1993.

[BGV11] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, August 2011, pages 111–131.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and Systems Sciences*, 47(3):549–595, December 1993. Prelim. version STOC 1990.

[Braun12] B. Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10. December 2012.

[BS08] E. Ben-Sasson and M. Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, May 2008.

[BZF11] M. Blanton, Y. Zhang, and K. Frikken. Secure and verifiable outsourcing of large-scale biometric computations. In *IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*, October 2011.

[CFHKKNPZ15] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: versatile verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2015.

[CL99] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002. Prelim. versions OSDI 1999, OSDI 2000.

[CMT12] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, January 2012.

[CRR11] R. Canetti, B. Riva, and G. Rothblum. Practical delegation of computation using multiple servers. In *ACM CCS*, October 2011.

[CT10] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, 2010.

[CTV15] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *Eurocrypt*, April 2015, pages 371–403.

[DFKP13] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In *Workshop on Language Support for Privacy-enhancing Technologies*, November 2013.

[Din07] I. Dinur. The PCP theorem by gap amplification. *Journal of the ACM*, 54(3), June 2007.

[FG12] D. Fiore and R. Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *ACM CCS*, May 2012, pages 501–512.

[FGLSS91] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43(2):268–292, March 1996. Prelim. version FOCS 1991.

[FGP14] D. Fiore, R. Gennaro, and V. Pastro. Efficiently verifiable computation on encrypted data. In *ACM CCS*, 2014.

[FL14] M. Fredrikson and B. Livshits. ZØ: an optimizing distributing zero-knowledge compiler. In *USENIX Security Symposium*, August 2014.

[Freivalds77] R. Freivalds. Probabilistic machines can use less running time. In *Proceedings of the IFIP Congress*, 1977, pages 839–842.

[GF95] A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *ACM PPoPP*, July 1995.

[GGP10] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *CRYPTO*, August 2010.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Eurocrypt*, 2013.

[GKR08] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, August 2015. Prelim. version STOC 2008.

[GLR11] S. Goldwasser, H. Lin, and A. Rubinstein. Delegation of computation without rejection problem from designated verifier CS-proofs. Cryptology ePrint Archive, Report 2011/456. 2011.

[GM01] P. Golle and I. Mironov. Uncheatable distributed computations. In *RSA Conference*, April 2001, pages 425–440.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. Prelim. version STOC 1985.

[Goldreich07] O. Goldreich. Probabilistic proof systems – a primer. *Foundations and trends in theoretical computer science*, 3(1):1–91, 2007.

[GOS06] J. Groth, R. Ostrovsky, and A. Sahai. New techniques for noninteractive zero-knowledge. *Journal of the ACM*, 59(3):11:1–11:35, June 2012. Prelim. versions CRYPTO 2006, Eurocrypt 2006.

[Groth10] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Asiacrypt*, 2010.

[GW11] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, June 2011.

[HKD07] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. In *SOSP*, October 2007, pages 175–188.

[IKO07] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.

[Kilian92] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, May 1992.

[Knijnenburg98] P. M. W. Knijnenburg. Flattening: VLIW code generation for imperfectly nested loops. In *CPC98*, June 1998.

[KNP05] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. Enhanced loop coalescing: a compiler technique for transforming non-uniform iteration spaces. In *ISHPC05/ALPS06*, September 2005.

[KP15] Y. T. Kalai and O. Paneth. Delegating RAM computations. Cryptology ePrint Archive, Report 2015/957. 2015.

[KPPSST14] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: faster verifiable set computations. In *USENIX Security Symposium*, August 2014.

[KR09] Y. T. Kalai and R. Raz. Probabilistically checkable arguments. In *CRYPTO*, 2009.

[KRR14] Y. T. Kalai, R. Raz, and R. Rothblum. How to delegate computations: the power of no-signaling proofs. In *STOC*, 2014.

[KSC09] G. O. Karame, M. Strasser, and S. Čapkun. Secure remote execution of sequential computations. In *International Conference on Information and Communications Security*, December 2009.

[KZMQCPPsS15] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. How to use SNARKs in universally composable protocols. Cryptology ePrint Archive, Report 2015/1093. 2015.

[Lipmaa11] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *IACR TCC*, 2011.

[Meir12] O. Meir. Combinatorial PCPs with short proofs. In *Conference on Computational Complexity (CCC)*, 2012.

[Merkle87] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, August 1987.

[Micali94] S. Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Prelim. version FOCS 1994.

[MR97] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998. Prelim. version STOC 1997.

[MSG07] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[MWR99] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *Network and Distributed System Security Symposium (NDSS)*, February 1999.

[PGHR13] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[PMP11] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping trust in modern computers*. Springer, 2011.

[Polychron87] C. D. Polychronopoulos. Loop coalescing: a compiler transformation for parallel machines. In *ICPP*, August 1987.

[SBVBPW13] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Eurosys*, April 2013.

[SBW11] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.

[Sion05] R. Sion. Query execution assurance for outsourced databases. In *International Conference on Very Large Databases (VLDB)*, August 2005, pages 601–612.

[SLSPDK05] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, October 2005.

[SMBW12] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Network and Distributed System Security Symposium (NDSS)*, February 2012.

[SSW10] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, June 2010.

[SVPBBW12] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, August 2012.

[Thaler13] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, August 2013.

[TRMP12] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.

[VSBW13] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[WB15] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2):74–84, February 2015.

[WHGsW15] R. Wahby, M. Howald, S. Garg, abhi shelat, and M. Walfish. Verifiable ASICs. Preprint. December 2015.

[WSRBW15] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium (NDSS)*, February 2015.

[YCFVEEGH08] B. Ylvisaker, A. Carroll, S. Friedman, B. Van Essen, C. Ebeling, D. Grossman, and S. Huack. Macah: a "C-level" language for programming kernels on coprocessor accelerators. Technical report. University of Washington, Department of CSE, 2008.