

Efficient RAM and control flow in verifiable outsourced computation

Riad S. Wahby^{*}, Srinath Setty^{†‡}, Zuocheng Ren[†],
Andrew J. Blumberg[†], and Michael Walfish^{*}

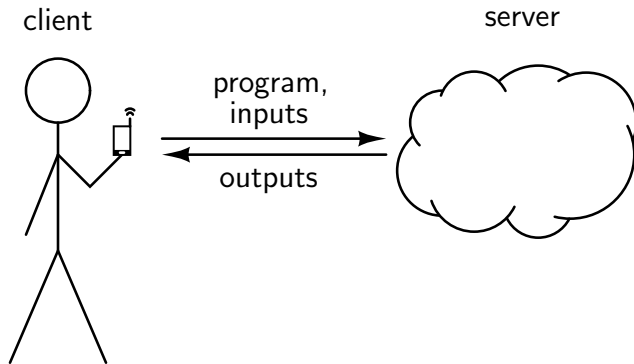
^{*}New York University

[†]The University of Texas at Austin

[‡]now at Microsoft Research

February 10, 2015

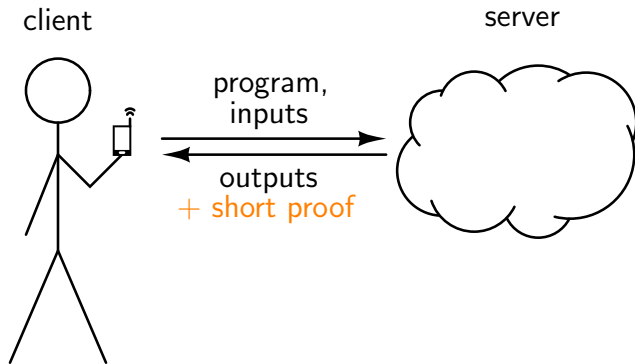
Proof-based verifiable computation enables outsourcing



Goal: A client wants to outsource a computation

- with strong correctness guarantees, and
- without assumptions about the server's hardware or how failures might occur.

Proof-based verifiable computation enables outsourcing



Approach: Server's response includes **short proof** of correctness.

This solution is based on powerful theoretical tools.

[GMR85, BCC88, BFLS91, ALMSS92, AS92, Kilian92, LFKN92, Shamir92, Micali00, BS05, BGHSV06, IKO07, GKR08]

Related work in proof-based verification


setup costs	applicable computations				
	regular structure	straight line	pure	stateful	general control flow
none	Thaler, CMT, TRMP [CRYPTO13, ITCS12, HotCloud12]				
low	Allspice [IEEE S&P13]				
med	Pepper [NDSS12]	Ginger [Security12]	Zaatar [Eurosys13], Pinocchio [IEEE S&P13]	Pantry [SOSP13]	
high					BCTV, BCGTV [Security14, CRYPTO13]

Related work in proof-based verification

setup costs	applicable computations				general control flow
	regular structure	straight line	pure	stateful	
none	Thaler, CMT, TRMP [CRYPTO13, ITCS12, HotCloud12]				
low		Allspice [IEEE S&P13]			
med	Pepper [NDSS12]	Ginger [Security12]	Zaatar [Eurosys13], Pinocchio [IEEE S&P13]	Pantry [SOSP13]	
high					BCTV, BCGTV [Security14, CRYPTO13]



Related work in proof-based verification

setup costs	applicable computations				
	regular structure	straight line	pure	stateful	general control flow
none	Thaler, CMT, TRMP [CRYPTO13, ITCS12, HotCloud12]				
low	Allspice [IEEE S&P13]				
med	Pepper [NDSS12]	Ginger [Security12]	Zaatar [Eurosys13], Pinocchio [IEEE S&P13]		
high				BCTV, BCGTV [Security14, CRYPTO13]	

Related work in proof-based verification

applicable computations

setup costs	applicable computations				
	regular structure	straight line	pure	stateful	general control flow
none	Thaler, CMT, TRMP [CRYPTO13, ITCS12, HotCloud12]				
low	Allspice [IEEE S&P13]				
med	Pepper [NDSS12]	Ginger [Security12]	Zaatar [Eurosys13], Pinocchio [IEEE S&P13]	Pantry [SOSP13]	Buffet (this work)
high				BCTV, BCGTV [Security14, CRYPTO13]	



Verifiable computation still faces challenges

Buffet

(this work)

Tension between expressiveness and efficiency

Substantially
mitigated

Large (amortized) setup costs for the client;
massive server overhead

Not addressed

The rest of this talk

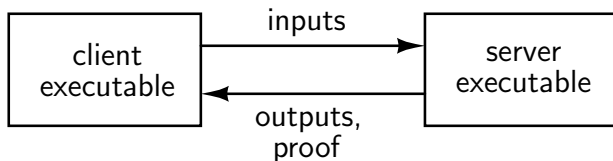
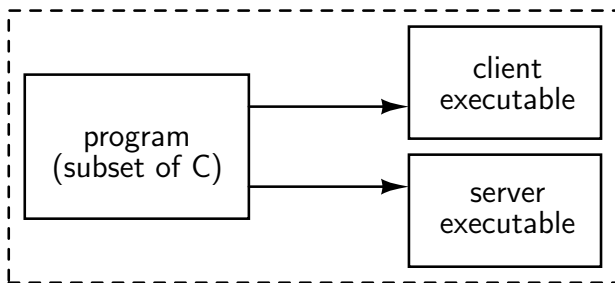
1. Background: the proof-based verification framework
2. Buffet: dynamic control flow in arithmetic circuits
3. Experimental results

The rest of this talk

1. Background: the proof-based verification framework
2. Buffet: dynamic control flow in arithmetic circuits
3. Experimental results

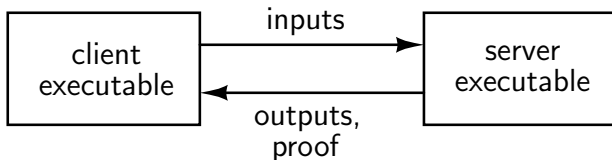
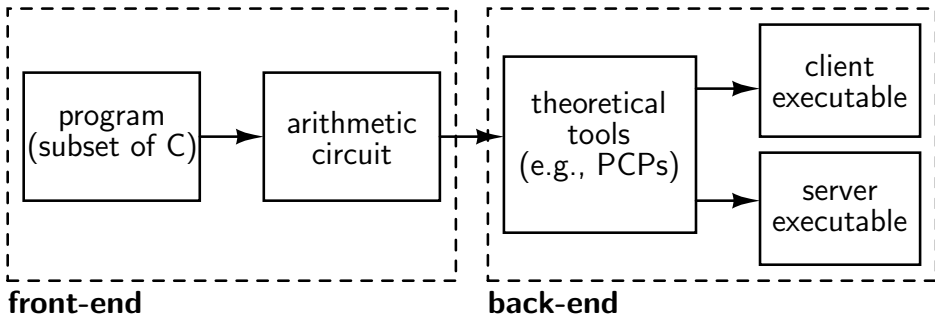
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.



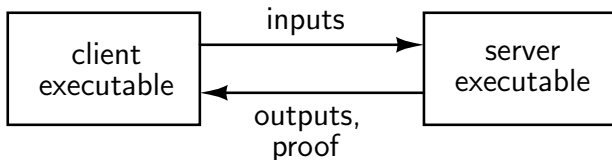
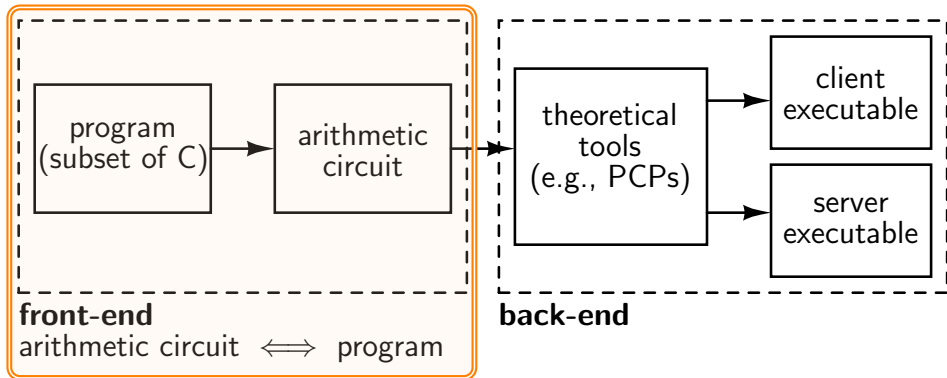
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.



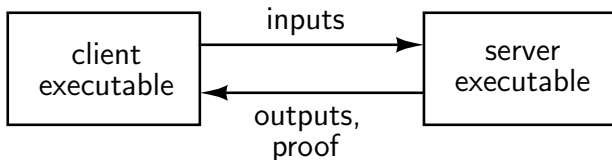
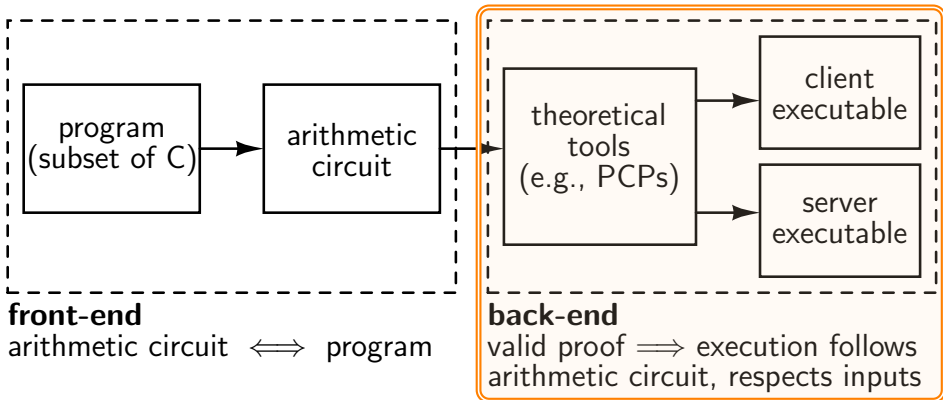
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.



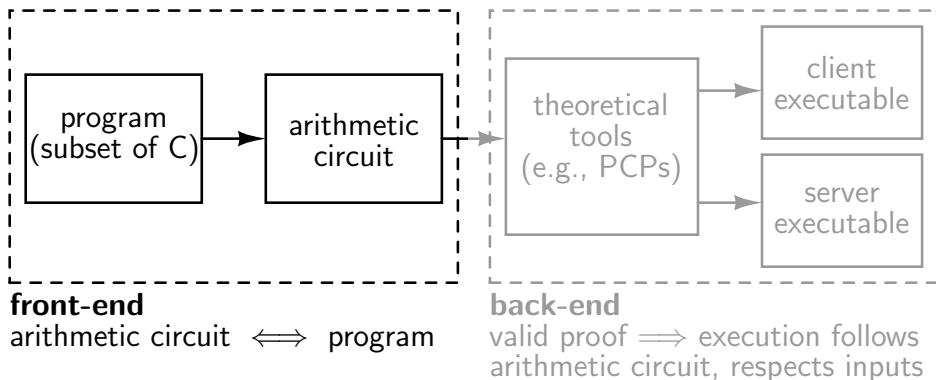
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.



Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.



Costs scale with arithmetic circuit size. So:

How can Buffet's front-end efficiently represent general-purpose C programs in arithmetic circuits?

Compiling programs to circuits in Pantry [SOSP13] (and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.

```
i = i + 1;
```

\implies

```
i1 = i0 + 1;
```


Compiling programs to circuits in Pantry [SOSP13] (and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.

```
if (i > 5)
    i = i + 1;
else
    i = i * 2;
```



```
i1 = i0 + 1;
i2 = i0 * 2;
i3 = (i0 > 5) ?
      i1 : i2;
```

Compiling programs to circuits in Pantry [SOSP13] (and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be **static**.

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



```
i = 0;
i0=i+1; // j == 0
i1=i0+1; // j == 1
...
i9=i8+1; // j == 9
```

Compiling programs to circuits in Pantry [SOSP13] (and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be **static**.
4. Arithmetic, inequalities, and logical operations are supported.

Compiling programs to circuits in Pantry [SOSP13] (and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be **static**.
4. Arithmetic, inequalities, and logical operations are supported.

Buffet's **key challenge**: how can we support *general* C programs with arbitrary control flow, including break, continue, and data dependent looping?

Compiling programs to circuits in Pantry [SOSP13] (and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be **static**.
4. Arithmetic, inequalities, and logical operations are supported.

Buffet's **key challenge**: how can we support *general* C programs with arbitrary control flow, including break, continue, and data dependent looping?

Buffet also adapts and refines a previous approach to verified RAM [BCGT12, BCGTV13, BCTV14] (see paper).

The rest of this talk

1. Background: the proof-based verification framework
2. Buffet: dynamic control flow in arithmetic circuits
3. Experimental results

Compiling nested loops

In a loop nest, 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;
  }
}
```

⇒

```
i = 0;
i0=i+1; // j == 0
i1=i0*2; // k == 0
i2=i1*2; // k == 1
i3=i2+1; // j == 1
i4=i3*2; // k == 0
i5=i4*2; // k == 1
...
```

Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” \Rightarrow “aaaaabb”

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

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


Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” \Rightarrow “aaaaabb”

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

1

1. Read (inchar,length) pair.

Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” \Rightarrow “aaaaabb”

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

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

Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” ⇒ “aaaaabb”

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

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

At one extreme, a single character's run length could be OUTLENGTH. so this must be the inner bound.

Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” \Rightarrow “aaaaabb”

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

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

/* bound=OUTLENGTH */

Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” \Rightarrow “aaaaabb”

```
i = j = 0;
while (j < OUTLENGTH) {           /* bound= ???      */
    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.

Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“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);
}
```

But: this code executes OUTLENGTH^2 inner loop iterations, and the resulting arithmetic circuit is quadratic in OUTLENGTH.

We can't eliminate unrolling. What about nesting?

We can't eliminate unrolling. What about nesting?

Consider:

1. Loop nests are equivalent to finite state machines.
2. Arithmetic circuits can efficiently represent FSMs.

We can't eliminate unrolling. What about nesting?

Consider:

1. Loop nests are equivalent to finite state machines.
2. Arithmetic circuits can efficiently represent FSMs.

Idea: transform loop nests into FSMs.

FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

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

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

FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

①

```
i = j = 0;  
while (j < OUTLENGTH) {
```

```
    inchar = input[i++];  
    length = input[i++];
```

②

```
    do {
```

```
        output[j++] = inchar;  
        length--;
```

```
    } while (length > 0);
```

```
}
```

1. Identify vertices: straight line code segments.

FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:



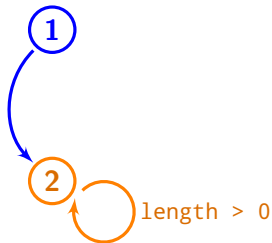
```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

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

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

FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:



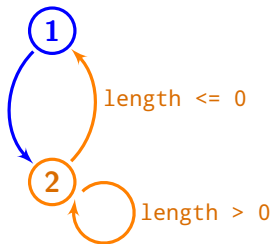
```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

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

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

FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

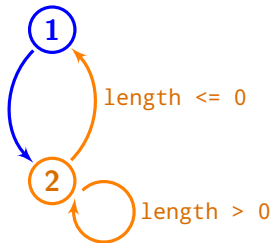


```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];
    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

1. Identify vertices: straight line code segments.
2. Identify edges: control flow between segments.
 - 1 transitions to 2 unconditionally.
 - 2 self-transitions when $\text{length} > 0$.
 - 2 transitions to 1 when $\text{length} \leq 0$.

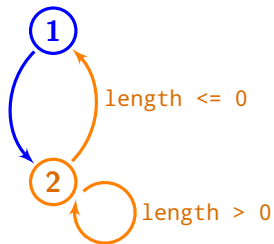
FSM Transformation: step 2

From the control flow graph



FSM Transformation: step 2

From the control flow graph, we can build a 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;
    }
  }
}
```


FSM Transformation: step 2

From the control flow graph, we can build a 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;
```

```
        }
```

```
    }
```

```
}
```

Buffet's FSM transformation: *loop flattening*

Buffet's transformation extends *loop flattening* [Ghuloum & Fisher, PPOPP95] with support for arbitrary loops, break, and continue.

Buffet's FSM transformation: *loop flattening*

Buffet's transformation extends *loop flattening* [Ghuloum & Fisher, PPOPP95] with support for arbitrary loops, break, and continue.

Caveats:

- Programmer must tell Buffet # of steps to unroll the FSM.
- No goto in Buffet's implementation (yet).
- No "program memory" \Rightarrow no function pointers.

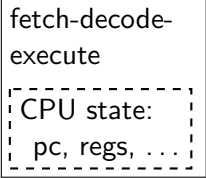
What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter.
What if we just had a program counter, registers, etc?

What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter.
What if we just had a program counter, registers, etc?

This is the approach of BCTV:
Represent a CPU transition



A rectangular box with a solid border. Inside the box, the text "fetch-decode-
execute" is written in two lines. Below this text is a smaller dashed rectangular box containing the text "CPU state:
pc, regs, ...".

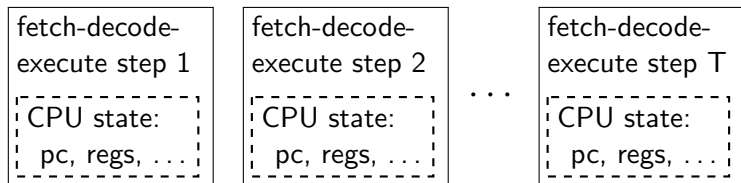
fetch-decode-
execute

CPU state:
pc, regs, ...

What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter.
What if we just had a program counter, registers, etc?

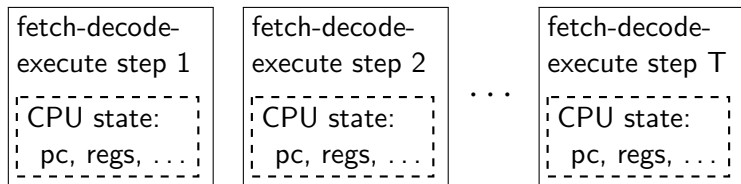
This is the approach of BCTV:
Represent a CPU transition, and unroll it.



What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter.
What if we just had a program counter, registers, etc?

This is the approach of BCTV:
Represent a CPU transition, and unroll it.

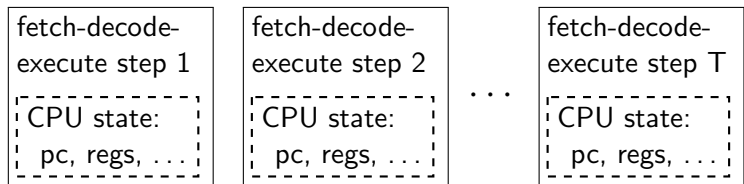


BCTV supports all of C, but like other systems requires bounded execution (programmer chooses # of CPU steps).

What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter.
What if we just had a program counter, registers, etc?

This is the approach of BCTV:
Represent a CPU transition, and unroll it.



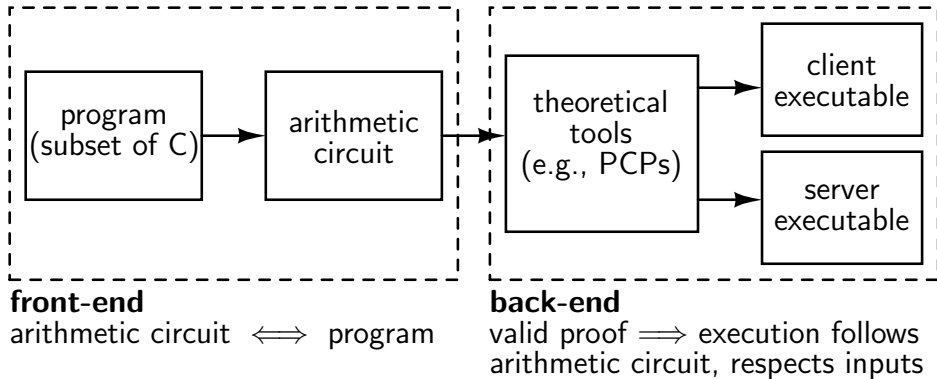
BCTV supports all of C, but like other systems requires bounded execution (programmer chooses # of CPU steps).

But: BCTV pays the cost of an entire CPU for each program step.

The rest of this talk

1. Background: the proof-based verification framework
2. Buffet: dynamic control flow in arithmetic circuits
3. Experimental results

Evaluation questions



Evaluation questions

Using the same back-end for Pantry, BCTV, and Buffet, how do the front-ends compare?

1. For a fixed arithmetic circuit size, what is the maximum computation size each system can handle?

Evaluation questions

Using the same back-end for Pantry, BCTV, and Buffet, how do the front-ends compare?

1. For a fixed arithmetic circuit size, what is the maximum computation size each system can handle?
2. For a fixed computation size, what is the server's cost under each system?

Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSP13].

FSM transform: source-to-source compiler built on top of clang.

Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSPP13].

FSM transform: source-to-source compiler built on top of clang.

For evaluation, we reimplemented the BCTV system, including

- a toolchain for the simulated CPU in Java and C
- a CPU simulator in C, compiled using Pantry

Our implementation's performance is within 15% of BCTV.

Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSP13].

FSM transform: source-to-source compiler built on top of clang.

For evaluation, we reimplemented the BCTV system, including

- a toolchain for the simulated CPU in Java and C
- a CPU simulator in C, compiled using Pantry

Our implementation's performance is within 15% of BCTV.

We use the Pinocchio back-end [Parno et al., IEEE S&P13].

(Highly optimized implementation from BCTV [Security14].)

Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSP13].

FSM transform: source-to-source compiler built on top of clang.

For evaluation, we reimplemented the BCTV system, including

- a toolchain for the simulated CPU in Java and C
- a CPU simulator in C, compiled using Pantry

Our implementation's performance is within 15% of BCTV.

We use the Pinocchio back-end [Parno et al., IEEE S&P13].

(Highly optimized implementation from BCTV [Security14].)

Evaluation platform:

- Texas Advanced Computing Center (TACC), Stampede cluster
- Linux machines with Intel Xeon E5-2680, 32 GB of RAM

What is the maximum computation size for each system?

For an arithmetic circuit of $\approx 10^7$ gates, we have:

	Pantry	BCTV	Buffet
matrix multiplication $m \times m$	$m = 215$	$m = 7$	$m = 215$
merge sort k elements	$k = 8$	$k = 32$	$k = 512$
Knuth-Morris-Pratt search needle length = n haystack length = ℓ	$n = 4,$ $\ell = 8$	$n = 16,$ $\ell = 160$	$n = 256,$ $\ell = 2900$

What is the maximum computation size for each system?

For an arithmetic circuit of $\approx 10^7$ gates, we have:

	Pantry	BCTV	Buffet
matrix multiplication $m \times m$	$m = 215$	$m = 7$	$m = 215$
merge sort k elements	$k = 8$	$k = 32$	$k = 512$
Knuth-Morris-Pratt search needle length = n haystack length = ℓ	$n = 4,$ $\ell = 8$	$n = 16,$ $\ell = 160$	$n = 256,$ $\ell = 2900$

What is the maximum computation size for each system?

For an arithmetic circuit of $\approx 10^7$ gates, we have:

	Pantry	BCTV	Buffet
matrix multiplication $m \times m$	$m = 215$	$m = 7$	$m = 215$
merge sort k elements	$k = 8$	$k = 32$	$k = 512$
Knuth-Morris-Pratt search needle length = n haystack length = ℓ	$n = 4,$ $\ell = 8$	$n = 16,$ $\ell = 160$	$n = 256,$ $\ell = 2900$

What is the maximum computation size for each system?

For an arithmetic circuit of $\approx 10^7$ gates, we have:

	Pantry	BCTV	Buffet
matrix multiplication $m \times m$	$m = 215$	$m = 7$	$m = 215$
merge sort k elements	$k = 8$	$k = 32$	$k = 512$
Knuth-Morris-Pratt search needle length = n haystack length = ℓ	$n = 4,$ $\ell = 8$	$n = 16,$ $\ell = 160$	$n = 256,$ $\ell = 2900$

What is the maximum computation size for each system?

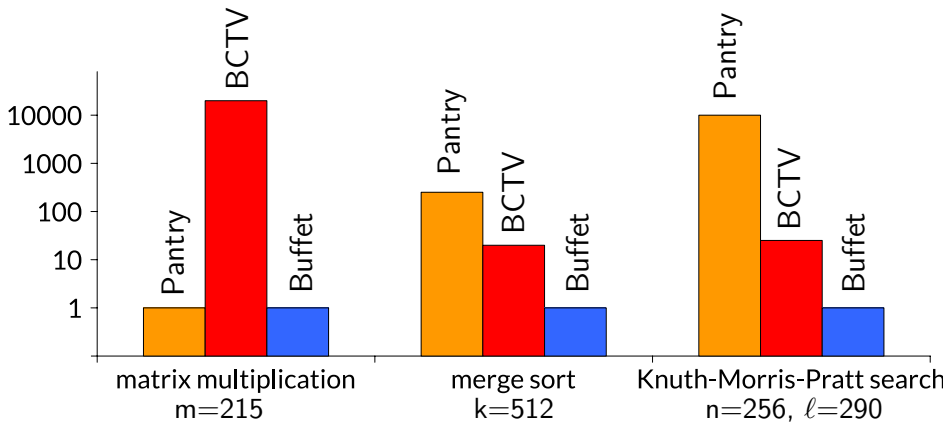
For an arithmetic circuit of $\approx 10^7$ gates, we have:

	Pantry	BCTV	Buffet
matrix multiplication $m \times m$	$m = 215$	$m = 7$	$m = 215$
merge sort k elements	$k = 8$	$k = 32$	$k = 512$
Knuth-Morris-Pratt search needle length = n haystack length = ℓ	$n = 4,$ $\ell = 8$	$n = 16,$ $\ell = 160$	$n = 256,$ $\ell = 2900$

These data establish ground truth. For apples-to-apples front-end comparison, we now extrapolate to Buffet's computation sizes.

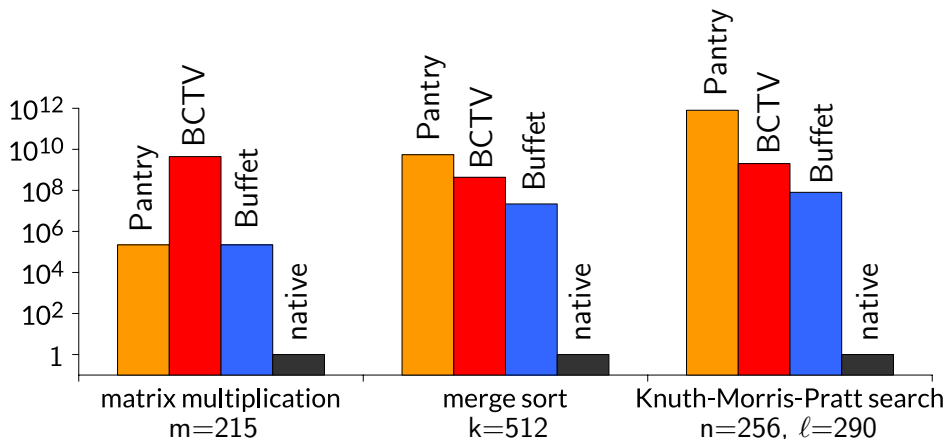
What is the server's cost for each system?

Extrapolated server execution time, normalized to Buffet



But we still have a long way to go!

Extrapolated server execution time, normalized to native execution



Recap

Buffet combines the **best aspects** of Pantry and BCTV.

- + Straight line computations are very efficient.
- + Buffet charges the programmer only for what is used.
- + General looping is transformed into FSM, efficiently compiled.
- + RAM interactions are efficient (see paper).

Buffet improves on Pantry and BCTV by 1–4 orders of magnitude.

Recap

Buffet combines the **best aspects** of Pantry and BCTV.

- + Straight line computations are very efficient.
- + Buffet charges the programmer only for what is used.
- + General looping is transformed into FSM, efficiently compiled.
- + RAM interactions are efficient (see paper).

Buffet improves on Pantry and BCTV by 1–4 orders of magnitude.

Buffet still has **limitations**:

- No support for goto or function pointers.
- Like all systems in the area, server overheads are still massive.

Recap

Buffet combines the **best aspects** of Pantry and BCTV.

- + Straight line computations are very efficient.
- + Buffet charges the programmer only for what is used.
- + General looping is transformed into FSM, efficiently compiled.
- + RAM interactions are efficient (see paper).

Buffet improves on Pantry and BCTV by 1–4 orders of magnitude.

Buffet still has **limitations**:

- No support for goto or function pointers.
- Like all systems in the area, server overheads are still massive.

<http://www.pepper-project.org/>