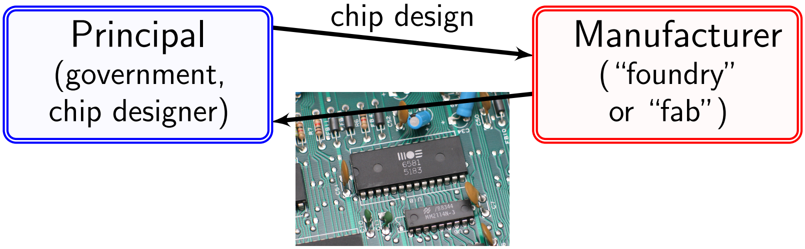
# Verifiable ASICs: trustworthy hardware with untrusted components

Riad S. Wahby[○⋆], Max Howald[†⋆],
Siddharth Garg[⋆], abhi shelat[‡], and Michael Walfish[⋆]

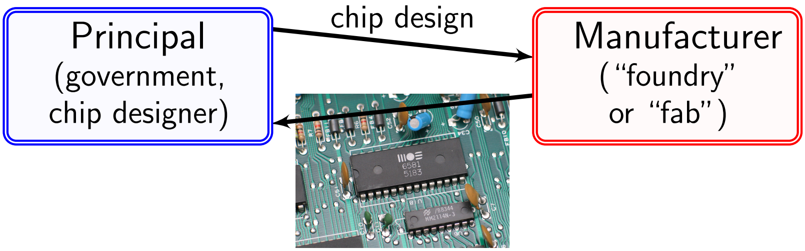
[○]Stanford University
[⋆]New York University
[†]The Cooper Union
[‡]The University of Virginia

June 10th, 2016

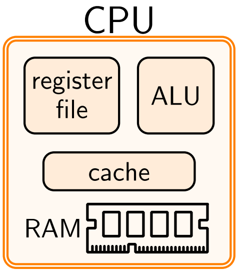
## Setting: ASICs with mutually distrusting designer, manufacturer

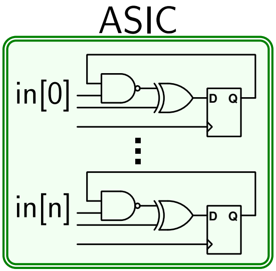


## Setting: ASICs with mutually distrusting designer, manufacturer

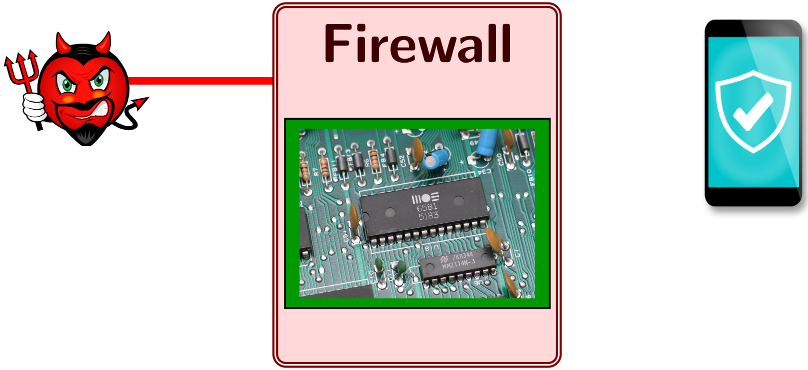


Here we are thinking about ASICs, not CPUs:



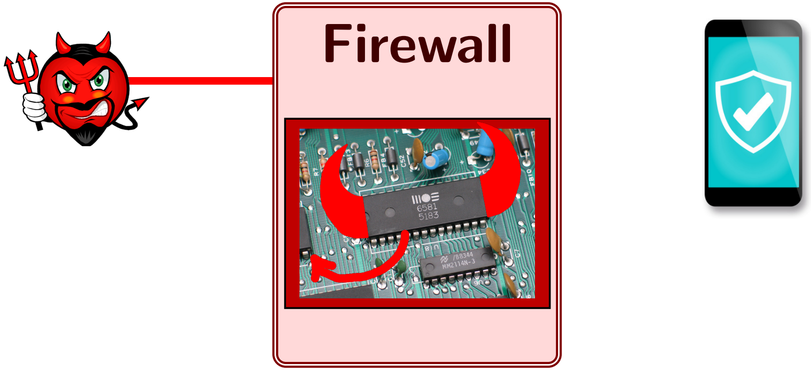


## Setting: ASICs with mutually distrusting designer, manufacturer



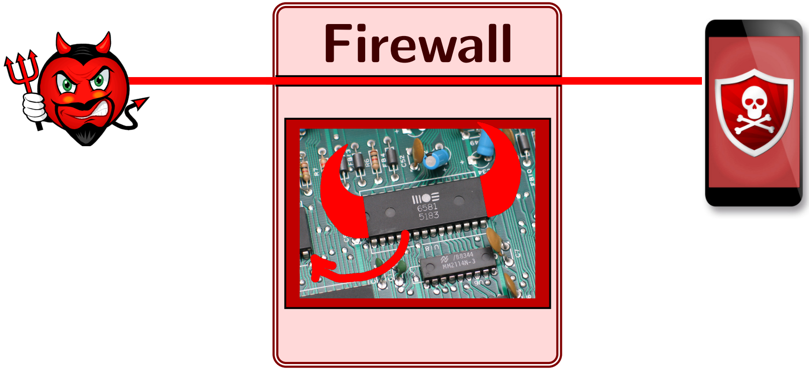
e.g., a network firewall appliance,
with a custom chip for packet processing

## Untrusted manufacturers can craft hardware Trojans



What if our packet processing chip has a **back door**?

## Untrusted manufacturers can craft hardware Trojans

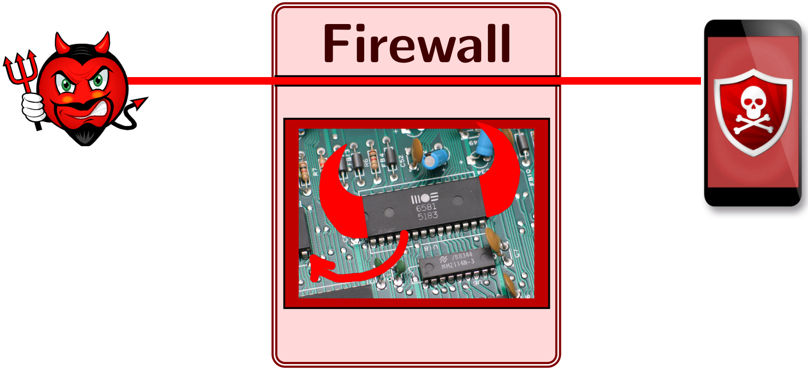


What if our packet processing chip has a **back door**?

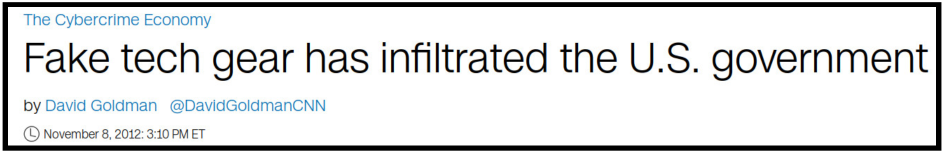
Threat: incorrect execution of the packet filter

(Other concerns, e.g., secret state, are important but orthogonal)

## Untrusted manufacturers can craft hardware Trojans



What if our packet processing chip has a **back door**?



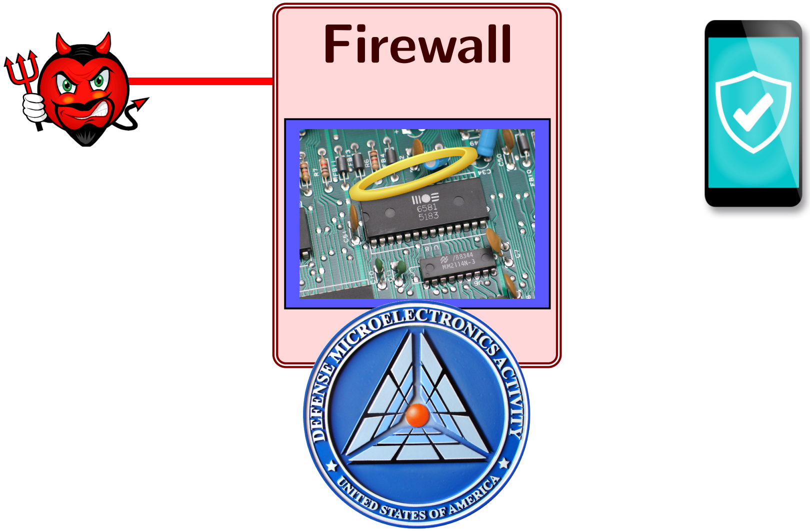
The Cybercrime Economy

Fake tech gear has infiltrated the U.S. government

by David Goldman @DavidGoldmanCNN

November 8, 2012: 3:10 PM ET

## Untrusted manufacturers can craft hardware Trojans



US DoD controls supply chain with **trusted foundries**.

## Trusted fabs are the only way to get strong guarantees

For example, stealthy trojans can thwart post-fab detection
[A2: Analog Malicious Hardware, Yang et al., IEEE S&P 2016; Stealthy Dopant-Level Trojans, Becker et al., CHES 2013]

## Trusted fabs are the only way to get strong guarantees

For example, stealthy trojans can thwart post-fab detection
[A2: Analog Malicious Hardware, Yang et al., IEEE S&P 2016; Stealthy Dopant-Level Trojans, Becker et al., CHES 2013]

But trusted fabrication is not a panacea:

- ✗ Only 5 countries have cutting-edge fabs on-shore
- ✗ Building a new fab takes $$$$$$, years of R&D

## Trusted fabs are the only way to get strong guarantees

For example, stealthy trojans can thwart post-fab detection [A2: Analog Malicious Hardware, Yang et al., IEEE S&P 2016; Stealthy Dopant-Level Trojans, Becker et al., CHES 2013]

## But trusted fabrication is not a panacea:

- ✗ Only 5 countries have cutting-edge fabs on-shore
- ✗ Building a new fab takes $$$$$$, years of R&D
- ✗ Semiconductor scaling: chip area and energy go with square and cube of transistor length ("critical dimension")
- ✗ So using an old fab means an enormous performance hit
  e.g., India's best on-shore fab is $10^8\times$ behind state of the art

## Trusted fabs are the only way to get strong guarantees

For example, stealthy trojans can thwart post-fab detection [A2: Analog Malicious Hardware, Yang et al., IEEE S&P 2016; Stealthy Dopant-Level Trojans, Becker et al., CHES 2013]

But trusted fabrication is not a panacea:

- ✗ Only 5 countries have cutting-edge fabs on-shore
- ✗ Building a new fab takes $$$$$$, years of R&D
- ✗ Semiconductor scaling: chip area and energy go with square and cube of transistor length ("critical dimension")
- ✗ So using an old fab means an enormous performance hit
  e.g., India's best on-shore fab is $10^8\times$ behind state of the art

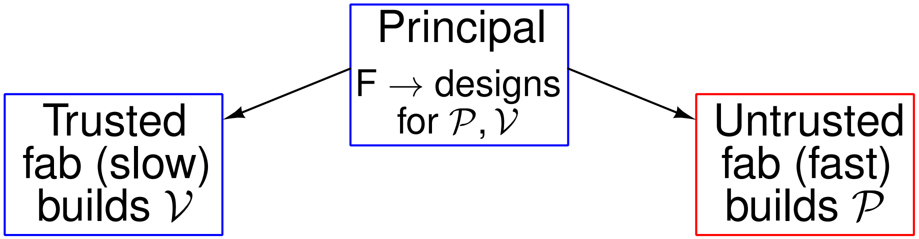
Can we get trust more cheaply?

## Verifiable ASICs

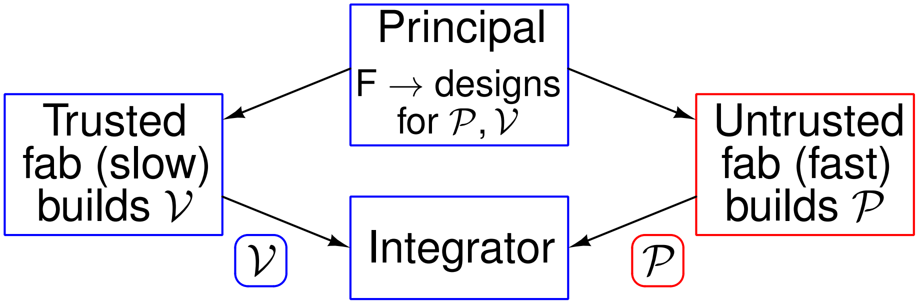
Principal

F $\rightarrow$ designs for $\mathcal{P}, \mathcal{V}$

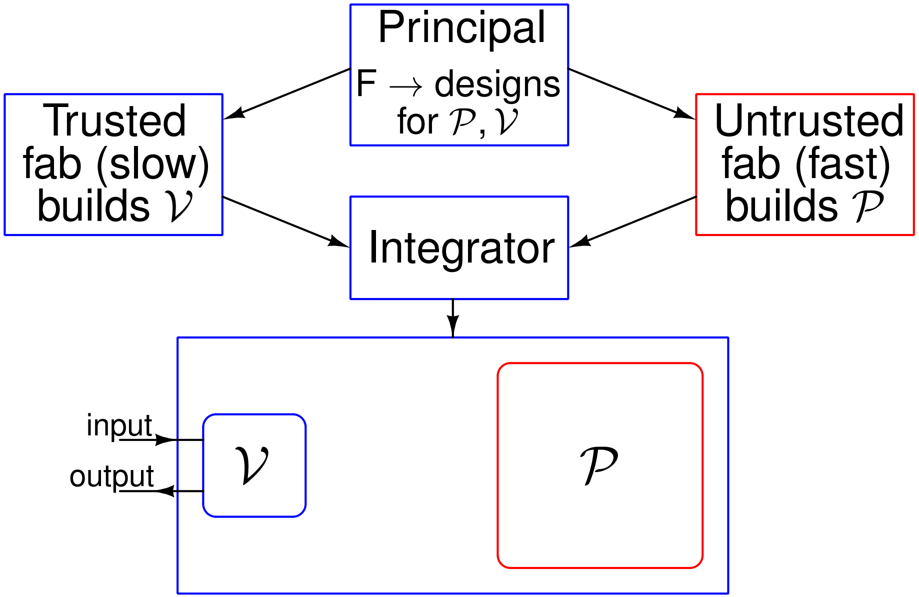
## Verifiable ASICs


Principal
F → designs for $\mathcal{P}, \mathcal{V}$
Trusted fab (slow) builds $\mathcal{V}$
Untrusted fab (fast) builds $\mathcal{P}$

## Verifiable ASICs

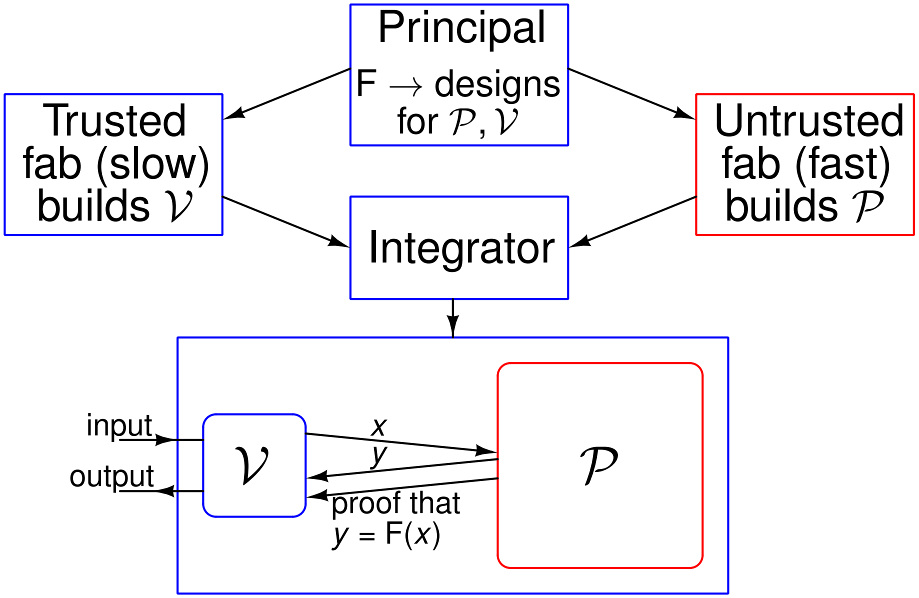

Principal
F $\rightarrow$ designs for $\mathcal{P}, \mathcal{V}$
Trusted fab (slow) builds $\mathcal{V}$
Untrusted fab (fast) builds $\mathcal{P}$
Integrator
$\mathcal{V}$
$\mathcal{P}$

## Verifiable ASICs

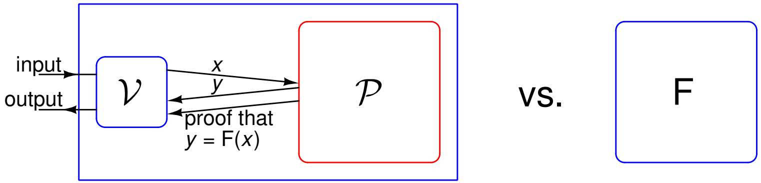


Principal
F $\rightarrow$ designs for $\mathcal{P}, \mathcal{V}$
Trusted fab (slow) builds $\mathcal{V}$
Untrusted fab (fast) builds $\mathcal{P}$
Integrator
input
output
$\mathcal{V}$
$\mathcal{P}$

## Verifiable ASICs


Principal
F $\rightarrow$ designs for $\mathcal{P}, \mathcal{V}$
Trusted fab (slow) builds $\mathcal{V}$
Untrusted fab (fast) builds $\mathcal{P}$
Integrator
input
output
$\mathcal{V}$
$x$
$y$
proof that $y = \mathrm{F}(x)$
$\mathcal{P}$

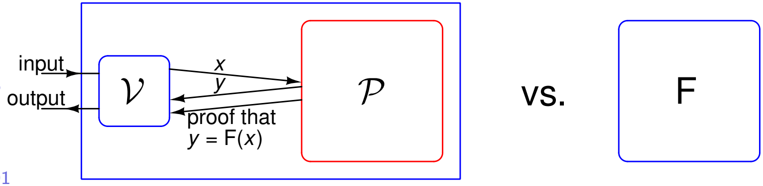
## Can we build Verifiable ASICs?



Makes sense if $\mathcal{V} + \mathcal{P}$ are cheaper than trusted F

## Can we build Verifiable ASICs?

Babai85
GMR85
BCC86
BFLS91
FGLSS91
Kilian92
ALMSS92
AS92
Micali94
BG02
GOS06
IKO07
GKR08
KR09
GGP10
Groth10
GLR11
Lipmaa11
BCCT12
GGPR13
BCCT13
KRR14
...

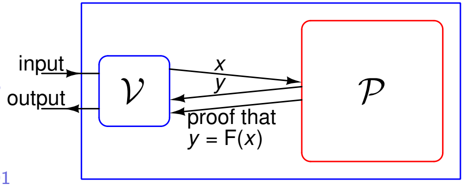


Makes sense if $\mathcal{V} + \mathcal{P}$ are cheaper than trusted F

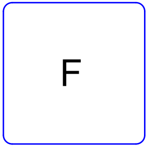
Reasons for hope:

- running time of $\mathcal{V} <$ F (asymptotically)

## Can we build Verifiable ASICs?



vs.



Makes sense if $\mathcal{V} + \mathcal{P}$ are cheaper than trusted F

Reasons for hope:

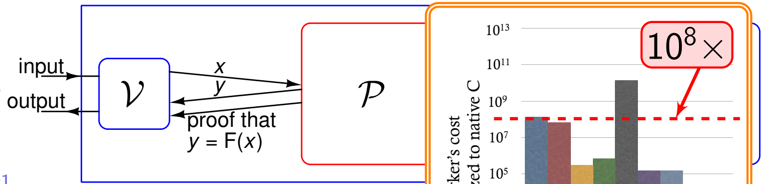
- running time of $\mathcal{V} <$ F (asymptotically)
- Implementations exist

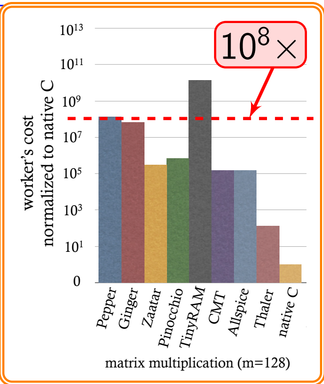
Babai85
GMR85
BCC86
BFLS91
FGLSS91
Kilian92
ALMSS92
AS92
Micali94
BG02
GOS06
IKO07
GKR08
KR09
GGP10
Groth10
GLR11
Lipmaa11
BCCT12
GGPR13
BCCT13
KRR14
...

SBW11
CMT12
SMBW12
TRMP12
SVPBBW12
SBVBPW13
VSBW13
PGHR13
Thaler13
BCGTV13
BFRSBW13
BFR13
DFKP13
BCTV14a
BCTV14b
BCGGMTV14
FL14
KPPSST14
FTP14
WSRHBW15
BBFR15
CFHKNPZ15
CTV15
KZMQCPPsS15

## Can we build Verifiable ASICs?

Babai85
GMR85
BCC86
BFLS91
FGLSS91
Kilian92
ALMSS92
AS92
Micali94
BG02
GOS06
IKO07
GKR08
KR09
GGP10
Groth10
GLR11
Lipmaa11
BCCT12
GGPR13
BCCT13
KRR14
. . .





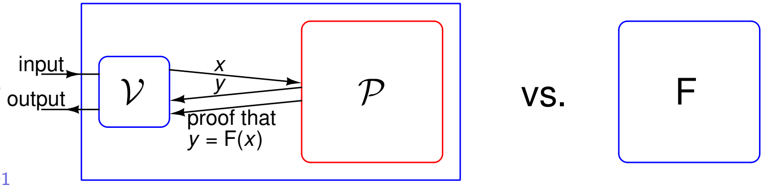
Makes sense if $\mathcal{V} + \mathcal{P}$ a
than trusted F

Reasons for hope:

- running time of $\mathcal{V} <$ F ( ymptotically)
- Implementations exist
- $\boxed{\mathcal{P}\text{ overheads are massive,}}$ but using an advanced fab might offset these costs

SBW11
CMT12
SMBW12
TRMP12
SVPBBW12
SBVBPW13
VSBW13
PGHR13
Thaler13
BCGTV13
BFRSBW13
BFR13
DFKP13
BCTV14a
BCTV14b
BCGGMTV14
FL14
KPPSST14
FTP14
WSRHBW15
BBFR15
CFHKNPZ15
CTV15
KZMQCPPsS15

## Can we build Verifiable ASICs?



Babai85
GMR85
BCC86
BFLS91
FGLSS91
Kilian92
ALMSS92
AS92
Micali94
BG02
GOS06
IKO07
GKR08
KR09
GGP10
Groth10
GLR11
Lipmaa11
BCCT12
GGPR13
BCCT13
KRR14
...

Makes sense if $\mathcal{V} + \mathcal{P}$ are cheaper than trusted F

Reasons for ~~hope~~ caution:

- Theory is silent about feasibility
- Onus is heavier than in prior work
- Hardware issues: energy, chip area
- Need physically realizable circuit design
- Need $\mathcal{V}$ to save for plausible computation sizes

SBW11
CMT12
SMBW12
TRMP12
SVPBBW12
SBVBPW13
VSBW13
PGHR13
Thaler13
BCGTV13
BFRSBW13
BFR13
DFKP13
BCTV14a
BCTV14b
BCGGMTV14
FL14
KPPSST14
FTP14
WSRHBW15
BBFR15
CFHKNPZ15
CTV15
KZMQCPPsS15

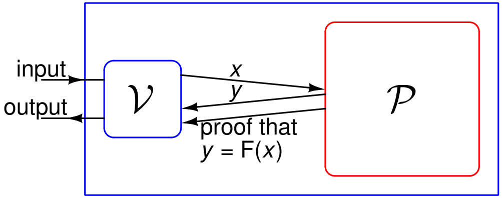
Zebra: a hardware design that saves costs

## A **qualified** success

Zebra: a hardware design that saves costs. . .

. . . sometimes.

## Probabilistic proof protocols, briefly

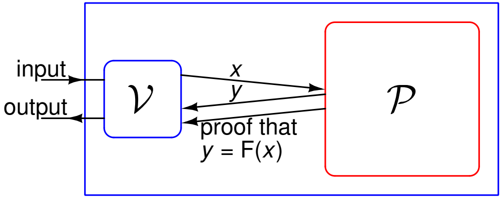


F must be expressed as an arithmetic circuit (AC)

AC satisfiable $\iff$ F was executed correctly

$\mathcal{P}$ convinces $\mathcal{V}$ that the AC is satisfiable

## Probabilistic proof protocols, briefly



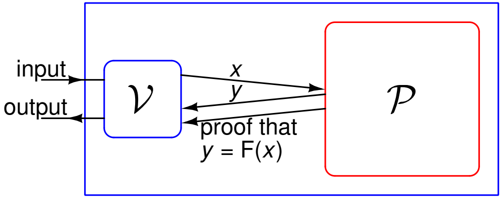
**Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14]

e.g., Zaatar, Pinocchio, libsnark

**IPs** [GKR08, CMT12, VSBW13]

e.g., Muggles, CMT, Allspice

## Probabilistic proof protocols, briefly



**Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14]

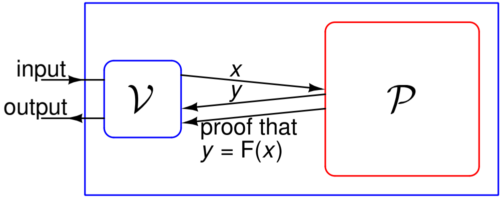
e.g., Zaatar, Pinocchio, libsnark

**IPs** [GKR08, CMT12, VSBW13]

e.g., Muggles, CMT, Allspice

What about other schemes? e.g.,
FHE [GGP10], MIP+FHE [BC12], MIP [BTWV14], PCIP [RRR16], IOP [BCS16], PIR [BHK16], . . .

## Probabilistic proof protocols, briefly



**Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14]

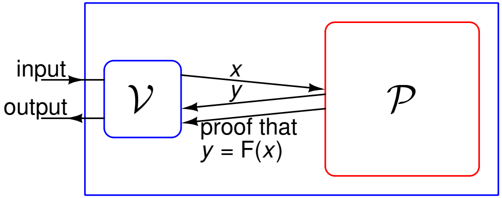
e.g., Zaatar, Pinocchio, libsnark

**IPs** [GKR08, CMT12, VSBW13]

e.g., Muggles, CMT, Allspice

What about other schemes? e.g.,
FHE [GGP10], MIP+FHE [BC12], MIP [BTWV14], PCIP [RRR16], IOP [BCS16], PIR [BHK16], . . .
These all seem a bit further from practicality.

## Probabilistic proof protocols, briefly



**Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14]

e.g., Zaatar, Pinocchio, libsnark

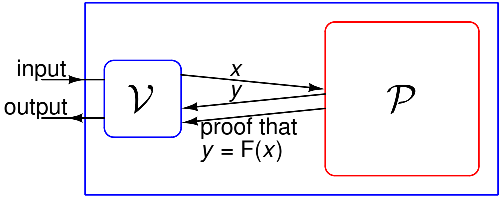
+ nondeterministic ACs, arbitrary connectivity
+ Few rounds ($\leq 3$)

**IPs** [GKR08, CMT12, VSBW13]

e.g., Muggles, CMT, Allspice

− deterministic ACs; layered, low depth
− Many rounds

## Probabilistic proof protocols, briefly



**Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14]

e.g., Zaatar, Pinocchio, libsnark

+ nondeterministic ACs, arbitrary connectivity

+ Few rounds ($\leq 3$)

**Unsuited to hardware implementation** ✗

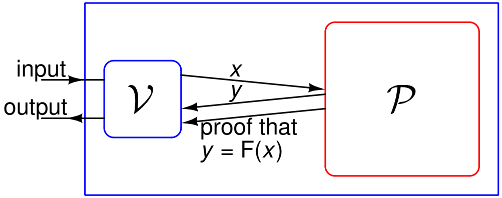
**IPs** [GKR08, CMT12, VSBW13]

e.g., Muggles, CMT, Allspice

– deterministic ACs; layered, low depth

– Many rounds

## Probabilistic proof protocols, briefly



**Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14]

e.g., Zaatar, Pinocchio, libsnark

+ nondeterministic ACs, arbitrary connectivity
+ Few rounds ($\leq 3$)

**Unsuited to hardware implementation** ✗

**IPs** [GKR08, CMT12, VSBW13]

e.g., Muggles, CMT, Allspice

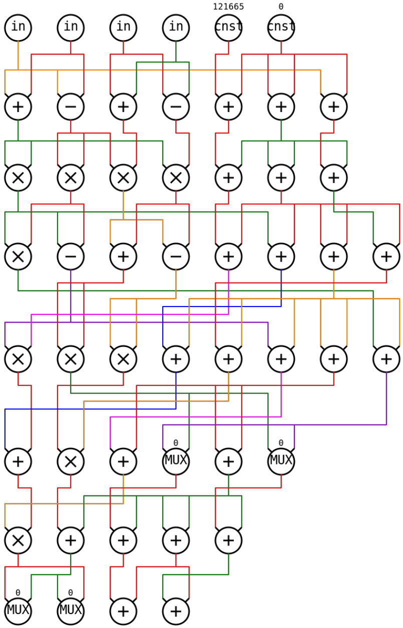
– deterministic ACs; layered, low depth

– Many rounds

**Suited to hardware implementation** ✓

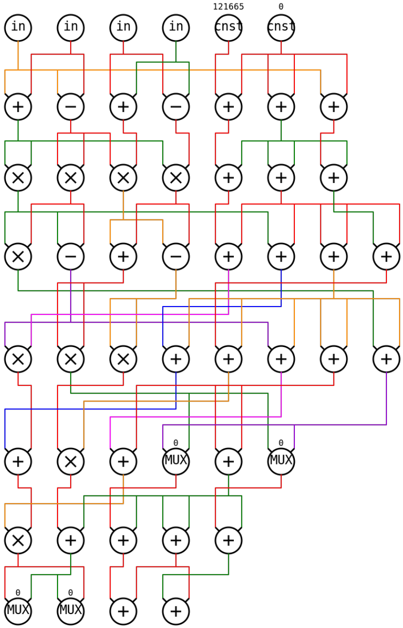
## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

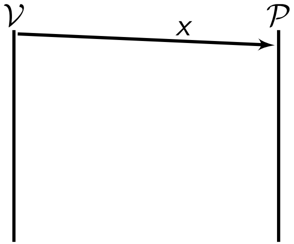
F must be expressed as a *layered* arithmetic circuit.



## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

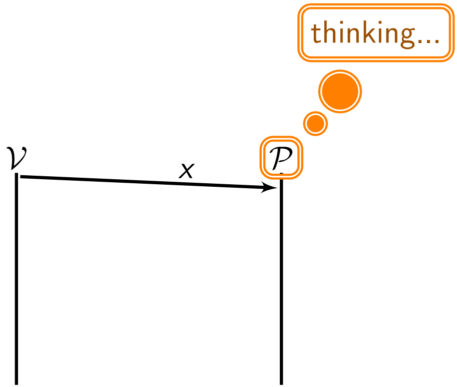
1. $\mathcal{V}$ sends inputs

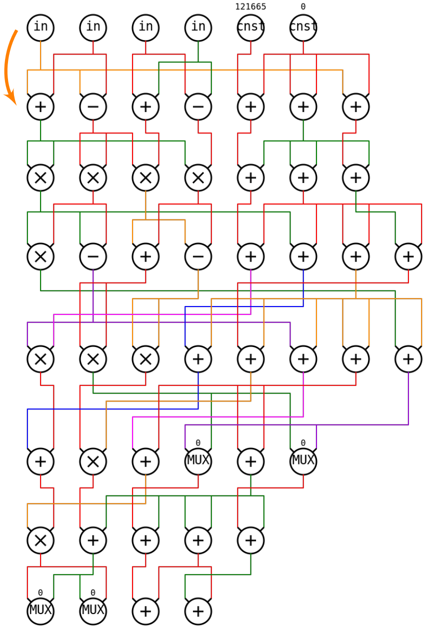




## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

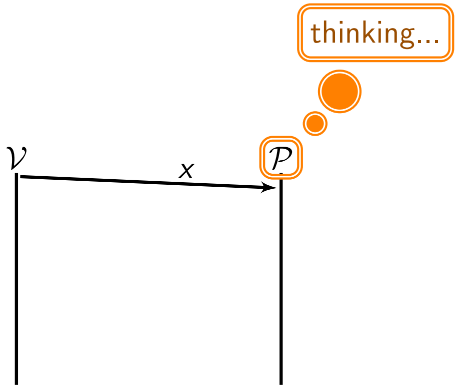
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates

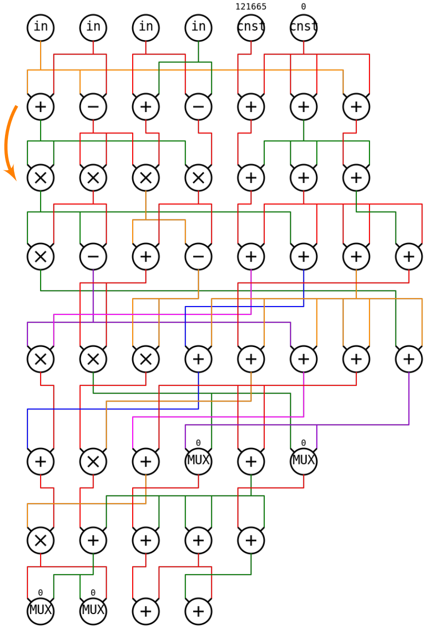




# Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

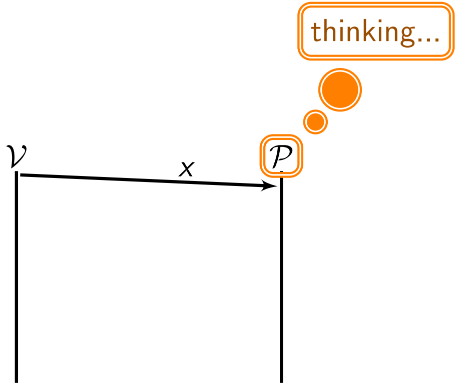
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates

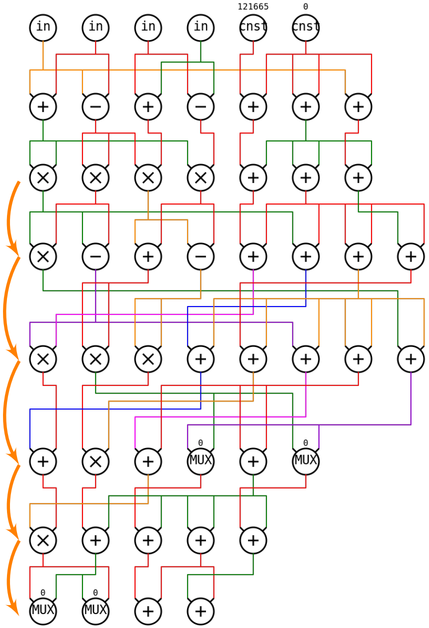




## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

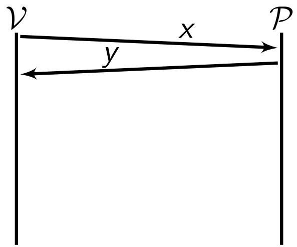
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates

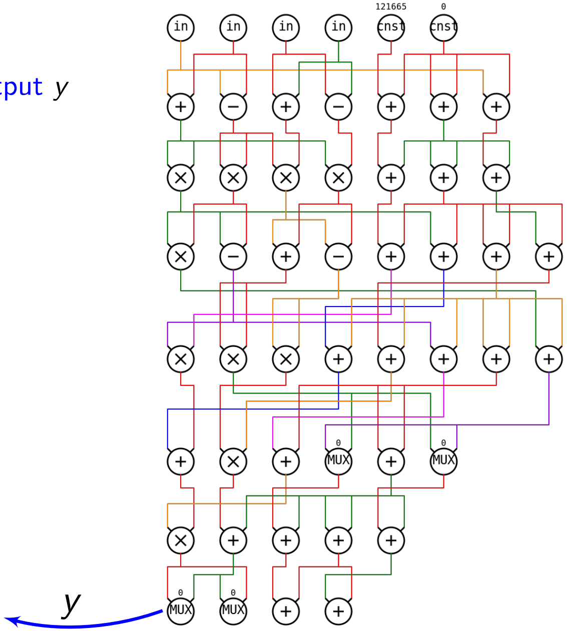




## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

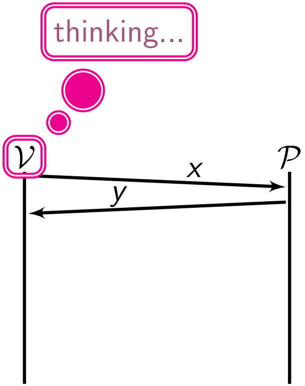
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$

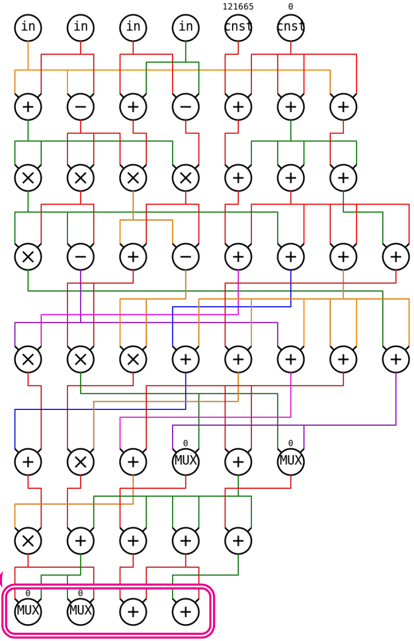




# Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

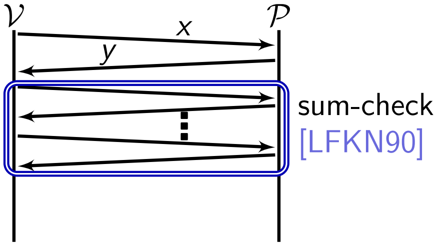
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires

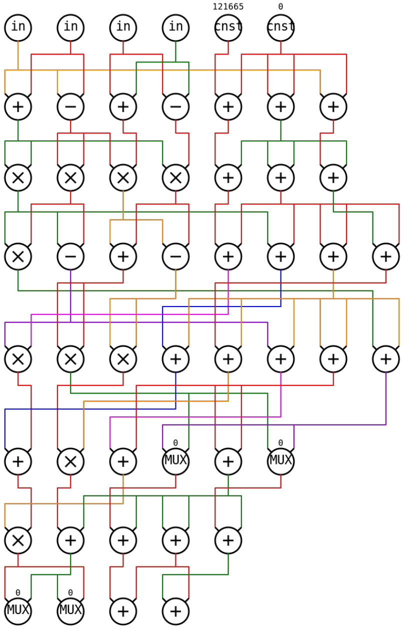




# Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

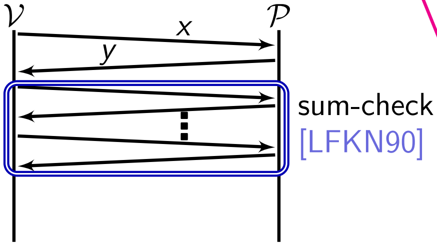
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires
4. $\mathcal{V}$ engages $\mathcal{P}$ in a sum-check

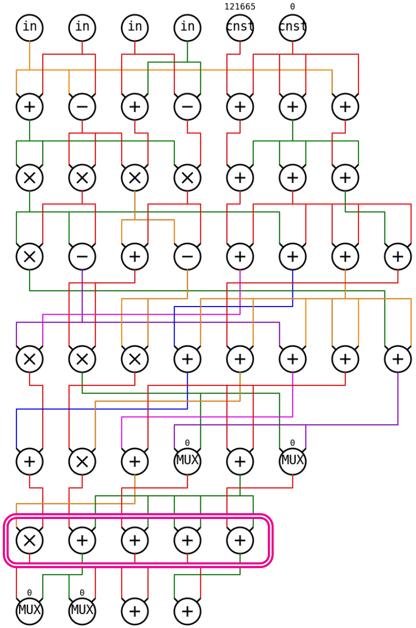




# Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

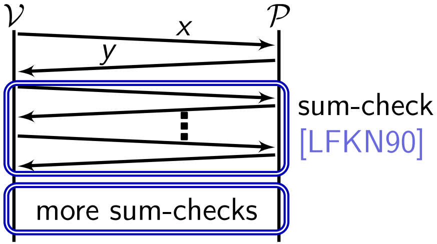
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires
4. $\mathcal{V}$ engages $\mathcal{P}$ in a sum-check, gets claim about second-last layer

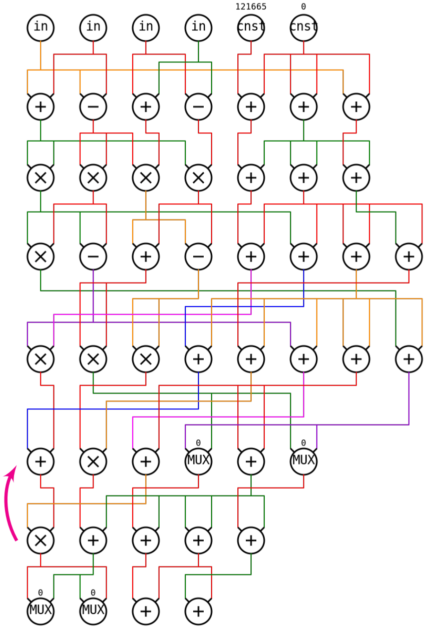




## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

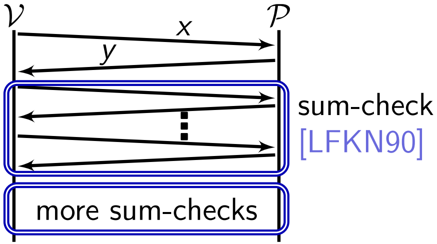
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires
4. $\mathcal{V}$ engages $\mathcal{P}$ in a sum-check, gets claim about second-last layer
5. $\mathcal{V}$ iterates

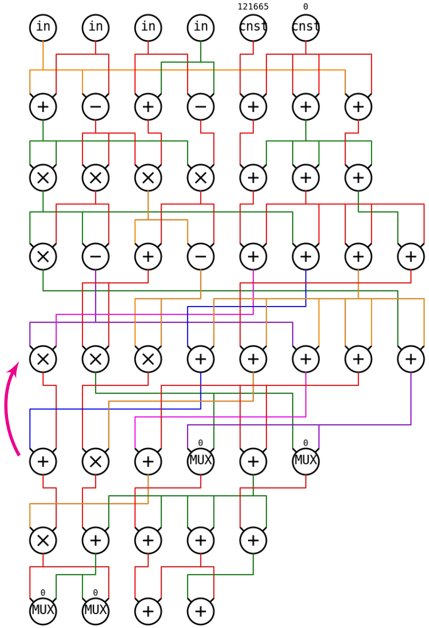




## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

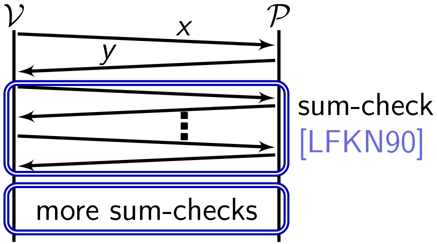
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires
4. $\mathcal{V}$ engages $\mathcal{P}$ in a sum-check, gets claim about second-last layer
5. $\mathcal{V}$ iterates

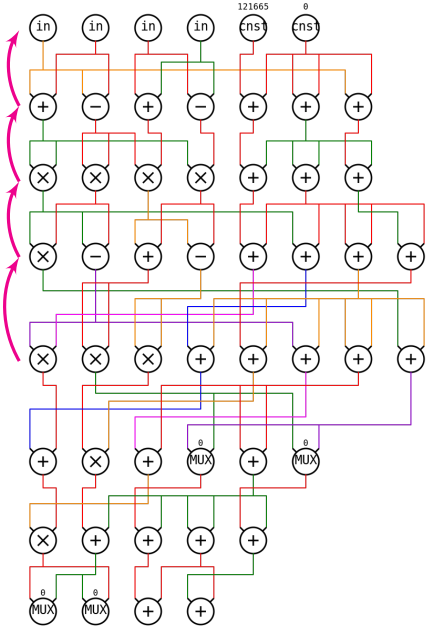




# Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

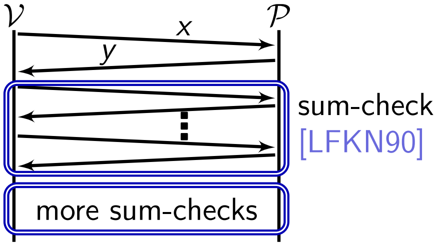
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires
4. $\mathcal{V}$ engages $\mathcal{P}$ in a sum-check, gets claim about second-last layer
5. $\mathcal{V}$ iterates

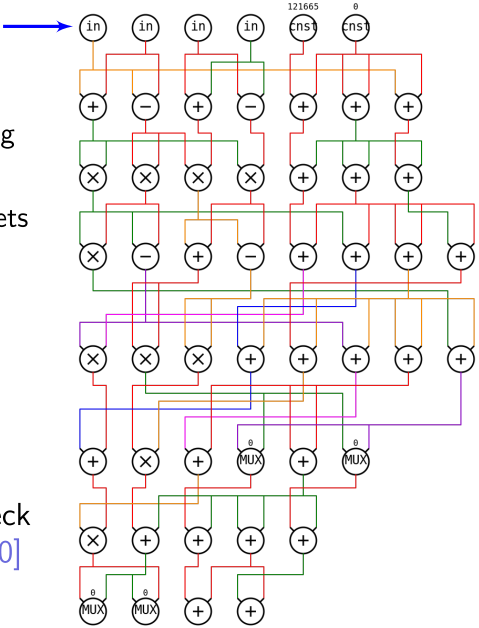




## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

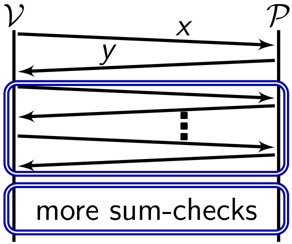
1. $\mathcal{V}$ sends inputs
2. $\mathcal{P}$ evaluates, returns output $y$
3. $\mathcal{V}$ constructs polynomial relating $y$ to last layer's input wires
4. $\mathcal{V}$ engages $\mathcal{P}$ in a sum-check, gets claim about second-last layer
5. $\mathcal{V}$ iterates, gets claim about inputs, which it can check



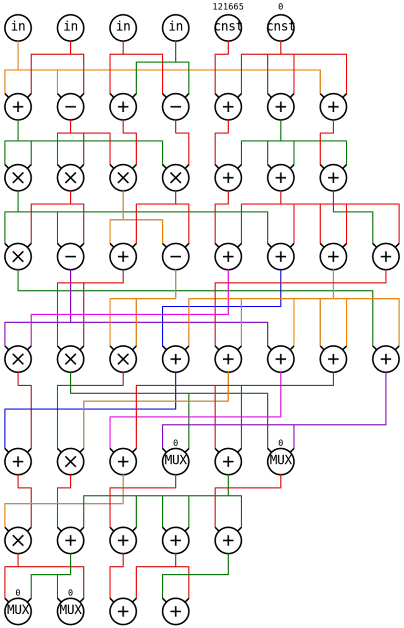


## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

Soundness error $\propto p^{-1}$



sum-check [LFKN90]

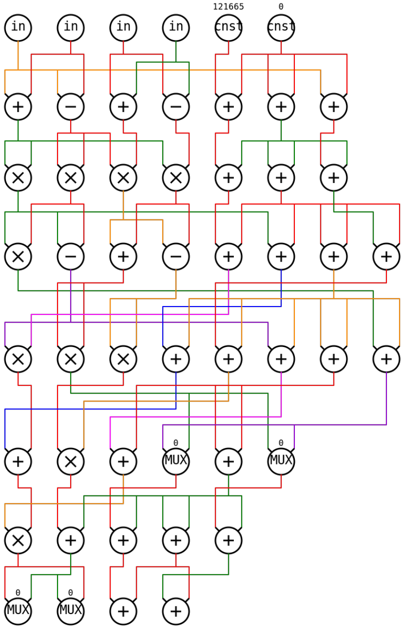


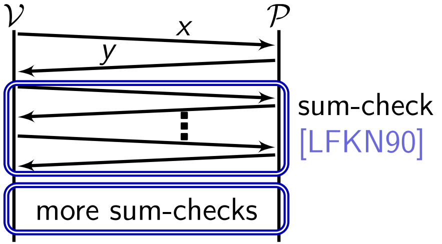
## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

Soundness error $\propto p^{-1}$

Cost to execute F directly:
O(depth · width)

$\mathcal{V}$'s sequential running time:
O(depth · log width + $|x|$ + $|y|$)
(assuming precomputed queries)





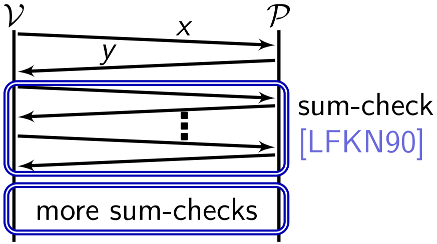
## Zebra builds on IPs of GKR [GKR08, CMT12, VSBW13]

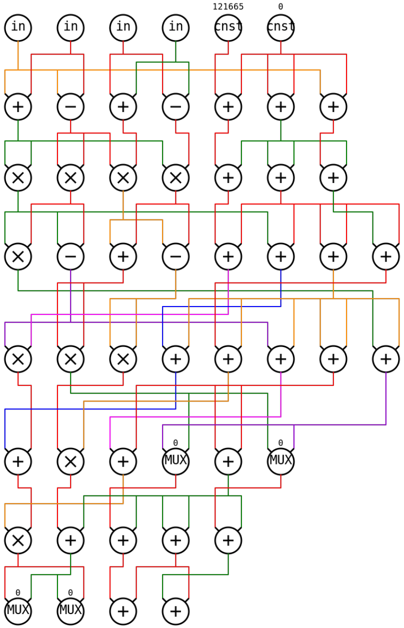
Soundness error $\propto p^{-1}$

Cost to execute F directly:
O(depth · width)

$\mathcal{V}$'s sequential running time:
O(depth · log width + $|x|$ + $|y|$)
(assuming precomputed queries)

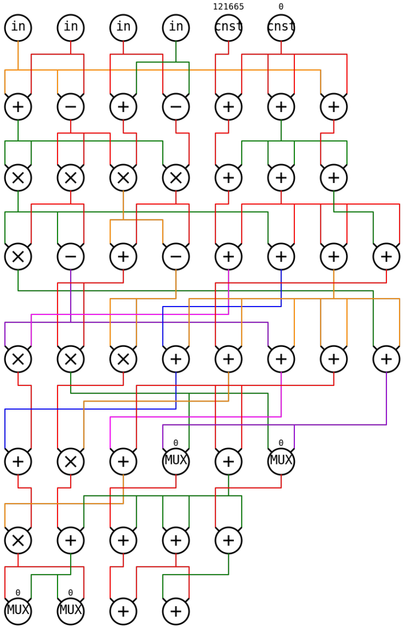
$\mathcal{P}$'s sequential running time:
O(depth · width · log width)





## Extracting parallelism in Zebra

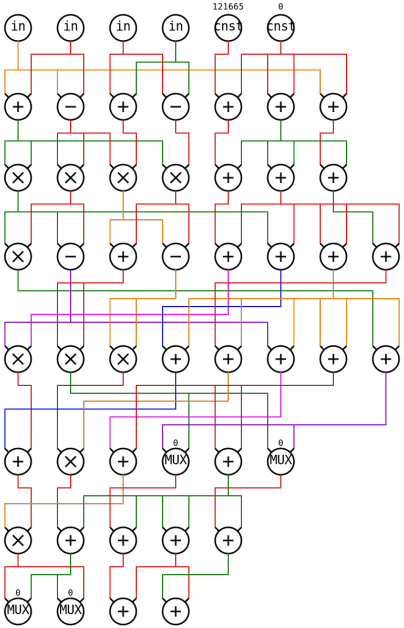
$\mathcal{P}$ executing AC: layers are sequential, but all gates at a layer can be executed in parallel



## Extracting parallelism in Zebra

$\mathcal{P}$ executing AC: layers are sequential, but all gates at a layer can be executed in parallel

Proving step: Can $\mathcal{V}$ and $\mathcal{P}$ interact about all of F's layers at once?

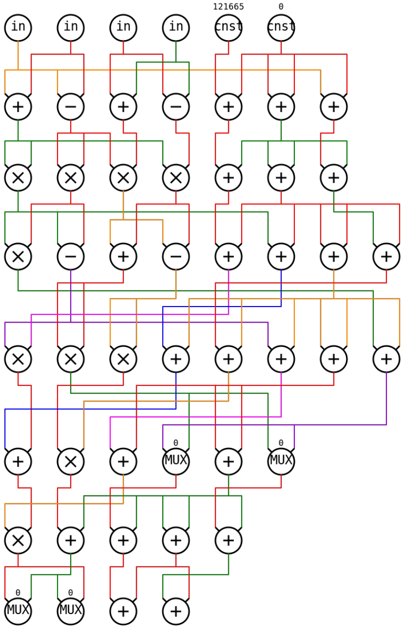


# Extracting parallelism in Zebra

$\mathcal{P}$ executing AC: layers are sequential, but all gates at a layer can be executed in parallel

Proving step: Can $\mathcal{V}$ and $\mathcal{P}$ interact about all of F's layers at once?

No. $\mathcal{V}$ must ask questions in order or soundness is lost.



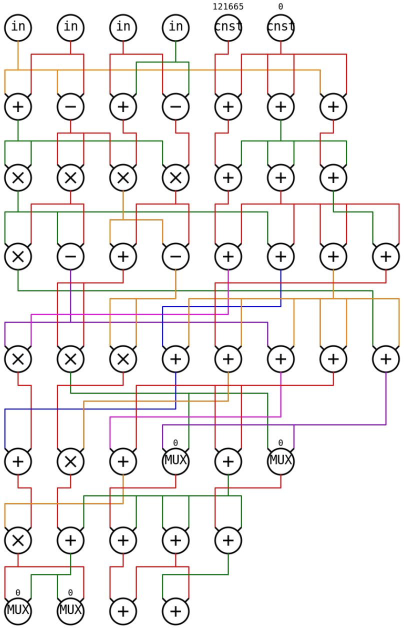
## Extracting parallelism in Zebra

$\mathcal{P}$ executing AC: layers are sequential, but all gates at a layer can be executed in parallel

Proving step: Can $\mathcal{V}$ and $\mathcal{P}$ interact about all of F's layers at once?

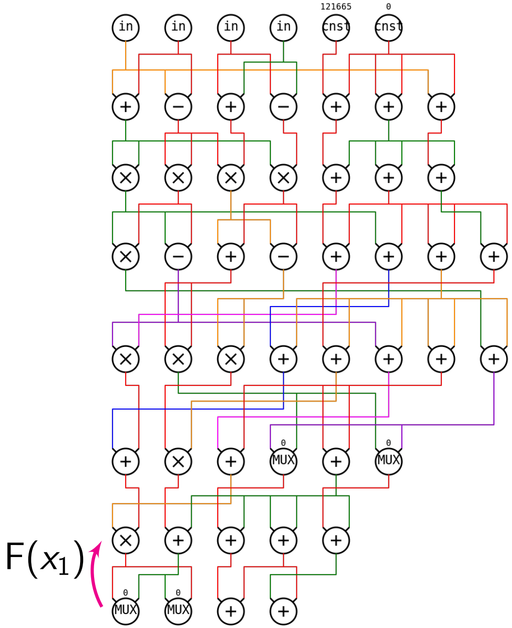
No. $\mathcal{V}$ must ask questions in order or soundness is lost.

But: there is still parallelism to be extracted...



## Extracting parallelism in Zebra's $\mathcal{P}$

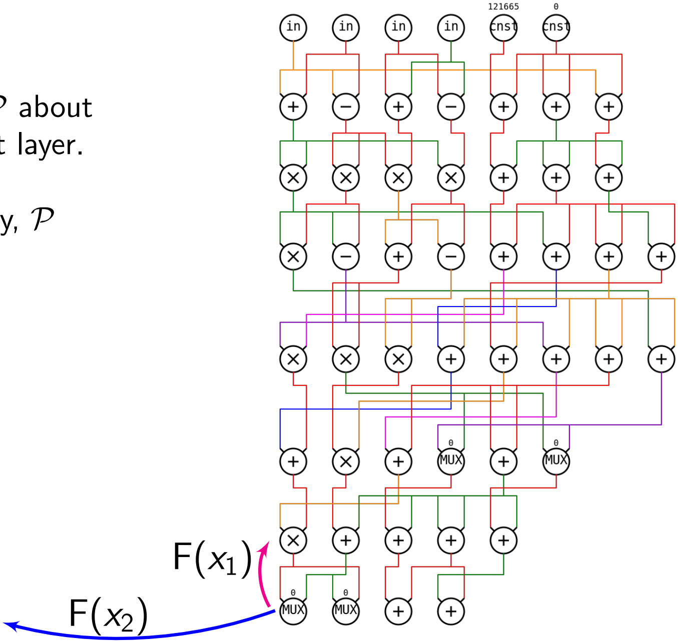
$\mathcal{V}$ questions $\mathcal{P}$ about $F(x_1)$'s output layer.



## Extracting parallelism in Zebra's $\mathcal{P}$

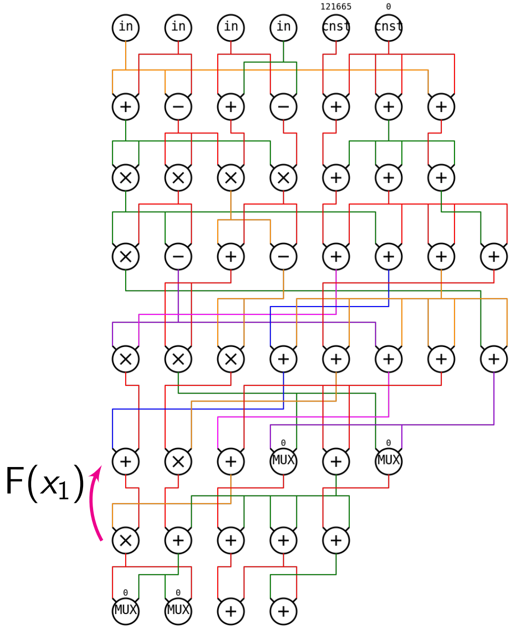
$\mathcal{V}$ questions $\mathcal{P}$ about $F(x_1)$'s output layer.

Simultaneously, $\mathcal{P}$ returns $F(x_2)$.



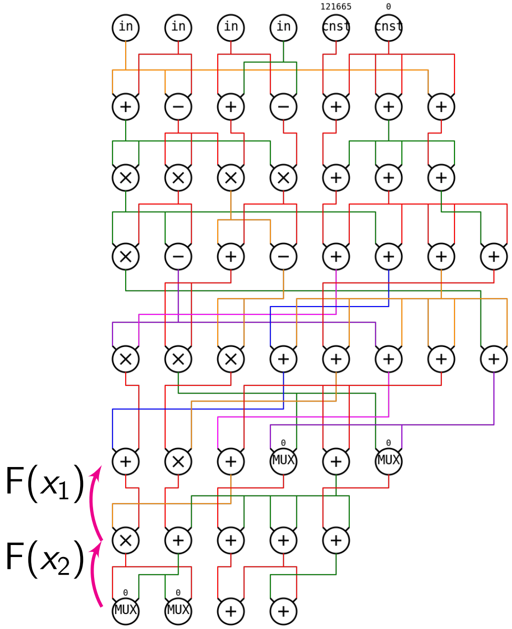
## Extracting parallelism in Zebra's $\mathcal{P}$

$\mathcal{V}$ questions $\mathcal{P}$ about $F(x_1)$'s next layer



## Extracting parallelism in Zebra's $\mathcal{P}$

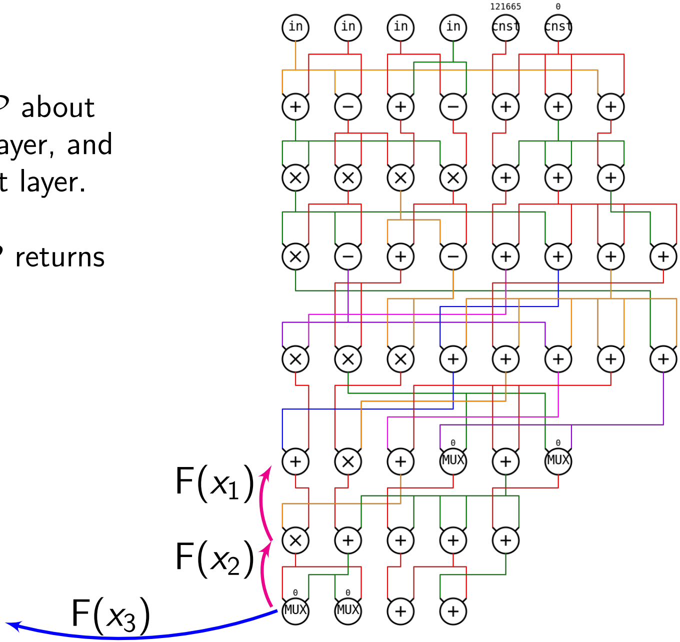
$\mathcal{V}$ questions $\mathcal{P}$ about $F(x_1)$'s next layer, and $F(x_2)$'s output layer.



# Extracting parallelism in Zebra's $\mathcal{P}$

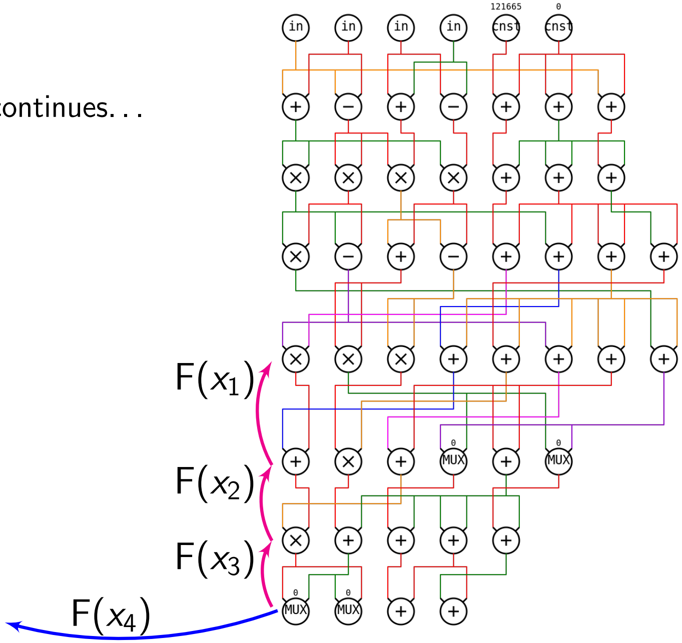
$\mathcal{V}$ questions $\mathcal{P}$ about $F(x_1)$'s next layer, and $F(x_2)$'s output layer.

Meanwhile, $\mathcal{P}$ returns $F(x_3)$.



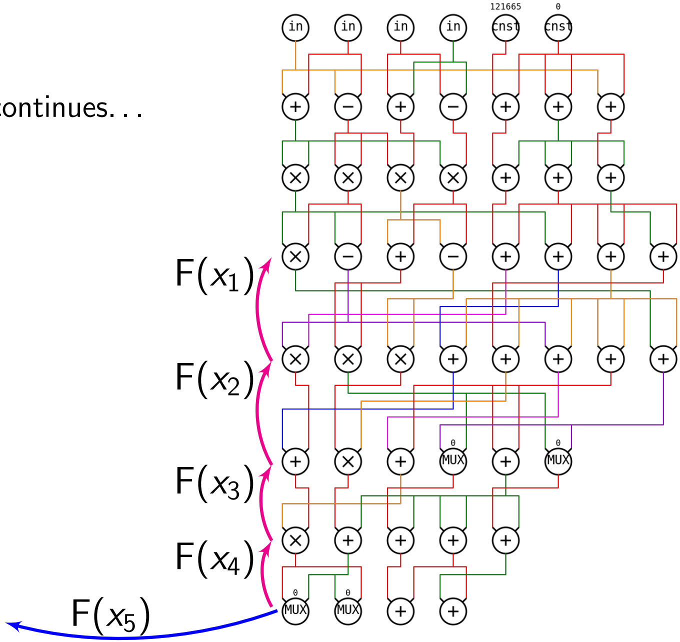
## Extracting parallelism in Zebra's $\mathcal{P}$

This process continues. . .



## Extracting parallelism in Zebra's $\mathcal{P}$

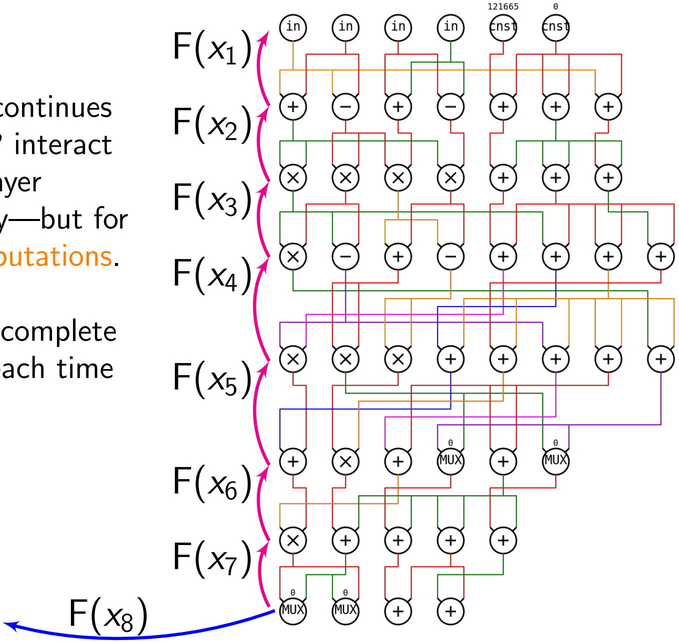
This process continues. . .



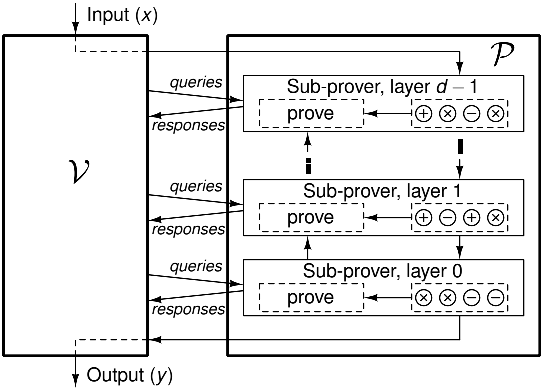
## Extracting parallelism in Zebra's $\mathcal{P}$

This process continues until $\mathcal{V}$ and $\mathcal{P}$ interact about every layer simultaneously—but for different computations.

$\mathcal{V}$ and $\mathcal{P}$ can complete one proof in each time step.

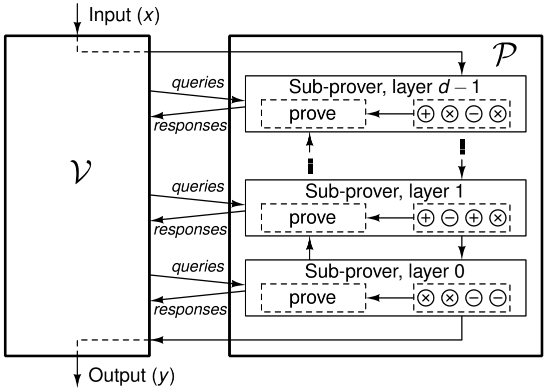


## Extracting parallelism in Zebra's $\mathcal{P}$ with pipelining



This approach is just a standard hardware technique, pipelining; it is possible because the protocol is naturally staged.

## Extracting parallelism in Zebra's $\mathcal{P}$ with pipelining

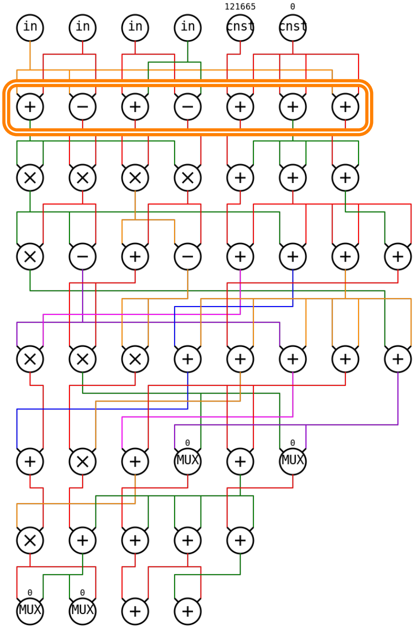


This approach is just a standard hardware technique, pipelining; it is possible because the protocol is naturally staged.

There are other opportunities to leverage the protocol's structure.

## Per-layer computations

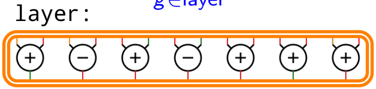
For each sum-check round, $\mathcal{P}$ sums over each gate in a layer



## Per-layer computations

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $\mathrm{H}[k]$, $k \in \{0, 1, 2\}$

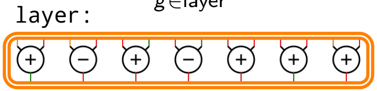
$$\mathrm{H}[k] = \sum_{g \in \mathrm{layer}} \delta(g, k)$$



## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$



In software:

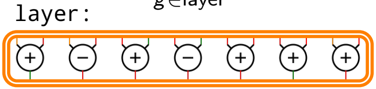
```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$

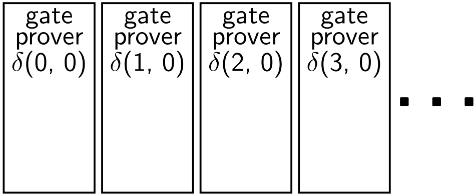


In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

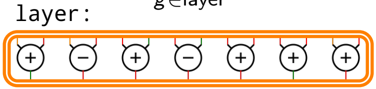
In hardware:


gate prover $\delta(0, 0)$
gate prover $\delta(1, 0)$
gate prover $\delta(2, 0)$
gate prover $\delta(3, 0)$
...

## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$

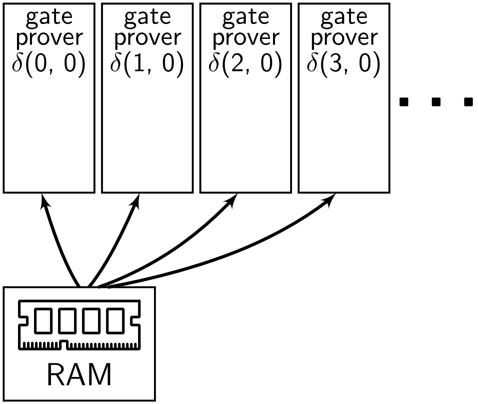


In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + δ(g, k)
    // δ uses state[g]

// update lookup table
// with 𝒱's random coin
for g ∈ layer:
  state[g] ← δ(g, r_j)
```

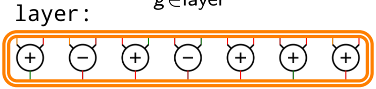
In hardware:



gate prover $\delta(0, 0)$
gate prover $\delta(1, 0)$
gate prover $\delta(2, 0)$
gate prover $\delta(3, 0)$
...
RAM

## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$



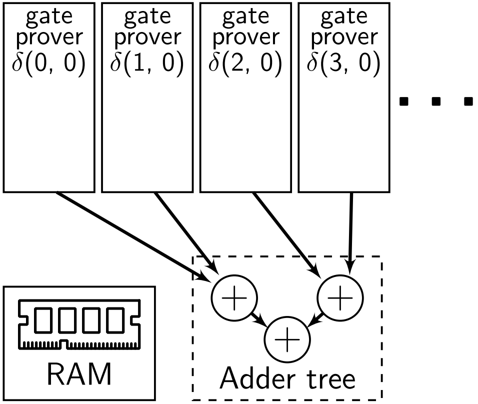
For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$

In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + δ(g, k)
    // δ uses state[g]

// update lookup table
// with 𝒱's random coin
for g ∈ layer:
  state[g] ← δ(g, r_j)
```

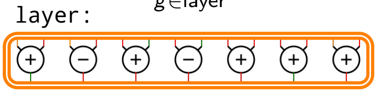
In hardware:


gate prover $\delta(0, 0)$
gate prover $\delta(1, 0)$
gate prover $\delta(2, 0)$
gate prover $\delta(3, 0)$
...
RAM
Adder tree

## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$

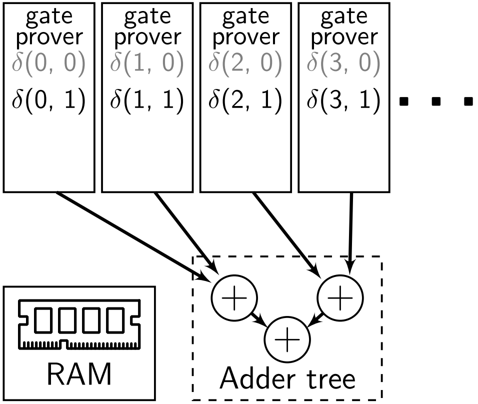


In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

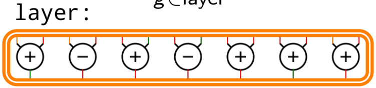
In hardware:



## Per-layer computations

$$\mathsf{H}[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $\mathsf{H}[k]$, $k \in \{0, 1, 2\}$

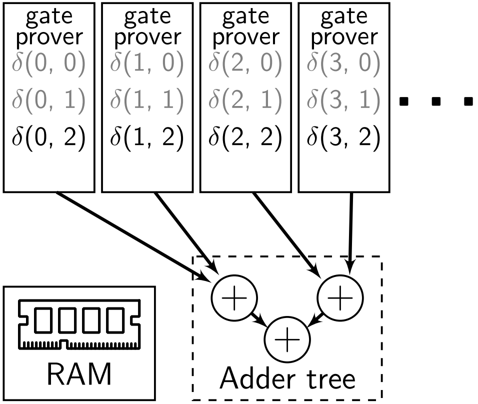


In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

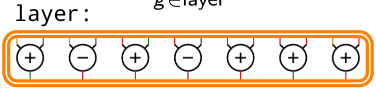
In hardware:



## Per-layer computations

$$\mathsf{H}[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $\mathsf{H}[k]$, $k \in \{0, 1, 2\}$

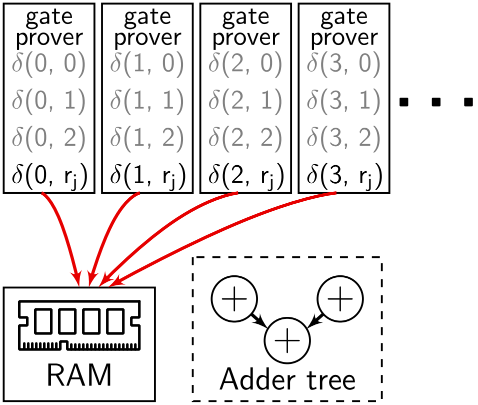


In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

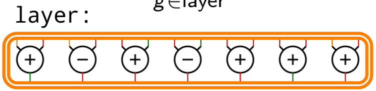
In hardware:



## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$

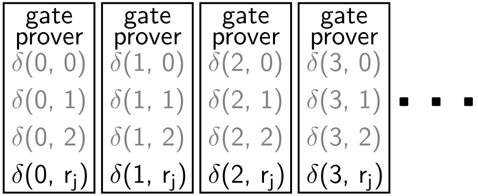


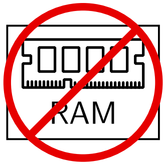
In software:

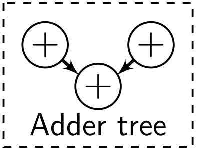
```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

In hardware:



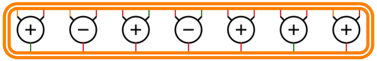




## Per-layer computations

$$H[k] = \sum_{g \in \text{layer}} \delta(g, k)$$

layer:



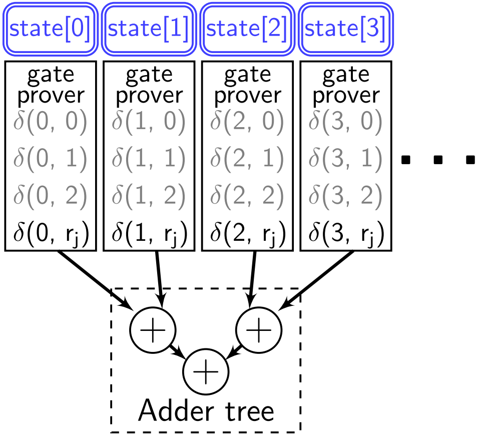
For each sum-check round, $\mathcal{P}$ sums over each gate in a layer, evaluating $H[k]$, $k \in \{0, 1, 2\}$

In software:

```
// compute H[0],H[1],H[2]
for k ∈ {0, 1, 2}:
  H[k] ← 0
  for g ∈ layer:
    H[k] ← H[k] + $\delta$(g, k)
    // $\delta$ uses state[g]

// update lookup table
// with $\mathcal{V}$'s random coin
for g ∈ layer:
  state[g] ← $\delta$(g, $r_j$)
```

In hardware:



## Zebra's design approach

✓ Extract parallelism

e.g., pipelined proving

e.g., parallel evaluation of $\delta$ by gate provers

✓ Exploit locality: distribute data and control

e.g., no RAM: data is kept close to places it is needed

## Zebra's design approach

✓ Extract parallelism

e.g., pipelined proving
e.g., parallel evaluation of $\delta$ by gate provers

✓ Exploit locality: distribute data and control

e.g., no RAM: data is kept close to places it is needed
e.g., *latency-insensitive* design: localized control

## Zebra's design approach

✓ Extract parallelism

e.g., pipelined proving
e.g., parallel evaluation of $\delta$ by gate provers

✓ Exploit locality: distribute data and control

e.g., no RAM: data is kept close to places it is needed
e.g., *latency-insensitive* design: localized control

✓ Reduce, reuse, recycle

e.g., computation: save energy by adding memoization to $\mathcal{P}$
e.g., hardware: save chip area by reusing the same circuits

## Architectural challenges

Interaction between $\mathcal{V}$ and $\mathcal{P}$ requires a lot of bandwidth

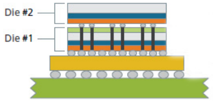
✗ $\mathcal{V}$ and $\mathcal{P}$ on circuit board? Too much energy, circuit area

## Architectural challenges

Interaction between $\mathcal{V}$ and $\mathcal{P}$ requires a lot of bandwidth

✗ $\mathcal{V}$ and $\mathcal{P}$ on circuit board? Too much energy, circuit area

✓ Zebra uses 3D integration

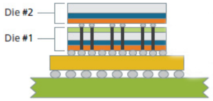


## Architectual challenges

Interaction between $\mathcal{V}$ and $\mathcal{P}$ requires a lot of bandwidth

✗ $\mathcal{V}$ and $\mathcal{P}$ on circuit board? Too much energy, circuit area

✓ Zebra uses 3D integration



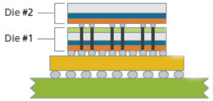
Protocol requires input-independent precomputation [VSBW13]

# Architectual challenges

Interaction between $\mathcal{V}$ and $\mathcal{P}$ requires a lot of bandwidth

✗ $\mathcal{V}$ and $\mathcal{P}$ on circuit board? Too much energy, circuit area

✓ Zebra uses 3D integration



Protocol requires input-independent precomputation [VSBW13]

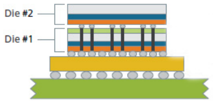
✓ Zebra amortizes precomputations over many $\mathcal{V}$-$\mathcal{P}$ pairs

## Architectual challenges

Interaction between $\mathcal{V}$ and $\mathcal{P}$ requires a lot of bandwidth

✗ $\mathcal{V}$ and $\mathcal{P}$ on circuit board? Too much energy, circuit area

✓ Zebra uses 3D integration

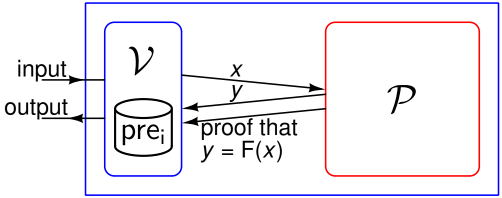


Protocol requires input-independent precomputation [VSBW13]

✓ Zebra amortizes precomputations over many $\mathcal{V}$-$\mathcal{P}$ pairs

Precomputations need secrecy, integrity

✗ Give $\mathcal{V}$ trusted storage? Cost would be prohibitive

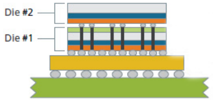


## Architectual challenges

Interaction between $\mathcal{V}$ and $\mathcal{P}$ requires a lot of bandwidth

✗ $\mathcal{V}$ and $\mathcal{P}$ on circuit board? Too much energy, circuit area

✓ Zebra uses 3D integration



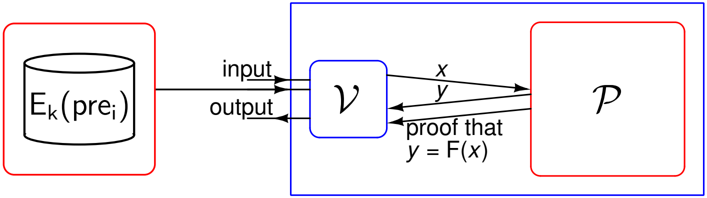
Protocol requires input-independent precomputation [VSBW13]

✓ Zebra amortizes precomputations over many $\mathcal{V}$-$\mathcal{P}$ pairs

Precomputations need secrecy, integrity

✗ Give $\mathcal{V}$ trusted storage? Cost would be prohibitive

✓ Zebra uses untrusted storage + authenticated encryption



## Implementation

Zebra's implementation includes

- a compiler that produces synthesizable Verilog for $\mathcal{P}$
- two $\mathcal{V}$ implementations
  - hardware (Verilog)
  - software (C++)
- library to generate $\mathcal{V}$'s precomputations
- Verilog simulator extensions to model software or hardware $\mathcal{V}$'s interactions with $\mathcal{P}$

. . . and it seemed to work really well!

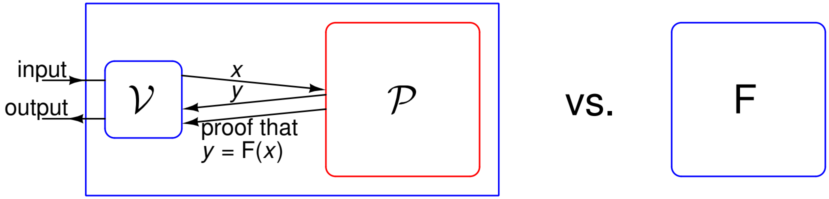
Zebra can produce 10k–100k proofs per second,
while existing systems take tens of seconds per proof!

. . . and it seemed to work really well!

Zebra can produce 10k–100k proofs per second,
while existing systems take tens of seconds per proof!

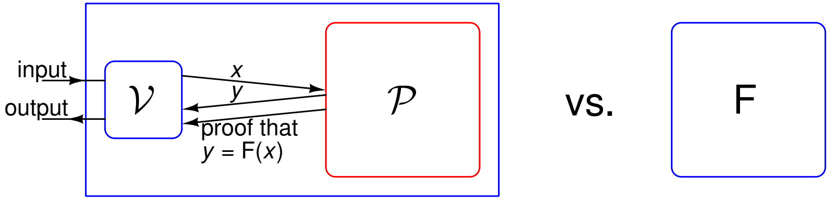
But that's not a serious evaluation. . .

## Evaluation method



Baseline: direct implementation of F in same technology as $\mathcal{V}$

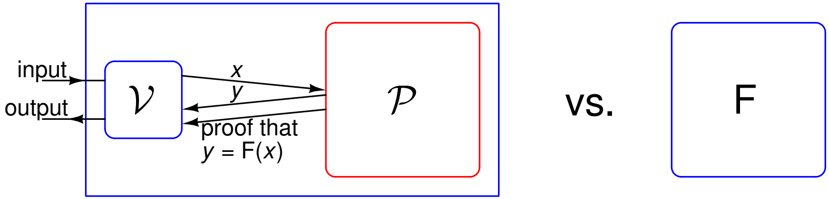
## Evaluation method



Baseline: direct implementation of F in same technology as $\mathcal{V}$

Metrics: energy, chip size per throughput (discussed in paper)

## Evaluation method



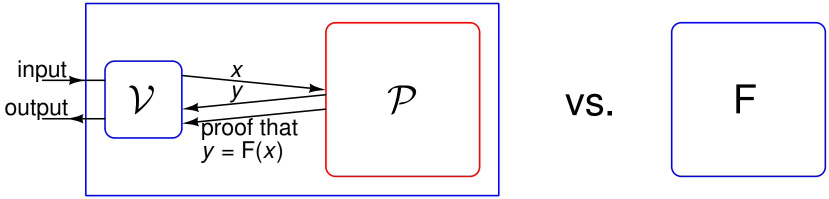
Baseline: direct implementation of F in same technology as $\mathcal{V}$

Metrics: energy, chip size per throughput (discussed in paper)

Measurements: based on circuit synthesis and simulation, published chip designs, and CMOS scaling models

Charge for $\mathcal{V}$, $\mathcal{P}$, communication; retrieving and decrypting precomputations; PRNG; Operator communicating with $\mathcal{V}$

## Evaluation method



Baseline: direct implementation of F in same technology as $\mathcal{V}$

Metrics: energy, chip size per tl

Measurements: based on circuit
published chip designs, and CM

Charge for $\mathcal{V}$, $\mathcal{P}$, communi
precomputations; PRNG; Operator communicating with

350 nm: 1997 (Pentium II)
7 nm: $\approx$ 2017 [TSMC]
$\approx$ 20 year gap between
trusted and untrusted fab

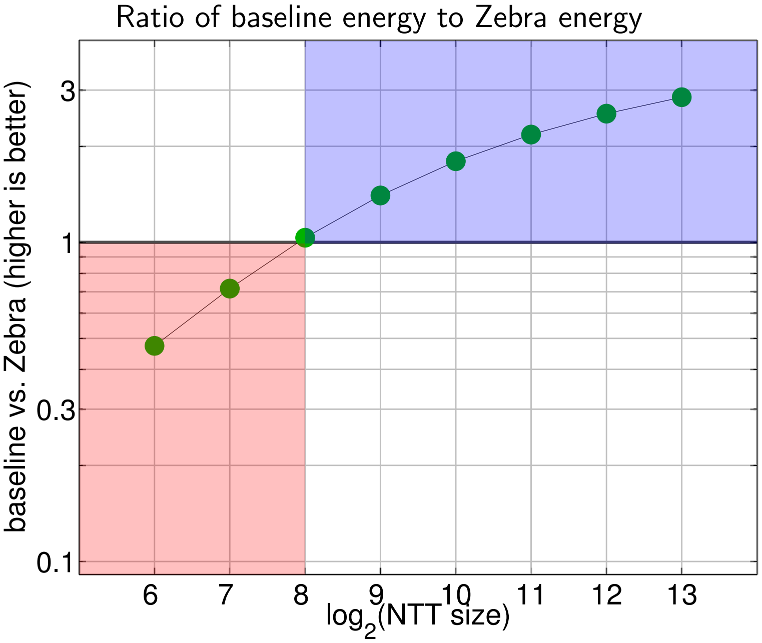
Constraints: trusted fab = 350 nm; untrusted fab = 7 nm
200 mm$^2$ max chip area; 150 W max total power

## Application #1: number theoretic transform

NTT: a Fourier transform over $\mathbb{F}_p$

Widely used, e.g., in computer algebra

## Application #1: number theoretic transform

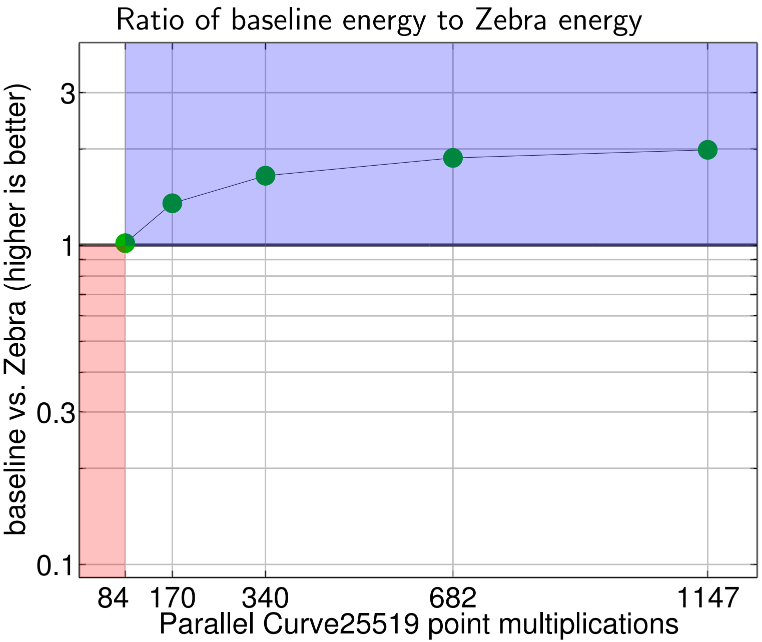


## Application #2: Curve25519 point multiplication

Curve25519: a commonly-used elliptic curve

Point multiplication: primitive, e.g., for ECDH

## Application #2: Curve25519 point multiplication



## A **qualified** success

Zebra: a hardware design that saves costs. . .

. . . **sometimes**.

## Summary of Zebra's applicability

1. Computation F must have a layered, shallow, deterministic AC
2. Must have a wide gap between cutting-edge fab (for $\mathcal{P}$) and trusted fab (for $\mathcal{V}$)
3. Amortizes precomputations over many instances
4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Summary of Zebra's applicability

**Applies to IPs, but not arguments**

1. Computation F must have a layered, shallow, deterministic AC
2. Must have a wide gap between cutting-edge fab (for $\mathcal{P}$) and trusted fab (for $\mathcal{V}$)
3. Amortizes precomputations over many instances
4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Arguments versus IPs, redux

| **Design principle** | **IPs** [GKR08, CMT12, VSBW13] | **Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14] |
|---|---|---|
| Extract parallelism | ✓ | ✓ |
| Exploit locality | ✓ | |
| Reduce, reuse, recycle | ✓ | |

Argument protocols seem friendly to hardware?

## Arguments versus IPs, redux

| **Design principle** | **IPs** [GKR08, CMT12, VSBW13] | **Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14] |
|---|---|---|
| Extract parallelism | ✓ | ✓ |
| Exploit locality | ✓ | ✗ |
| Reduce, reuse, recycle | ✓ | |

Argument protocols seem unfriendly to hardware:

$\mathcal{P}$ computes over entire AC at once $\implies$ need RAM

## Arguments versus IPs, redux

| Design principle | IPs [GKR08, CMT12, VSBW13] | Arguments [GGPR13, SBVBPW13, PGHR13, BCTV14] |
|---|---|---|
| Extract parallelism | ✓ | ✓ |
| Exploit locality | ✓ | ✗ |
| Reduce, reuse, recycle | ✓ | ✗ |

Argument protocols seem unfriendly to hardware:

$\mathcal{P}$ computes over entire AC at once $\implies$ need RAM

$\mathcal{P}$ does crypto for every gate in AC $\implies$ special crypto circuits

## Arguments versus IPs, redux

| **Design principle** | **IPs** [GKR08, CMT12, VSBW13] | **Arguments** [GGPR13, SBVBPW13, PGHR13, BCTV14] |
|---|---|---|
| Extract parallelism | ✓ | ✓ |
| Exploit locality | ✓ | ✗ |
| Reduce, reuse, recycle | ✓ | ✗ |

Argument protocols seem unfriendly to hardware:

$\mathcal{P}$ computes over entire AC at once $\implies$ need RAM

$\mathcal{P}$ does crypto for every gate in AC $\implies$ special crypto circuits

…but we hope these issues are surmountable!

## Summary of Zebra's applicability

1. Computation F must have a layered, shallow, deterministic AC
2. Must have a wide gap between cutting-edge fab (for $\mathcal{P}$) and trusted fab (for $\mathcal{V}$)
3. Amortizes precomputations over many instances
4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

**Common to essentially all built proof systems**

## Summary of Zebra's applicability

1. Computation F must have a layered, shallow, deterministic AC
2. Must have a wide gap between cutting-edge fab (for $\mathcal{P}$) and trusted fab (for $\mathcal{V}$)
3. $\boxed{\text{Amortizes}}$ precomputations over many instances
4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Summary of Zebra's applicability

1. Computation F
2. Must have a wi
   and trusted fab
3. Amortizes preco
4. Computation
5. Computation F

| System | Amortization regime | Advice |
|---|---|---|
| Zebra | many $\mathcal{V}$-$\mathcal{P}$ pairs | short |
| Allspice [VSBW13] | batch of instances of a particular F | short |
| Bootstrapped SNARKs [BCTV14a, CTV15] | all computations | long |
| BCTV [BCTV14b] | all computations of the same length | long |
| Pinocchio [PGHR13] | all future instances of a particular F | long |
| Zaatar [SBVBPW13] | batch of instances of a particular F | long |

Exception: [CMT12] with logspace-uniform ACs

## Summary of Zebra's applicability

1. Computation F must have a layered, shallow, deterministic AC
2. Must have a wide gap between cutting-edge fab (for $\mathcal{P}$) and trusted fab (for $\mathcal{V}$)
3. Amortizes precomputations over many instances
4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Summary of Zebra's applicability

For example, libsnark [BCTV14b], a highly optimized implementation of [GGPR13] and Pinocchio [PGHR13]:

$\mathcal{V}$'s work: $6\ \text{ms} + (|x| + |y|) \cdot 3\ \mu\text{s}$ on a 2.7 GHz CPU

4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Summary of Zebra's applicability

For example, libsnark [BCTV14b], a highly optimized implementation of [GGPR13] and Pinocchio [PGHR13]:

$\mathcal{V}$'s work: 6 ms + $(|x| + |y|) \cdot 3\ \mu$s on a 2.7 GHz CPU

$\Rightarrow$ break-even point $\geq 16 \times 10^6$ CPU ops

4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Summary of Zebra's applicability

For example, libsnark [BCTV14b], a highly optimized implementation of [GGPR13] and Pinocchio [PGHR13]:

$\mathcal{V}$'s work: $6$ ms $+ (|x| + |y|) \cdot 3\ \mu$s on a 2.7 GHz CPU

$\Rightarrow$ break-even point $\geq 16 \times 10^6$ CPU ops

With 32 GB RAM, libsnark handles ACs with $\leq 16 \times 10^6$ gates

4. Computation F must be very large for $\mathcal{V}$ to save work
5. Computation F must be efficient as an arithmetic circuit

## Summary of Zebra's applicability

For example, libsnark [BCTV14b], a highly optimized implementation of [GGPR13] and Pinocchio [PGHR13]:

$\mathcal{V}$'s work: $6 \text{ ms} + (|x| + |y|) \cdot 3\ \mu\text{s}$ on a 2.7 GHz CPU

$\Rightarrow$ break-even point $\geq 16 \times 10^6$ CPU ops

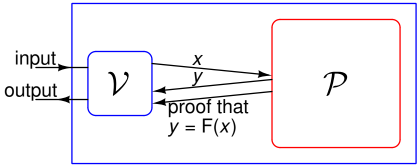
With 32 GB RAM, libsnark handles ACs with $\leq 16 \times 10^6$ gates

$\Rightarrow$ breaking even requires $> 1$ CPU op per AC gate, e.g., computations over $\mathbb{F}_p$ rather than machine integers

4. Computation F must be very large for $\mathcal{V}$ to save work

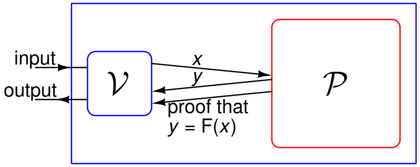
5. Computation F must be efficient as an arithmetic circuit

## Recap



- Verifiable ASICs: a new approach to building trustworthy hardware under a strong threat model
- First hardware design for a probabilistic proof protocol
- Improves performance compared to trusted baseline

## Recap

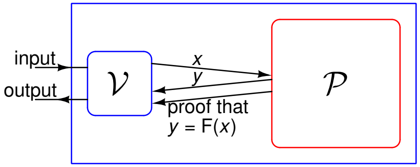


+ Verifiable ASICs: a new approach to building trustworthy hardware under a strong threat model
+ First hardware design for a probabilistic proof protocol
+ Improves performance compared to trusted baseline

– Improvement compared to the baseline is modest
– Applicability is limited:

  precomputations must be amortized
  computation needs to be “big enough”
  large gap between trusted and untrusted technology
  does not apply to all computations

## Recap



+ Verifiable ASICs: a new approach to building trustworthy hardware under a strong threat model
+ First hardware design for a probabilistic proof protocol
+ Improves performance compared to trusted baseline

– Improvement compared to the baseline is modest
– Applicability is limited:
  - precomputations must be amortized
  - computation needs to be “big enough”
  - large gap between trusted and untrusted technology
  - does not apply to all computations

Bottom line: Zebra is plausible—when it applies

`https://www.pepper-project.org/`