Constructing an Adversary Solver for Equihash

Xiaofei Bai, **Jian Gao**, Chenglong Hu, Liang Zhang
Fudan University

Fairness of PoW Systems

- Don’t service those who don’t work
- Adversaries should not be doing less work than honest users

- Cheating adversaries eat away miner profit
  - Drive miners away
  - Form monopoly -> 51% attack

ASIC Solvers and ASIC resistance

- Adversaries build highly-efficient ASIC solvers
  - not selling to most users
  - creating profitability advantage
- New systems try to limit ASIC solver advantage
  - craft puzzle schemas with special characteristics
  - spawned almost every PoW system after SHA256d
  - designing ASIC solvers is significantly harder
  - memory-hard and bandwidth-hard theories
- Alternative methods exist

Equihash

- Designed to be ASIC resistant
- Uses a single-list General Birthday Problem
- Applied in cryptocurrency systems including Zcash and BTG
- Has efficient ASIC solvers under (200, 9), but not under (144, 5)

- Our design can handle all parameters

Goals

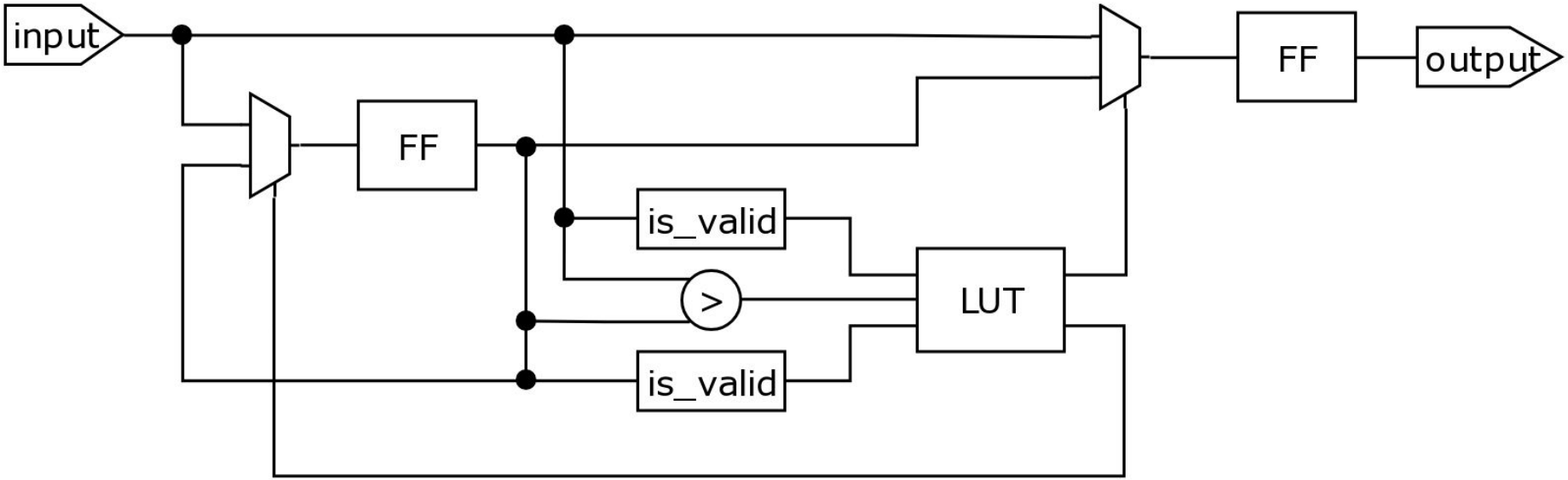
- Demonstrate how adversaries design their solvers
  - the difficulties they face
  - how they work around problems
- Understand PoW systems better
- Designing puzzle schemas with better fairness
- Eventually offer better protection to underlying systems

Input : list L of N n-bit strings ($N \ll 2^n$)
begin
  Enumerate $L^{(0)}$ as $\{(x_i, \{i\}) | i = 1, 2, ..., N\}$
  $r \leftarrow 1$
  while $r < k$ **do**
    join
      Sort $L^{(r-1)}$, finding all unordered pairs $((x_i, S_i), (x_j, S_j))$ such that x_i collides with x_j on the first $\frac{rn}{k+1}$ bits, and that $S_i \cap S_j = \varnothing$
      $L^{(r)} \leftarrow \{(x_i \oplus x_j, S_i \cup S_j) | ((x_i, S_i), (x_j, S_j))$ is a found pair $\}$
      $r \leftarrow r + 1$
  join
    Sort $L^{(k-1)}$, finding all unordered pairs $((x_i, S_i), (x_j, S_j))$ such that $x_i = x_j$, and that $S_i \cap S_j = \varnothing$
    $R \leftarrow \{(S_i \cup S_j) | ((x_i, S_i), (x_j, S_j))$ is a found pair $\}$
Output: list R of sets of distinct indices

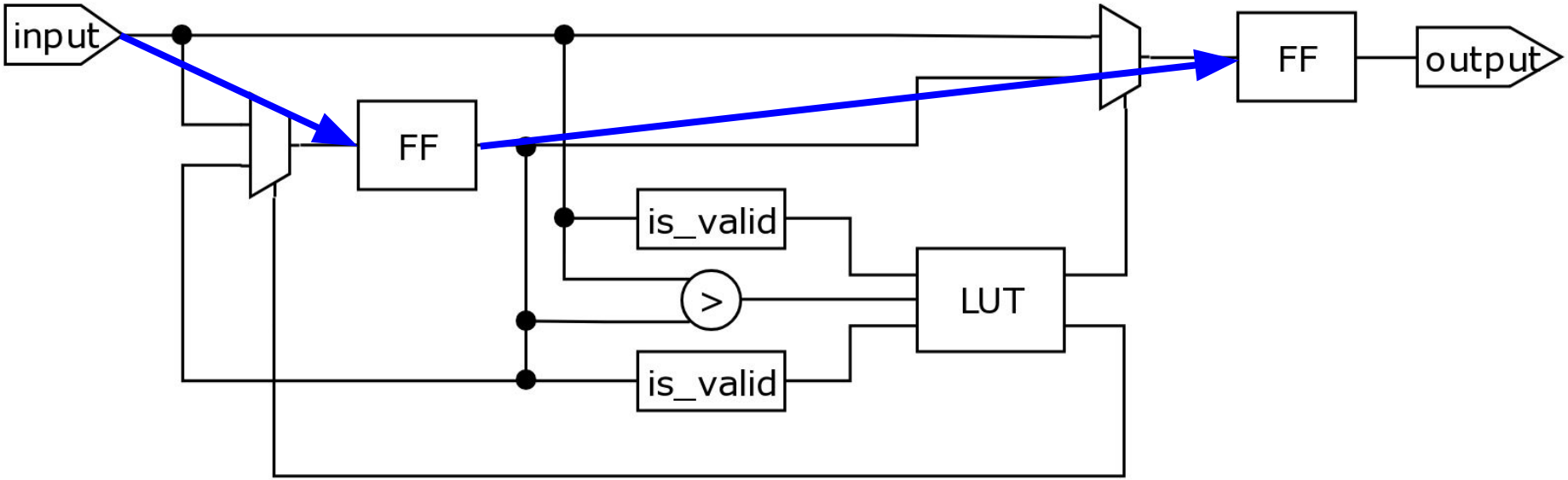
The `join` Step

- Under large parameter sets, list itself too large for on-chip storage
  - off-chip memory access is slow and energy-expensive
- Hashing or sorting to cover this subroutine
  - honest software solvers prefer hashing
- ASIC hashing is basically stripping CPU/GPU of unused logic
  - not much advantage

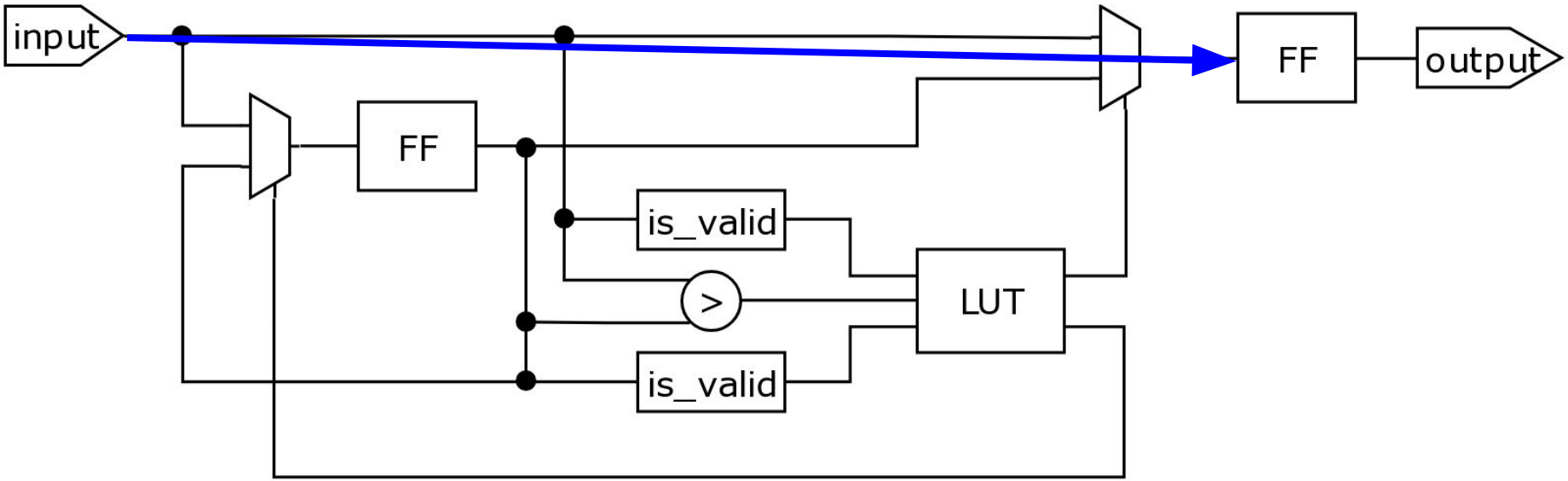
Smartcell (1)



Smartcell (2)



Smartcell (3)



Smartcell Chains

- Can sort longer lists
  - greater values overtake lesser ones
- Worst case: last to enter, first to exit
  - jumps over every other item
- Need N-smartcell chain to sort N-item lists

Smartcell Sorting Recap

Pros:

- Linear time
- One sequential read

Cons:

- Linear logic
  - too large for one chip
  - use too much energy

Merge

- Reduce list length
- Major operation is trivial
- Need to store input in memory
  - output pattern of smartcell chains and input pattern of merge modules mismatch
- Add prefetch queues to merging lanes
  - avoid stalling for read latency
  - input is random, so cache is not helpful
  - implemented with SRAMs/eDRAMs
    - possibly prefetch again into FFs

Pair Generation

- To record indices and compute XOR
- Trivial on MCUs
  - because colliding entries are already sorted together
- Output list length is random
  - expected to stay the same
  - if shrink: add bubbles to pipeline
  - if expand: breaks sorting in next round

Tail Cutting

- Cut the longer intermediate lists
- The better cases are cut
  - they (should) tend to yield more solutions
  - best case can have trillions of pairs
    - virtually impossible
    - handling is unrealistic
  - hurt yields, but not significantly

Evaluation

- Correctness: Proven. Algorithm unchanged
- Implementation: Verified. Via simulation
- Memory usage: Calculated.
- Performance: Calculated.
- Frequency and power usage: ?

Evaluation: Off-Chip Memory

	(200, 9)	(144, 5)
Capacity	~500MB	~8GB
Total Bandwidth	~70B/tick	~100B/tick

* Uniform memory, with pipelining considered.
** Solver core ticks, not memory bus ticks.

Evaluation: Power

- ME software to estimate power usage
  - based on design , clock speed and tech lib
  - serve as reference for IC designers
- Synopsis Design Compiler
  - with 28nm HKMG tech lib
  - Applied to core components
- 1.5-2 W at 1 GHz
  - scales very well with frequency

Evaluation: Efficiency (1)

frequency	/ throughput	/ k	* yield	* (1-loss)	/ power
tick/s	/ (tick/round)	/ (round/puzzle)	* (sol/puzzle)	* %	/ W

$$\frac{\text{= efficiency}}{\text{= (sol/J)}}$$

Evaluation: Efficiency (2)

	(200, 9)	(144, 5)
Adversary (ASIC)	~53 sol/J	~5.3 sol/J
Honest User (GPU)	~4 sol/J	~0.4 sol/J

* At 500MHz core frequency.

Conclusion

- A highly efficient adversary solver for Equihash
  - multi-chip ASIC
  - works under all parameter sets
- Equihash-related systems and assets should be carefully valued
- Exposed design procedure
  - PoW routines are subject to tweaks and optimizations
  - Adversary-side design challenges, as candidates for future PoW fairness
  - proving ASIC-resistance is extremely hard, if not impossible

THANK YOU!

Q&A