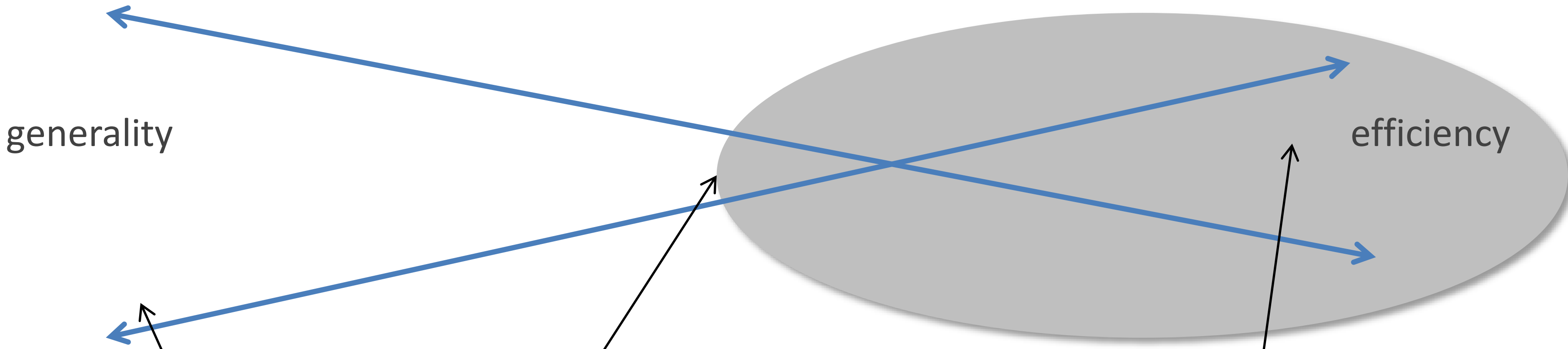
Secure Computation of MIPS Machine Code

Gordon, Katz, McIntosh, Wang

Efficiency vs. Generality



Domain specific languages that approximate certain high level languages.

Constructions tailored towards particular applications.

Machine code / legacy code

Legacy Code

Moving to the RAM model offers the possibility of securely emulating real architectures.

In theory, we can support "real" languages, their existing libraries, and existing compilers.

What would this take in practice?

Ideal world: the programmer has never heard the words "secure computation".

Oblivious RAM [GO96,…]

client $(v_1, d_1), (v_2, d_2) \ldots., (v_n, d_n)$ server

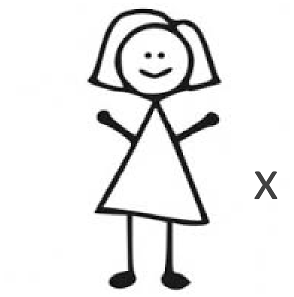
access pattern 1: $(r, v_5), (r, v_2), (w, v_2, d_1) \ldots., (w, v_7, d_2)$

access pattern 2: $(r, v_1), (r, v_1), (r, v_1) \ldots., (r, v_1)$

ANY 2 access patterns are indistinguishable

ORAM in secure computation

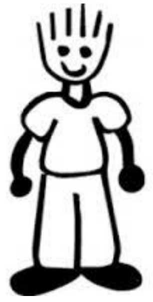
Who should hold the ORAM?
Recall, the client fetches items from the server.
Alice shouldn't see Bob's items, and Bob Shouldn't see Alice's.



x

F(x,y)

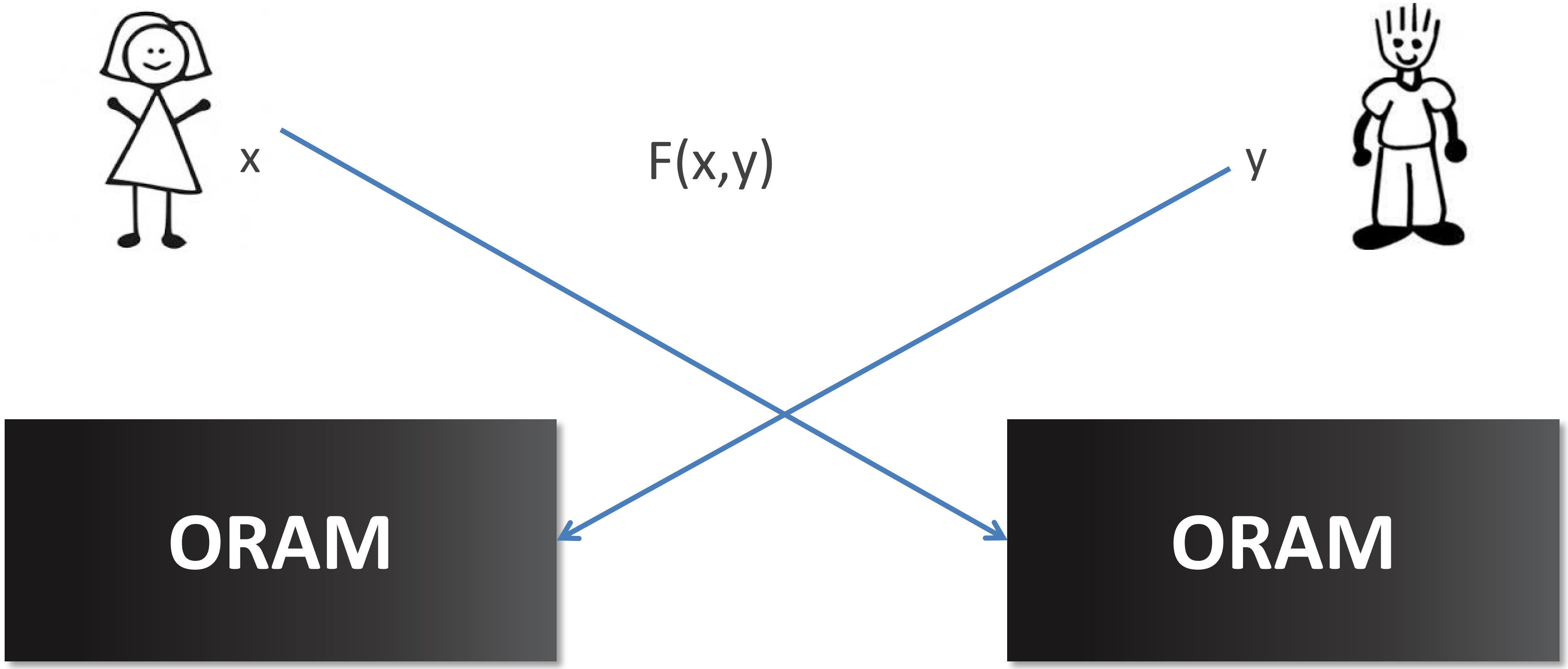
y



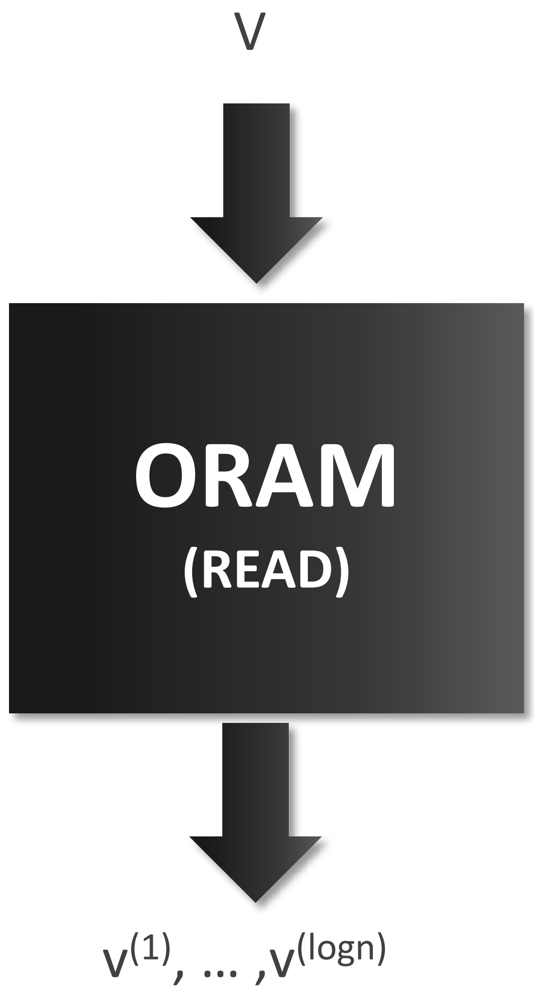
ORAM

ORAM in Secure Computation

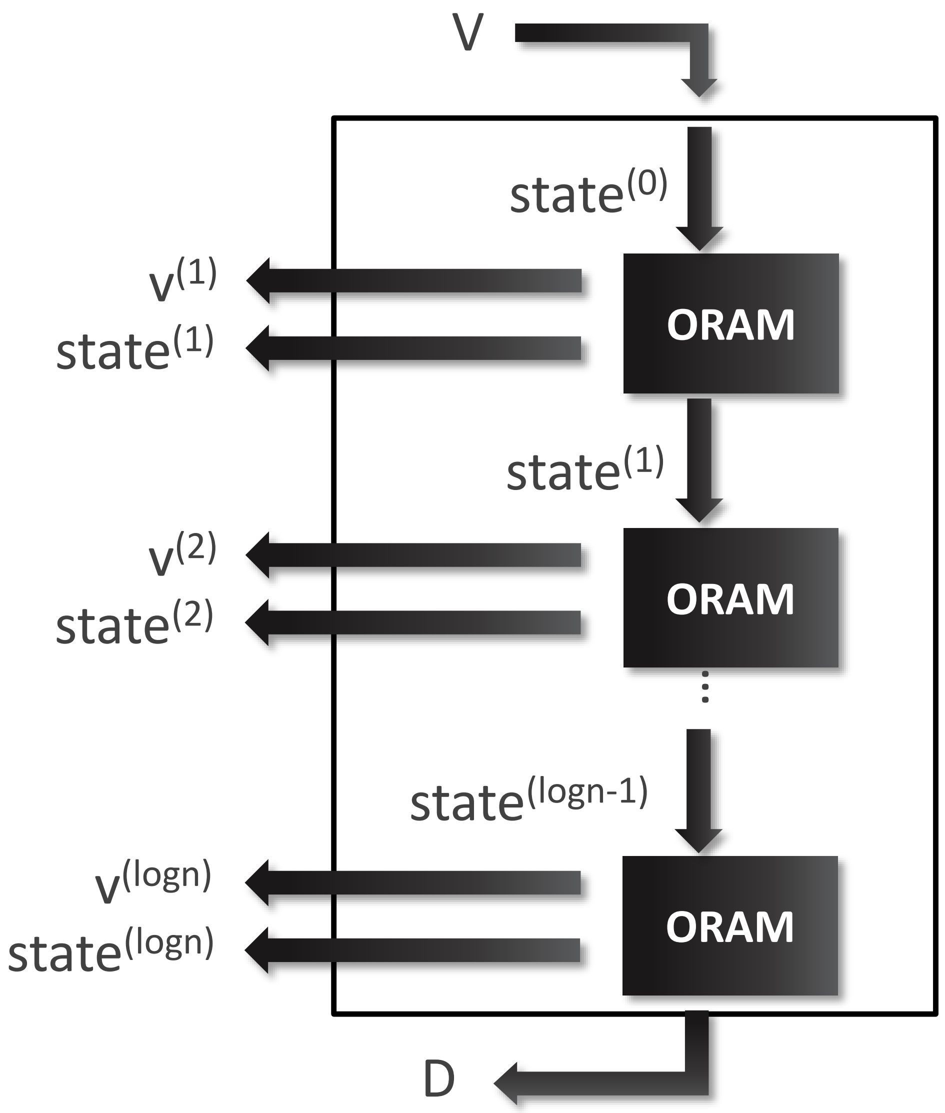
But even if Alice sees which of her **own** items are fetched
she learns something about y.
(Consider a binary search for y among the items in X)



Oblivious RAM (abstraction)



1 of the log n will match, output D

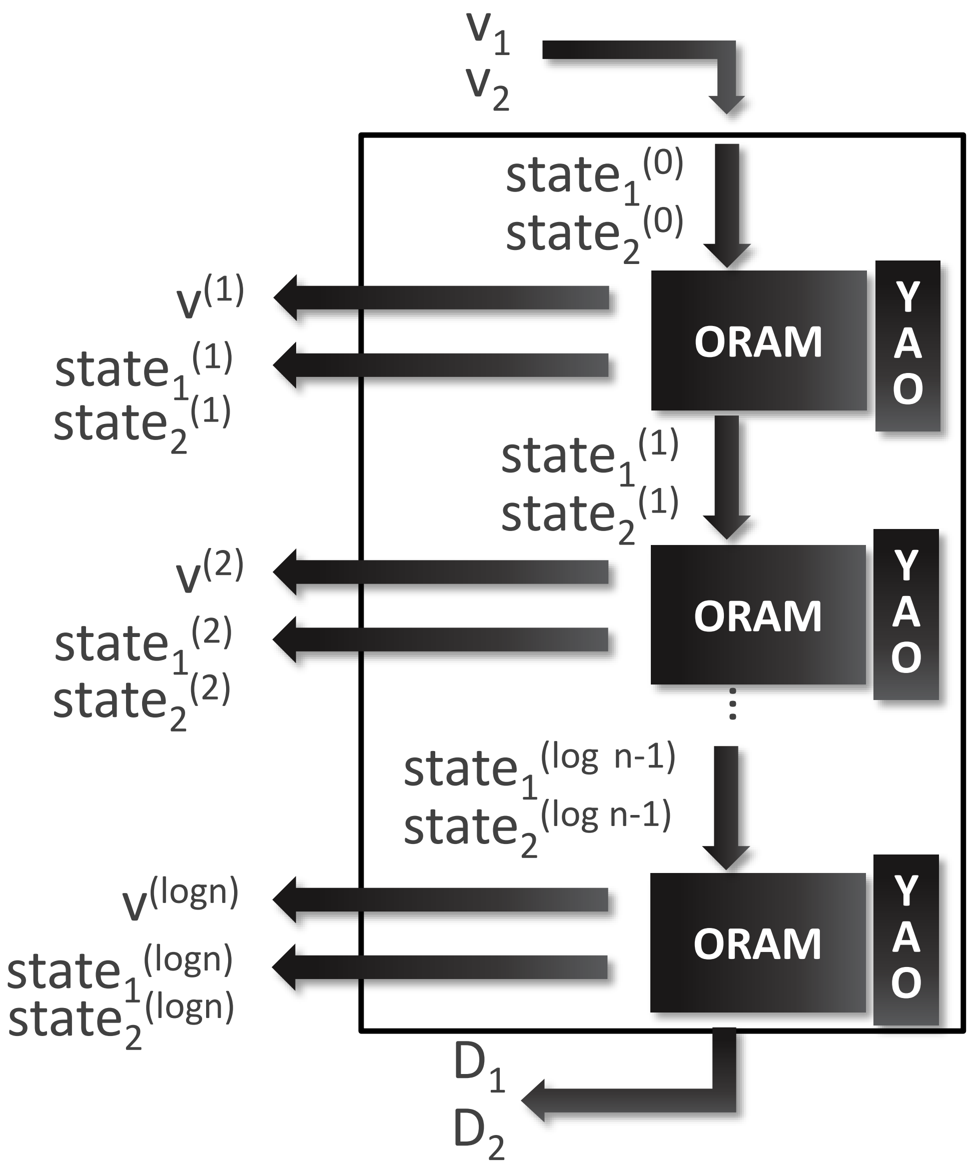


CCS 2012

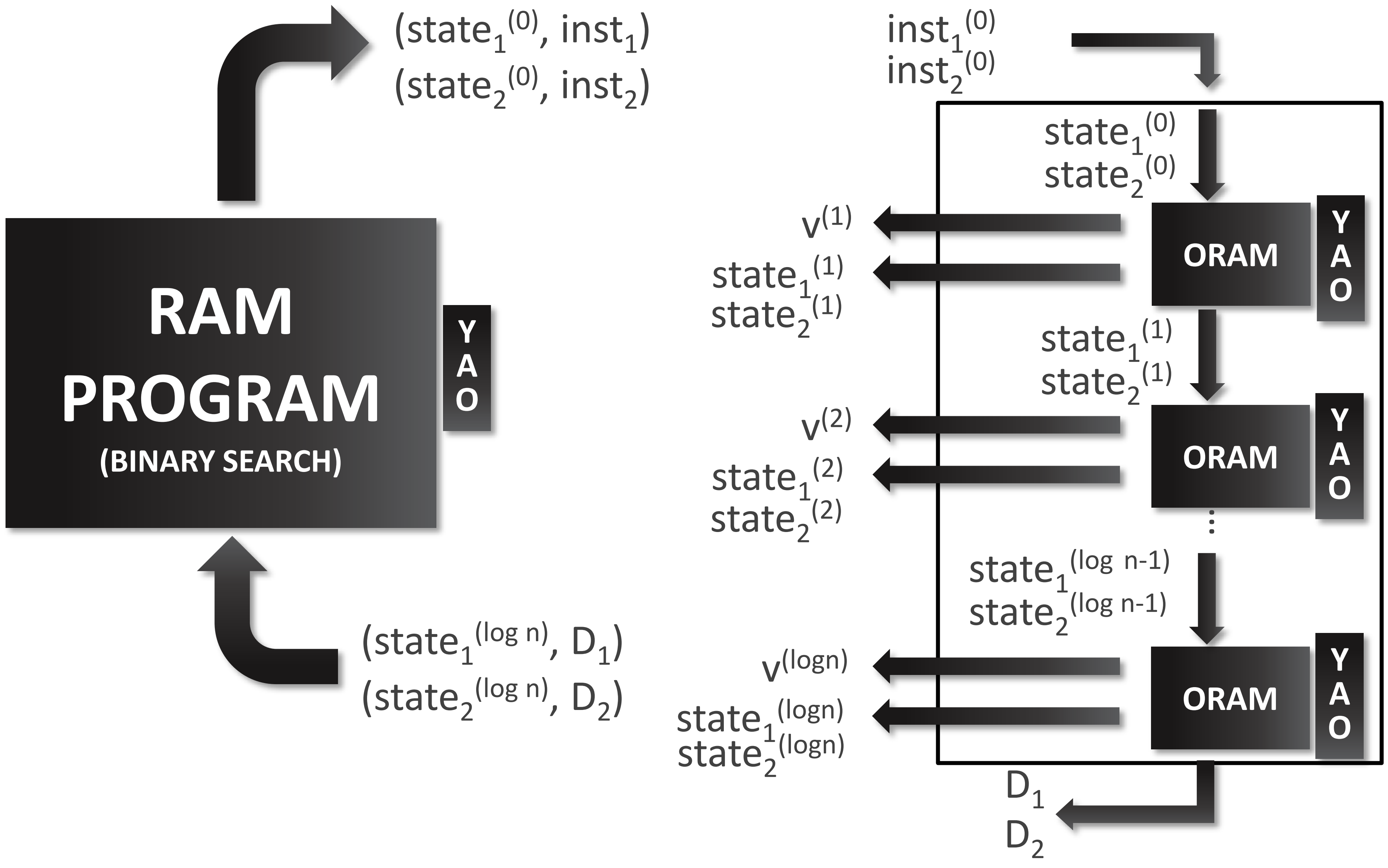
"secret shares"

$v_1 \oplus v_2 = v$

$state_1 \oplus state_2 = state$

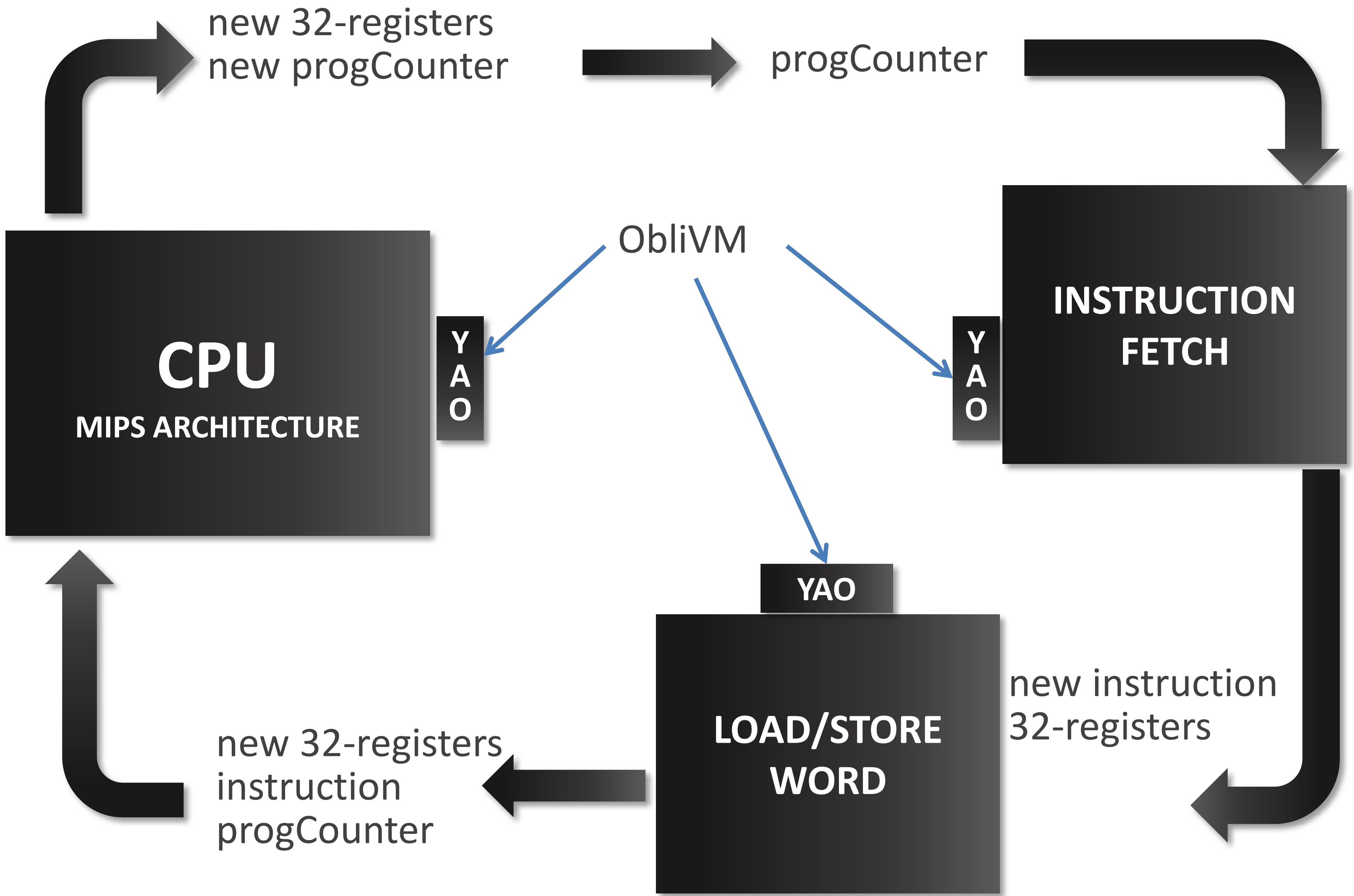


CCS 2012

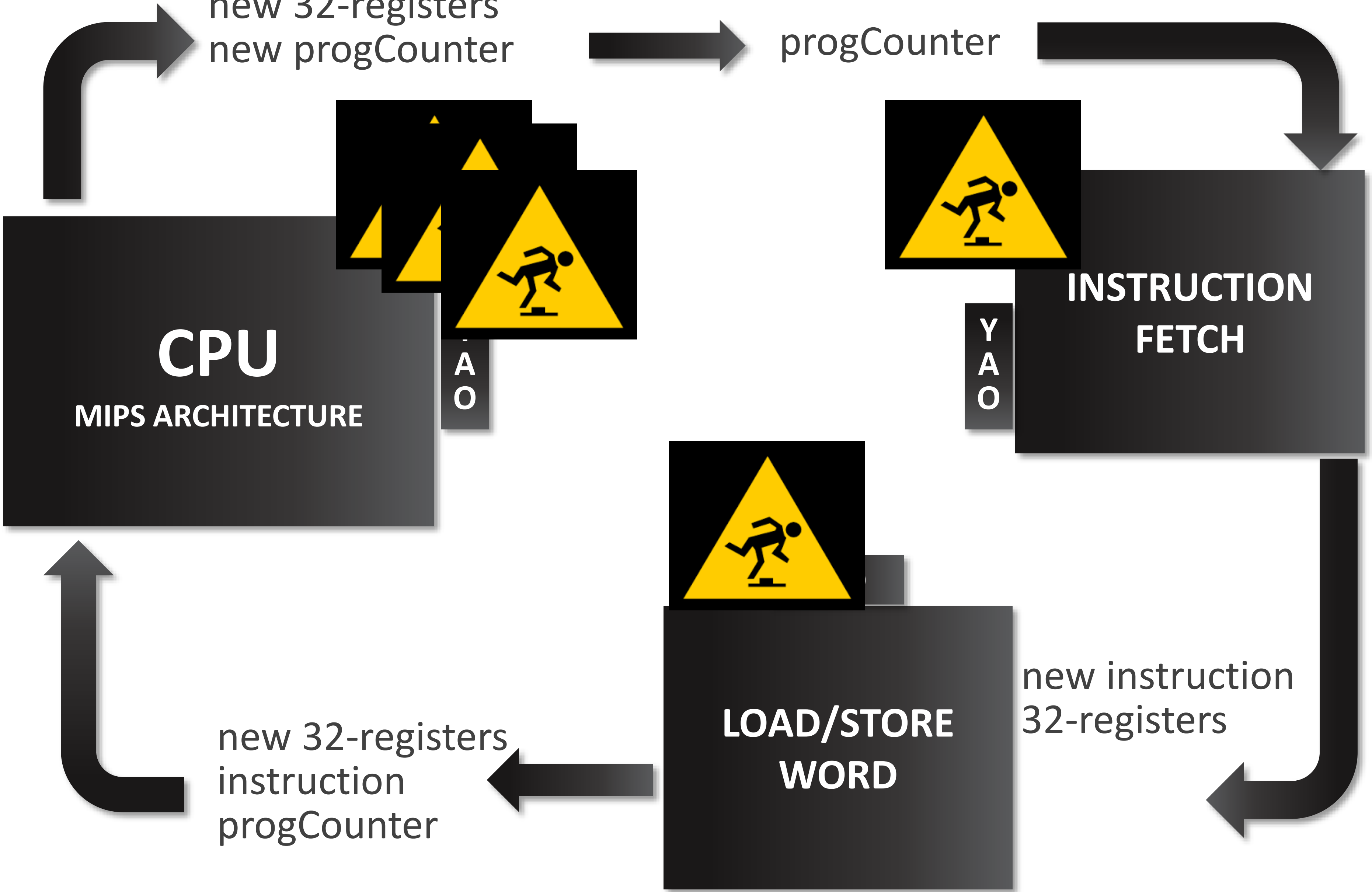


RAM PROGRAM
(BINARY SEARCH)
YAO
$(\text{state}_1^{(0)}, \text{inst}_1)$
$(\text{state}_2^{(0)}, \text{inst}_2)$
$(\text{state}_1^{(\log n)}, D_1)$
$(\text{state}_2^{(\log n)}, D_2)$
$\text{inst}_1^{(0)}$
$\text{inst}_2^{(0)}$
$\text{state}_1^{(0)}$
$\text{state}_2^{(0)}$
ORAM
YAO
$v^{(1)}$
$\text{state}_1^{(1)}$
$\text{state}_2^{(1)}$
$\text{state}_1^{(1)}$
$\text{state}_2^{(1)}$
ORAM
YAO
$v^{(2)}$
$\text{state}_1^{(2)}$
$\text{state}_2^{(2)}$
$\text{state}_1^{(\log\ n-1)}$
$\text{state}_2^{(\log n-1)}$
ORAM
YAO
$v^{(\log n)}$
$\text{state}_1^{(\log n)}$
$\text{state}_2^{(\log n)}$
D_1
D_2

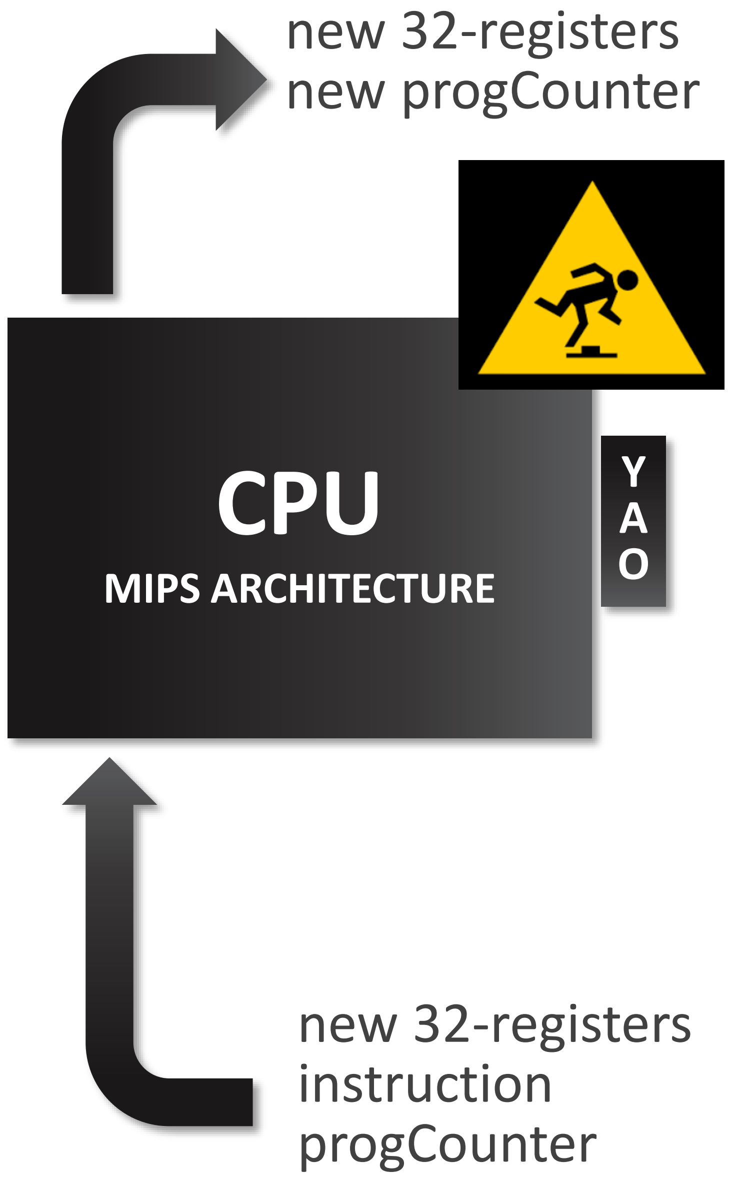
Current Work (DARPA: PROCEED)



Current Work



Current Work



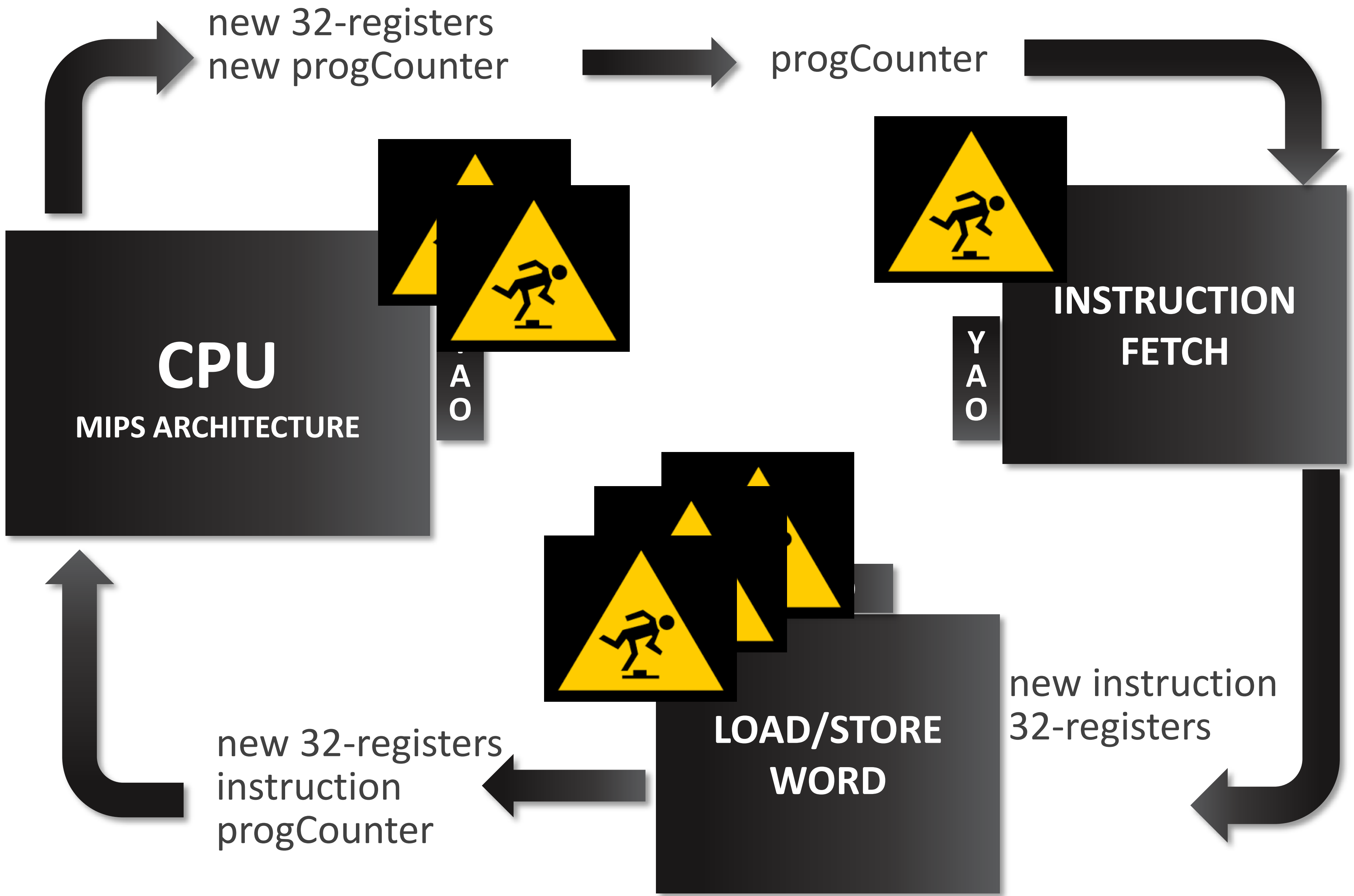
Why MIPS?

- Fixed register space = fixed circuit.
- With approximately 15 instructions, we can compute:
Djikstra, longest common sub-string, set-intersection, stable marriage, binary search, decision trees…
- Easy to implement!
- 15 instructions = small circuit.
- We first proposed LLVM, but instructions in LLVM are polymorphic objects.

On the other hand:

- ARM or x86 would give bigger circuits, but smaller programs. Ultimately, I don't know which is best.

Current Work



Component Run-Times

ALU

$\sim$ 15 instructions $\sim$ 7K AND gates

Memory

Fetch from 1024 32-bit words $\sim$ 43K AND gates

Improvement #1: Instruction Mapping

Divide all instructions into separate “banks”

Bank_i contains instructions that **could** be executed in the i^{th} cycle.

Instruction Mapping

```
If (x > 5)
    instr1
    instr2
else
    instr3
    instr4
```

If x is tainted:
instr1 and instr3 must go in the same ORAM bank

and
instr2 and instr4 must go in the same ORAM bank

```
for (i = 1 to x)
    instr1
    instr2
end for
instr3
instr4
instr5
```

t = 1: instr1

t = 2: instr2

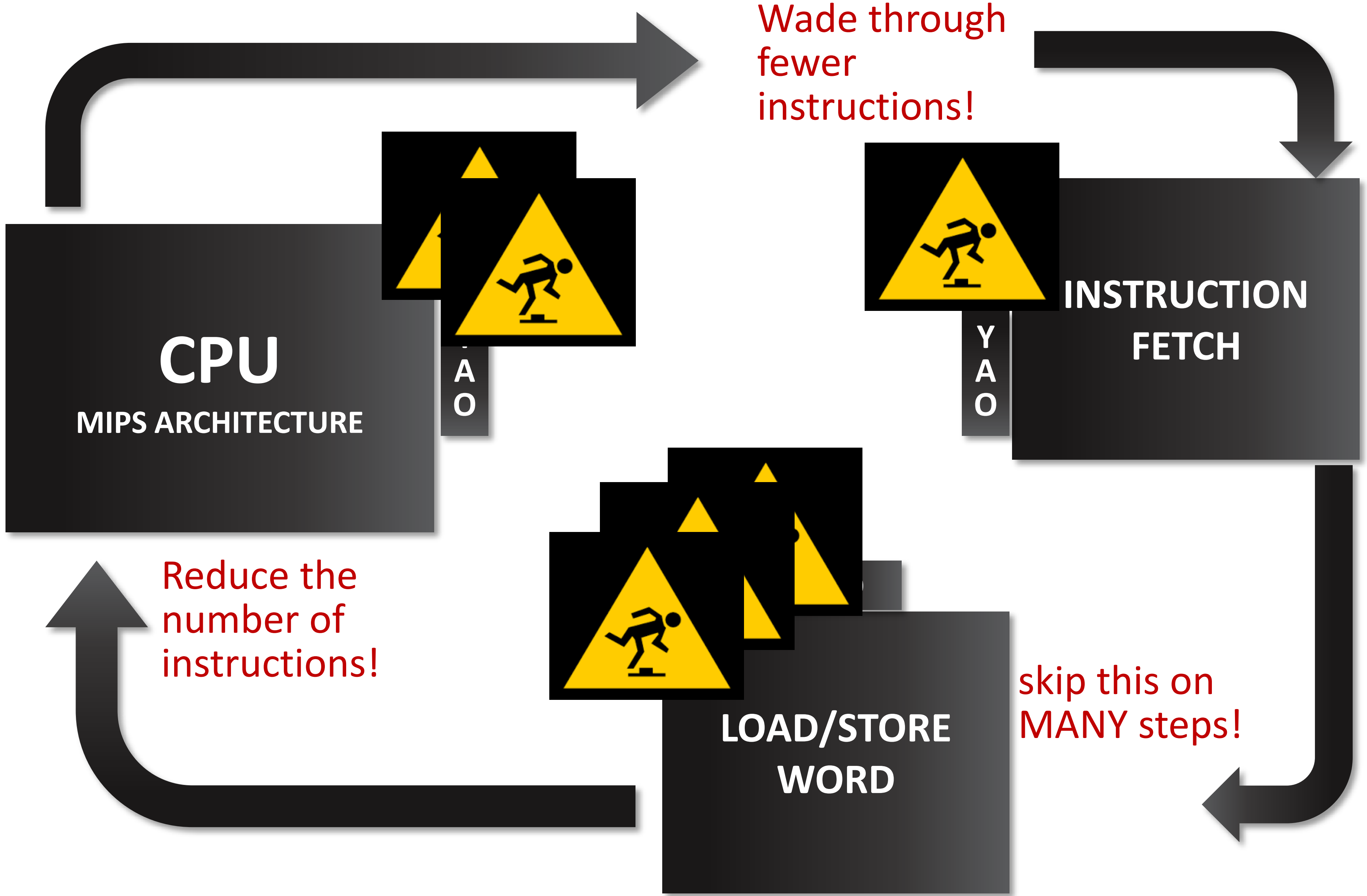
t = 3: instr1 or instr3

t = 4: instr2 or instr4

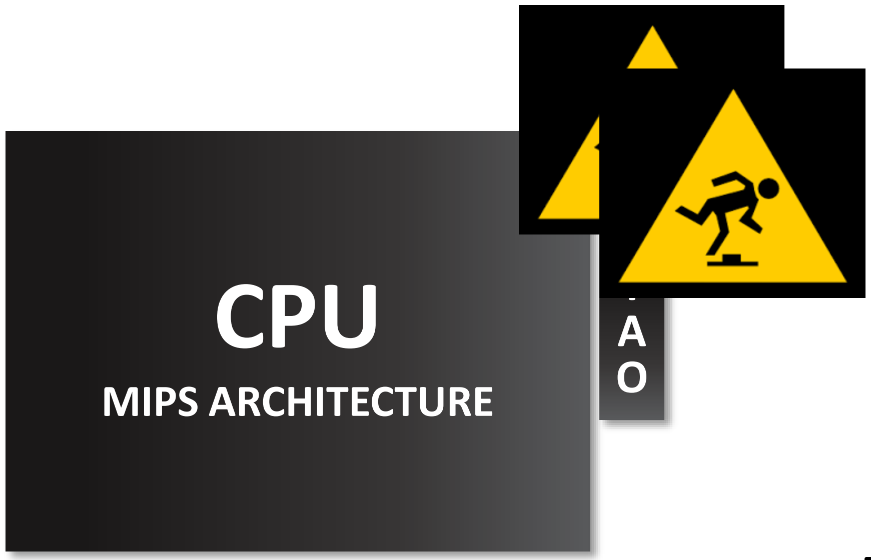
t = 5: instr1 or instr3 or intsr5

loop size t, program length n:
n/t banks, each of size t.

Instruction Mapping



Instruction Mapping



Reduce the number of instructions!

<u>Set Intersection:</u>

Reduces the average ALU size from 6727 to 1848 AND gates. (3.5X)

Instruction Mapping

Wade through fewer instructions!

<u>Set Intersection:</u>
The full program has about 150 instructions.
The largest instruction bank after mapping has 31 instructions.
More than half the instruction banks have fewer than 20 instructions.

Instruction Mapping

Unfortunately, even after instruction mapping,
load/store operations still might occur
in almost every time step.

skip this on
MOST steps! ☹

Improvement #2: padding

```
for (i = 1 to x)
  If (x > 5)
      instr1 ←
      instr2 ←
  else
      instr3 ←
      instr4 ←
      instr5 ←
```

If two branches are relatively prime, one of length k_1, the other k_2, then in less than k_1k_2 time-steps, we will cover the entire loop.

By padding branches such that the lengths are relatively composite, we can greatly reduce the number of instructions per bank:

for set intersection, we go from ~ 40 down to 4.

Padding

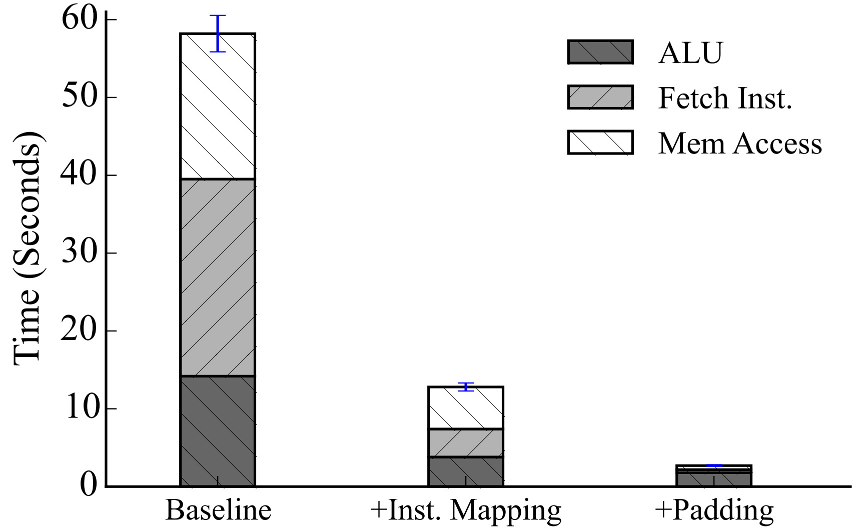
We padded 2 of 3 branches that appear in the main loop using a total of 6 NOP instructions.

Before padding we found that a load/store operation might be executed in almost every time step.

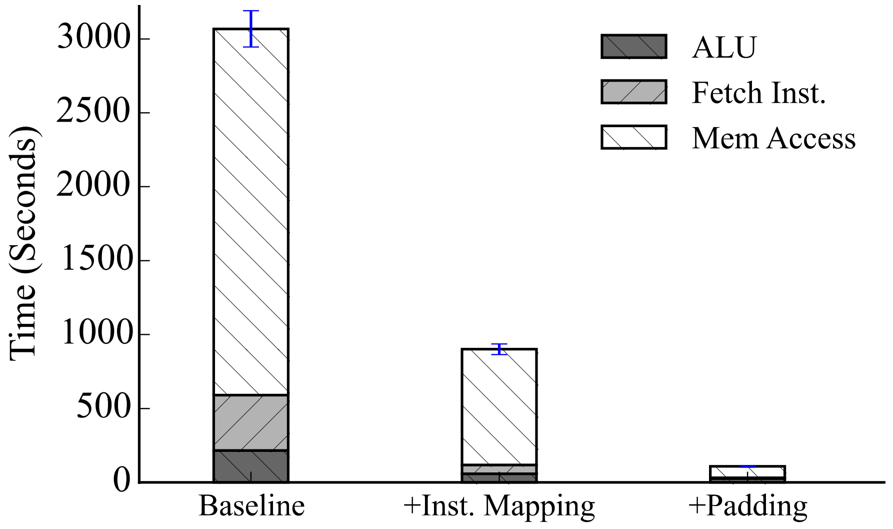
After padding, we find that for only 1/10 of all time steps require a load/store operation.

skip this on MOST steps! ☺

Set Intersection

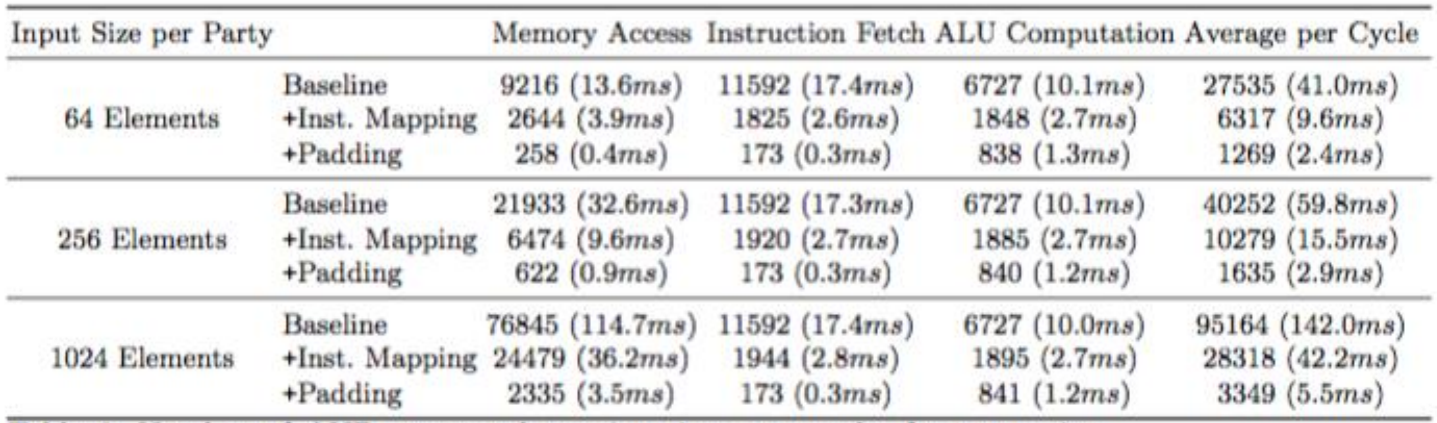


Run-time decomposition for computing set-intersection size when each party's input consists of 64 32-bit integers.



Run-time decomposition for computing set-intersection size when each party's input consists of 1024 32-bit integers.

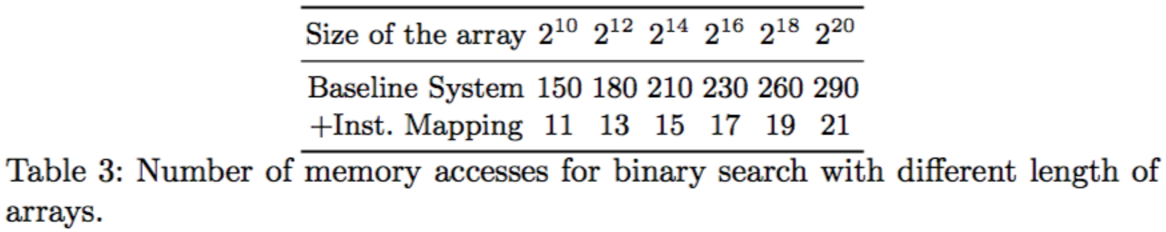
Set Intersection



Input Size per Party		Memory Access	Instruction Fetch	ALU Computation	Average per Cycle
64 Elements	Baseline	9216 (13.6ms)	11592 (17.4ms)	6727 (10.1ms)	27535 (41.0ms)
	+Inst. Mapping	2644 (3.9ms)	1825 (2.6ms)	1848 (2.7ms)	6317 (9.6ms)
	+Padding	258 (0.4ms)	173 (0.3ms)	838 (1.3ms)	1269 (2.4ms)
256 Elements	Baseline	21933 (32.6ms)	11592 (17.3ms)	6727 (10.1ms)	40252 (59.8ms)
	+Inst. Mapping	6474 (9.6ms)	1920 (2.7ms)	1885 (2.7ms)	10279 (15.5ms)
	+Padding	622 (0.9ms)	173 (0.3ms)	840 (1.2ms)	1635 (2.9ms)
1024 Elements	Baseline	76845 (114.7ms)	11592 (17.4ms)	6727 (10.0ms)	95164 (142.0ms)
	+Inst. Mapping	24479 (36.2ms)	1944 (2.8ms)	1895 (2.7ms)	28318 (42.2ms)
	+Padding	2335 (3.5ms)	173 (0.3ms)	841 (1.2ms)	3349 (5.5ms)

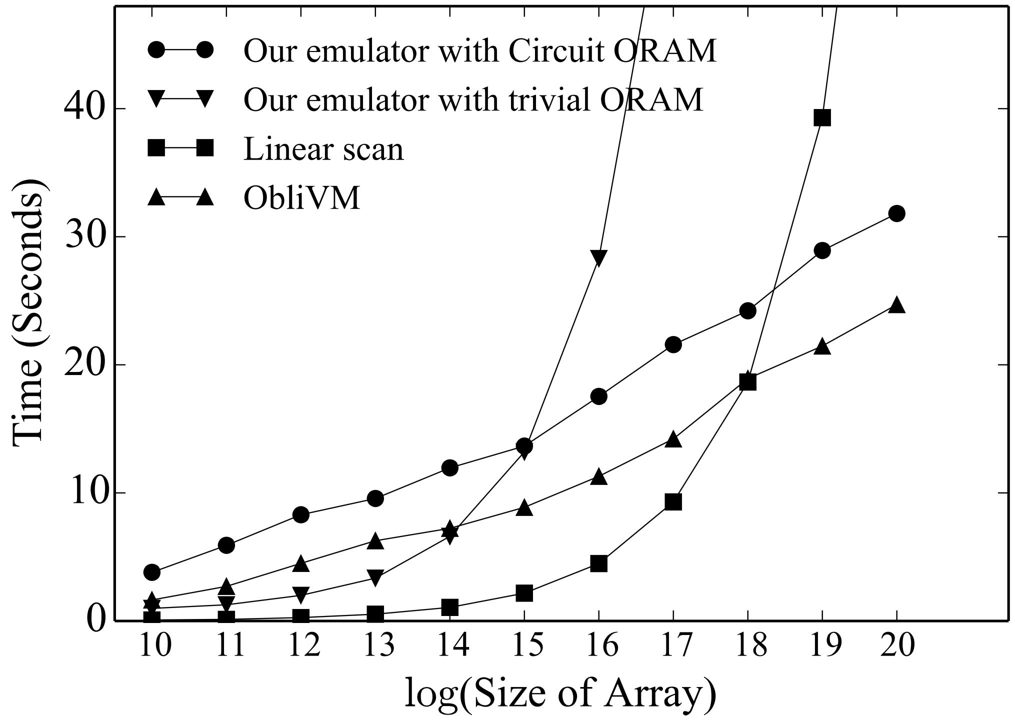
Table 2: Number of AND gates and running time, per cycle, for computing set-intersection size. Sets of 32-bit integers with different sizes are used.

Binary Search



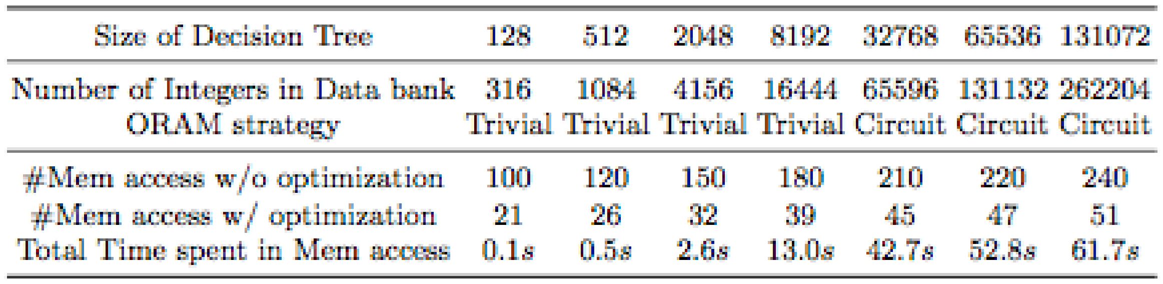
Size of the array	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Baseline System	150	180	210	230	260	290
+Inst. Mapping	11	13	15	17	19	21

Table 3: Number of memory accesses for binary search with different length of arrays.



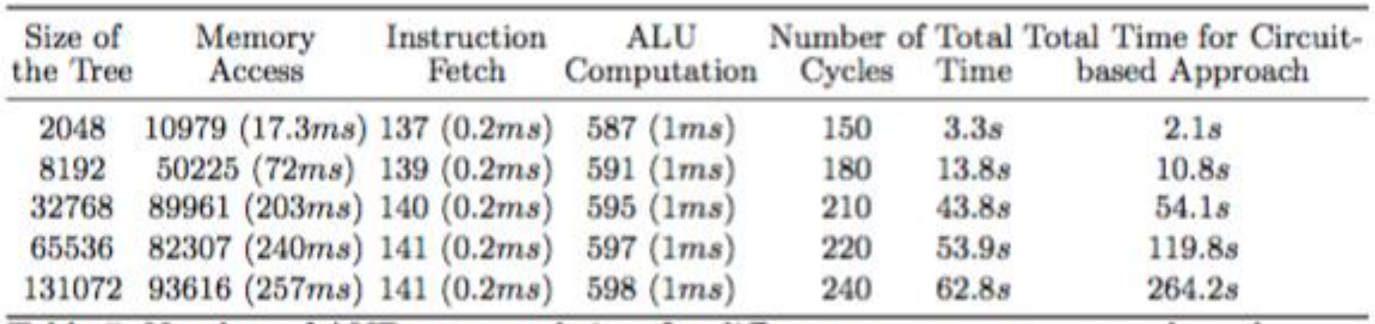
Comparing the performance of secure binary search. One party holds an array of 32-bit integers, while the other holds a value to search for.

Decision Trees



Size of Decision Tree	128	512	2048	8192	32768	65536	131072
Number of Integers in Data bank	316	1084	4156	16444	65596	131132	262204
ORAM strategy	Trivial	Trivial	Trivial	Trivial	Circuit	Circuit	Circuit
#Mem access w/o optimization	100	120	150	180	210	220	240
#Mem access w/ optimization	21	26	32	39	45	47	51
Total Time spent in Mem access	$0.1s$	$0.5s$	$2.6s$	$13.0s$	$42.7s$	$52.8s$	$61.7s$

Table 6: Memory accesses for decision-tree evaluation.



Size of the Tree	Memory Access	Instruction Fetch	ALU Computation	Number of Cycles	Total Time	Total Time for Circuit-based Approach
2048	10979 ($17.3ms$)	137 ($0.2ms$)	587 ($1ms$)	150	$3.3s$	$2.1s$
8192	50225 ($72ms$)	139 ($0.2ms$)	591 ($1ms$)	180	$13.8s$	$10.8s$
32768	89961 ($203ms$)	140 ($0.2ms$)	595 ($1ms$)	210	$43.8s$	$54.1s$
65536	82307 ($240ms$)	141 ($0.2ms$)	597 ($1ms$)	220	$53.9s$	$119.8s$
131072	93616 ($257ms$)	141 ($0.2ms$)	598 ($1ms$)	240	$62.8s$	$264.2s$

Table 5: Number of AND gates and time for different components per cycle and total running time, for evaluating binary decision trees of different sizes.

A True Universal Circuit

One more benefit of the general approach:
We have a true universal circuit!

1. Compile the private input function to MIPS,
2. Supply a function pointer as input to the emulator.
3. Our optimizations no longer apply: the analysis leaks information.

Thanks!