

Some advances in and use scenarios for practical MPC and ZK



Vlad Kolesnikov

Georgia Tech

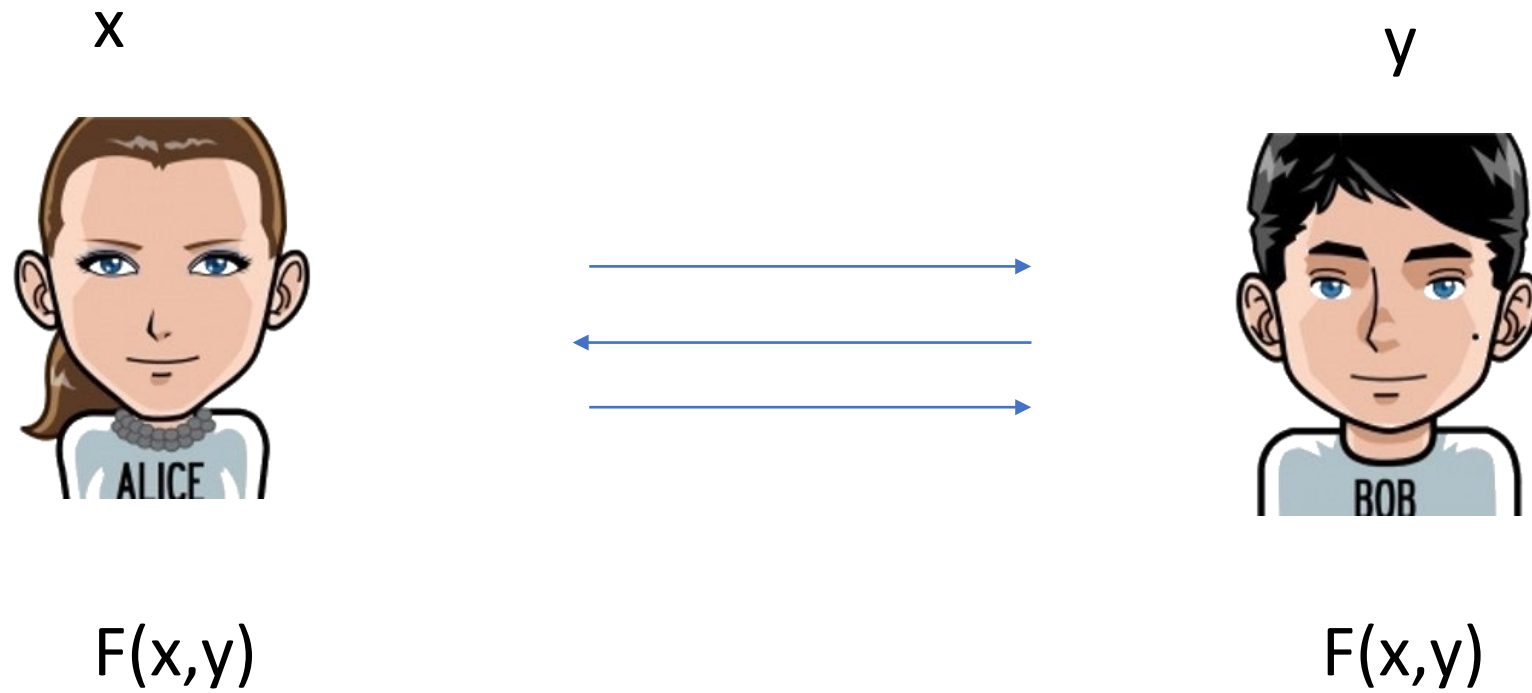
MAPPS Workshop

ICERM, Brown

Outline

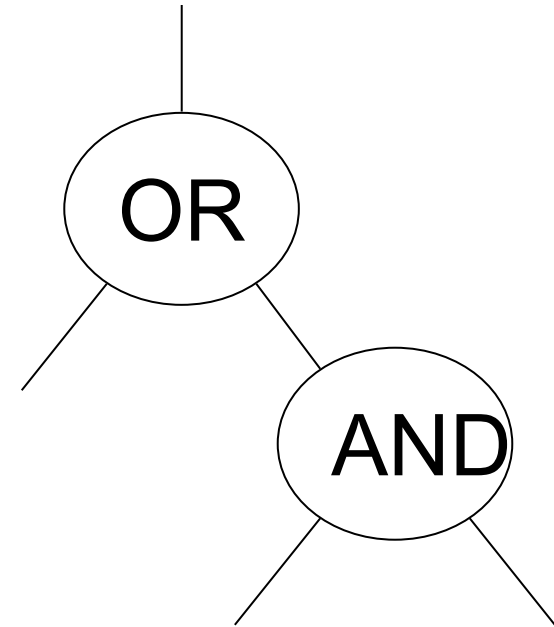
- Garbled Circuits (GC): basic technique for MPC
- Generalization to RAM programs
- Stacked Garbling: free branching
- ZK
- Performance

MPC

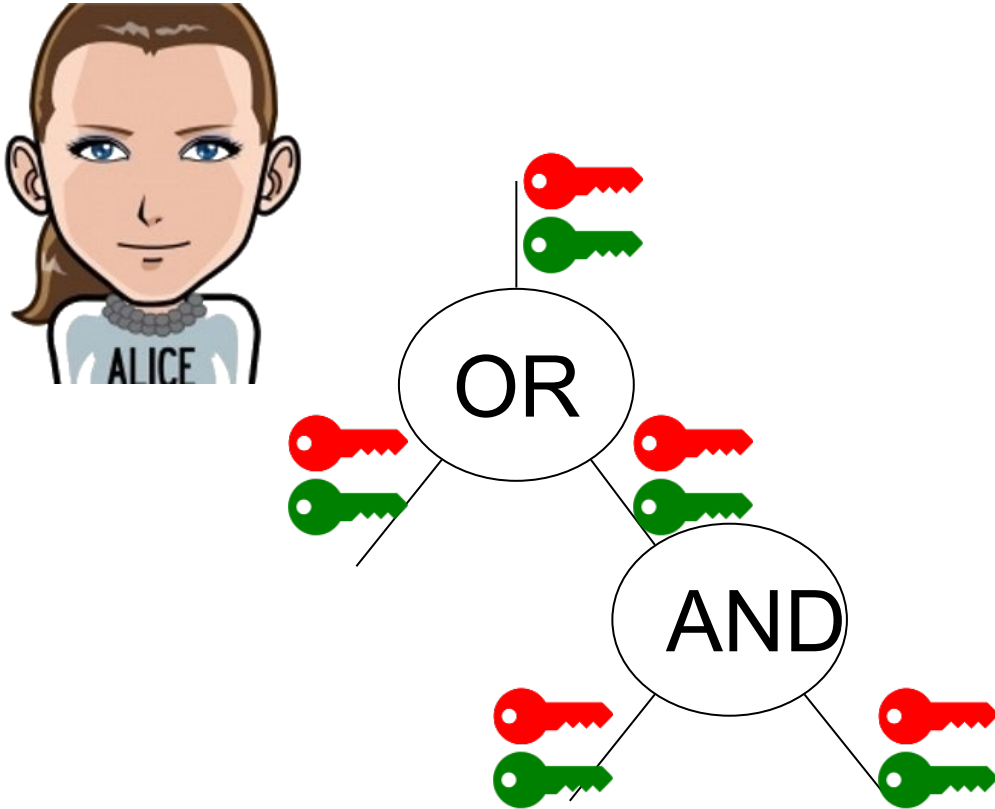


Functions are Boolean circuits

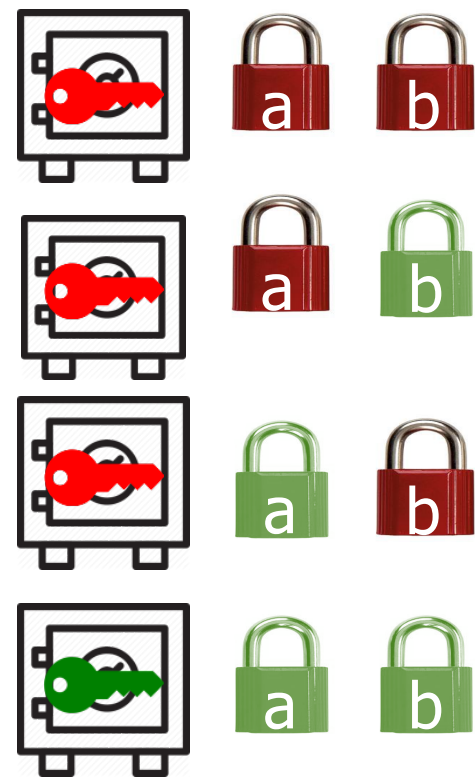
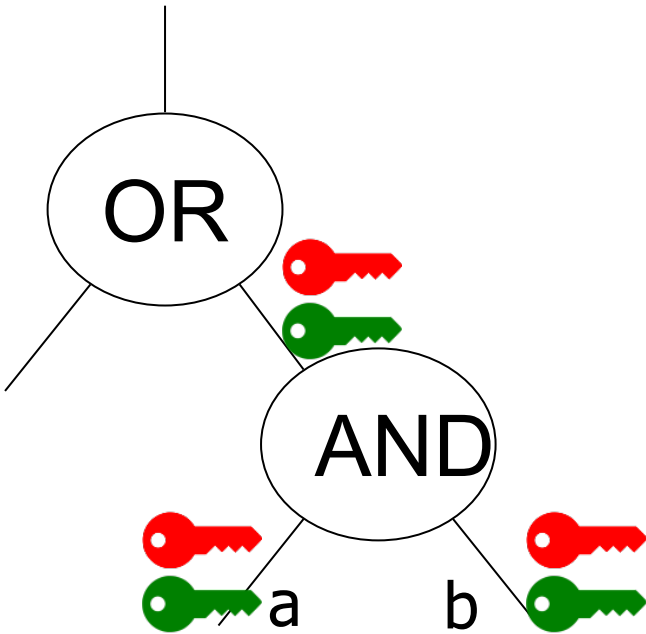
$F(x, y)$



GC intuition: computing on encrypted values

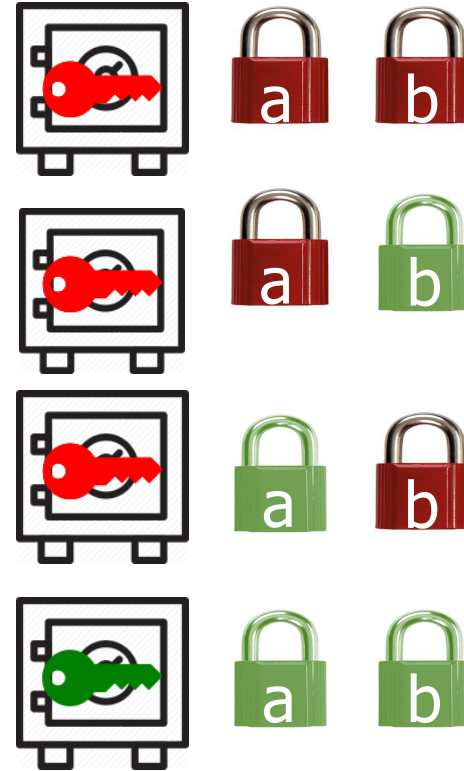
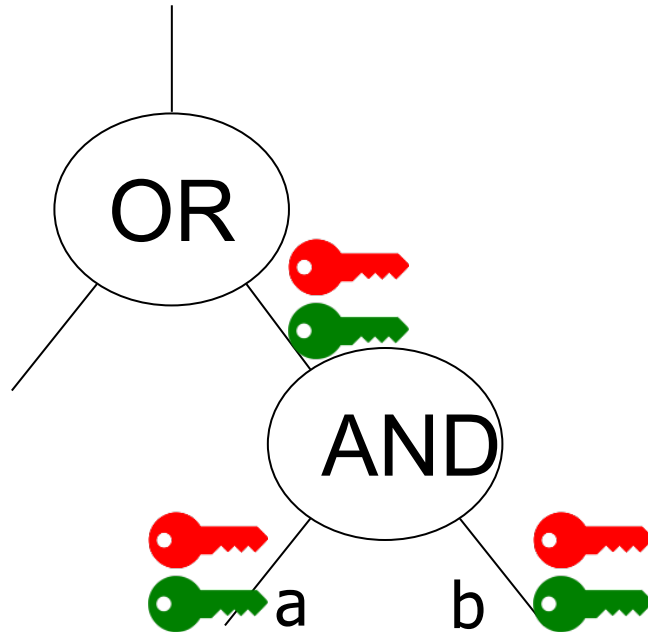


GC intuition: computing on encrypted values



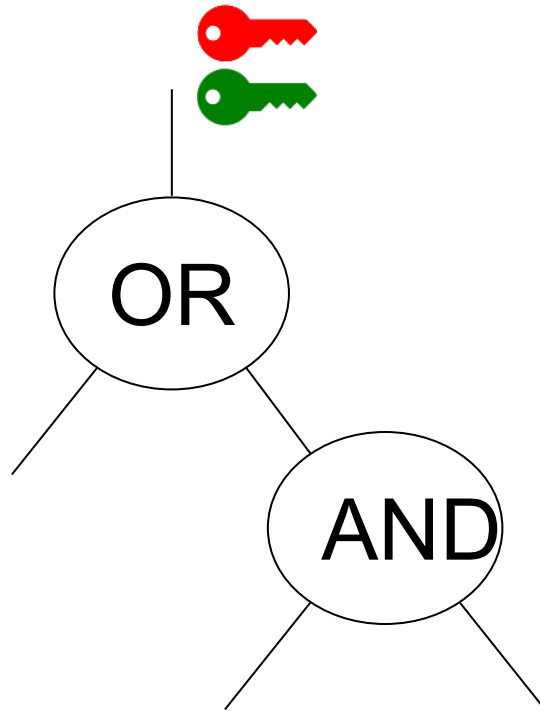
a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1



GC intuition: computing on encrypted values



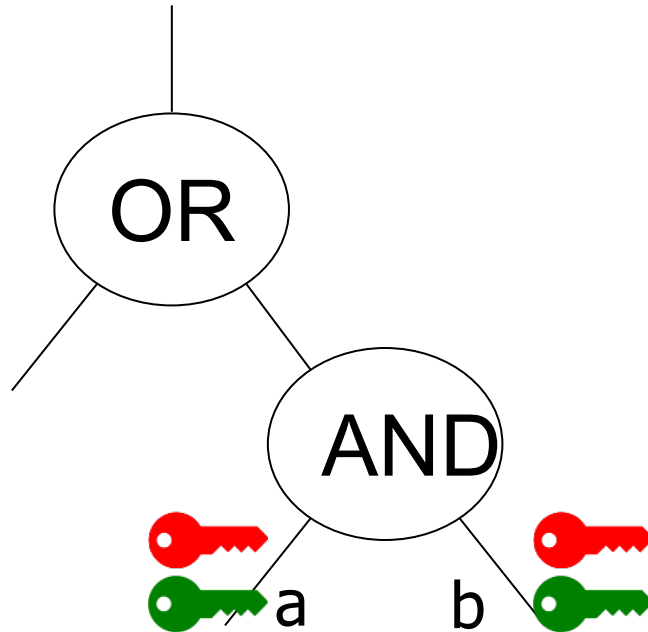
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

GC intuition: decoding encrypted output



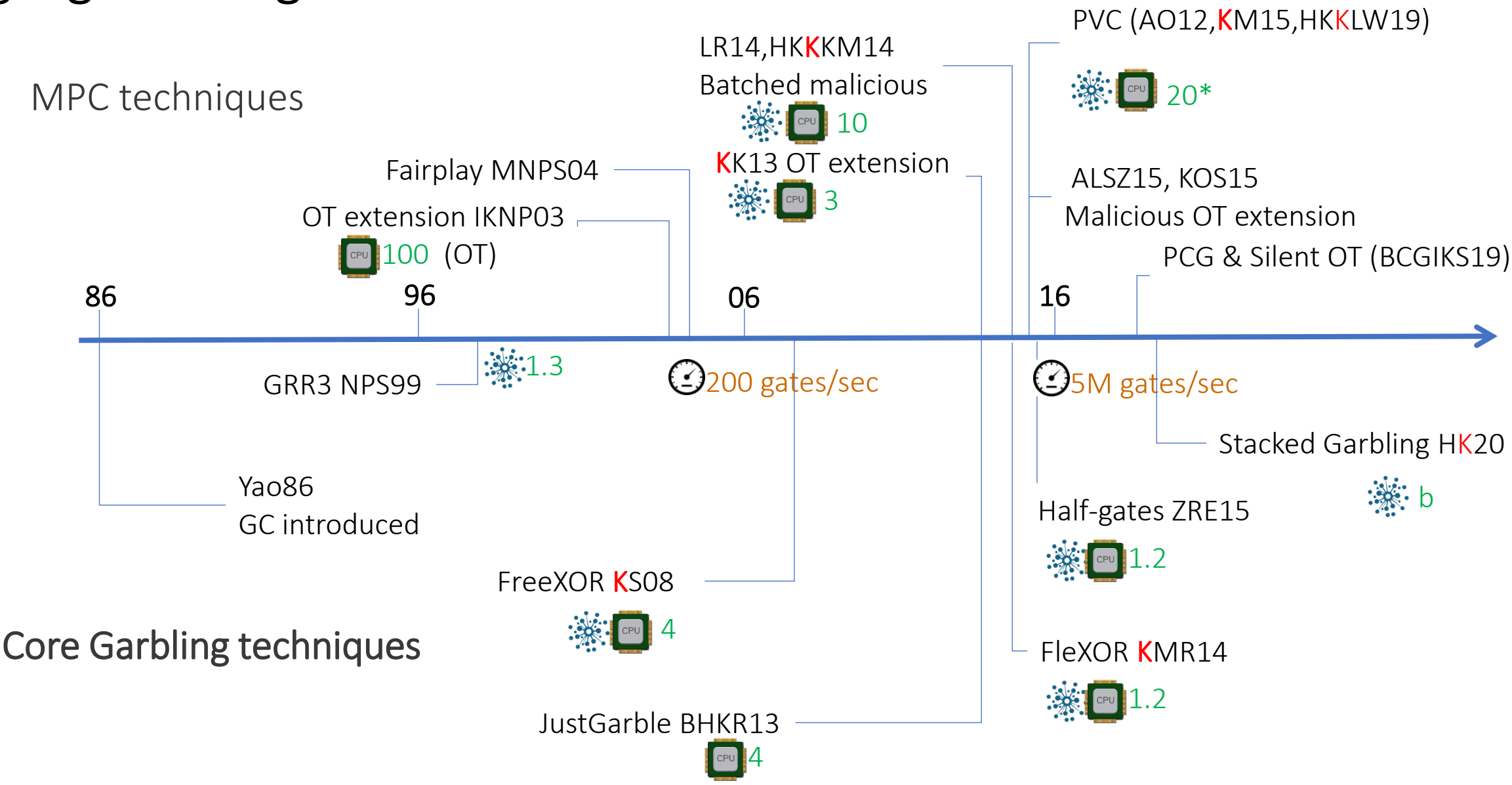
	0
	1

GC intuition: OT for transferring input labels



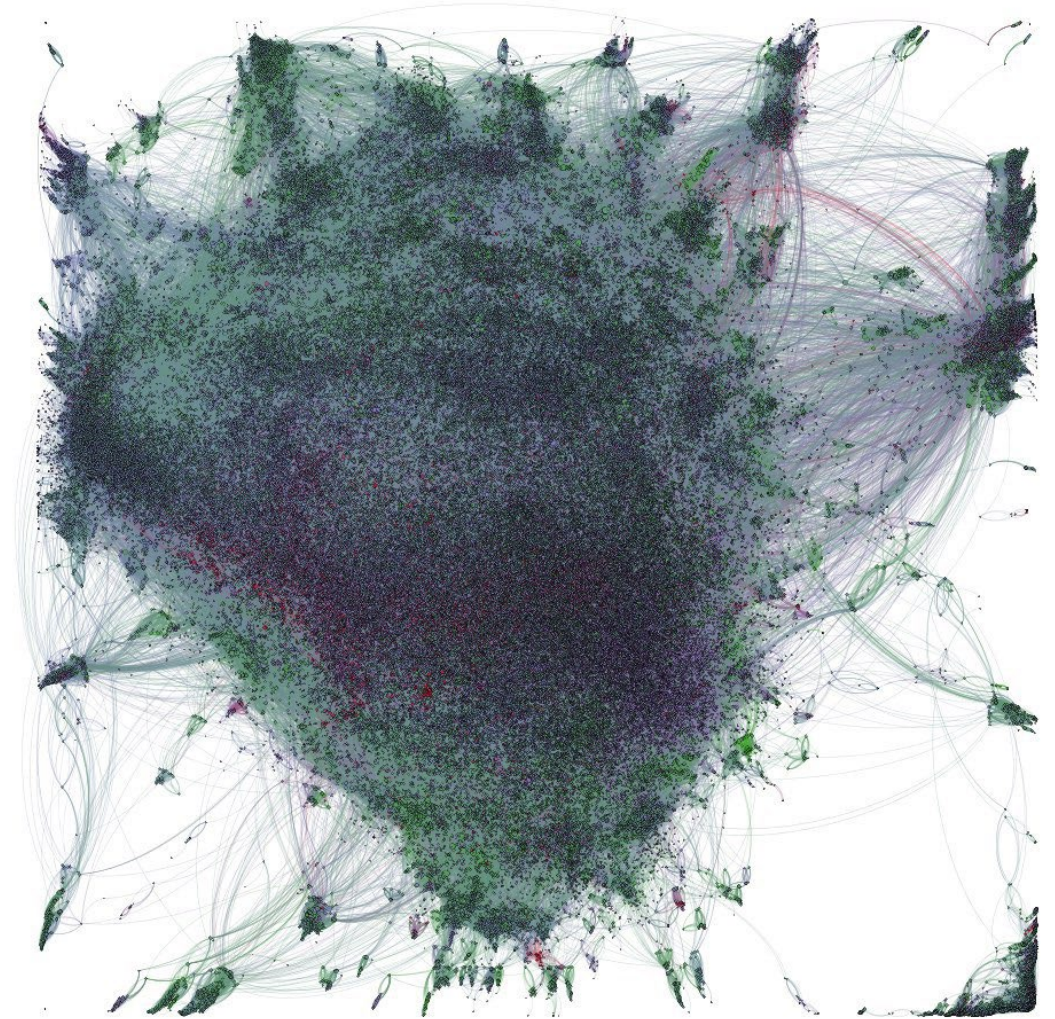
Garbled circuits are pretty stable

Highlights of algorithmic GC advances



Functions are (potentially HUGE) Boolean circuits

KMP, RegEx,...→



Functions are (small) RAM programs

KMP, RegEx,...→
Because of control flow

```
// C++ program for implementation of KMP patt
// algorithm
#include <bits/stdc++.h>

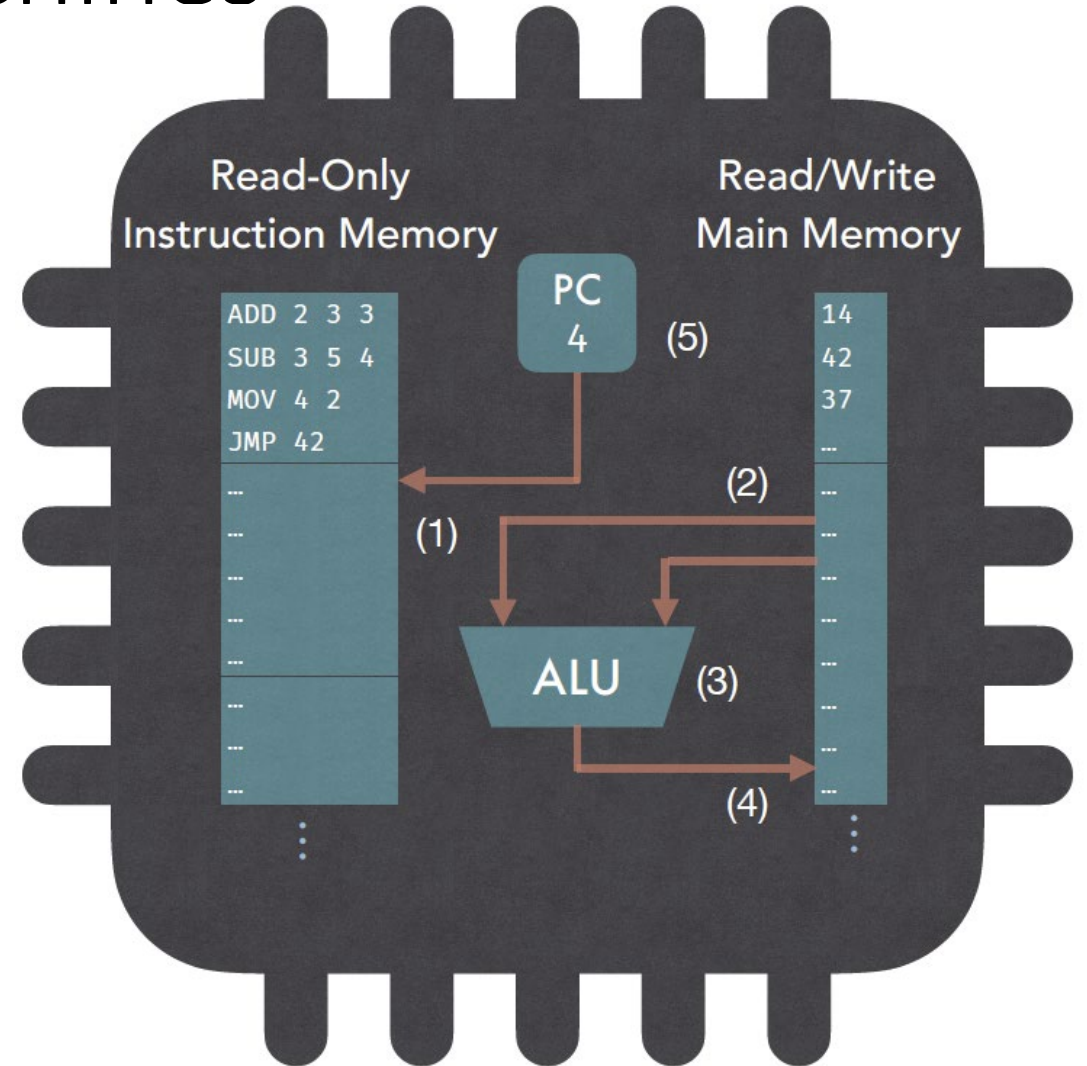
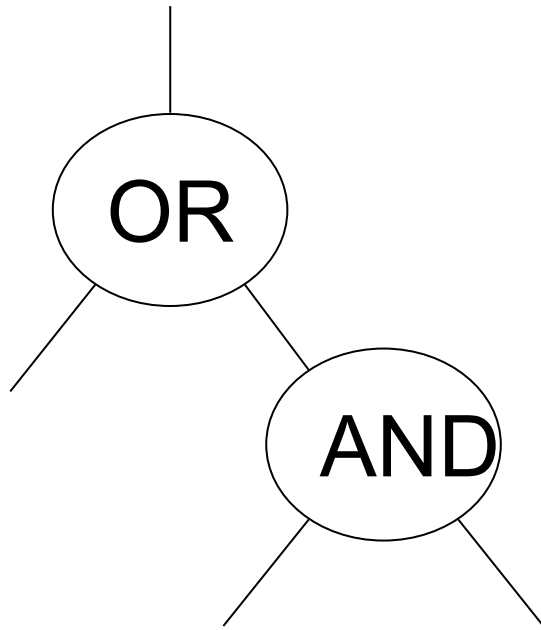
void computeLPSArray(char* pat, int M, int* l
// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[
computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while ((N - i) >= (M - j)) {
        if (pat[j] == txt[i]) {
```

From circuits to RAM machines



Need: compute F
Represent F as circuit vs as a C program

Free branching via Stacked garbling

[K18,HK20ab,21]

- Sequence of works [K18,HK20a,HK20b, HK21]
- Consider circuits with conditionals

Let C_0, C_1 be two arbitrary circuits. The space of circuits is defined as follows:

$$C ::= \text{Netlist}(\cdot) \mid \text{Cond}(C_0, C_1) \mid \text{Seq}(C_0, C_1)$$

Belief (1986-2020): You must pay full price for inactive branches.

Stacked garbling [K18,HK20ab,21]

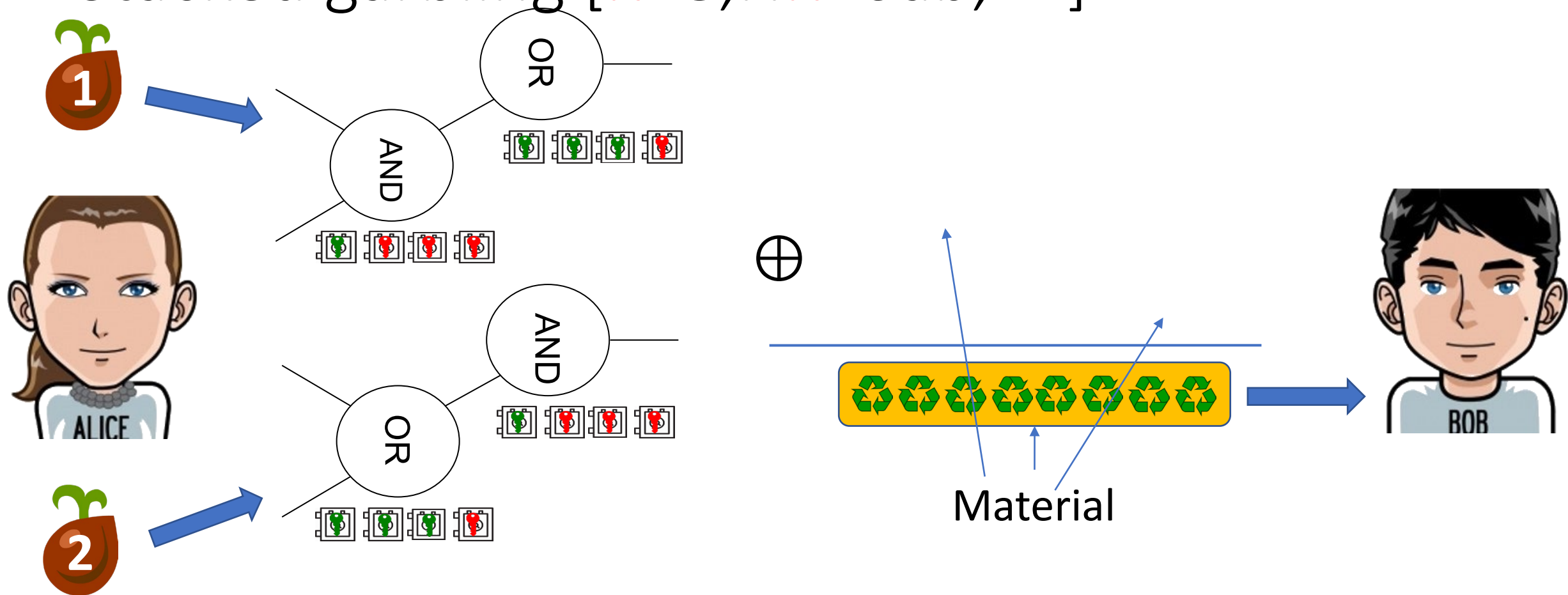
$$C ::= \text{Netlist}(\cdot) \mid \text{Cond}(C0, C1) \mid \text{Seq}(C0, C1)$$

HK20: Can evaluate $\text{Cond}(C0, C1)$ while transmitting only one branch

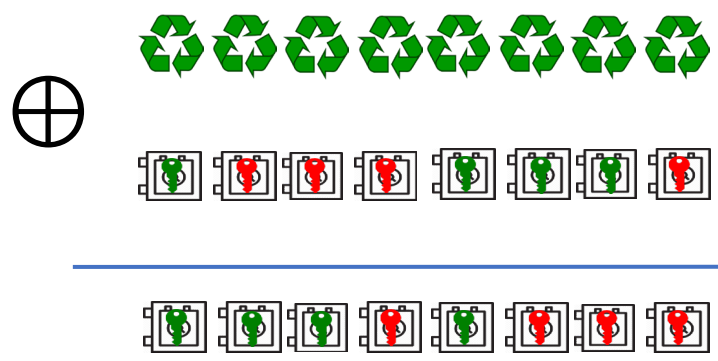
Idea:

- * the same GC *material* M is used for evaluation of $C0$ and $C1$.
- * GC outputs a key to Eval which converts material M to a valid GC and to a random-looking string for inactive branch
- * Eval evaluates both $C0, C1$. One of them will produce garbage labels. They are canceled (garbage-collected) by gadgets constructed by Garbler.
- * **Material reuse** (novel general idea; works for other protocols as well)

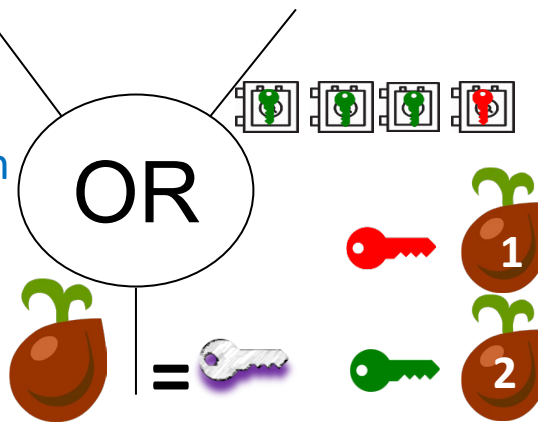
Stacked garbling [K18,HK20ab,21]



Stacked garbling [K18,HK20ab,21]



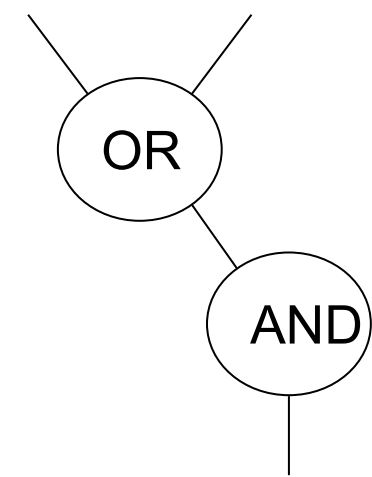
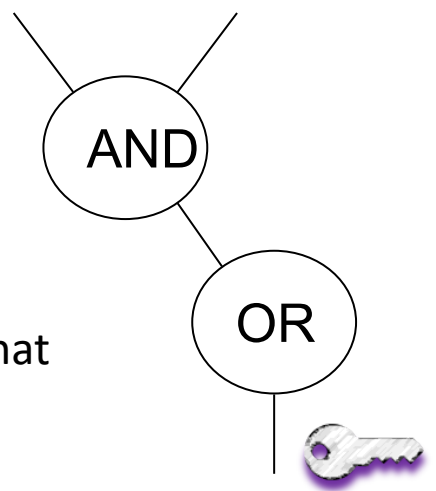
Branch
condition



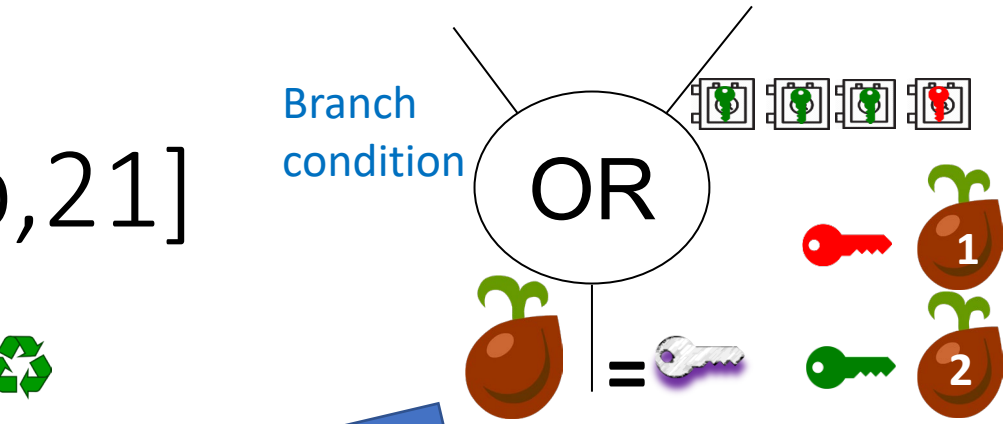
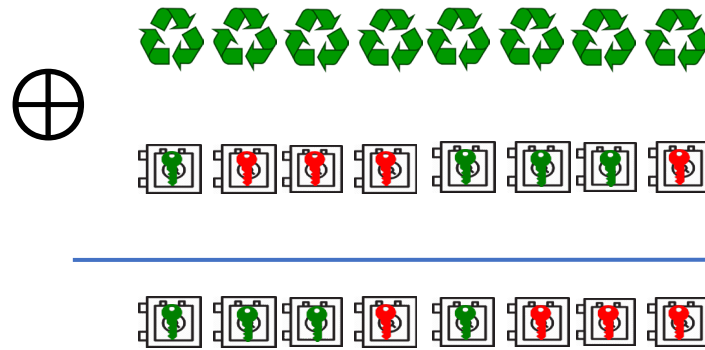
Guess active
branch 1/seed 2



Idea: Alice programs branch condition key so that Bob obtains the seed for the **inactive** branch

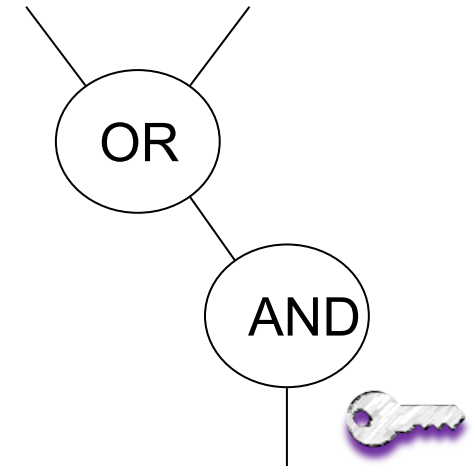
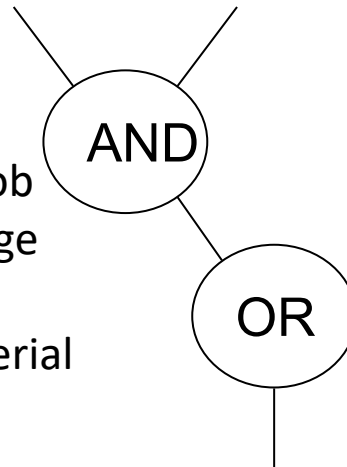


Stacked garbling [K18,HK20ab,21]



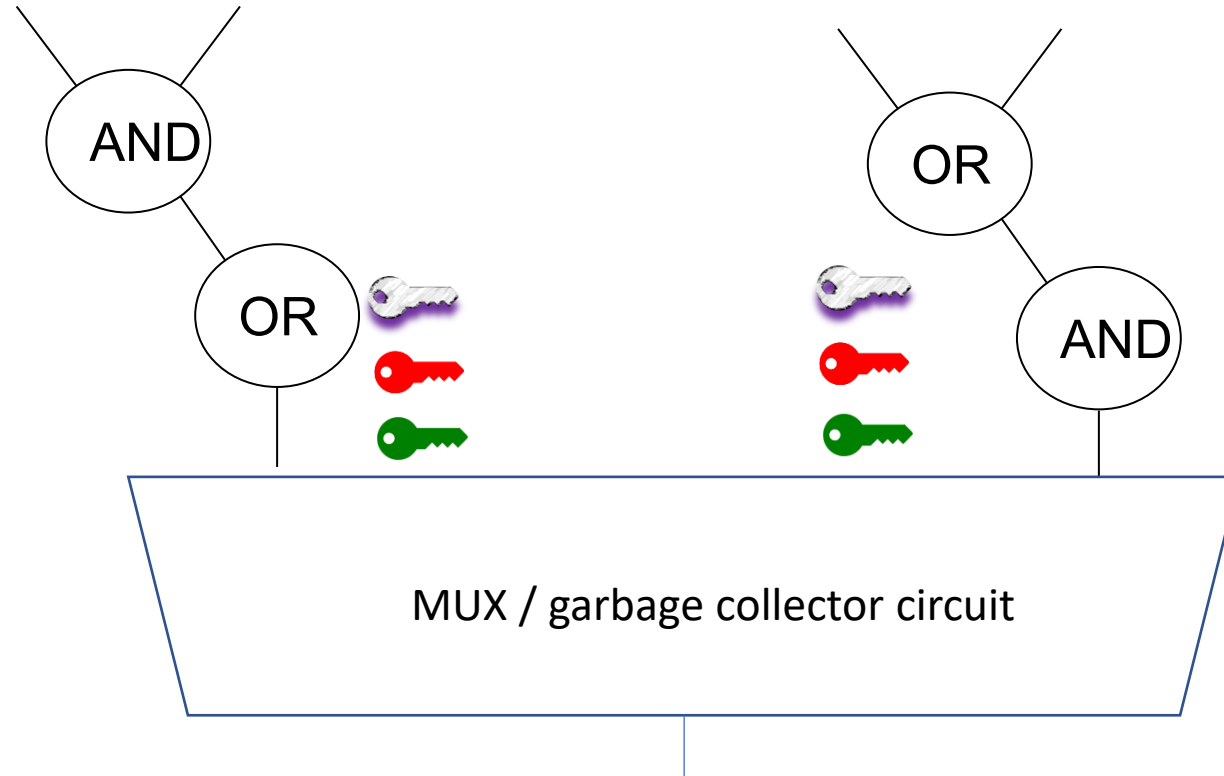
For each branch, if it is active (guess is correct), Bob gets a good output label, otherwise he gets garbage output label.

He can't tell which is which (requires that GC material and labels look random – standard property)



Stacked garbling [K18,HK20ab,21]

For active branch guessed correctly, Bob gets a valid label, otherwise he gets garbage output label.



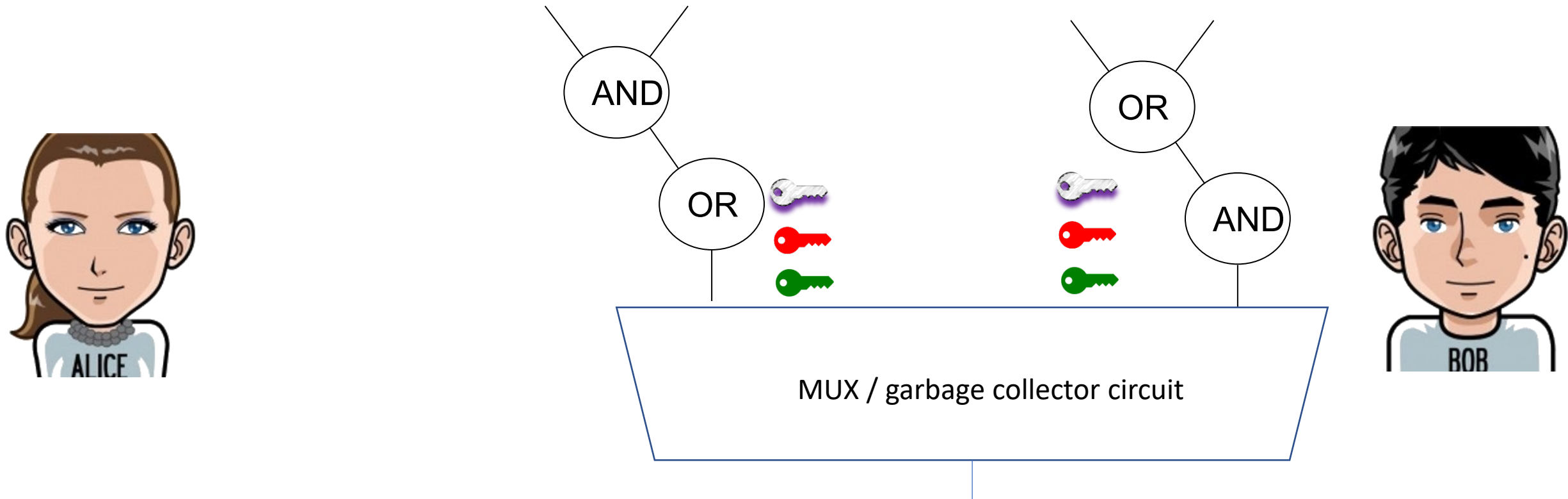
We need to obviously discard garbage of evaluation of one of b branches.

Key idea: Bob is deterministic and Alice can emulate him and *predict* the possible garbage keys

Indeed, for each possible active branch (b options), Bob makes b guesses

Then Alice constructs a MUX gadget which collects garbage ($b \times b$ possible combinations of keys only)

Stacked garbling [K18,HK20ab,21]



Actually, no: In how many ways for Bob to get garbage key?

Guess wrong (1 way) and each possible branch input (2^n ways)

Solution: set input of inactive branch to all 0.

This works.

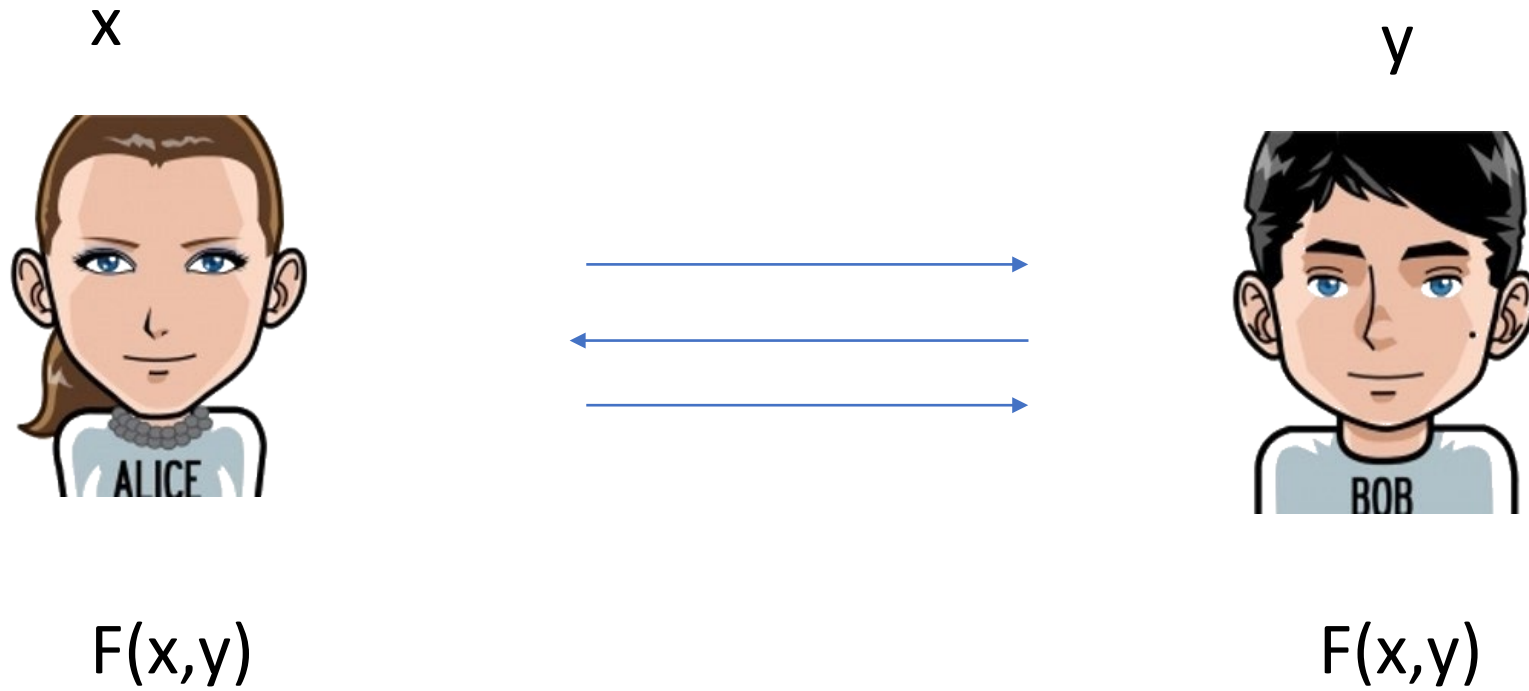
Ok, now we only pay for a single longest branch

In HK21 we show how to achieve computation $O(b \log b)$

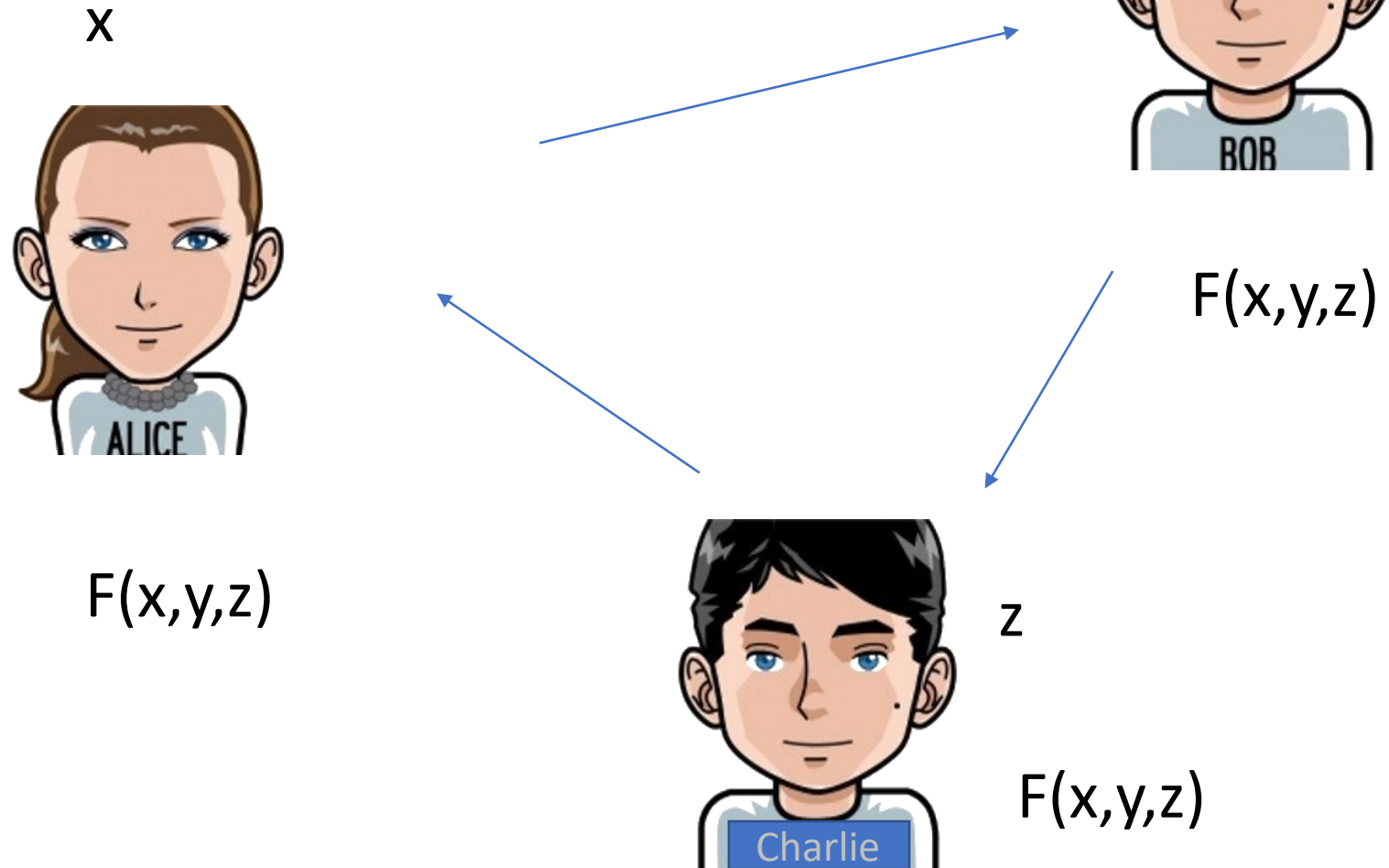
Summary

- Free Branching for GC (incidentally, also for GMW HKP20, HKP21)

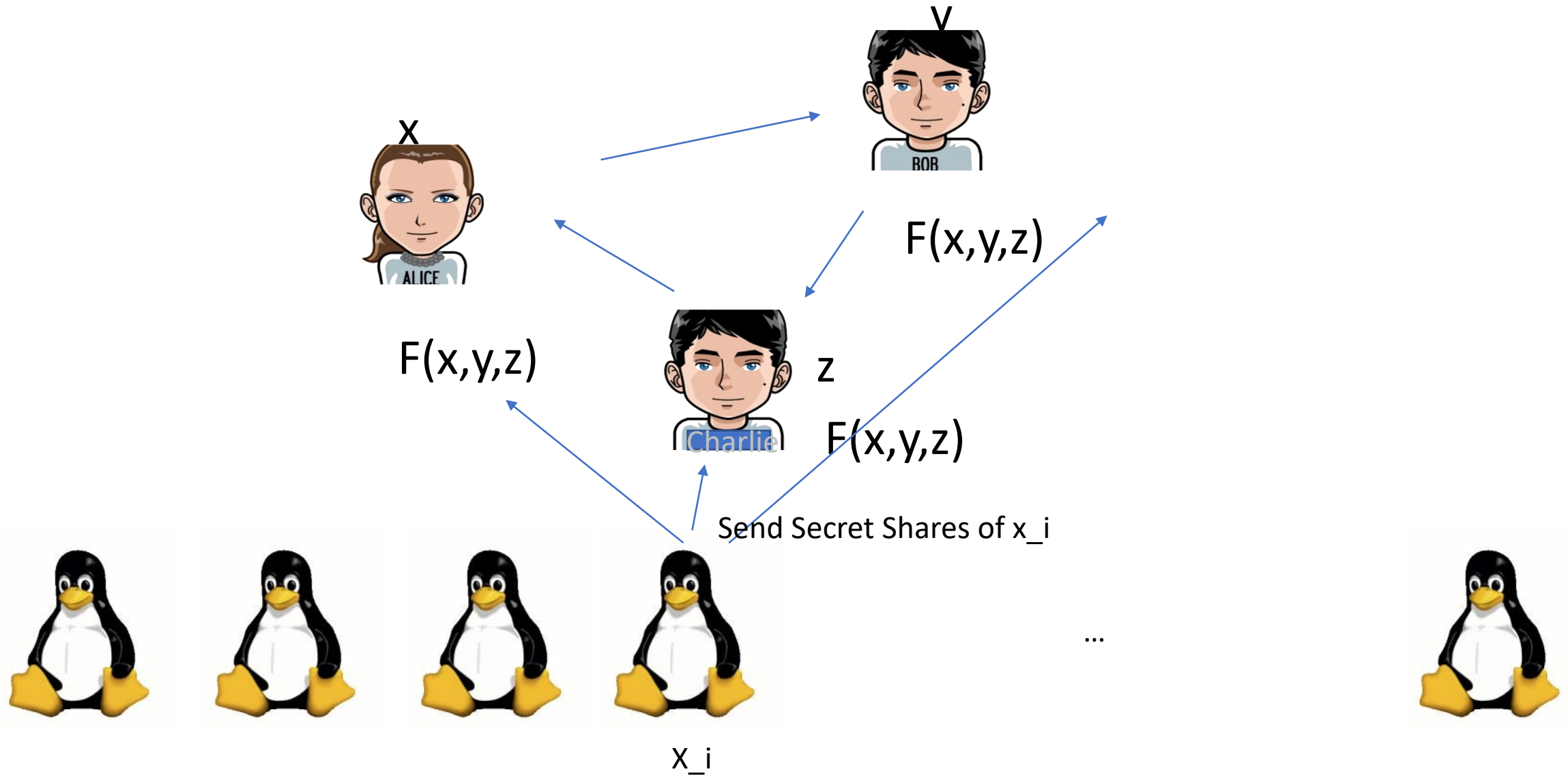
Use scenarios



Use scenarios



Use scenarios

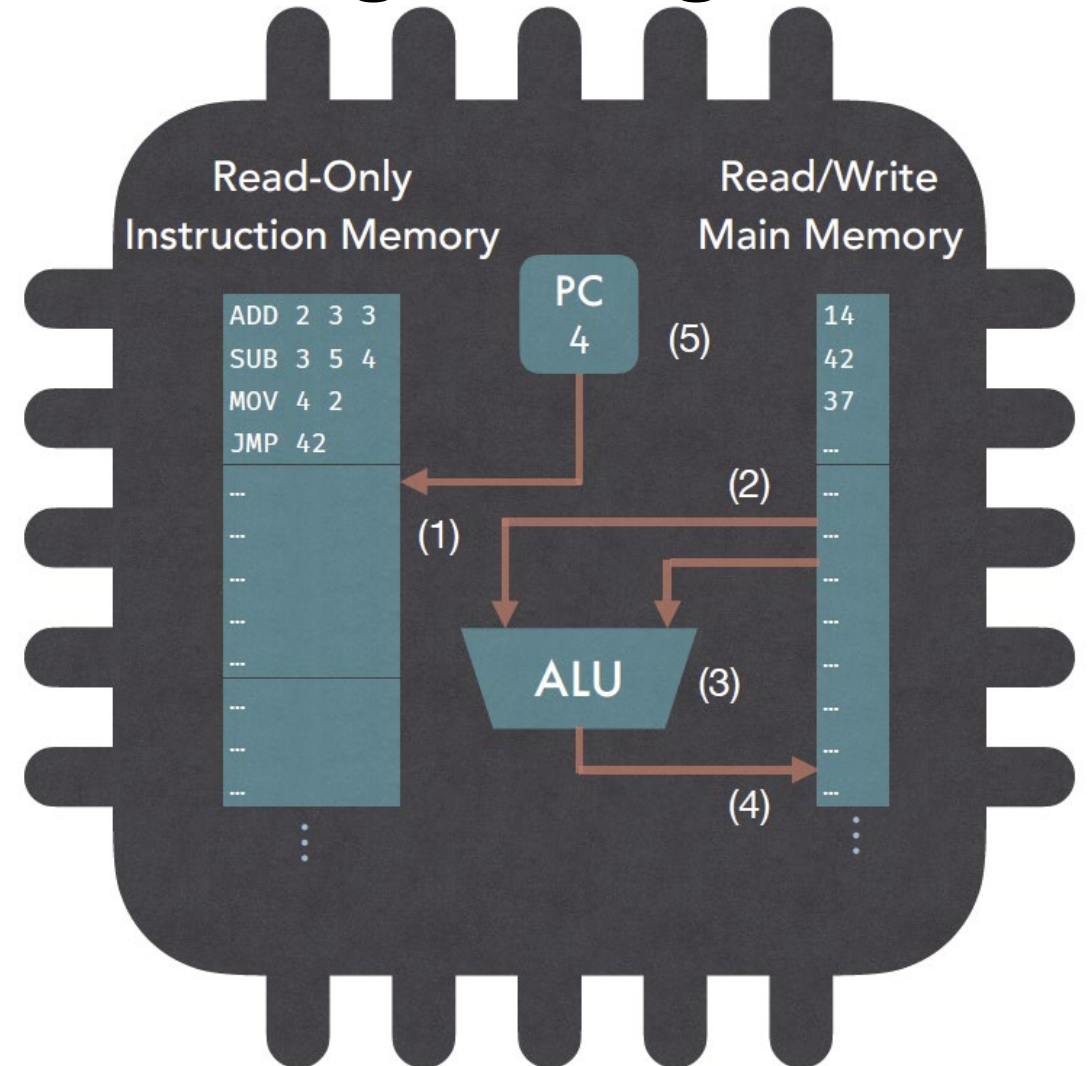


Implementing CPU with Stacked garbling

[HK20] For circuit $C = \text{Cond}(C_0, C_1, \dots, C_{b-1})$
Performance improvement factor b

CPU is such a conditional circuit!

Implement N CPU steps as sequence of N circuits. Each circuit ALU is now as large as a single instruction!



Zero knowledge proofs

A special case of MPC

$F(x,y)$ is a Boolean predicate

x



Prover

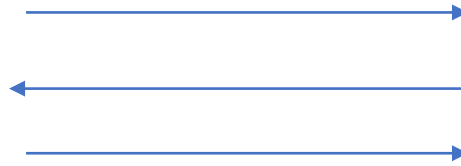
Knows everything
(V has no secrets)

\perp

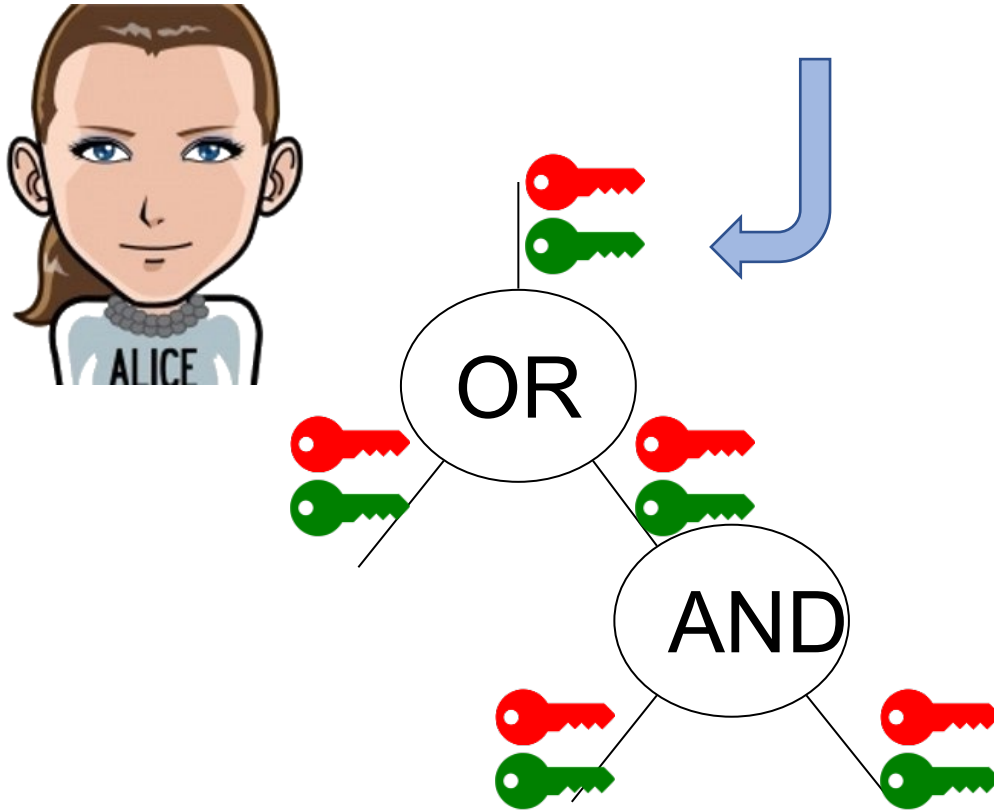



Verifier

learn $F(x,y) == 1$



Zero Knowledge proofs from GC



GC generator is V
If P can obtain 
on the output wire and show to
 V , she accepts the proof

Indeed, the only way to obtain it
is to correctly evaluate F on P 's
input.

Performance

- Garbled Circuit: On a laptop on 1Gbps LAN. Latency doesn't matter
2PC: about $\sim 2\text{M}$ AND gates/sec

Anyone knows what you can compute with 2M AND gates?

AES is 6k AND gates

500 AES evaluations/sec

Easy to scale

Performance

- RAM programs: Branching is cheap
- RAM access is expensive
- improvements expected

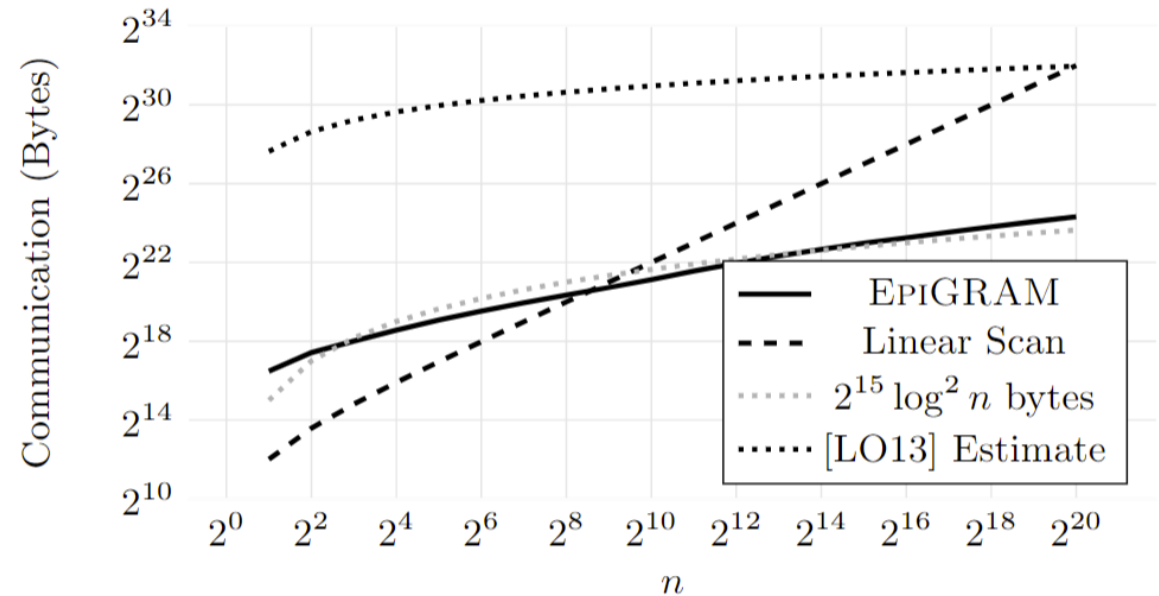


Fig. 12. Estimated concrete communication cost of our GRAM. We fix the word size $w = 128$ and plot per-access amortized communication as a function of n . For comparison we include an estimate of [LO13]’s performance (our estimate is favorable to [LO13], see Supplementary Material A for our analysis).

Variable Instruction Set Architecture (VISA)

This is our recent Assembly compiler

<i>G</i> 's Pattern Length	<i>E</i> 's String Length	EMP #AND Gates ($\times 10^9$)	EMP Total Time(s)	GAR Total Time(s)	Impr.
50	400	0.142	7.4	19.4	0.4×
150	700	2.264	120.0	46.4	2.6×
250	1500	13.11	732.7	119.8	6.1×
500	7000	233.0	12843.3	765.3	16.8×

Figure 9: Comparison of GAR with EMP's straight-line execution on KMP. Our improvement over EMP increases with larger inputs. Note, EMP implements array lookup with linear scans, not GRAM. For very small arrays, linear scans outperform EPIGRAM, which explains EMP's performance in the smallest instance.

ZK Performance

Zero-knowledge for ANSI C [HYD^K21]

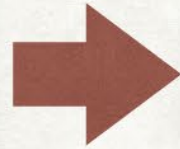
Compile C into small ISA and execute each step as a circuit between Prover P and Verifier V

- Runs at about 10KHz (previous machines ran at ~1Hz) with 2MB RAM support
- Run arbitrary Linux programs
- You can program arbitrary statements to be proven
- Or prove bugs
 - `gzip`: The bug (CVE-2005-1228 [CVE05]) allows an attacker to illegally write to an arbitrary directory. When `gzip` decompresses a zip file, the output directory is intended to be named according to a prefix of the input file name. Under certain inputs, `gzip` will erroneously write to an arbitrary directory chosen by the attacker.. We detect the bug by placing string comparison logic immediately before opening an output file.

Big Picture (ZEE ZK proof system [HYDK21])

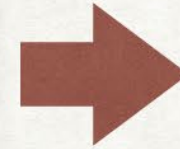
source ANSI C program

```
// program.c
int main(int argc, char** argv) {
  ...
}
```

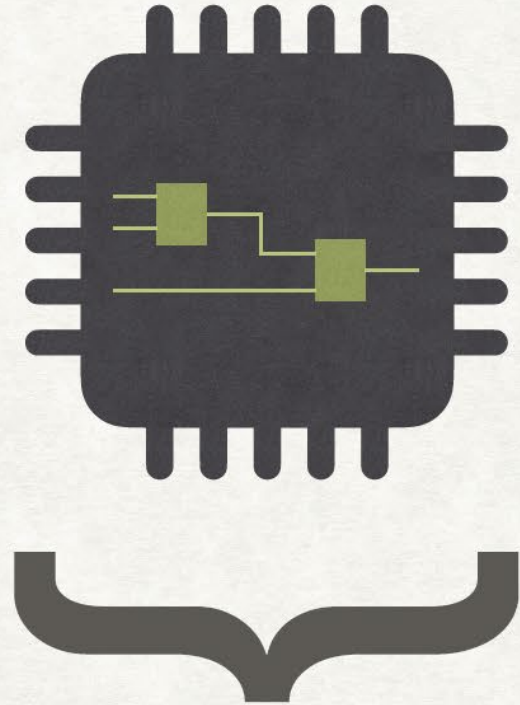


compiled assembly code
in custom assembly language

```
// program.as
...
loop_15:
  ADD $GR0 $GR1
  MUL $GR0 #15
...
```



ZK Machine



Circuits are implemented
by share algebra

Run ZK machine at about 11Khz over a rich ISA

gzip Benchmark

Bug allows attacker to write to arbitrary file in target system on some inputs CVE-2005-1228

ZKM demonstrates existence of the bug in 6.5s

```
// in correct program ofname must be prefix of ifname
int len_if = strlen(ifname);
int len_of = strlen(ofname);
if (len_of > len_if) {QED;}
for (int i = 0; i < len_of; ++i) {
    if (ifname[i] != ofname[i]) {QED;}
}
```

sed Benchmark

- On certain inputs, `sed` would trigger an infinite loop that overwrites all of memory (Software-artifact Infrastructure Repository)
- On the ZKM, this eventually causes `sed` to write to protected memory
- The ZKM triggers a “hardware interrupt” on this illegal memory access, leading to a proof
- ZKM demonstrates existence of the bug in 36.1s

	Syntax	Semantics
Algebra	MOV $tar \{src\}$	$\mathcal{R}[tar] \leftarrow \mathcal{V}(src)$
	CMOV $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \begin{cases} \mathcal{V}(src_1), & \text{if } \mathcal{R}[src_0] \neq 0 \\ \mathcal{R}[tar], & \text{otherwise} \end{cases}$
	ADD $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] + \mathcal{V}(src_1)$
	SUB $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] - \mathcal{V}(src_1)$
	MUL $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \cdot \mathcal{V}(src_1)$
	XOR $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \oplus \mathcal{V}(src_1)$
	AND $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \wedge \mathcal{V}(src_1)$
	OR $tar \ src_0 \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \vee \mathcal{V}(src_1)$
	EQZ $tar \ src$	$\mathcal{R}[tar] \leftarrow \begin{cases} 1, & \text{if } \mathcal{R}[src] = 0 \\ 0, & \text{otherwise} \end{cases}$
	MSB $tar \ src$	$\mathcal{R}[tar] \leftarrow \begin{cases} 1, & \text{if } \mathcal{R}[src] \geq 2^{31} \\ 0, & \text{otherwise} \end{cases}$
	POW2 $tar \ src$	$\mathcal{R}[tar] \leftarrow 2^{\mathcal{R}[src]}$
Control Flow	JMP $\{dst\}$	$pc \leftarrow \mathcal{V}(dst)$
	BNZ $src \{dst\}$	$pc \leftarrow \begin{cases} \mathcal{V}(dst), & \text{if } \mathcal{R}[src] \neq 0 \\ pc + 1, & \text{otherwise} \end{cases}$
	PC $tar \{src\}$	$\mathcal{R}[tar] \leftarrow pc + \mathcal{V}(src) ; pc \leftarrow pc + 1$
	HALT	– no effect, pc unchanged –
	QED	– no effect, pc unchanged –
Memory	LOAD $tar \ addr_0 \{addr_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{M}[\mathcal{R}[addr_0] + \mathcal{V}(addr_1)]$
	STORE $src \ addr_0 \{addr_1\}$	$\mathcal{M}[\mathcal{R}[addr_0] + \mathcal{V}(addr_1)] \leftarrow \mathcal{R}(src)$
\mathcal{P} Input	INPUT tar	$\mathcal{R}[tar] \leftarrow x$ where $x \in \{0..2^{32} - 1\}$ is chosen by \mathcal{P}
	ORACLE $\{id\}$	honest \mathcal{P} privately calls oracle procedure $\mathcal{V}(id)$; $pc \leftarrow pc + 1$

Design Considerations:

- Which instructions should we include?
- Some instructions are more efficient in our protocol than others
- How many instructions should we include?
- Additional instructions require additional circuitry and can increase cost

We opt for a bare bones architecture

- Highly efficient • Easy to maintain

MPC for targeted functions

- PSI and variants
- DB queries
- Can be as fast as plaintext evaluation
 - Computer RAM access is slower than computing symmetric crypto
 - Can be especially fast if
 - some RAM access patterns are revealed
 - A 3rd party is used in the computation as an oblivious helper

Questions