Efficient Cryptographic Computation for Real-World Programs and People

Advancing Algorithms and Systems

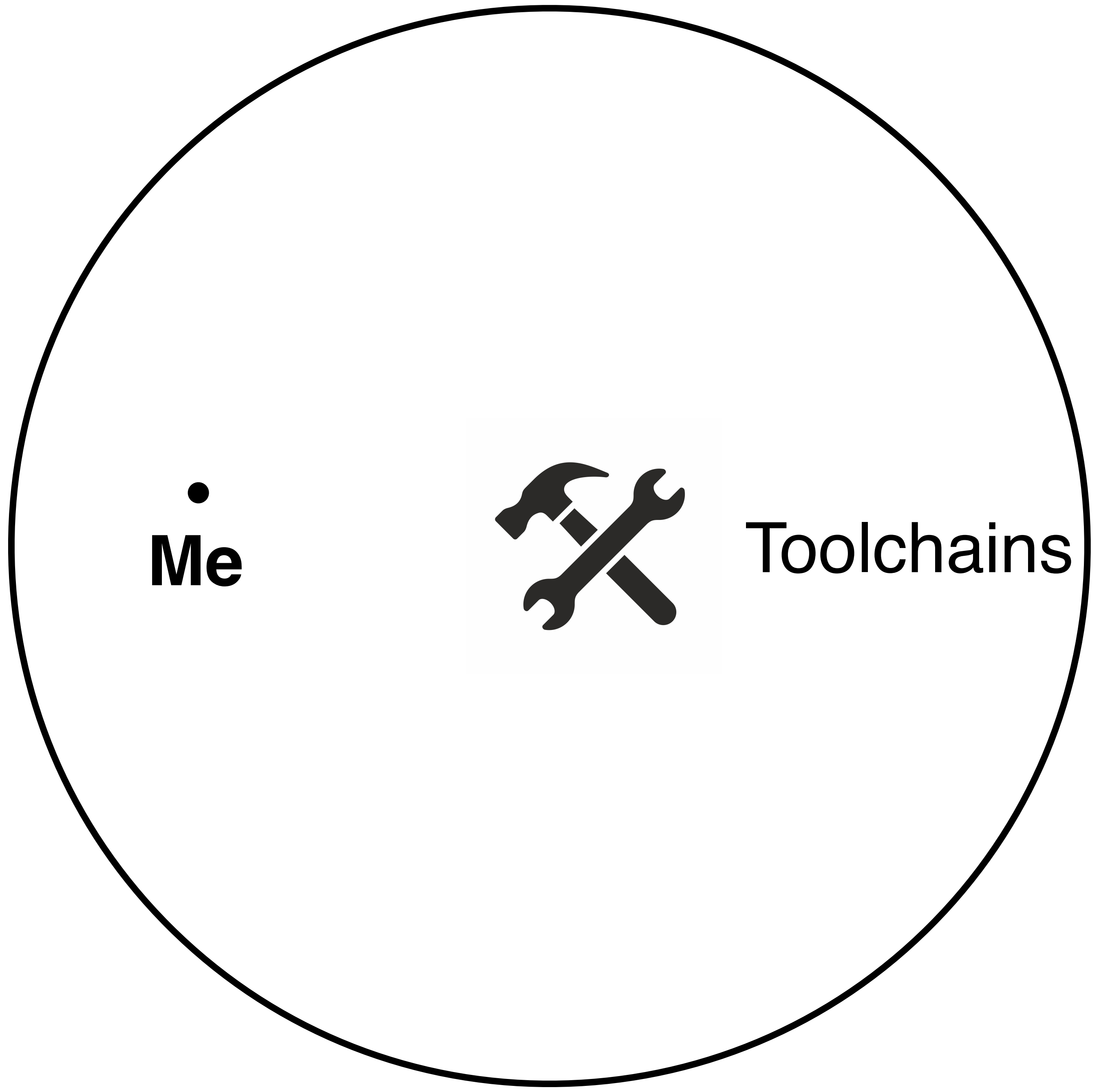
Yibin Yang

Georgia Tech

About Me

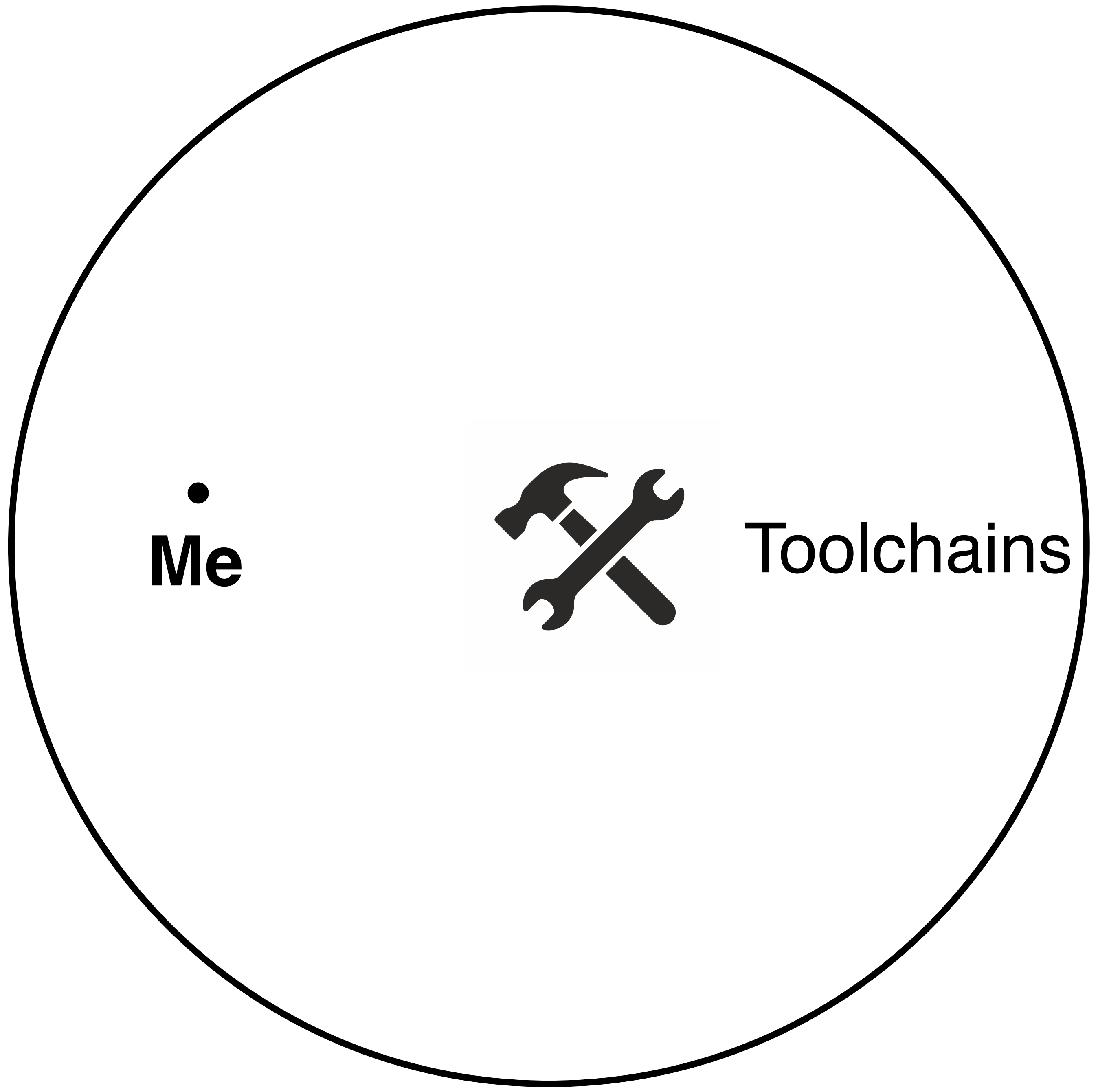
About Me

Applied Cryptographer



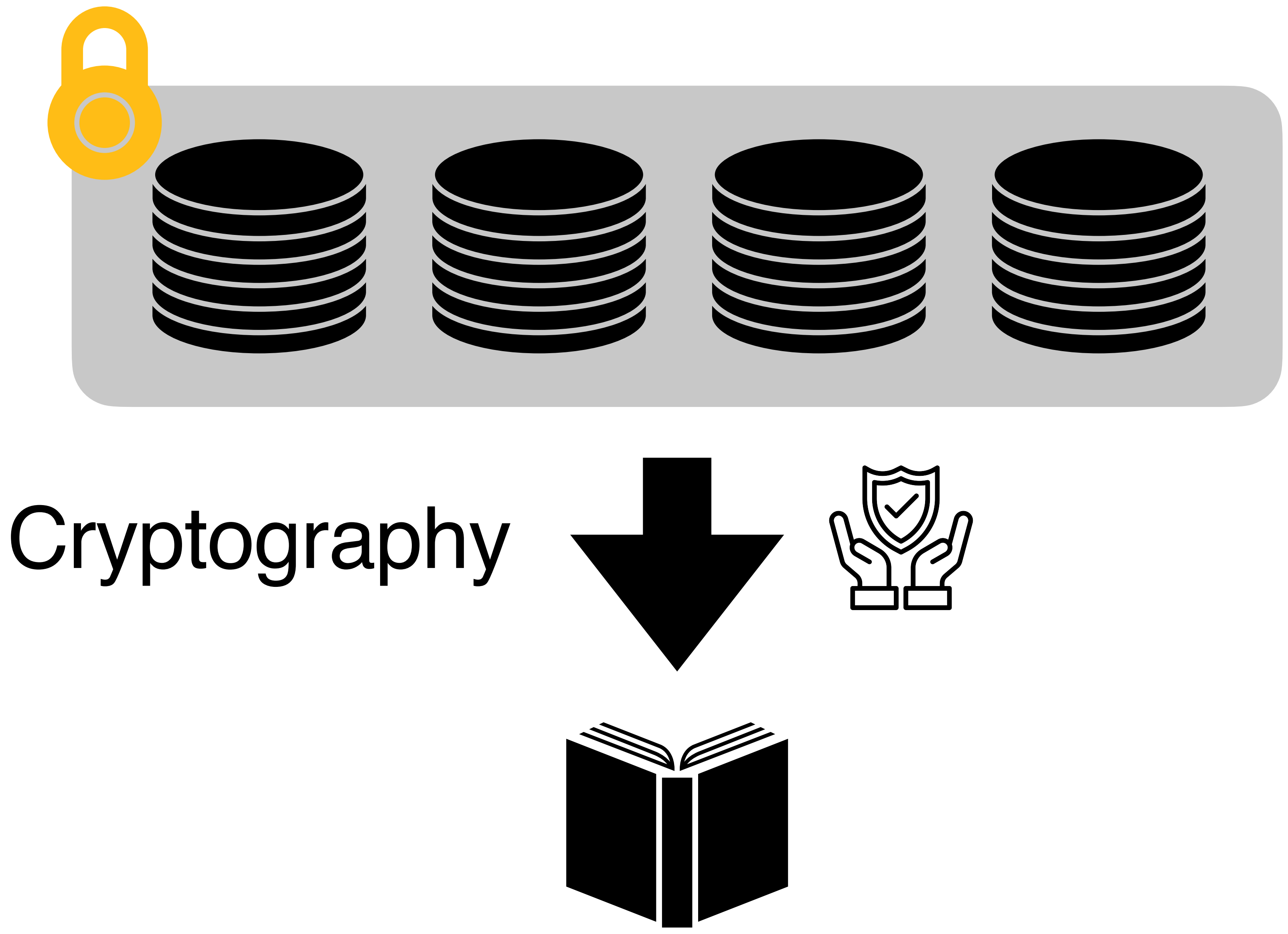
About Me

Applied Cryptographer



Me

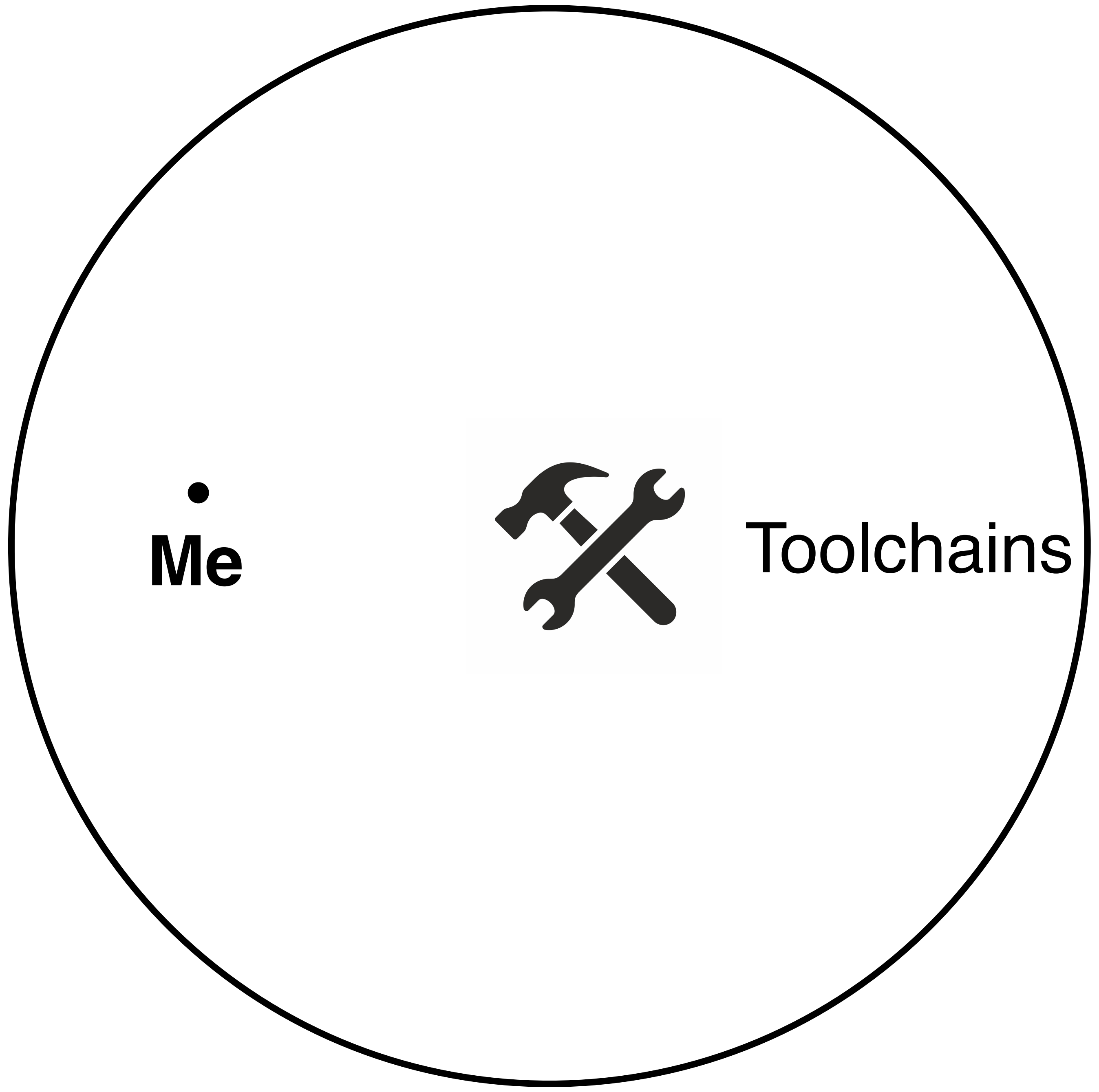
Toolchains

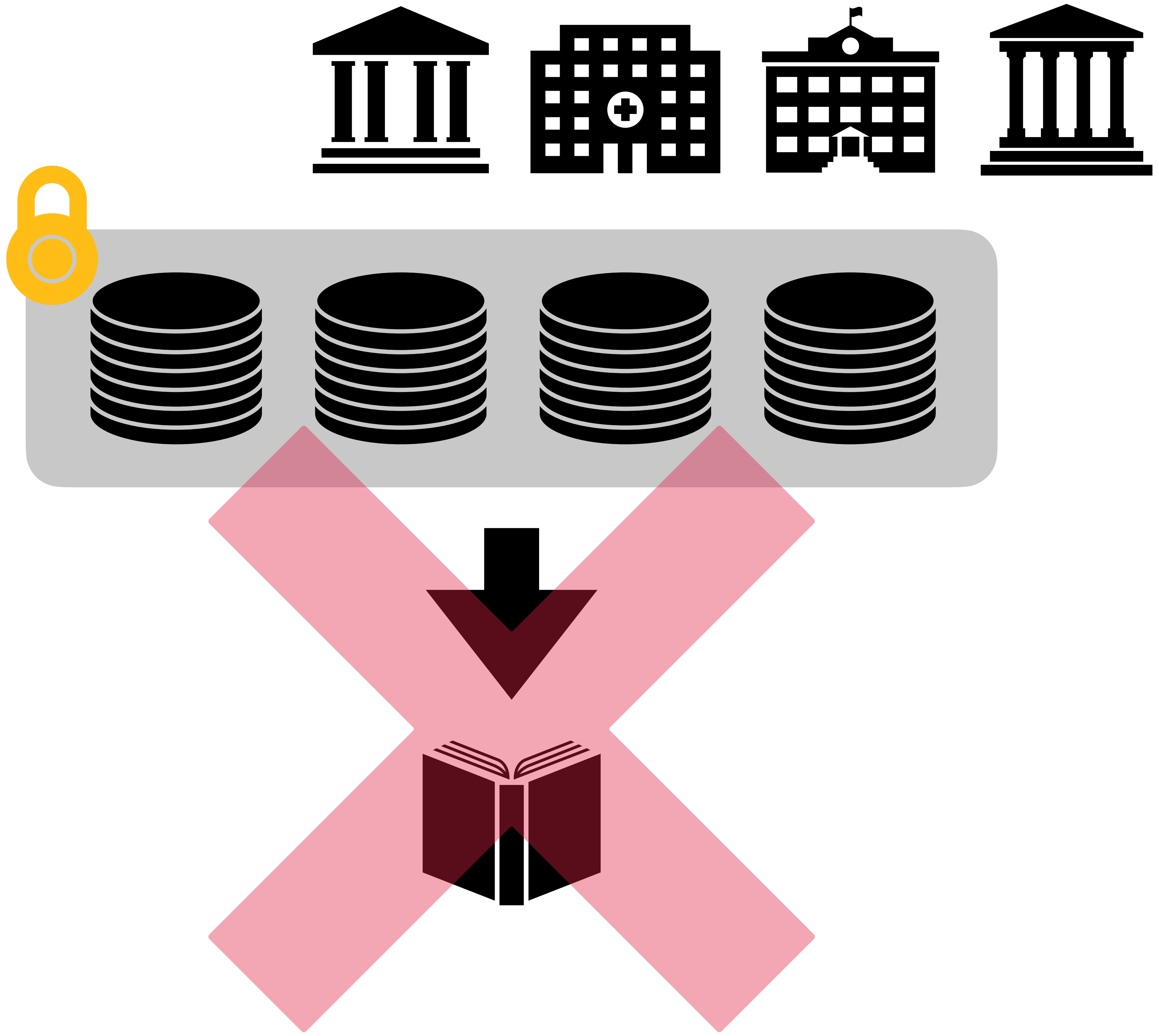


Cryptography

About Me

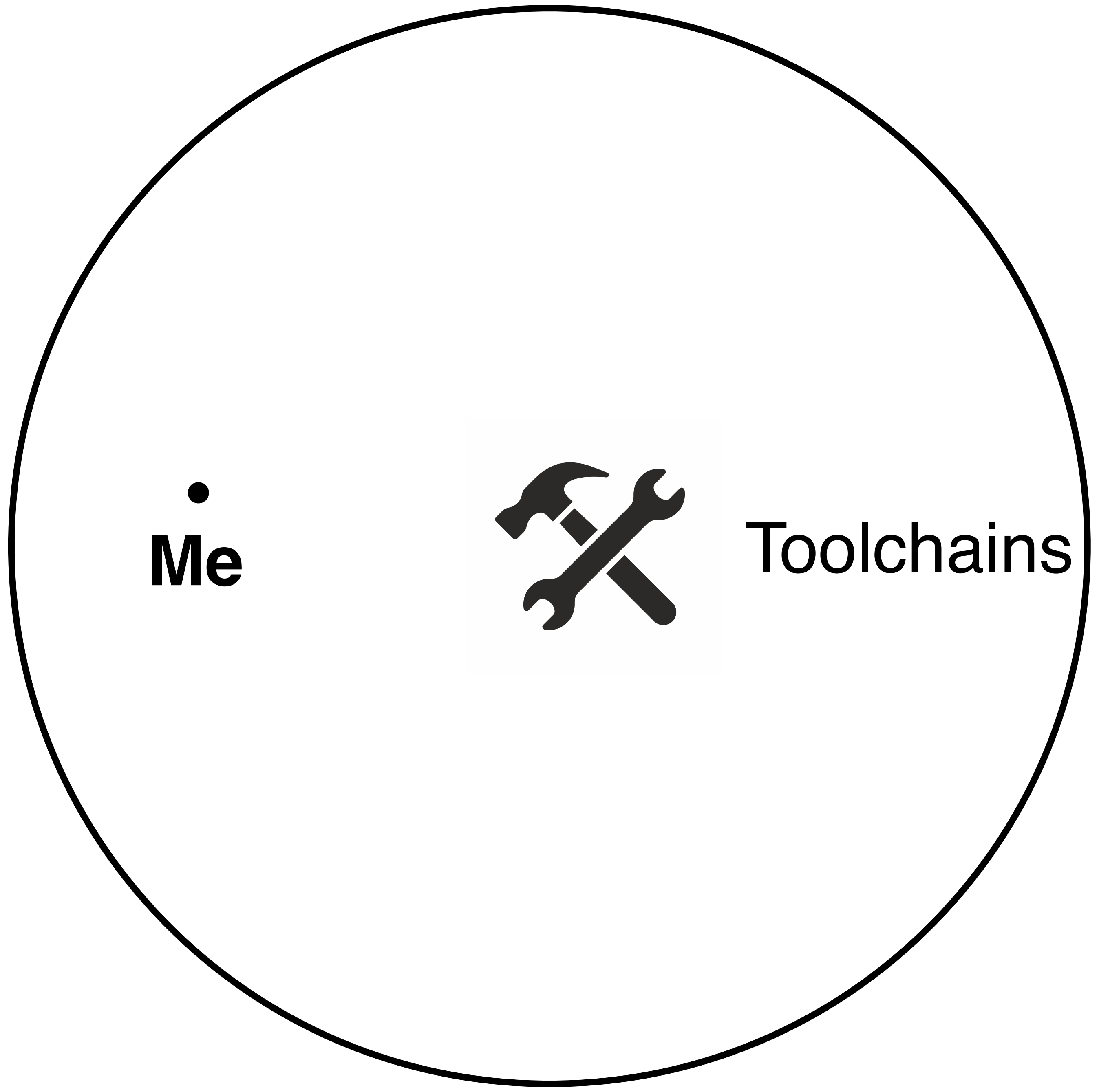
Applied Cryptographer

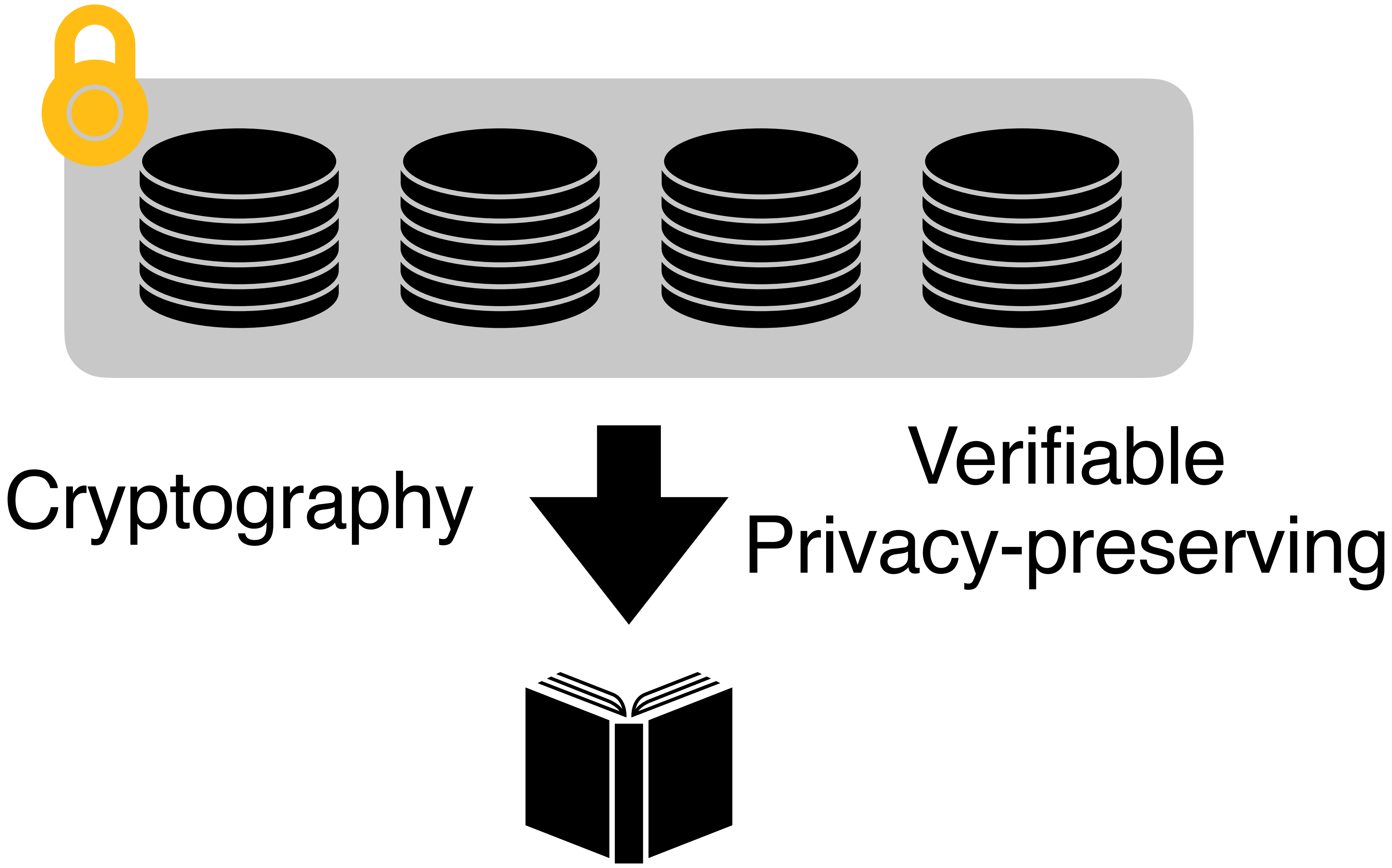




About Me

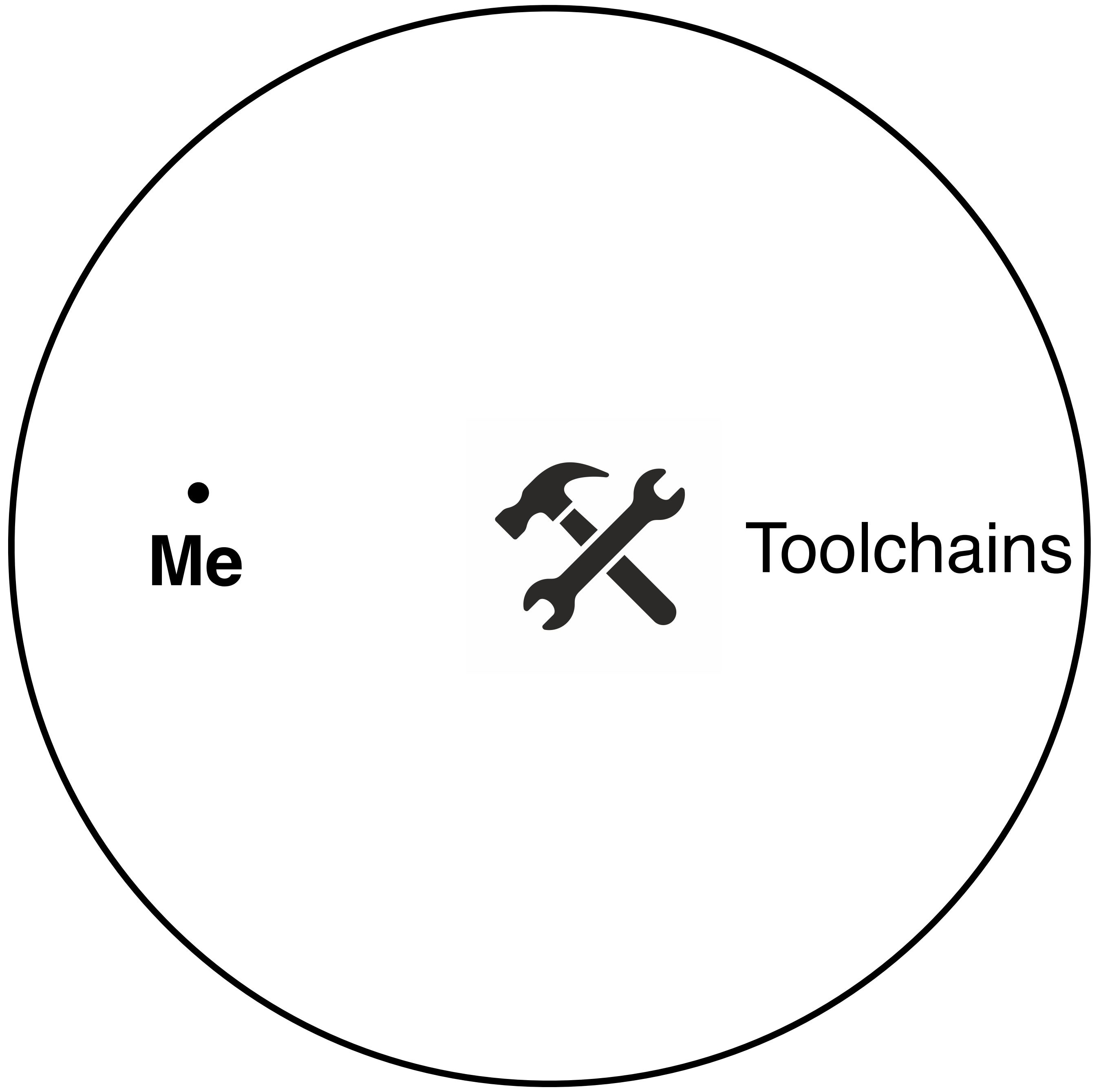
Applied Cryptographer

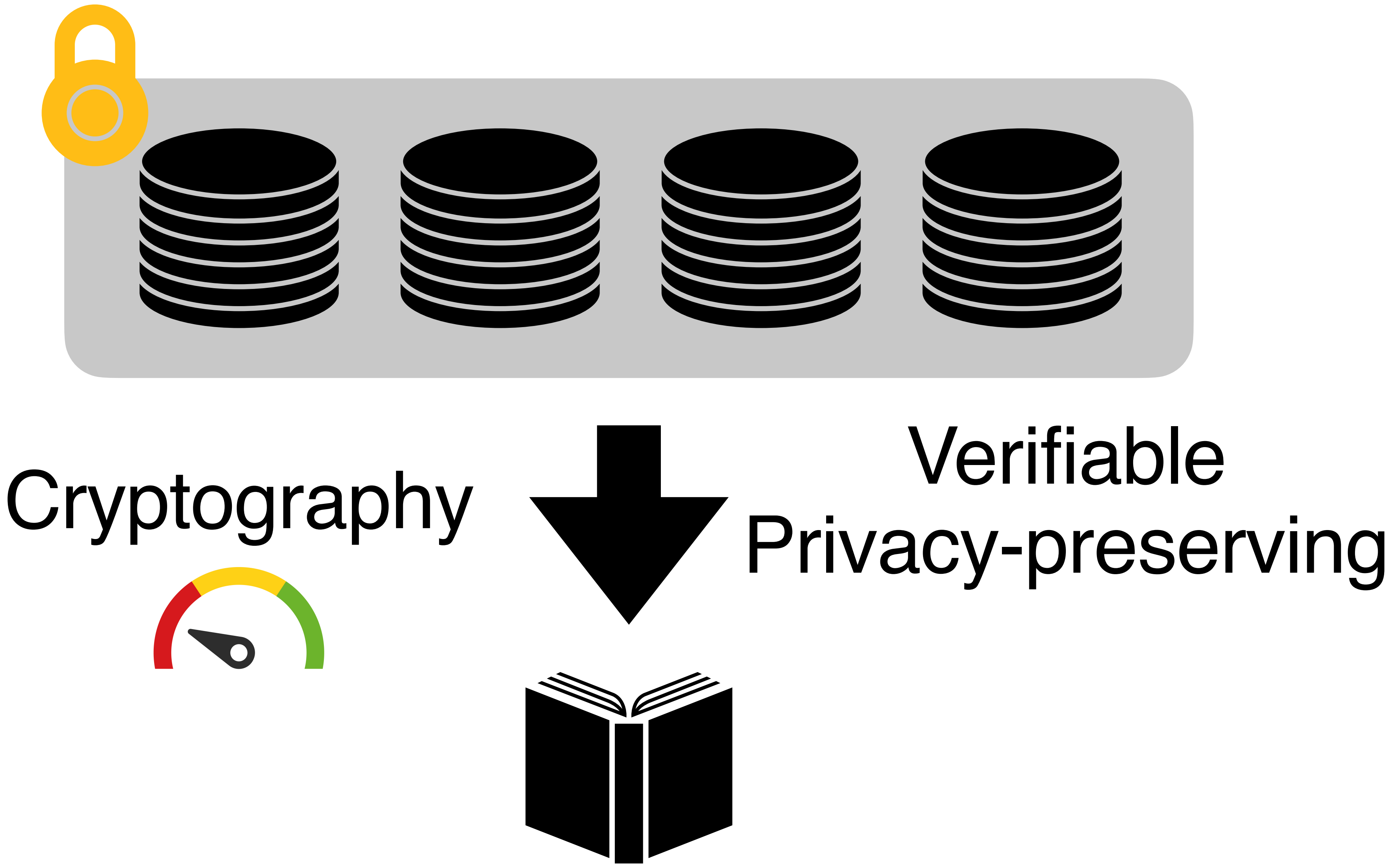




About Me

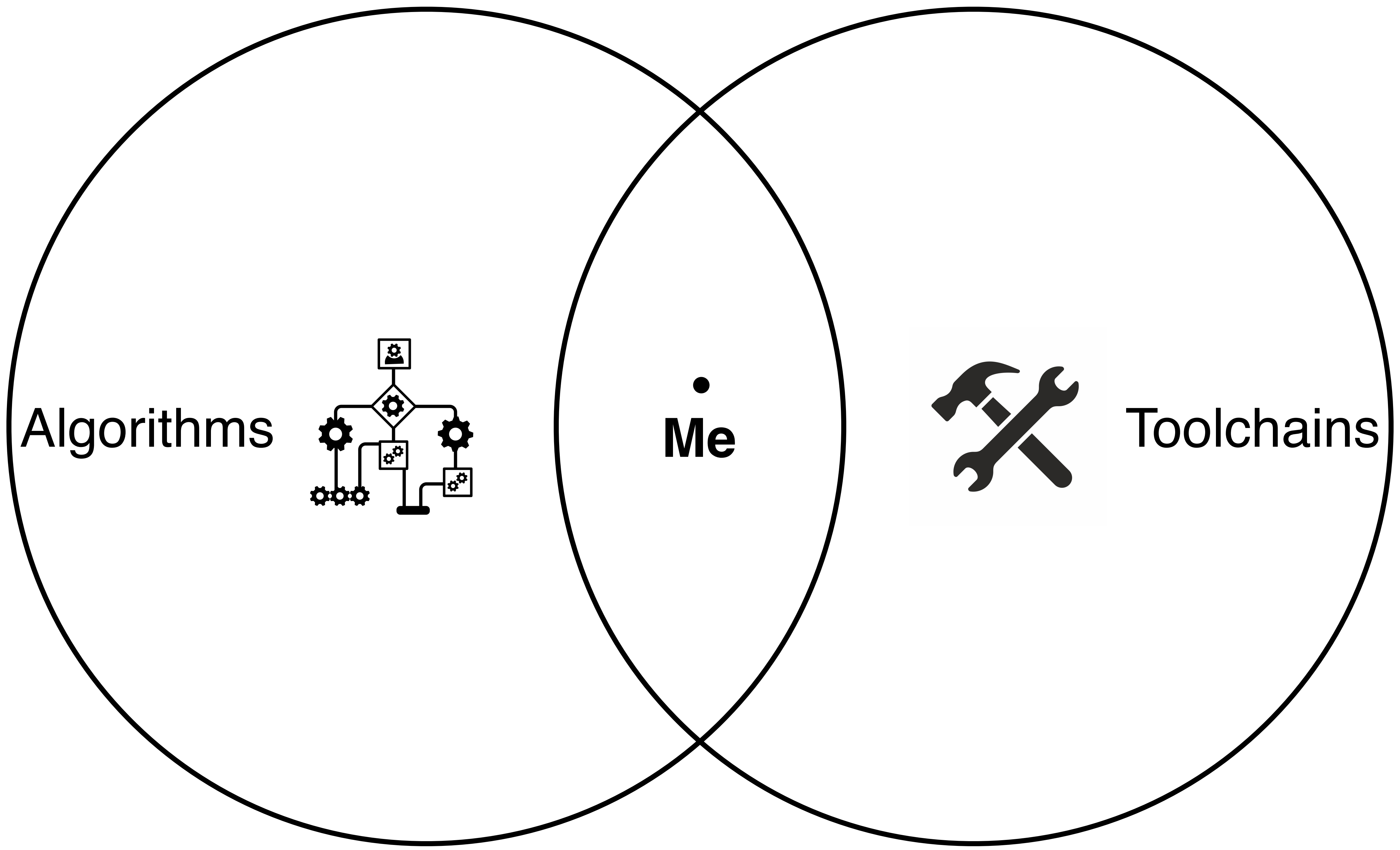
Applied Cryptographer





About Me

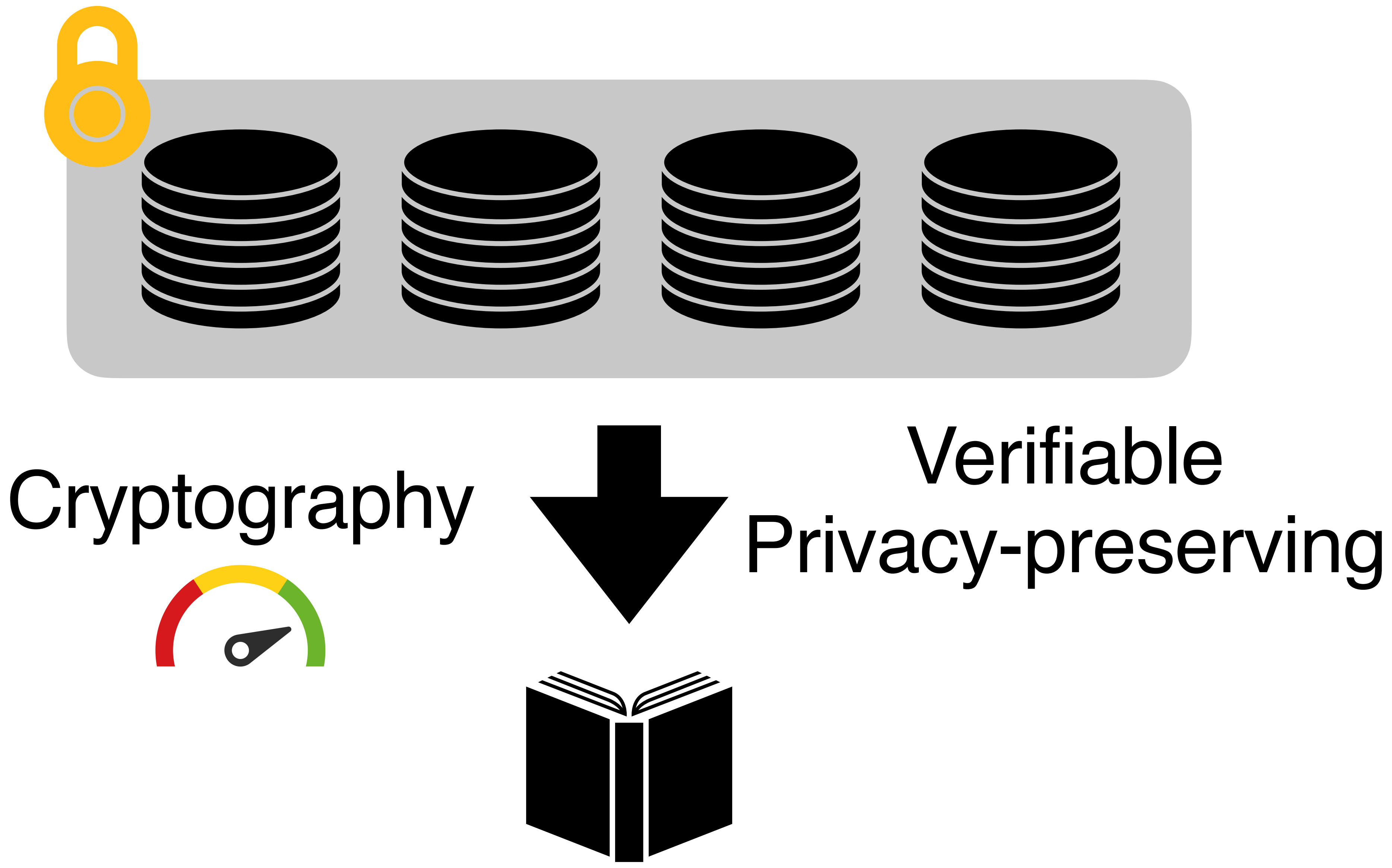
Applied Cryptographer



Algorithms

Me

Toolchains



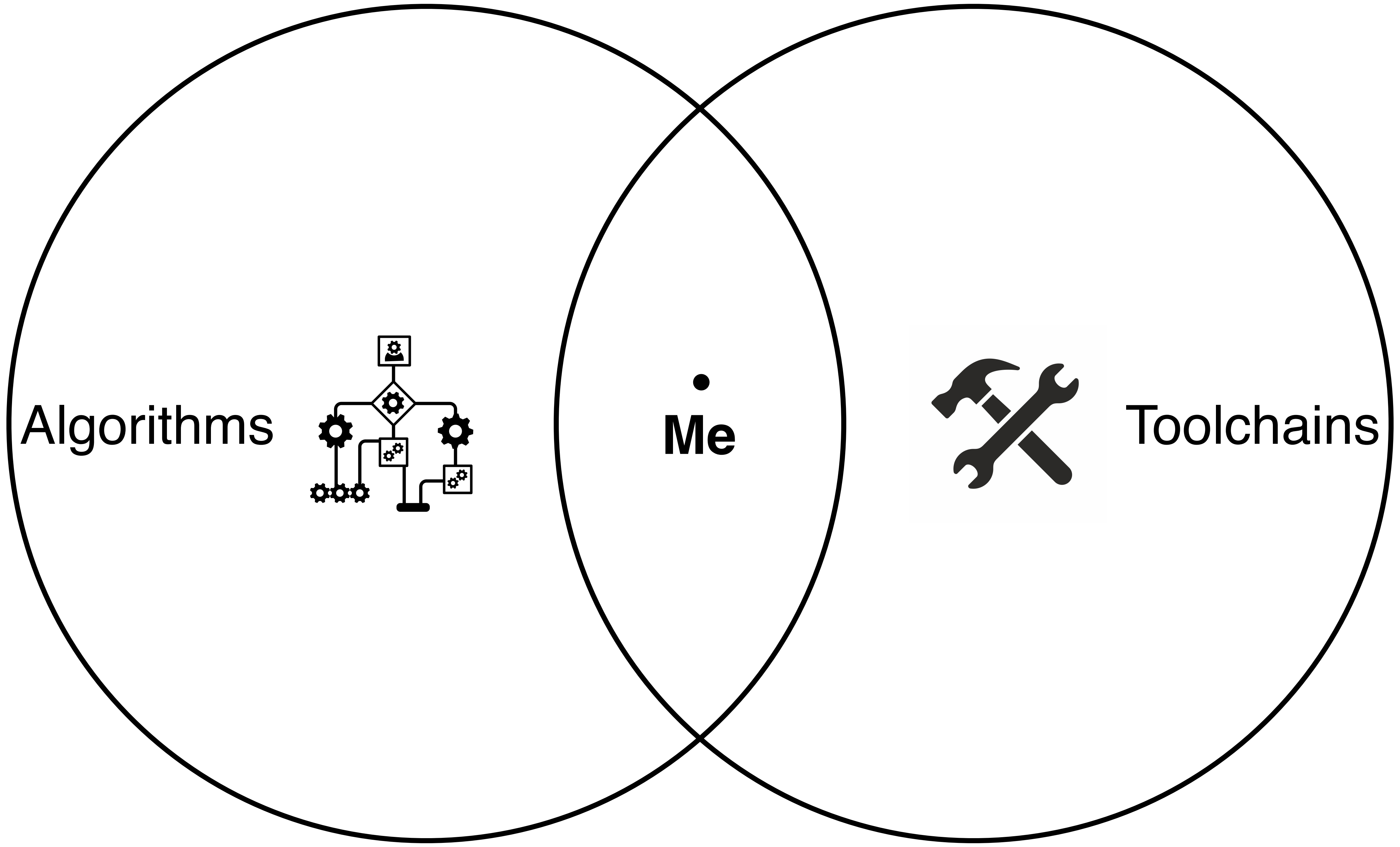
Cryptography

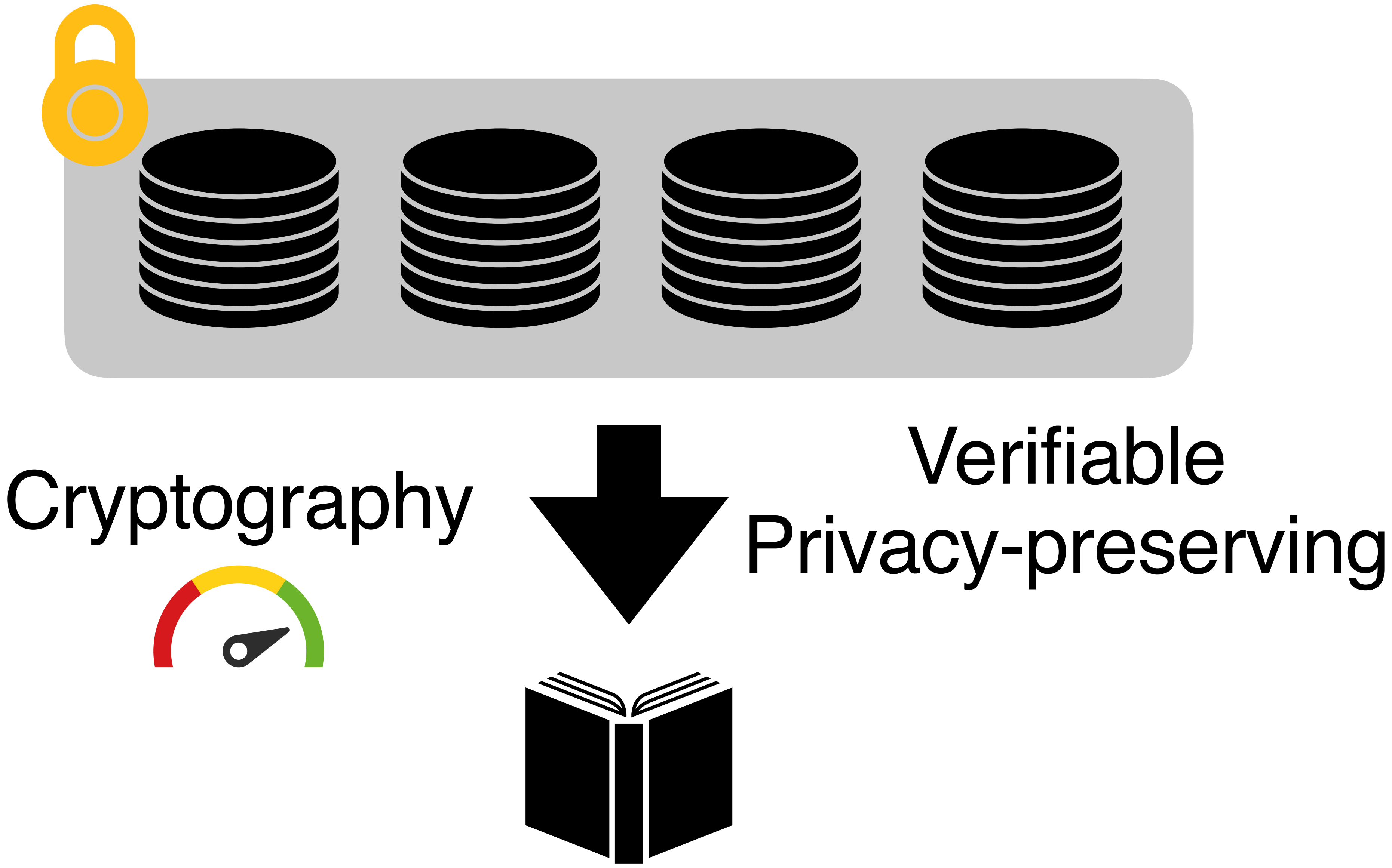
Verifiable

Privacy-preserving

About Me

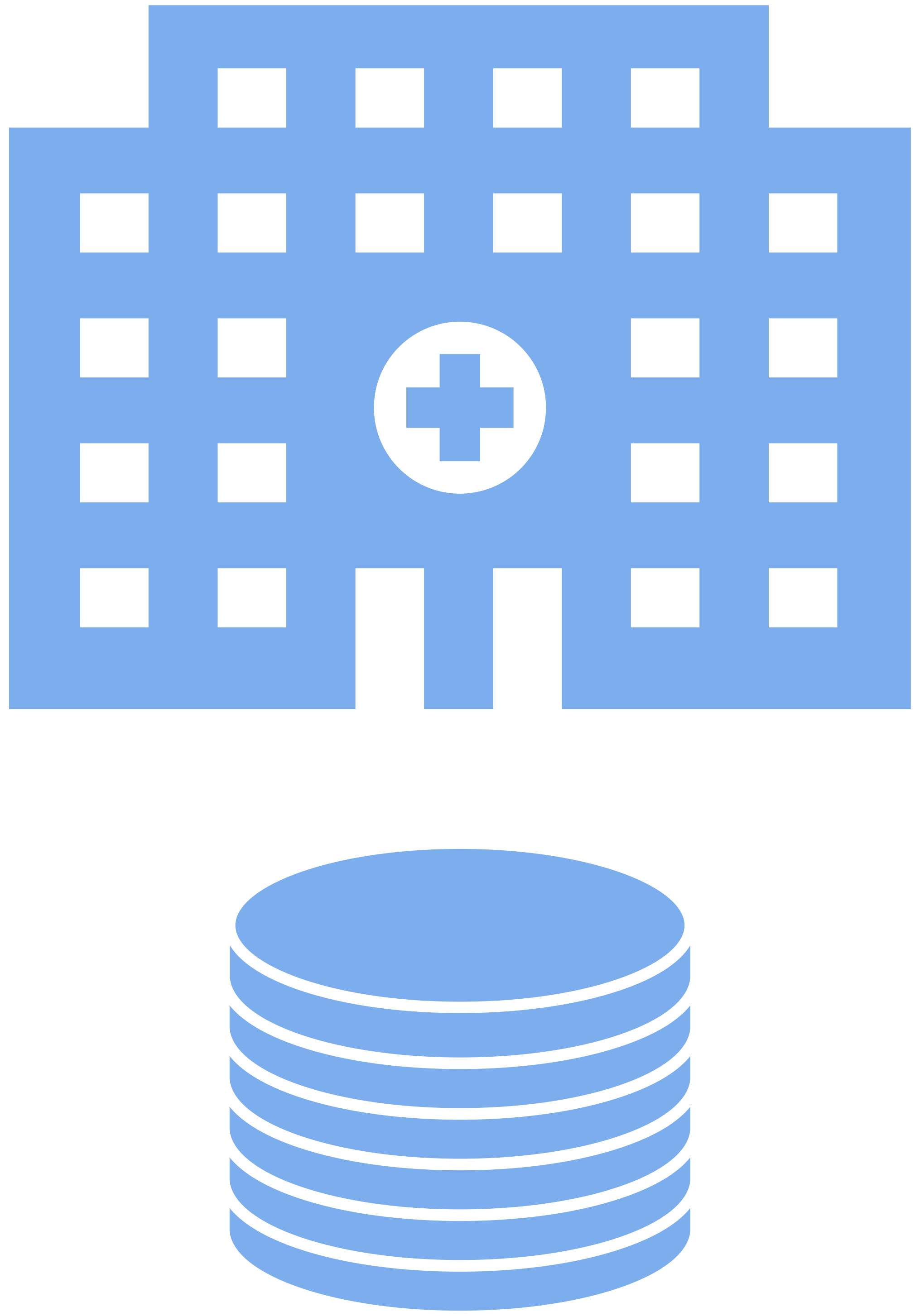
Applied Cryptographer



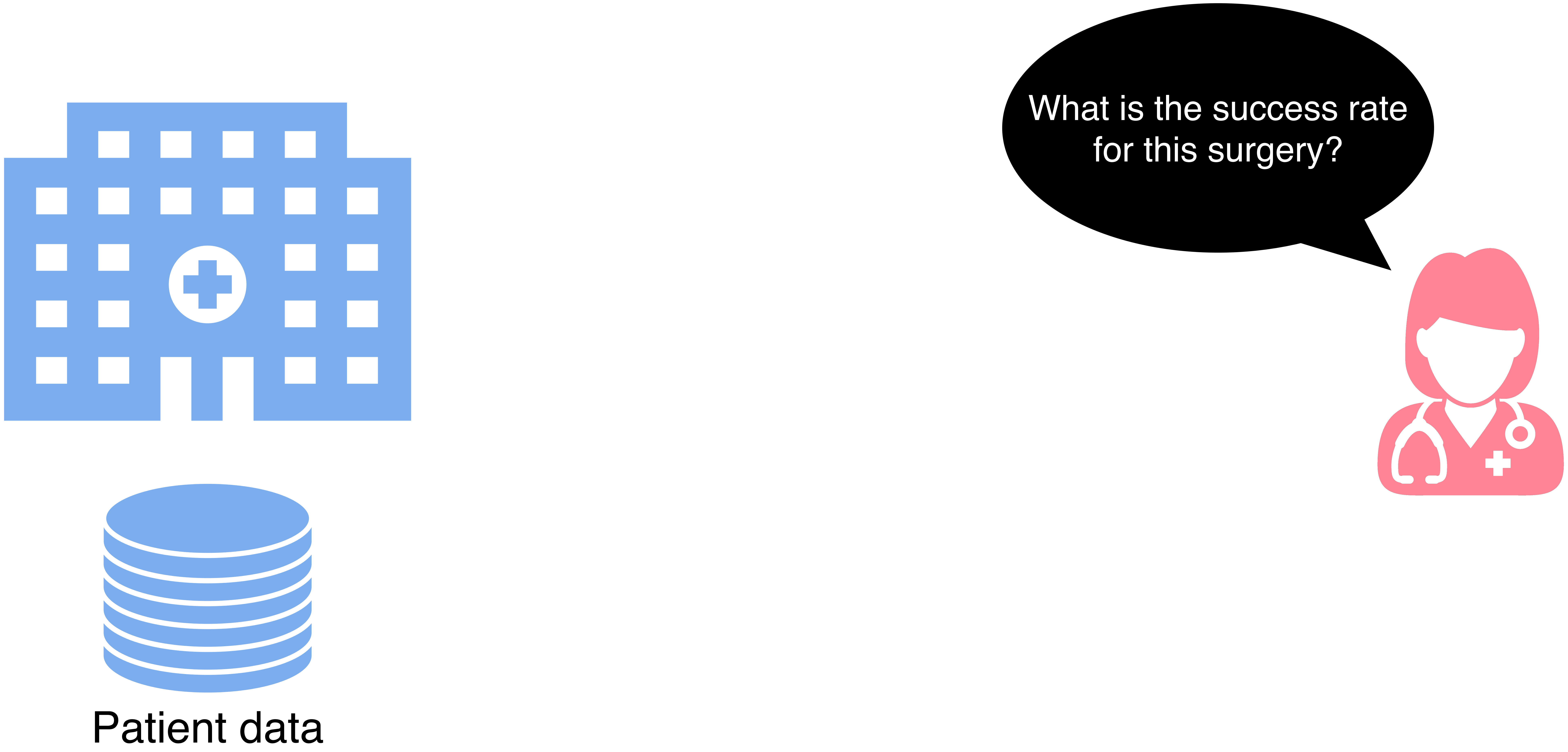


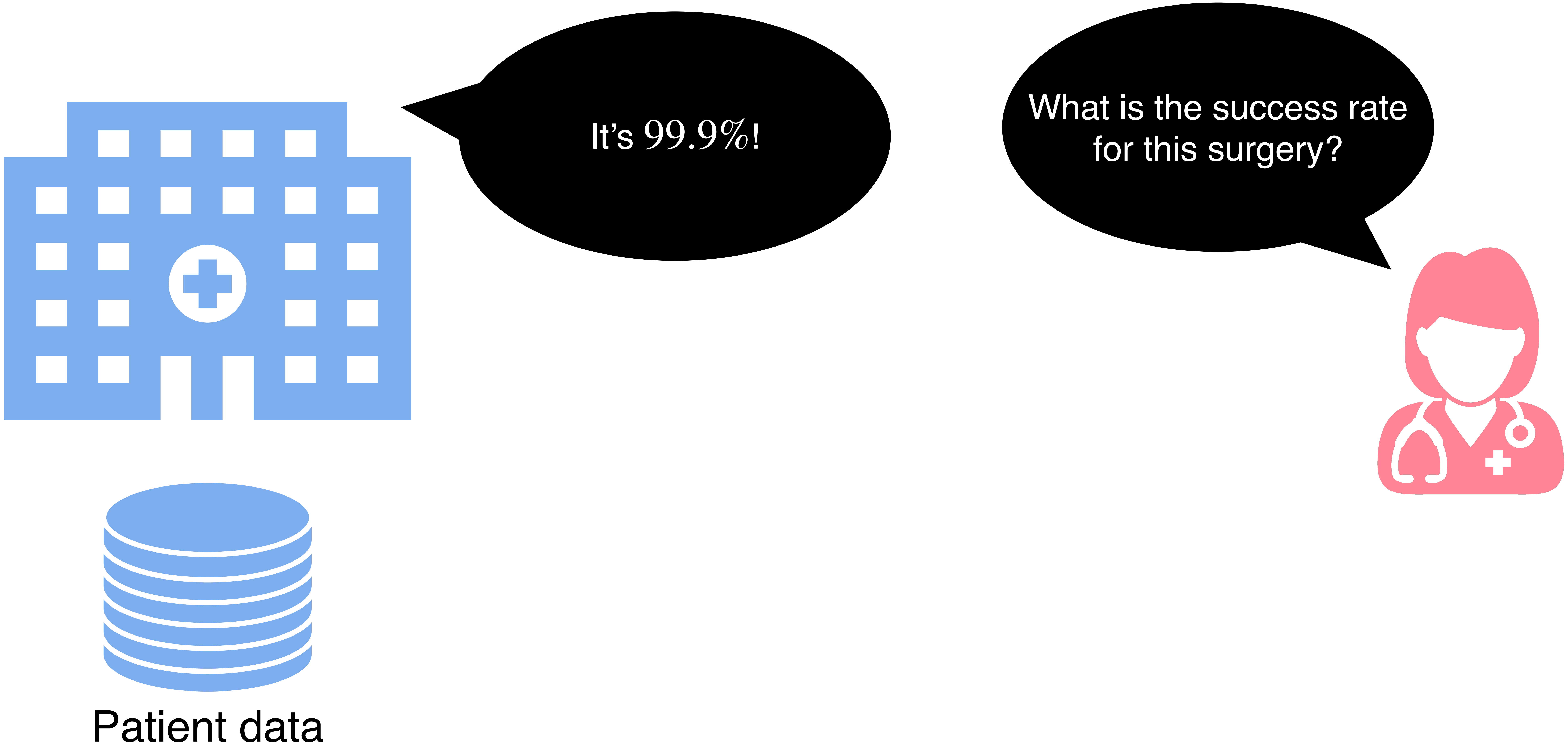
Zero-Knowledge Proof

Secure Multi-Party Computation



Patient data

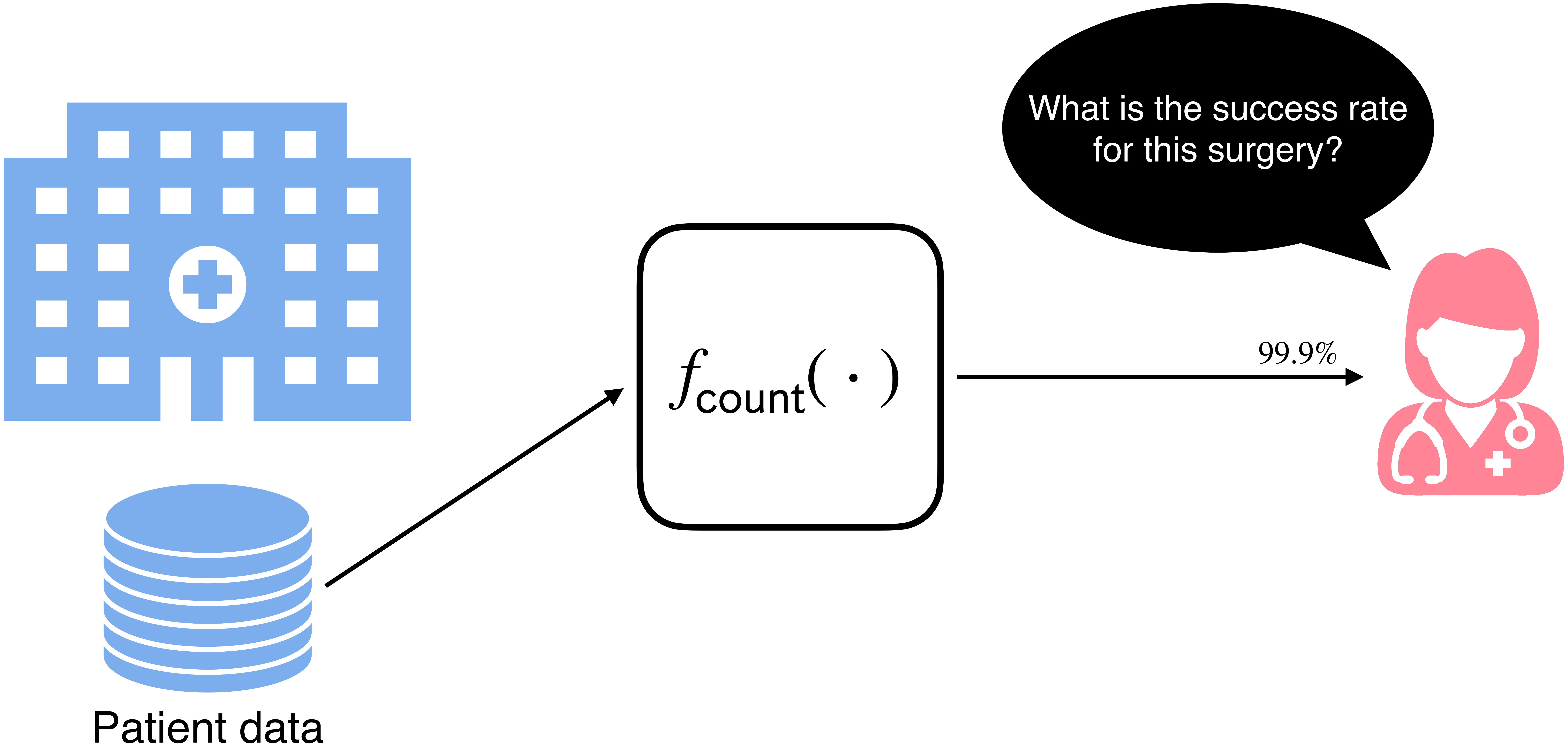

What is the success rate for this surgery?
Patient data



It's 99.9%!

What is the success rate for this surgery?

Patient data



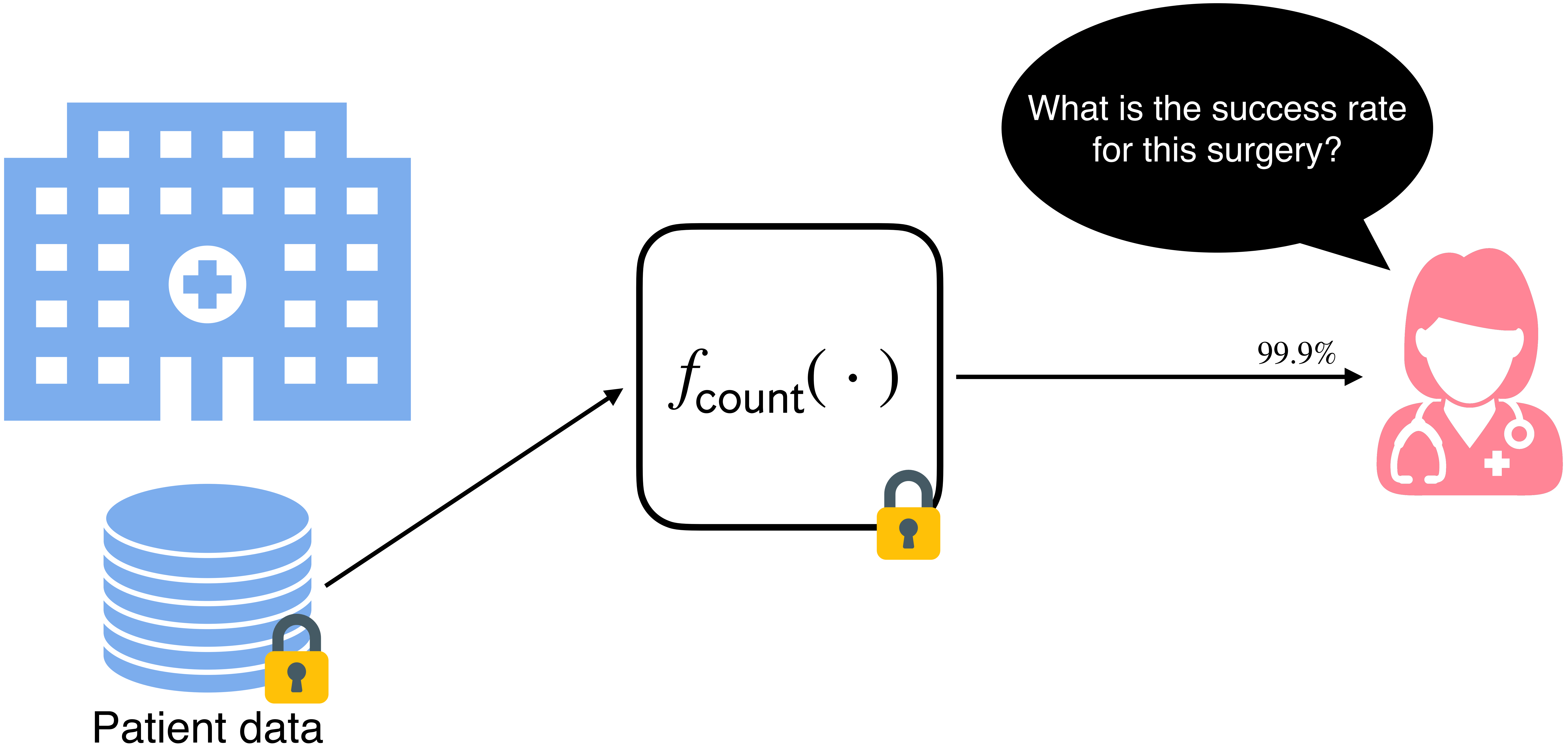
Patient data

$f_{\text{count}}(\,\cdot\,)$

99.9%

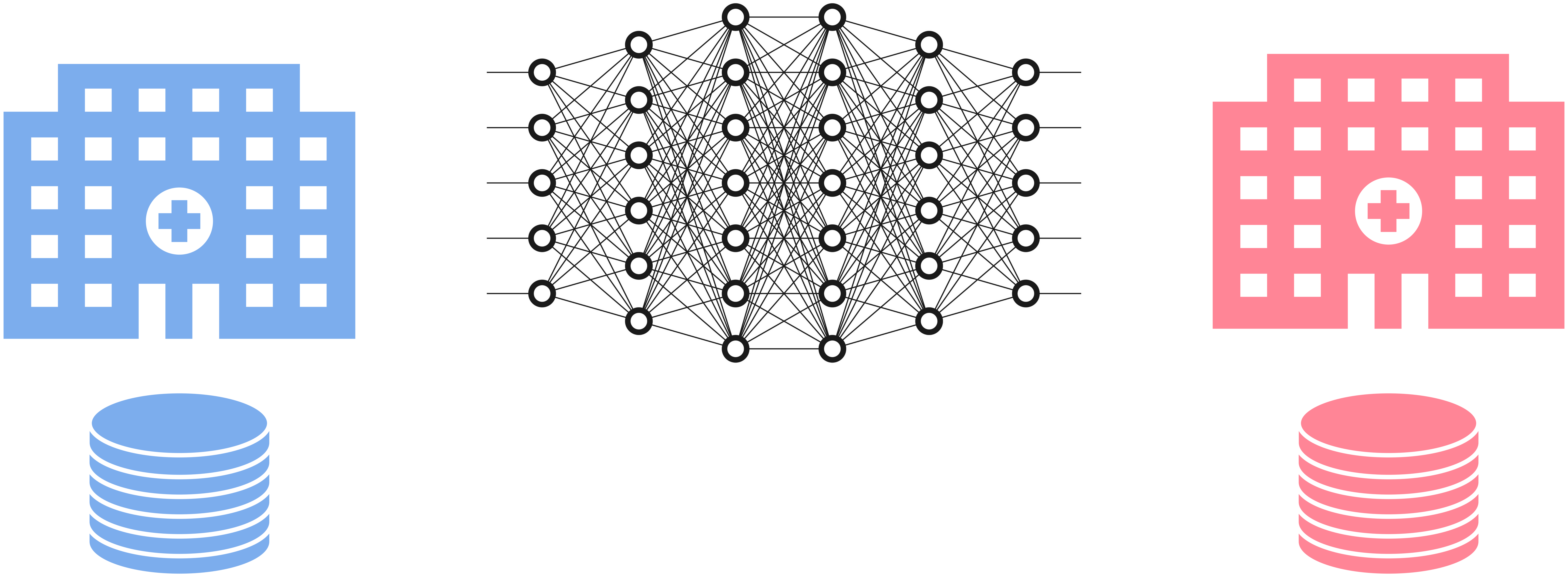
What is the success rate for this surgery?

Zero-Knowledge Proof (ZKP) [GMR85]


What is the success rate for this surgery?
$f_{\text{count}}(\cdot)$
99.9%
Patient data

Patient data

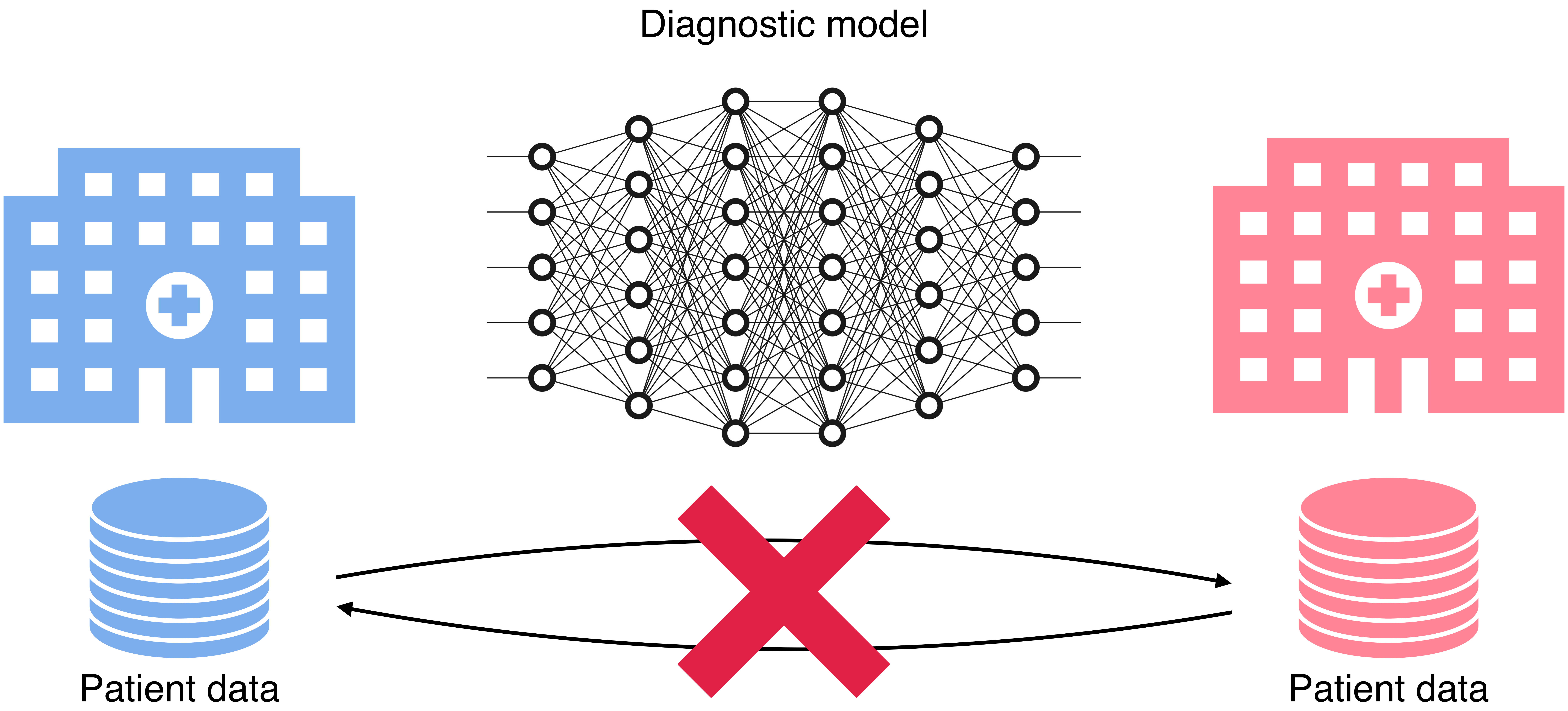
Patient data



Diagnostic model

Patient data

Patient data

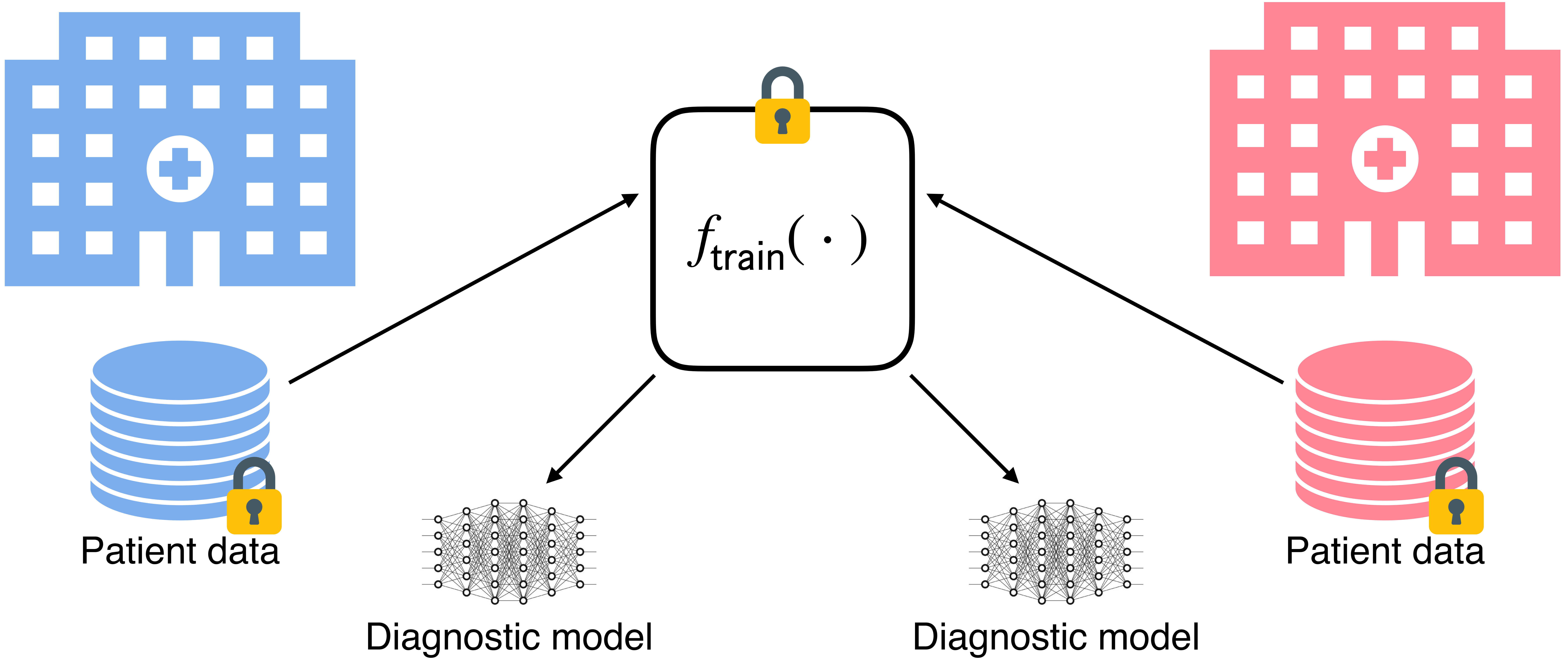


Diagnostic model

Patient data

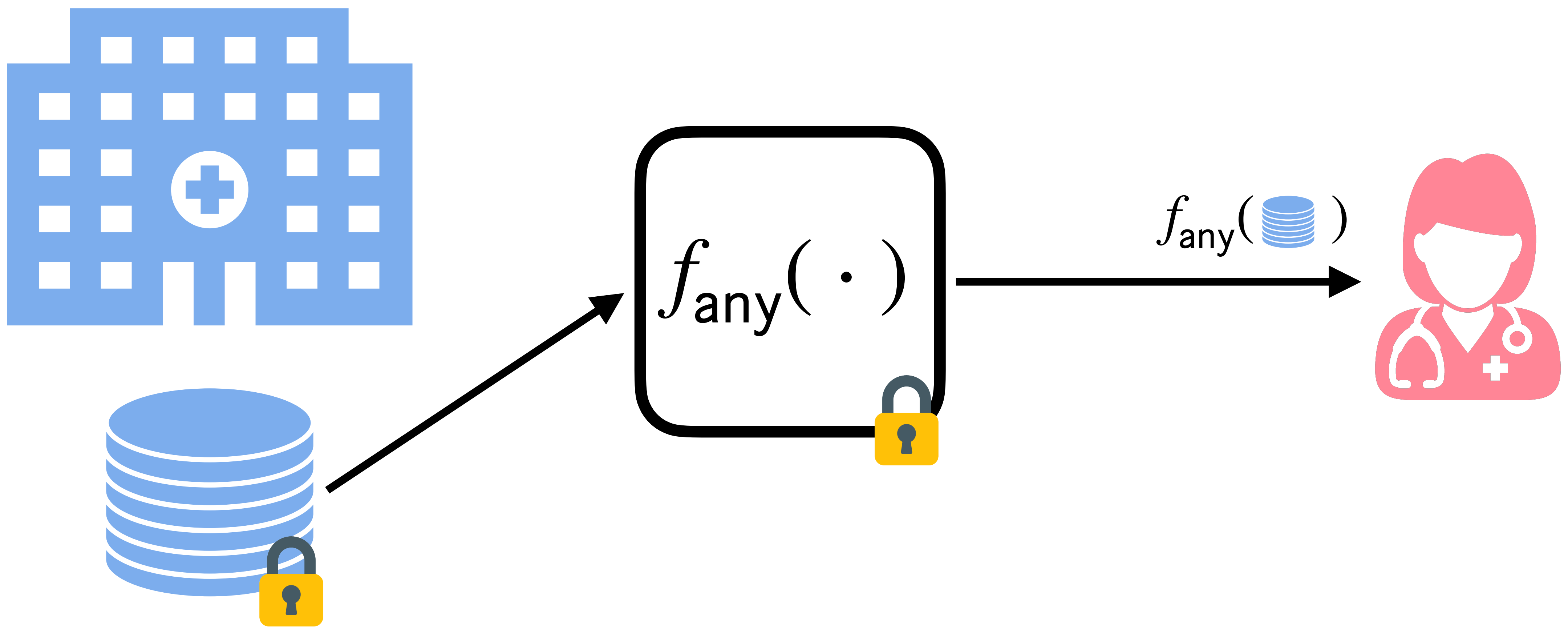
Patient data

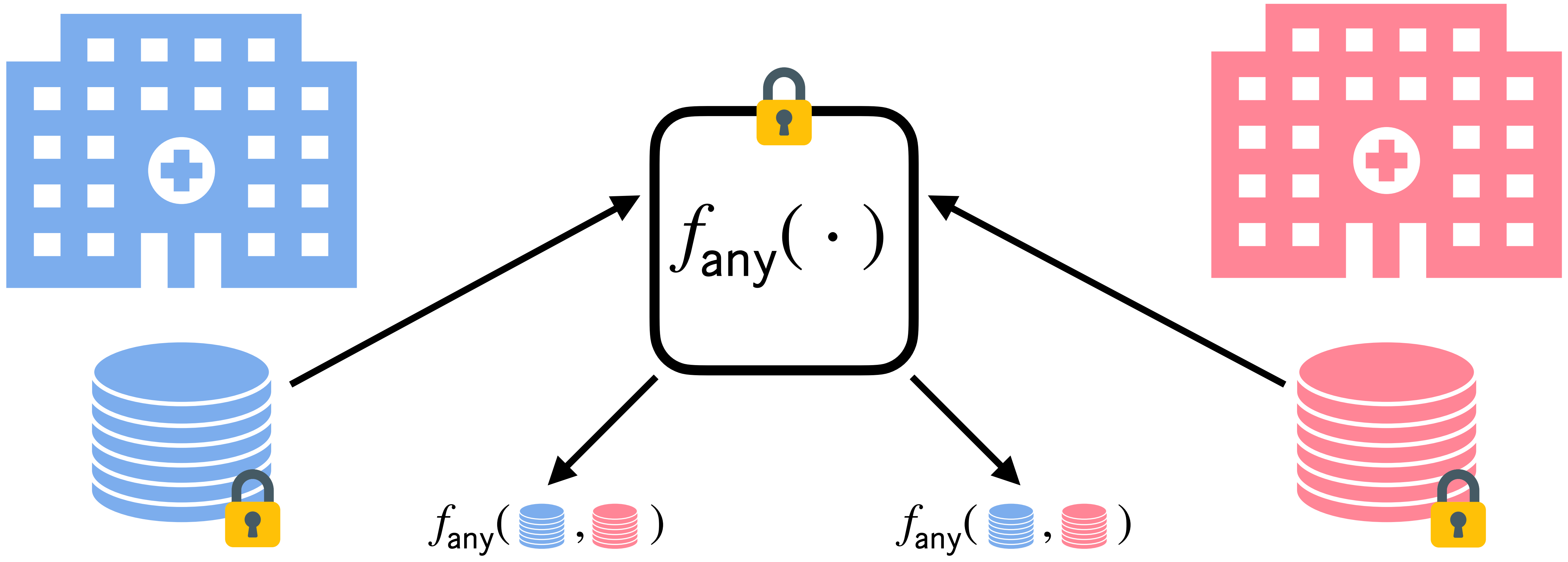
Secure Multi-Party Computation (MPC) [Yao86]

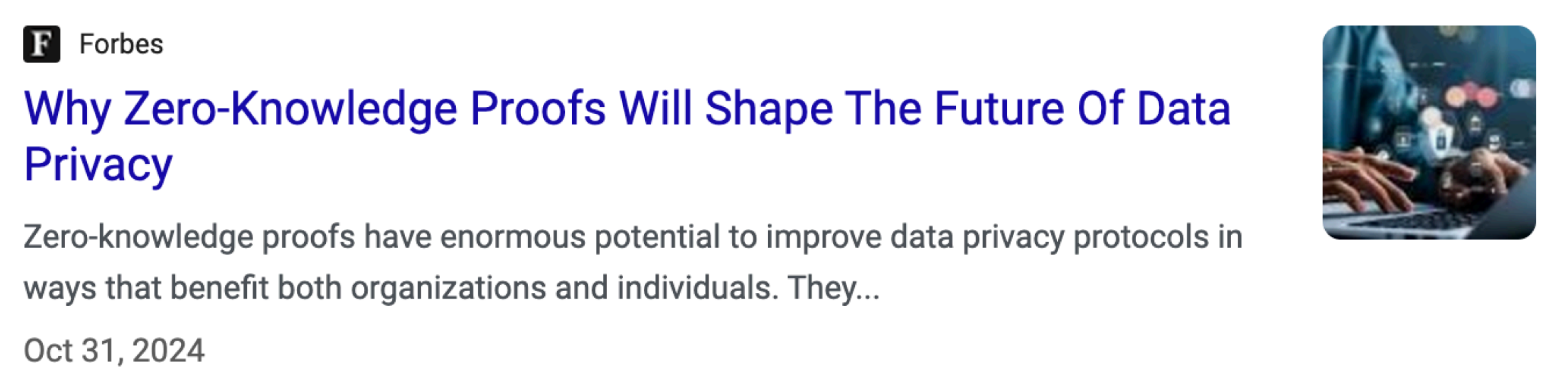


ZKP and MPC are *Generic*: Being capable of *any* function

ZKP and MPC are *Generic*: Being capable of *any* function





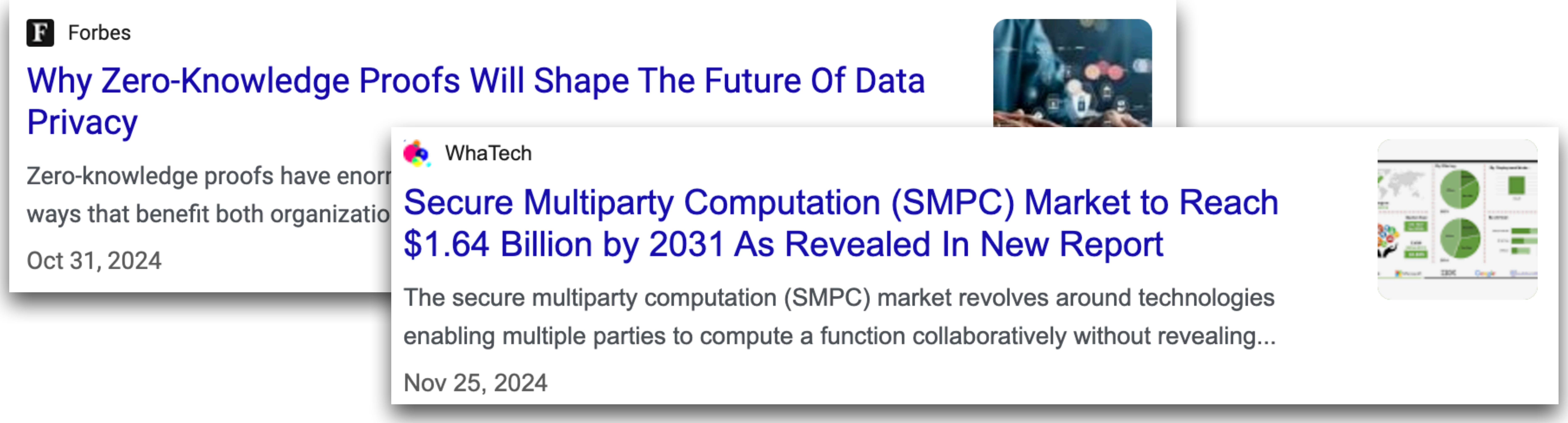


Forbes

Why Zero-Knowledge Proofs Will Shape The Future Of Data Privacy

Zero-knowledge proofs have enormous potential to improve data privacy protocols in ways that benefit both organizations and individuals. They...

Oct 31, 2024



Forbes

Why Zero-Knowledge Proofs Will Shape The Future Of Data Privacy

Zero-knowledge proofs have enorr
ways that benefit both organizatio

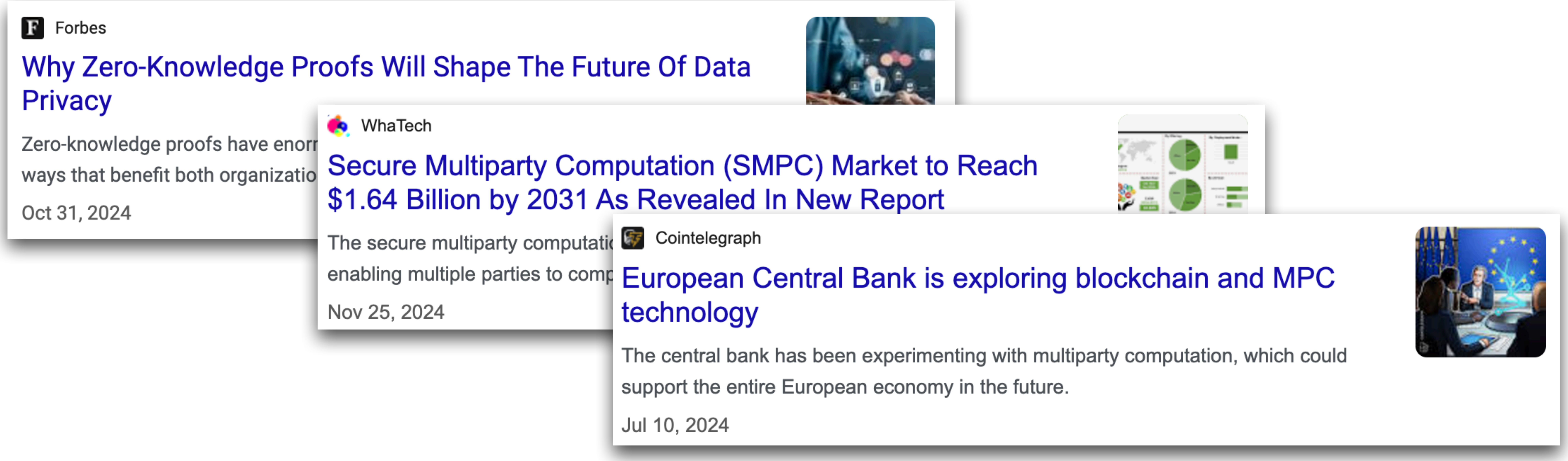
Oct 31, 2024

WhaTech

Secure Multiparty Computation (SMPC) Market to Reach $1.64 Billion by 2031 As Revealed In New Report

The secure multiparty computation (SMPC) market revolves around technologies enabling multiple parties to compute a function collaboratively without revealing...

Nov 25, 2024



Forbes

Why Zero-Knowledge Proofs Will Shape The Future Of Data Privacy

Zero-knowledge proofs have enorr
ways that benefit both organizatio

Oct 31, 2024

WhaTech

Secure Multiparty Computation (SMPC) Market to Reach $1.64 Billion by 2031 As Revealed In New Report

The secure multiparty computatic
enabling multiple parties to comp

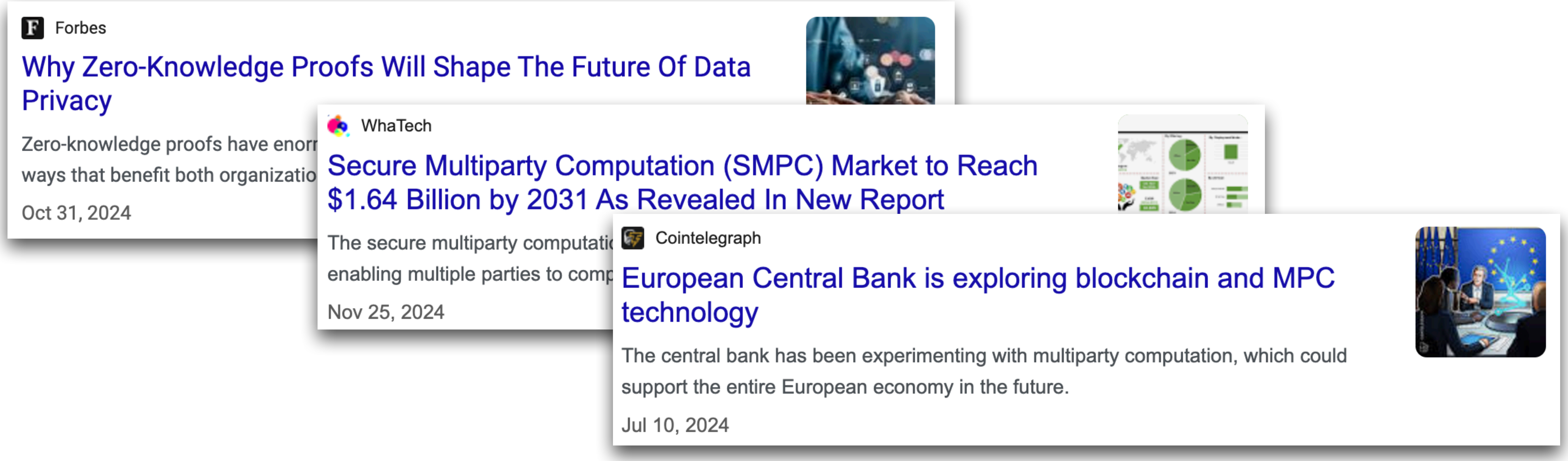
Nov 25, 2024

Cointelegraph

European Central Bank is exploring blockchain and MPC technology

The central bank has been experimenting with multiparty computation, which could support the entire European economy in the future.

Jul 10, 2024



Forbes

Why Zero-Knowledge Proofs Will Shape The Future Of Data Privacy

Zero-knowledge proofs have enorr
ways that benefit both organizatio

Oct 31, 2024

WhaTech

Secure Multiparty Computation (SMPC) Market to Reach $1.64 Billion by 2031 As Revealed In New Report

The secure multiparty computati
enabling multiple parties to comp

Nov 25, 2024

Cointelegraph

European Central Bank is exploring blockchain and MPC technology

The central bank has been experimenting with multiparty computation, which could support the entire European economy in the future.

Jul 10, 2024

Meta Microsoft coinbase NEXUS

amazon VISA Ligero Algorand

ZKP and MPC deployments are rare

ZKP and MPC deployments are rare

Poor Usability

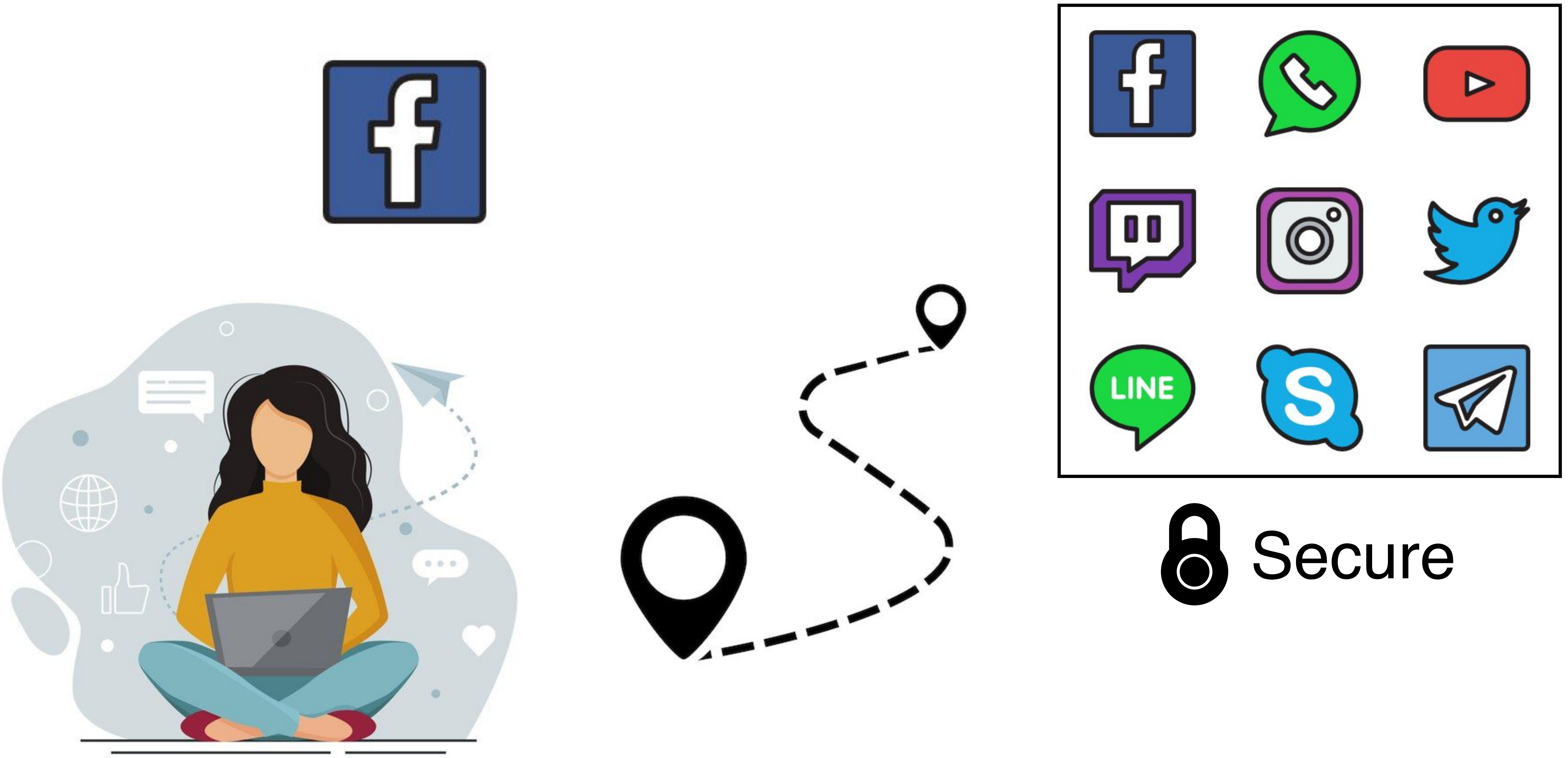
Mastering these techniques requires a notably steep learning curve

Poor Usability

Mastering these techniques requires a notably steep learning curve

Poor Usability

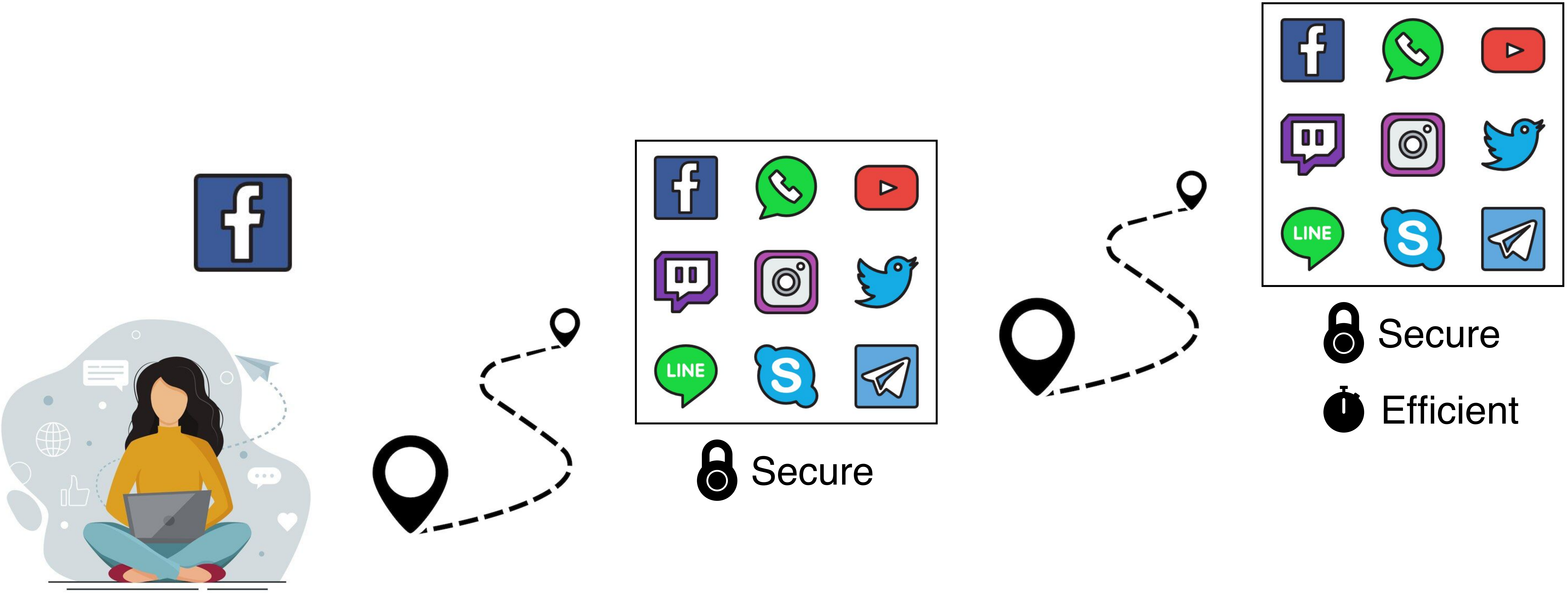
Mastering these techniques requires a notably steep learning curve



Secure

Poor Usability

Mastering these techniques requires a notably steep learning curve

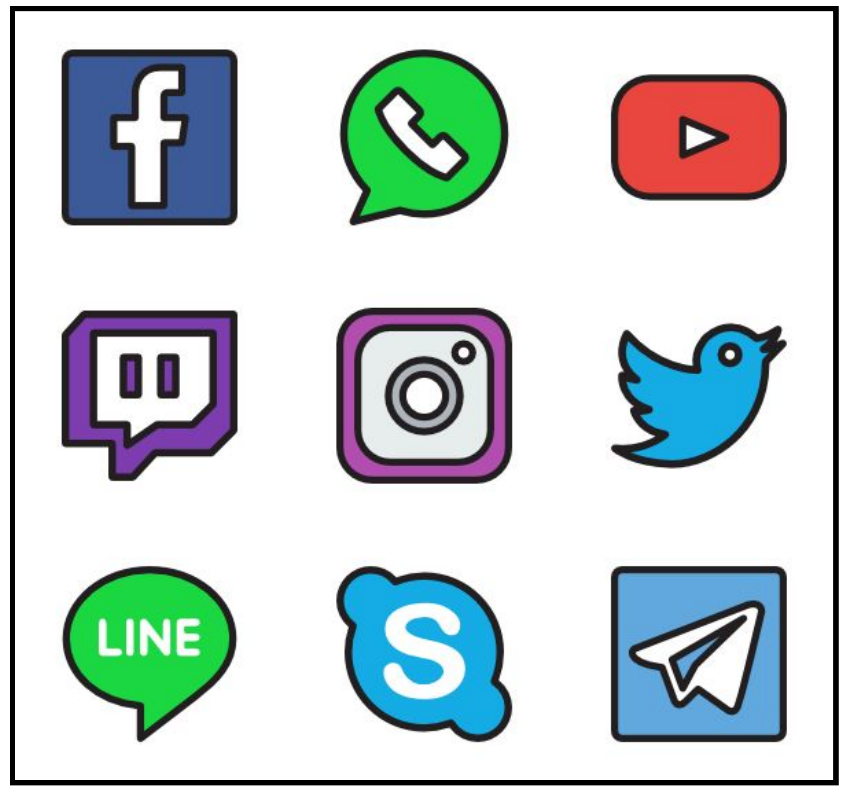


Secure

Secure

Efficient

My Research Focus

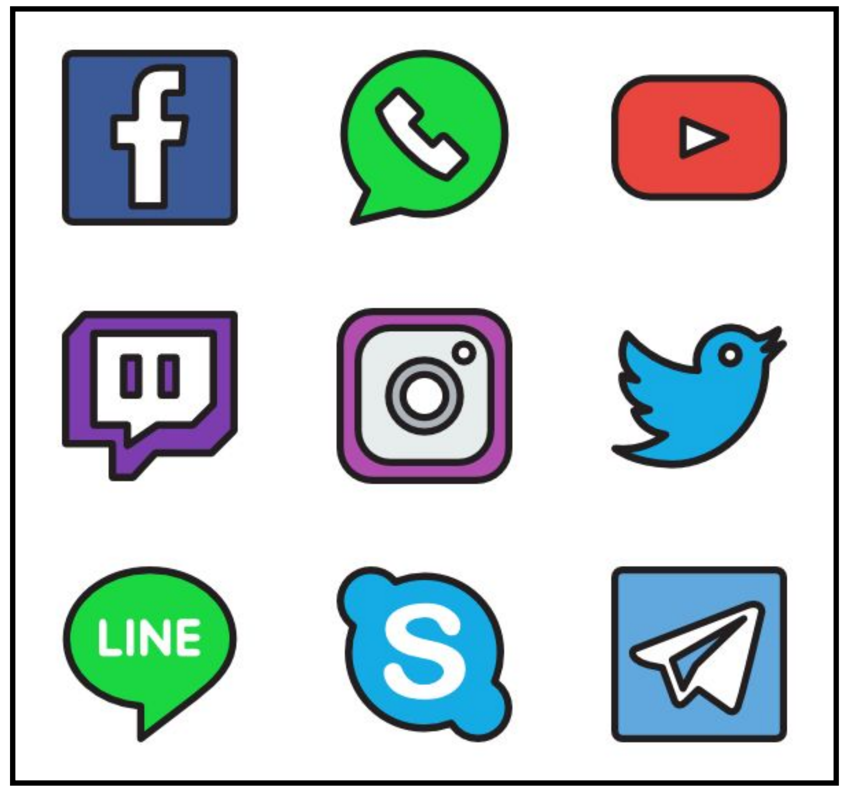


🔒 Secure

⏱ Efficient

My Research Focus

Generic Toolchains:
Being capable of *any real-world* computation

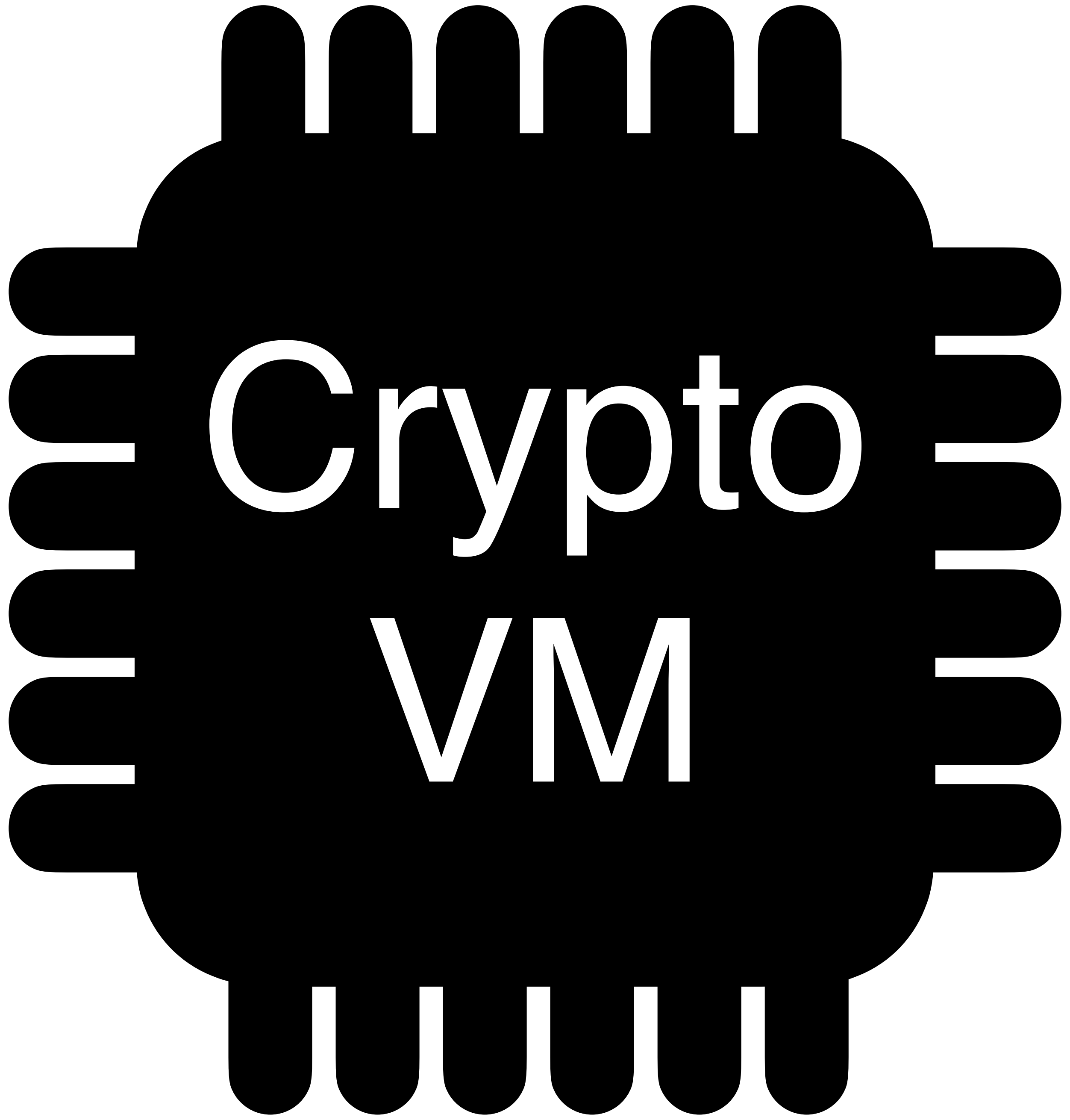


🔒 Secure

⏱ Efficient

My Research Focus

Generic Toolchains:
Being capable of *any real-world* computation



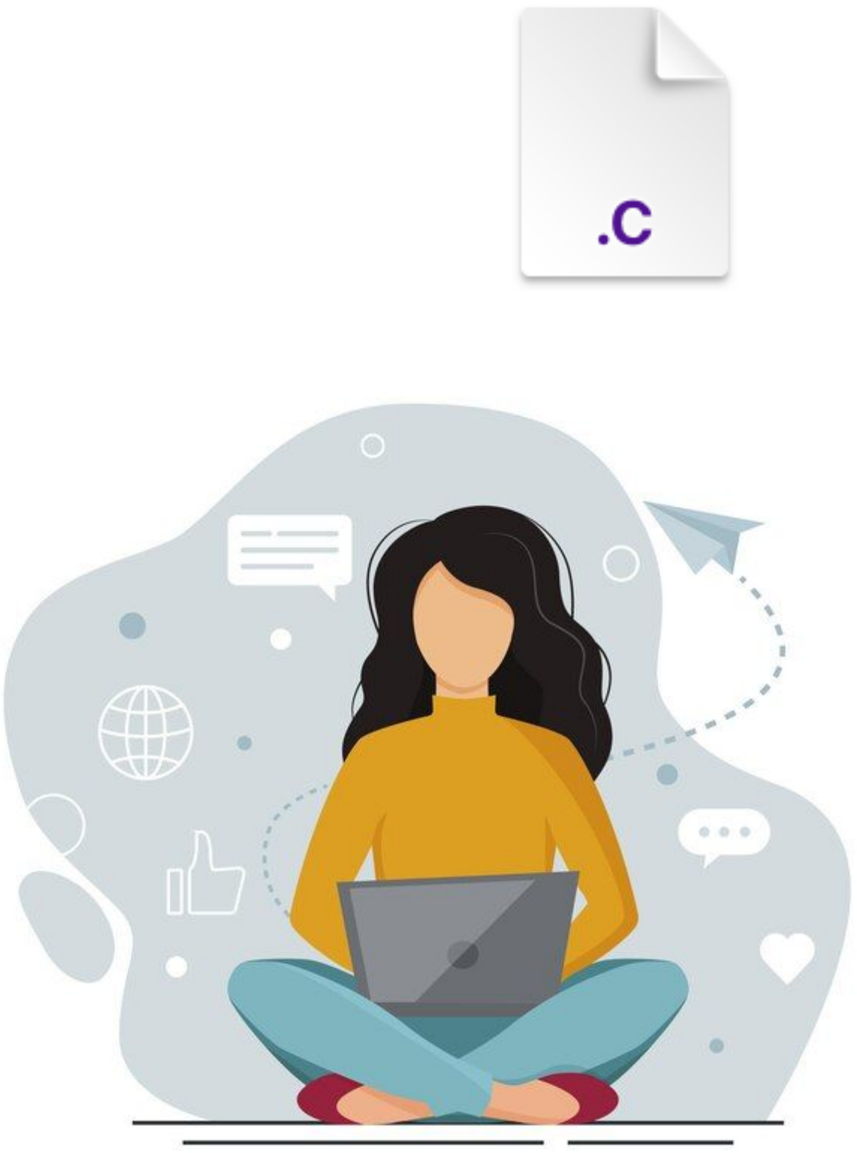
Crypto
VM

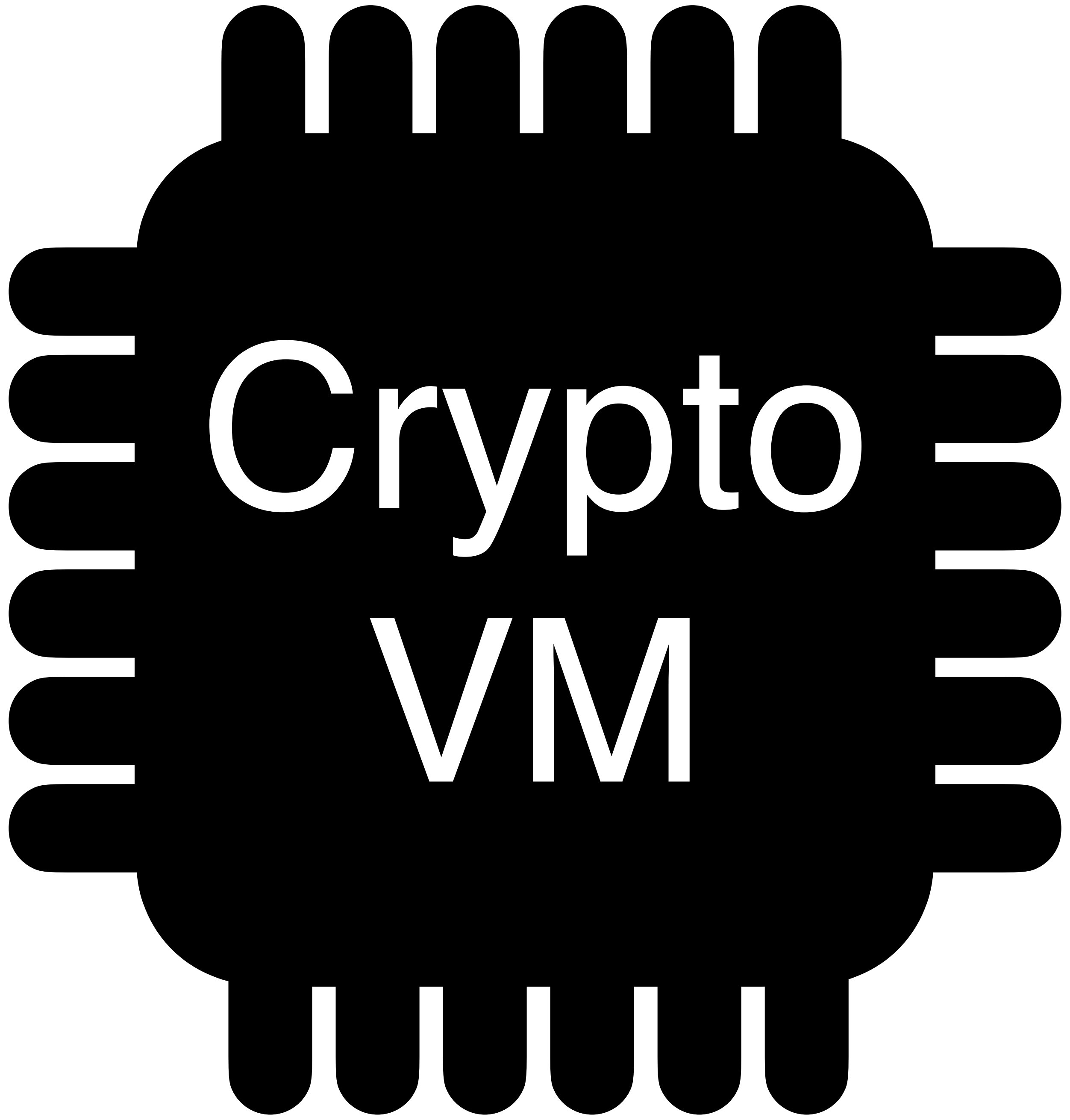
Secure

Efficient

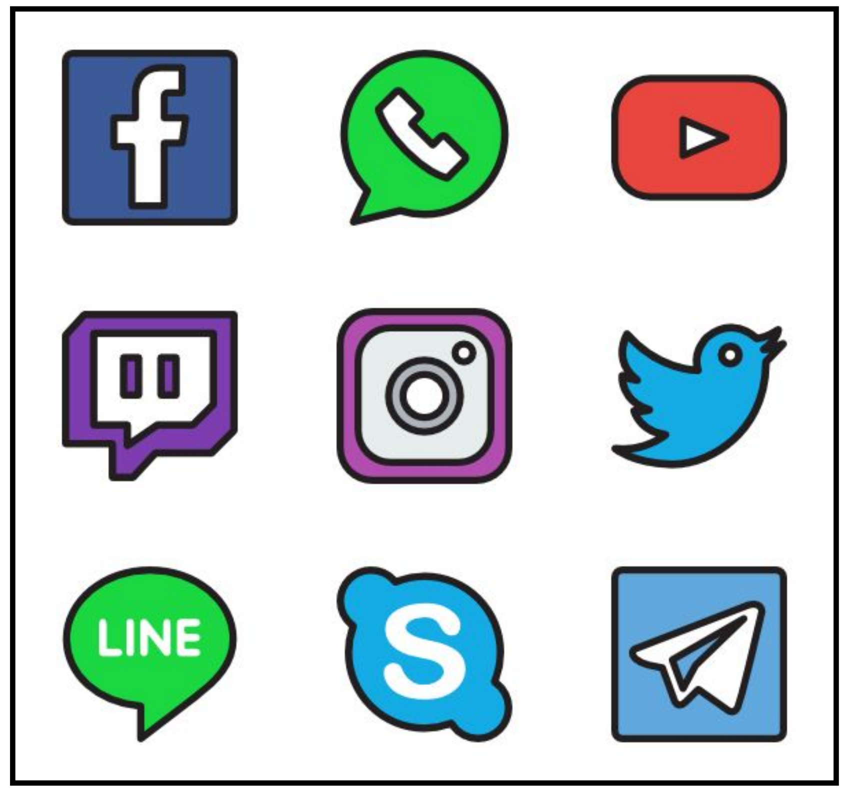
My Research Focus

Generic Toolchains:
Being capable of *any real-world* computation





Crypto VM

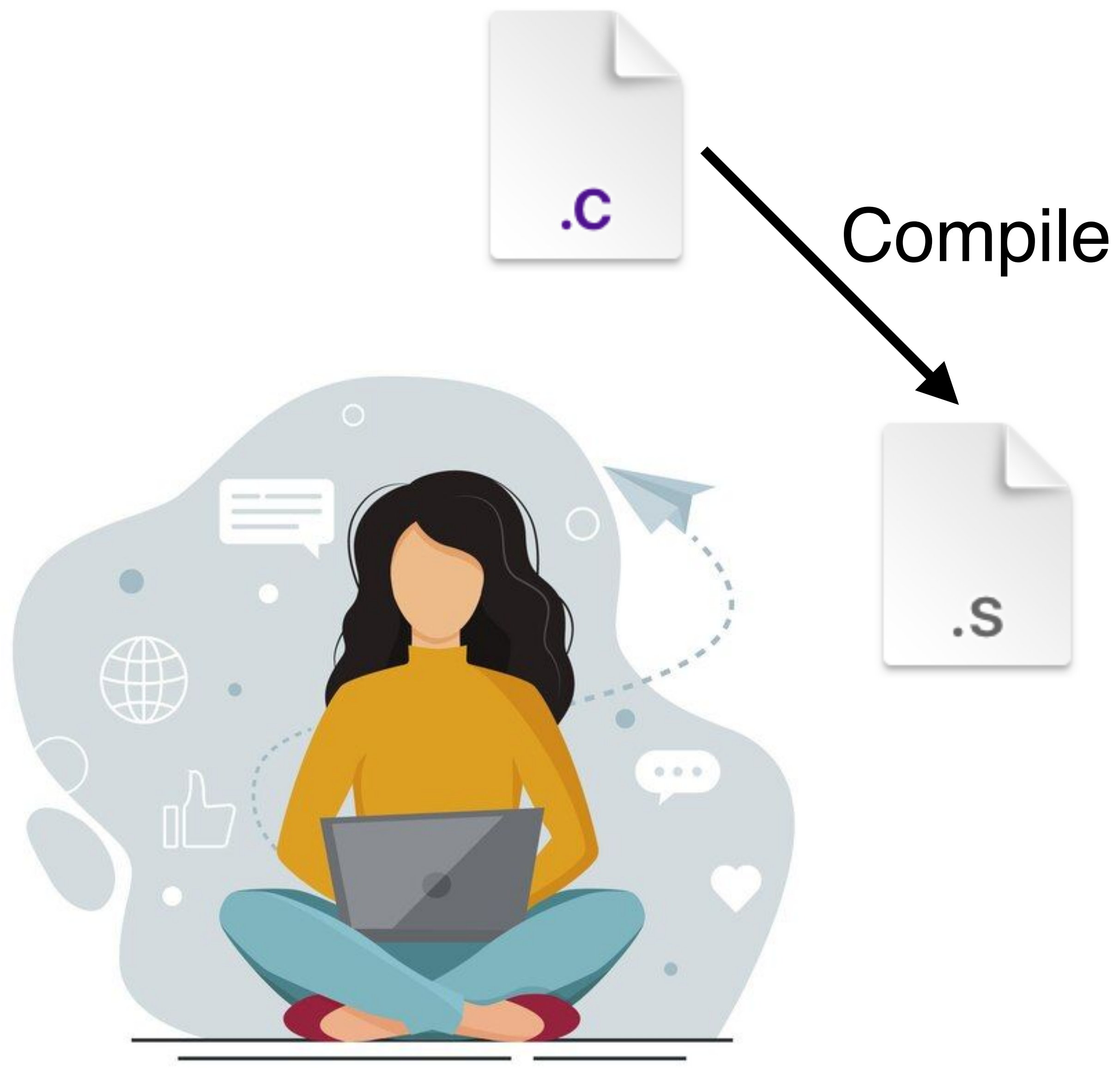


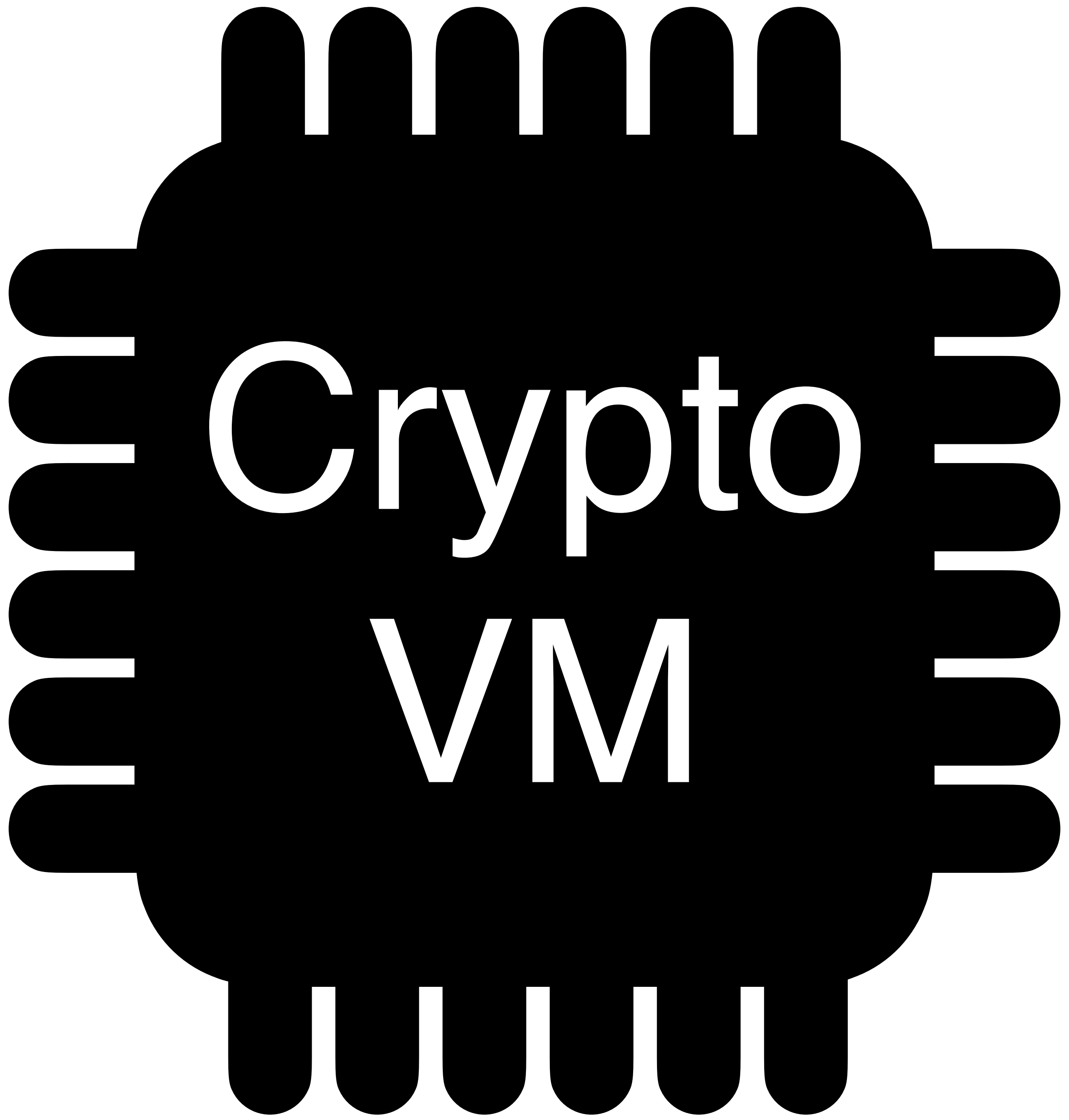
🔒 Secure

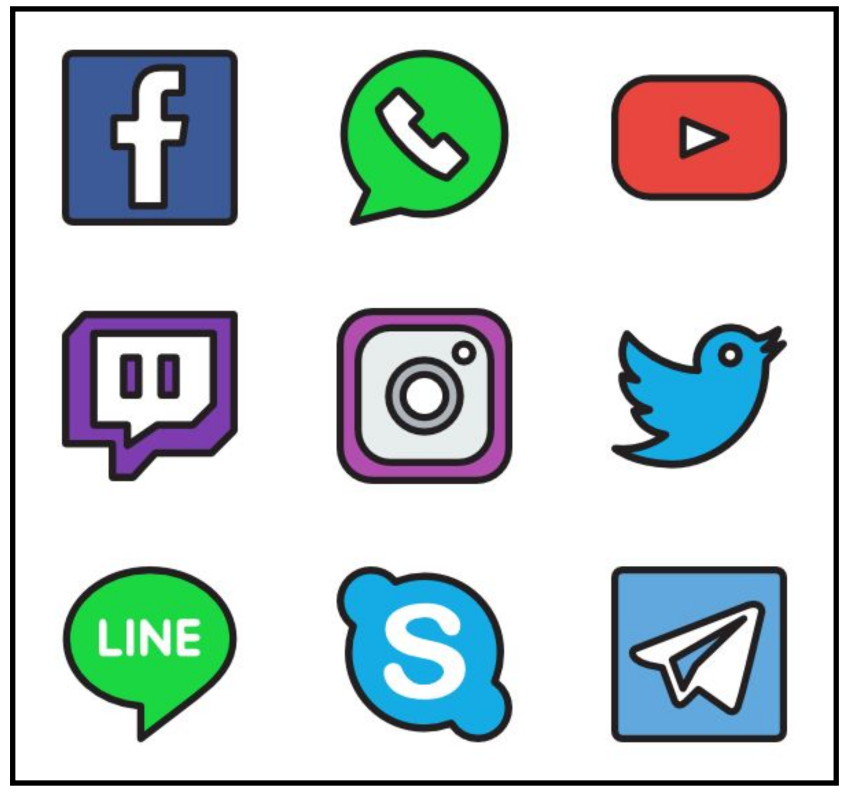
⏱ Efficient

My Research Focus

Generic Toolchains:
Being capable of *any real-world* computation





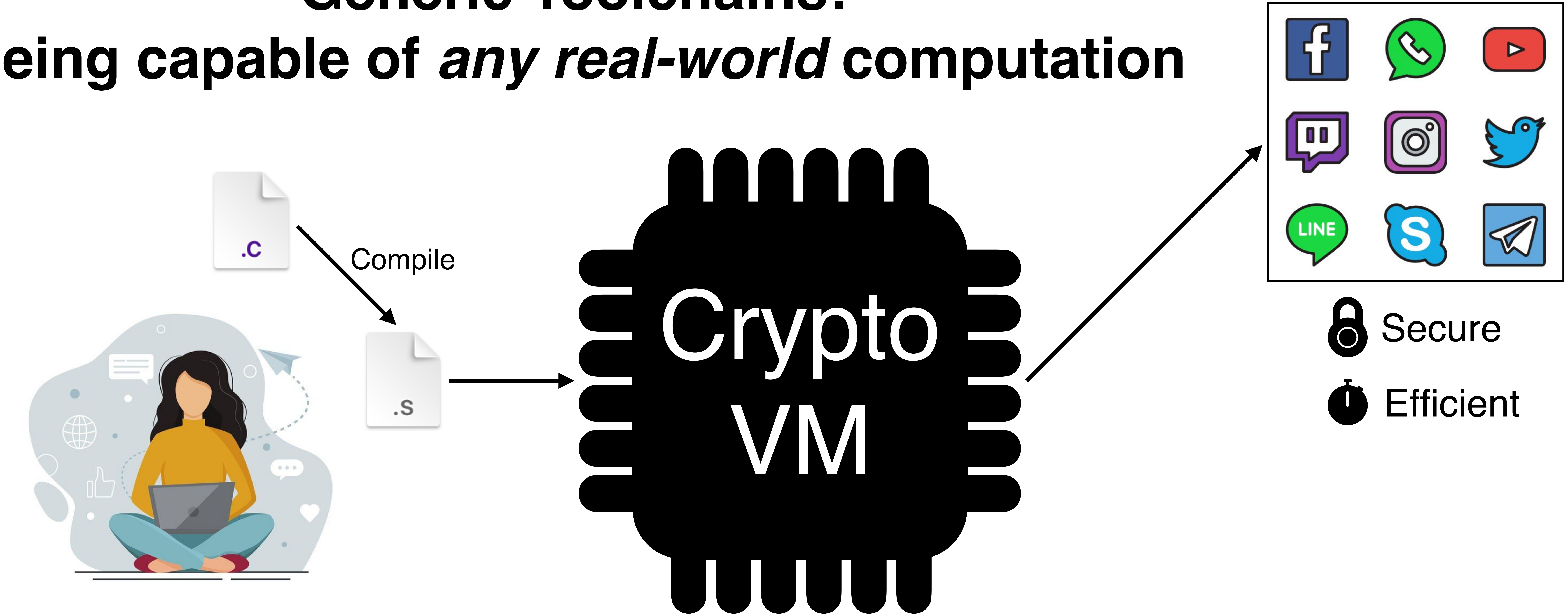


🔒 Secure

⏱ Efficient

My Research Focus

Generic Toolchains:
Being capable of *any real-world* computation



.c

Compile

.s

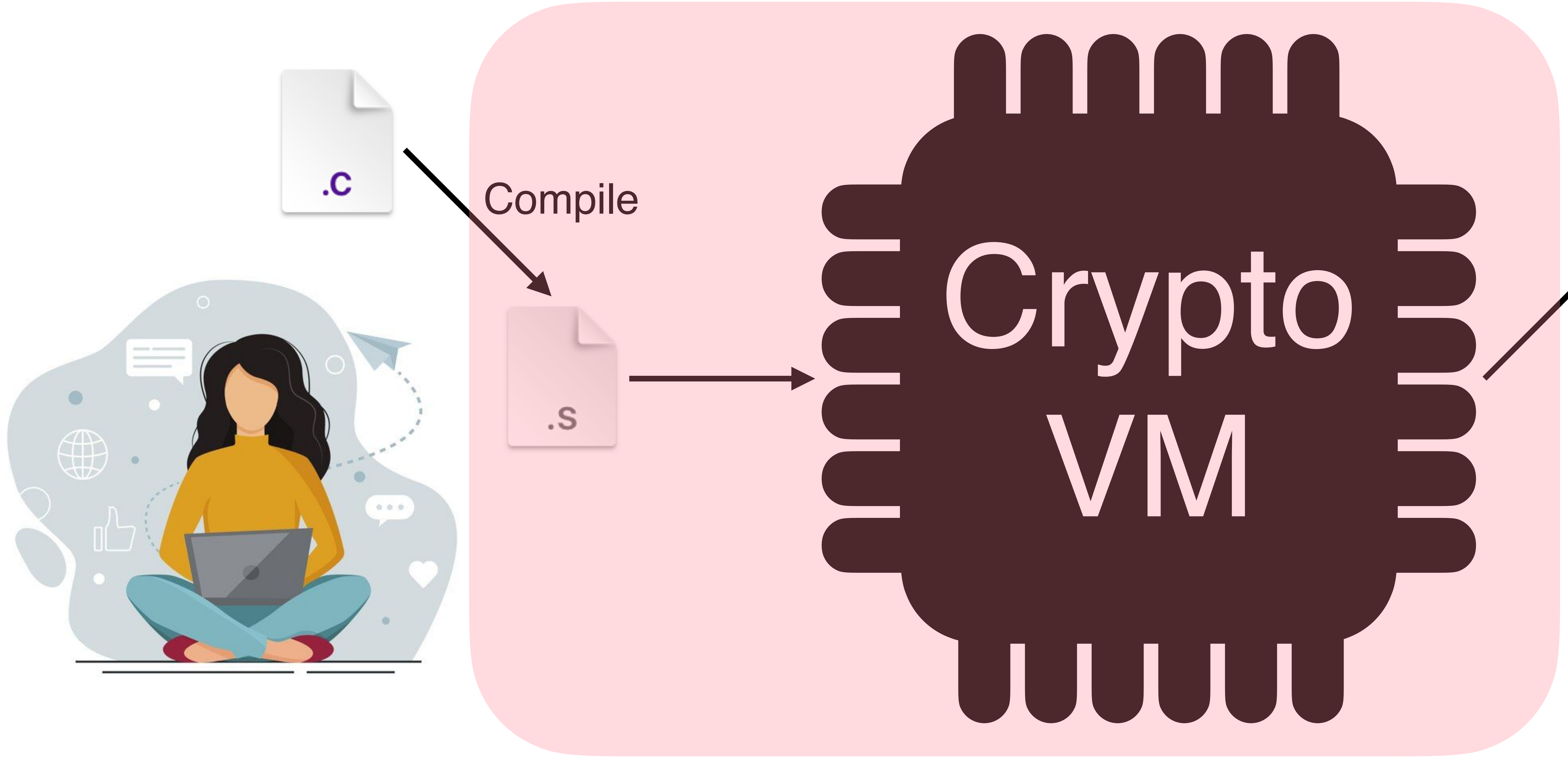
Crypto VM

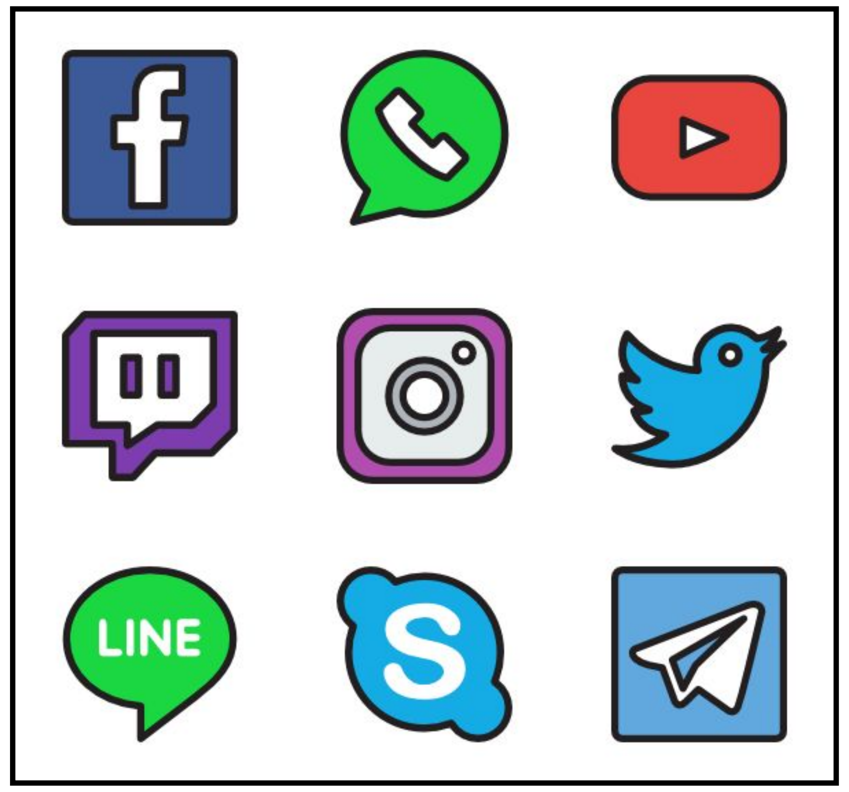
Secure

Efficient

My Research Focus

Generic Toolchains:
Being capable of *any real-world* computation

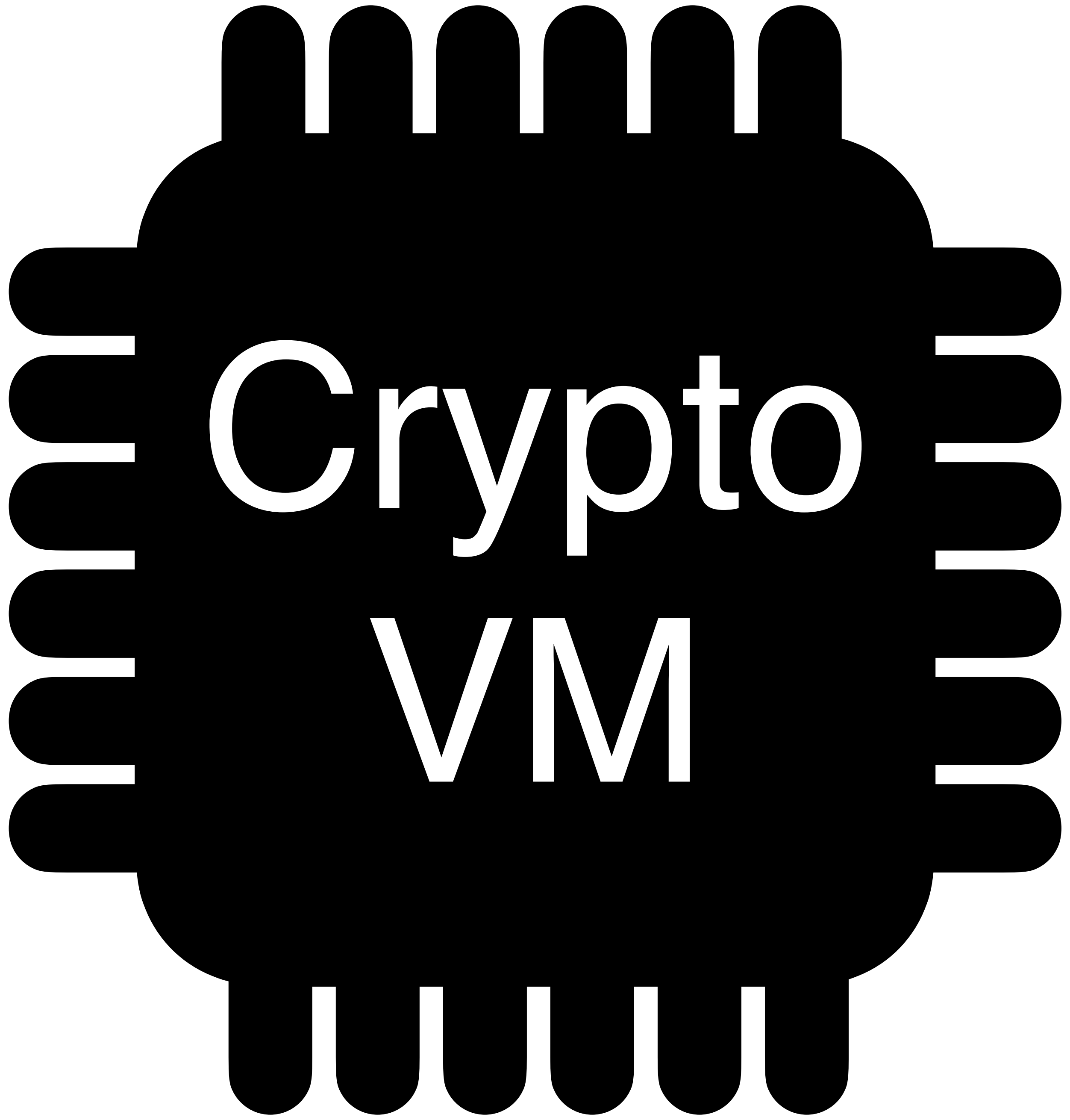




🔒 Secure

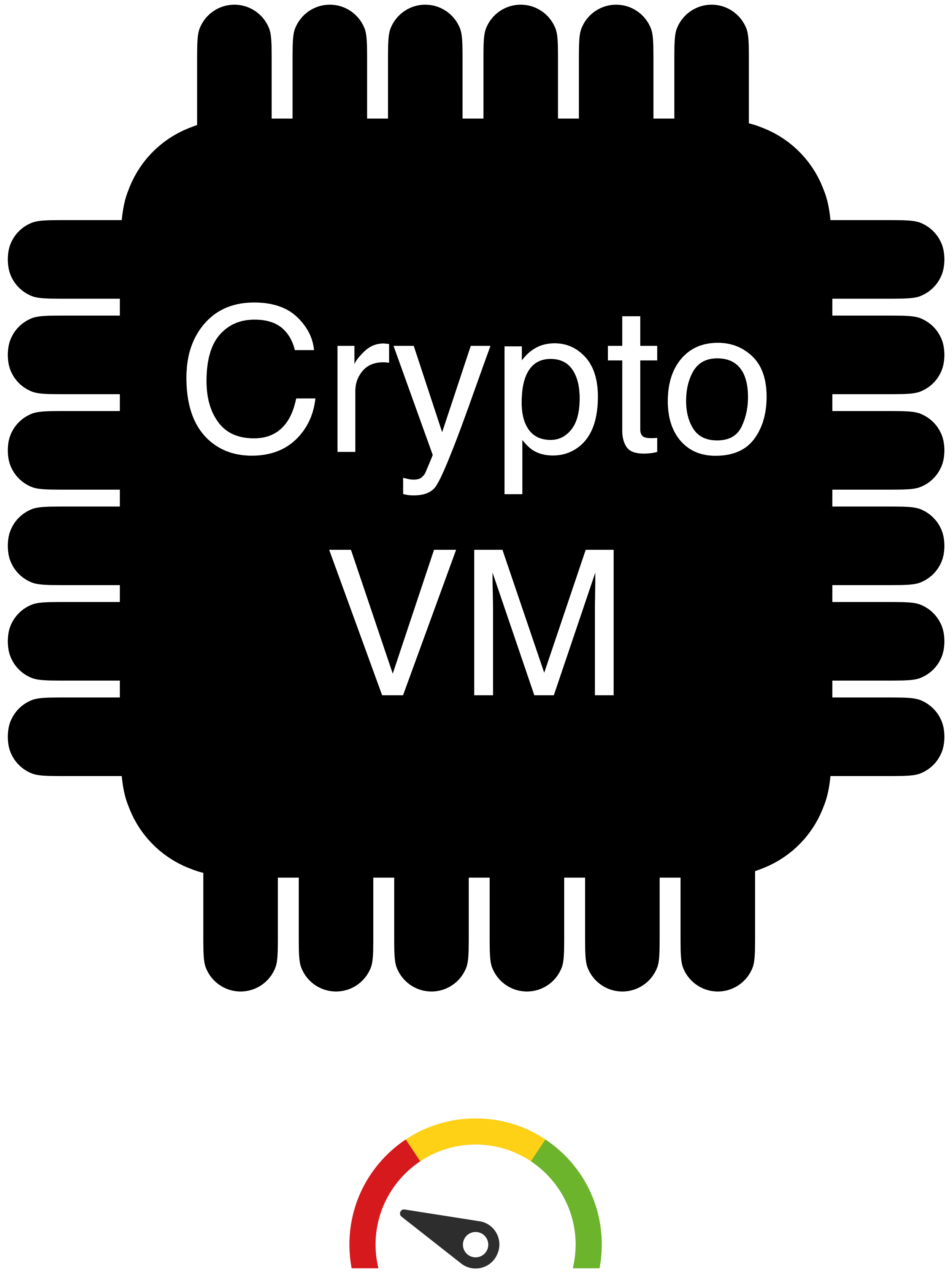
⏱ Efficient

Do We Already Have Such a VM?

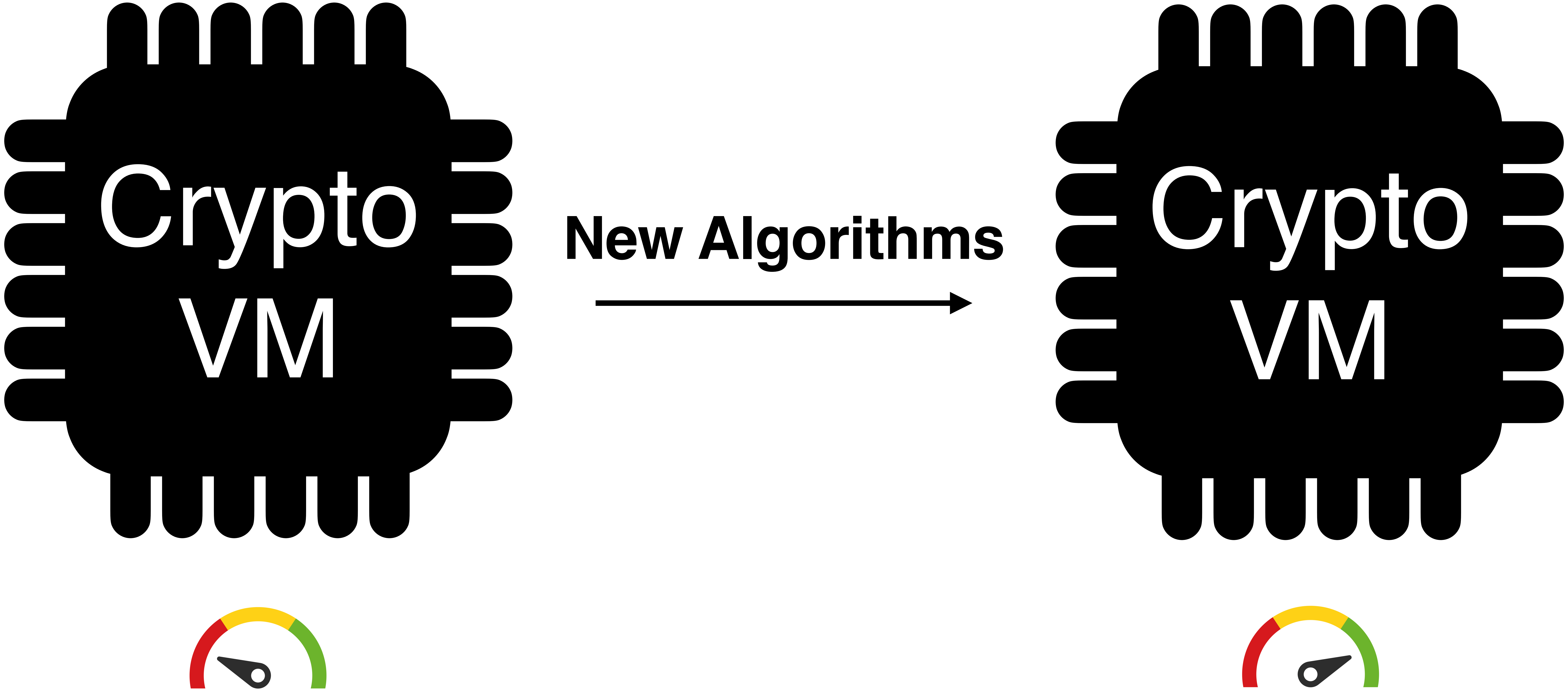


Crypto VM

Do We Already Have Such a VM?



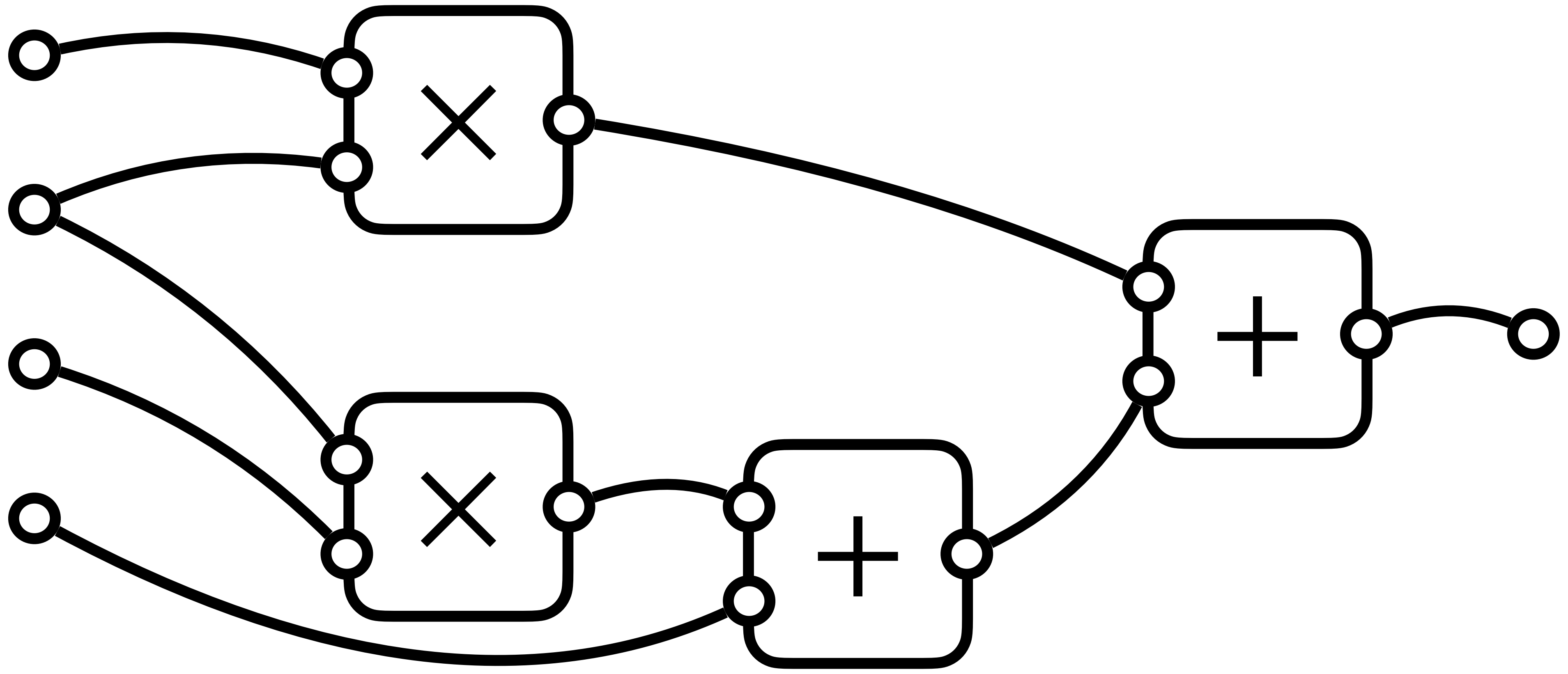
Do We Already Have Such a VM?



Existing Generic Methods: Being capable of *any* computation

My Generic Toolchains: Being capable of *any real-world* computation

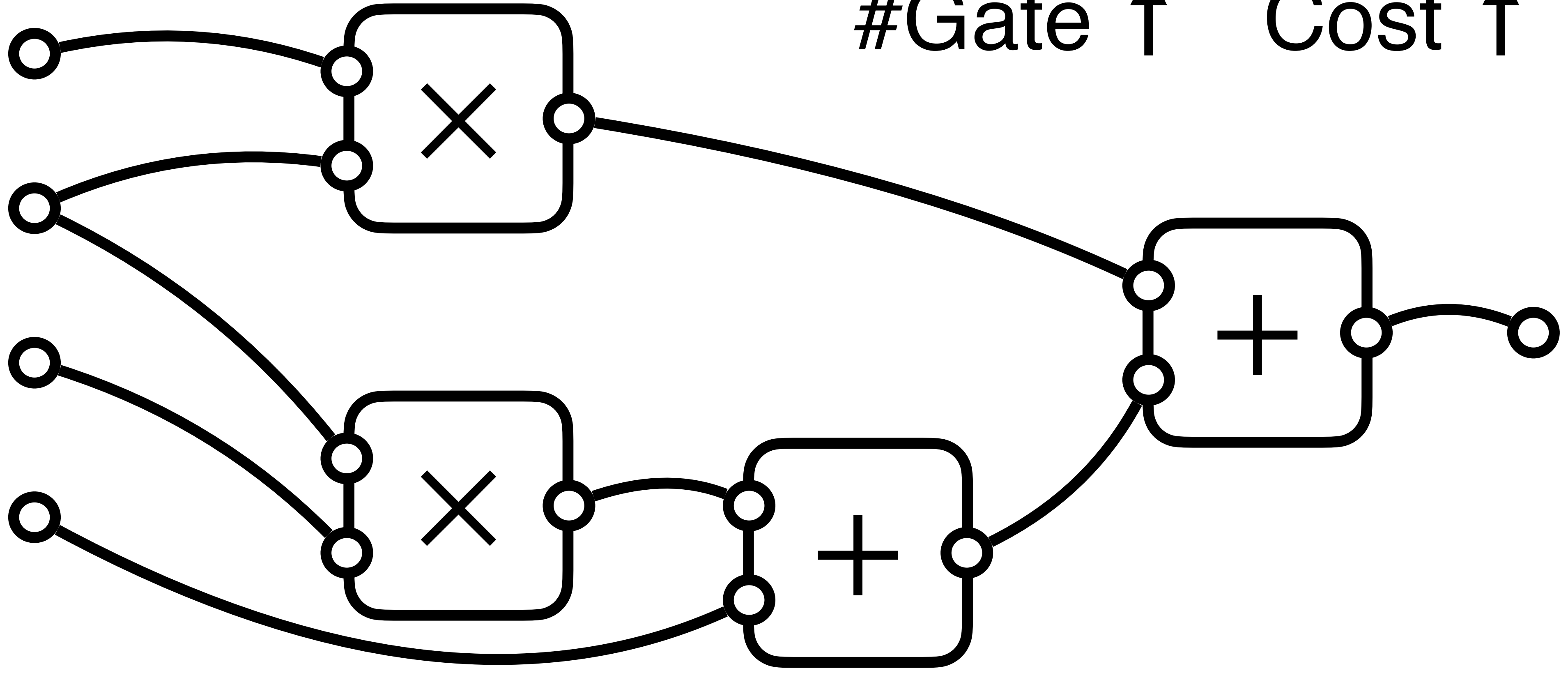
Existing Generic Methods: Being capable of *any* computation



My Generic Toolchains: Being capable of *any real-world* computation

Existing Generic Methods:
Being capable of *any* computation

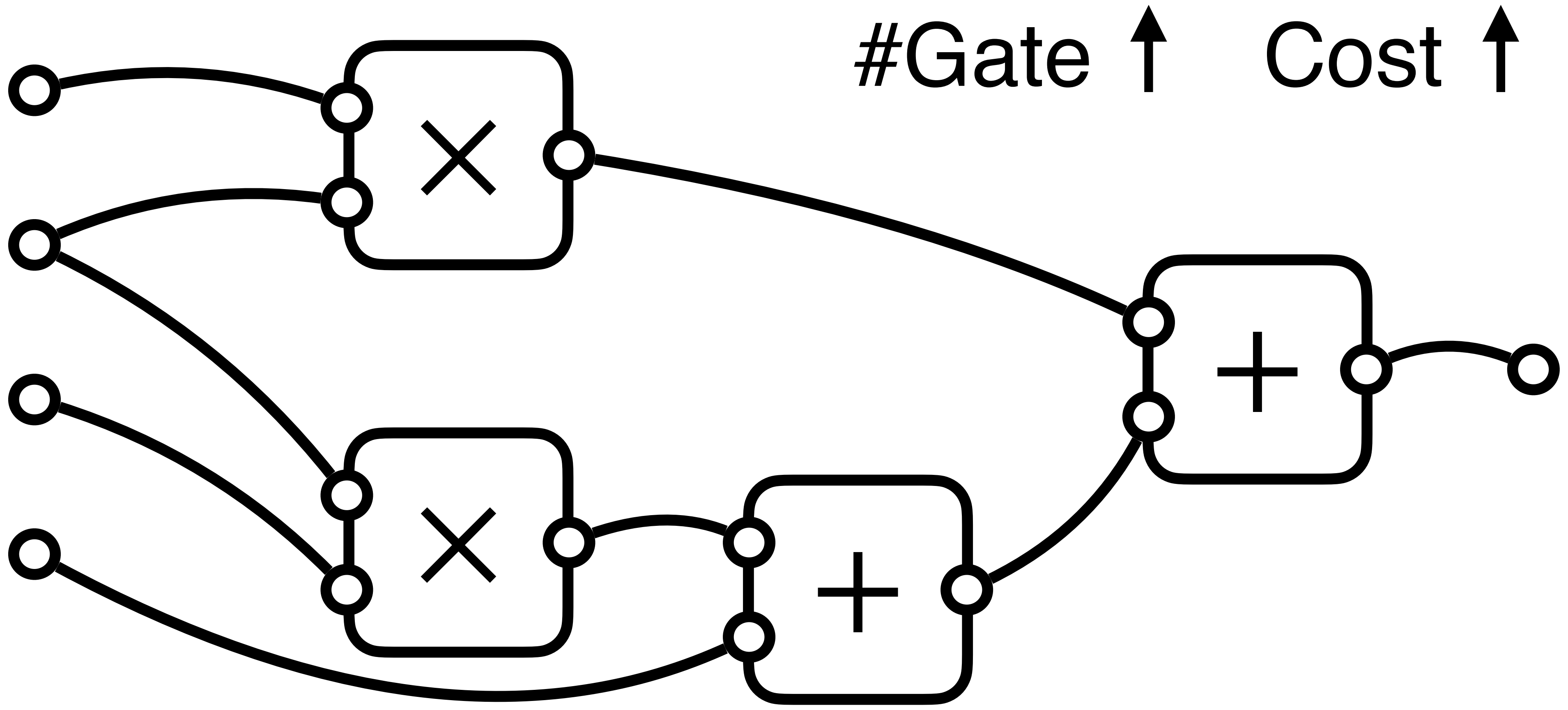
My Generic Toolchains:
Being capable of *any real-world* computation



#Gate ↑ Cost ↑

Existing Generic Methods: Being capable of *any* computation

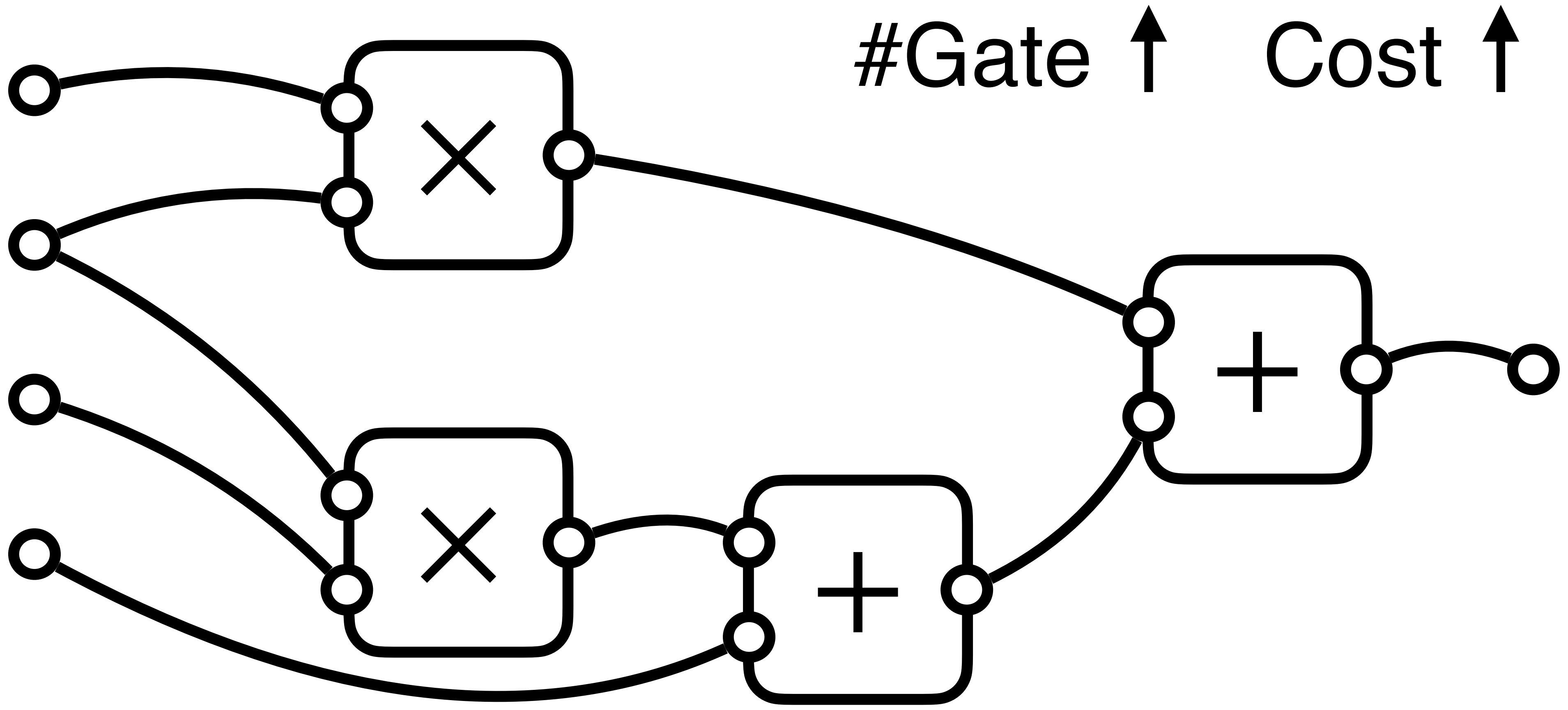
#Gate ↑ Cost ↑



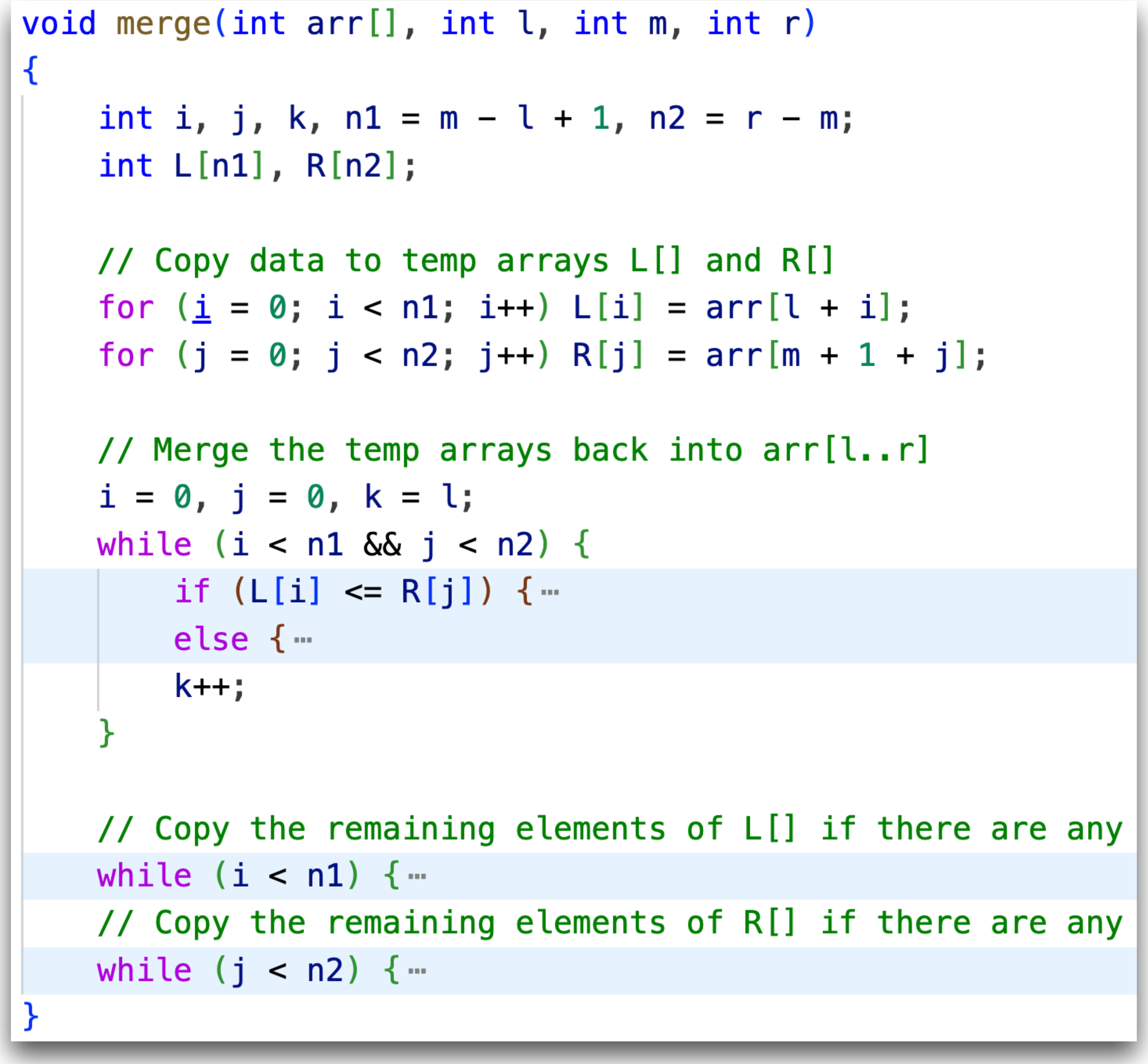
My Generic Toolchains: Being capable of *any real-world* computation

.c

Existing Generic Methods: Being capable of *any* computation



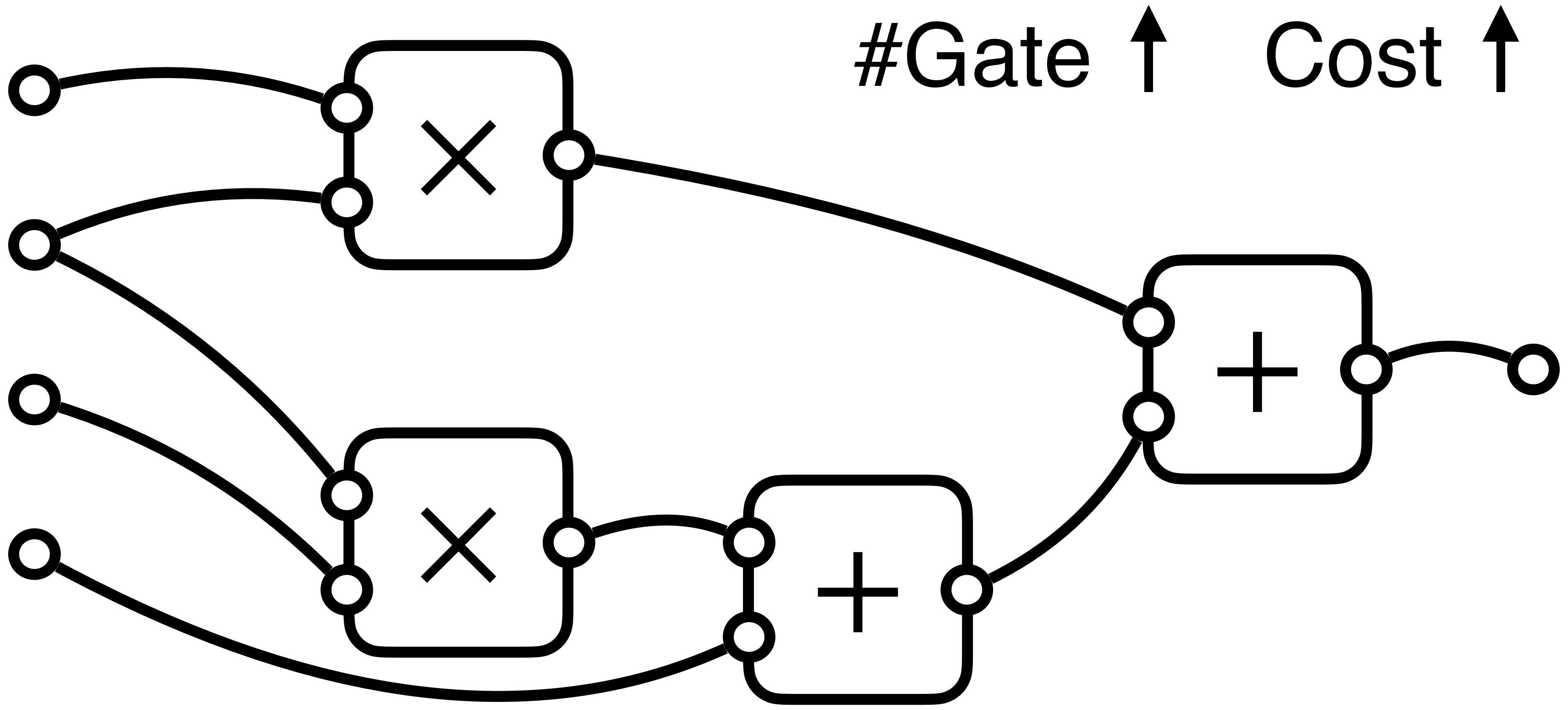
My Generic Toolchains: Being capable of *any real-world* computation



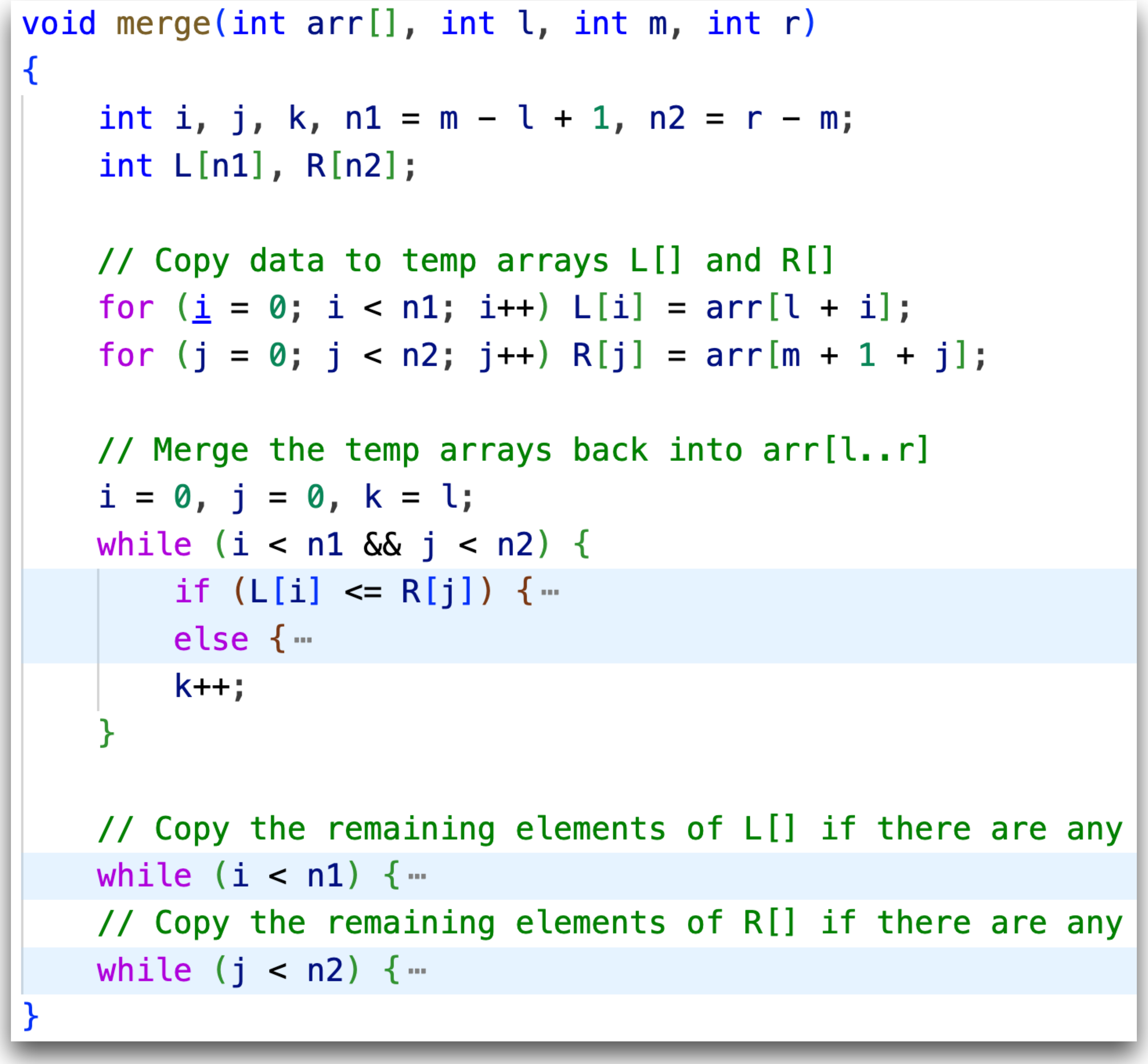
```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Existing Generic Methods:
Being capable of *any* computation

My Generic Toolchains:
Being capable of *any real-world* computation

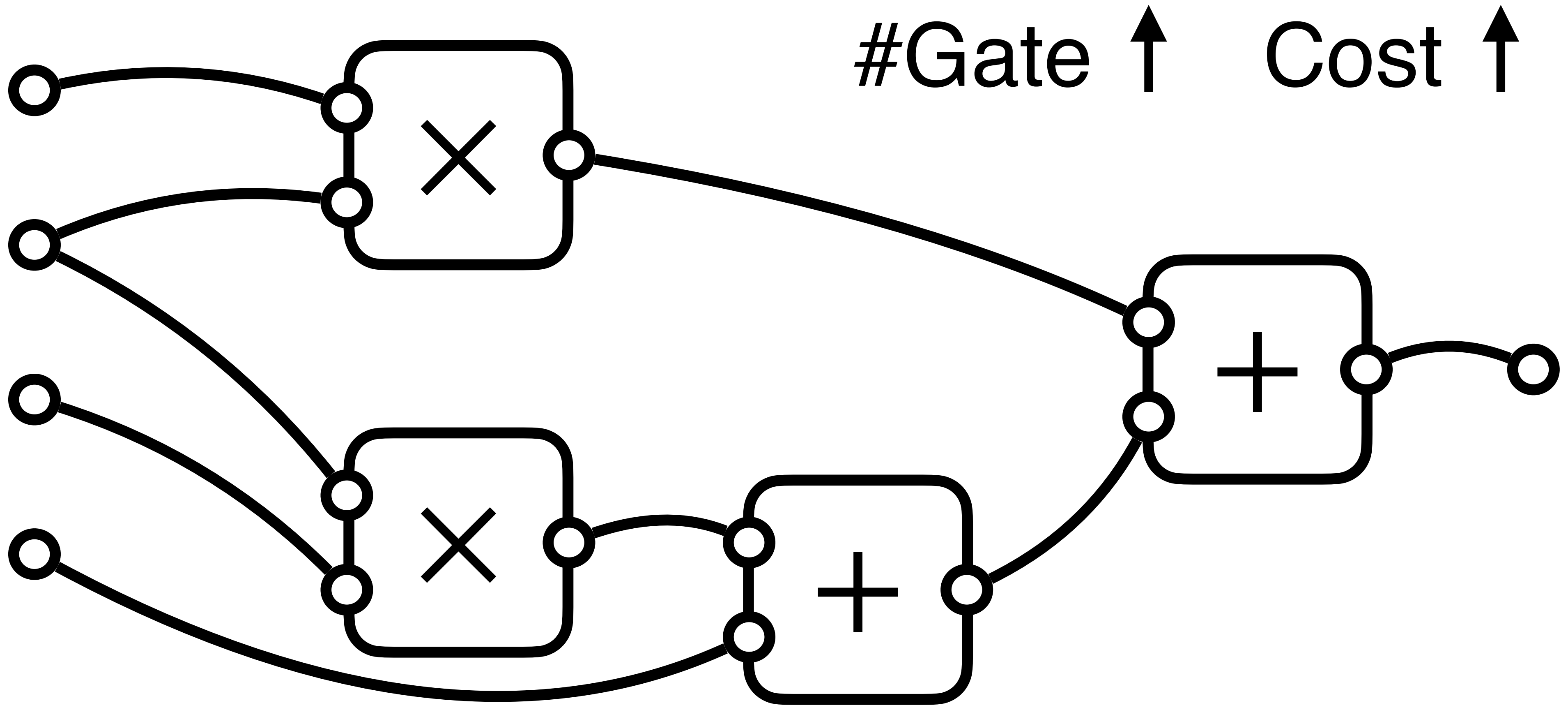


#Gate ↑ Cost ↑



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Existing Generic Methods: Being capable of *any* computation

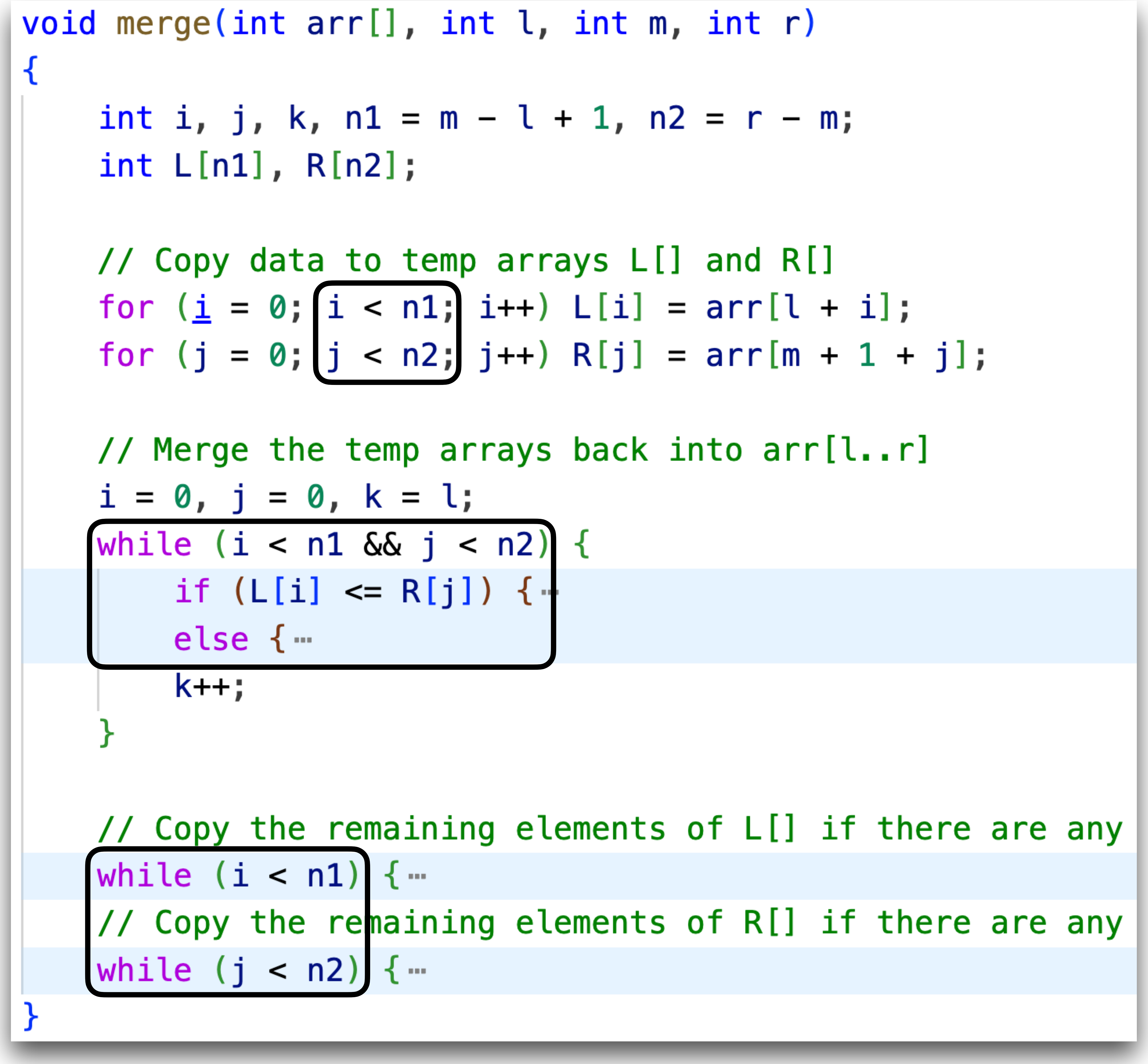


#Gate ↑ Cost ↑

Branching

if C_0 else C_1

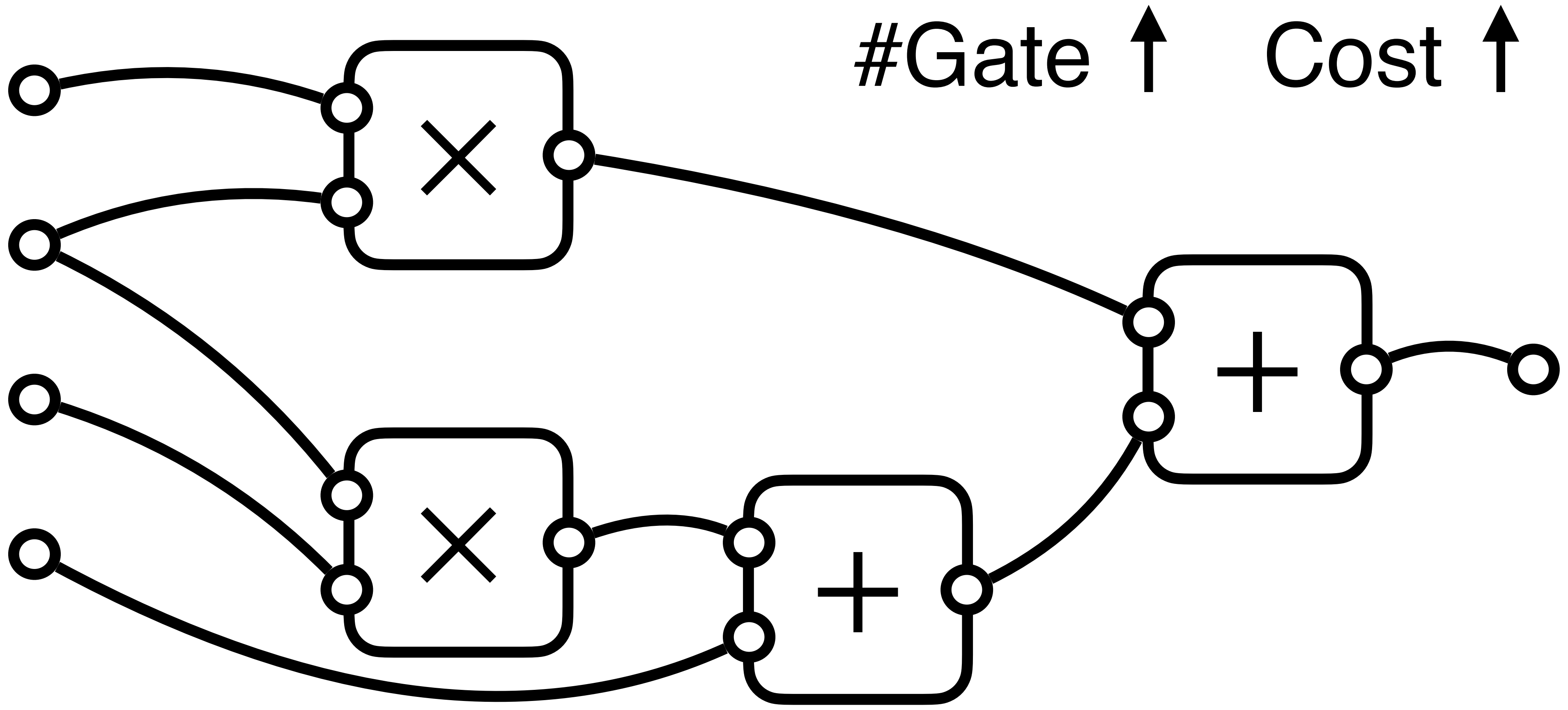
My Generic Toolchains: Being capable of *any real-world* computation



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Existing Generic Methods: Being capable of *any* computation

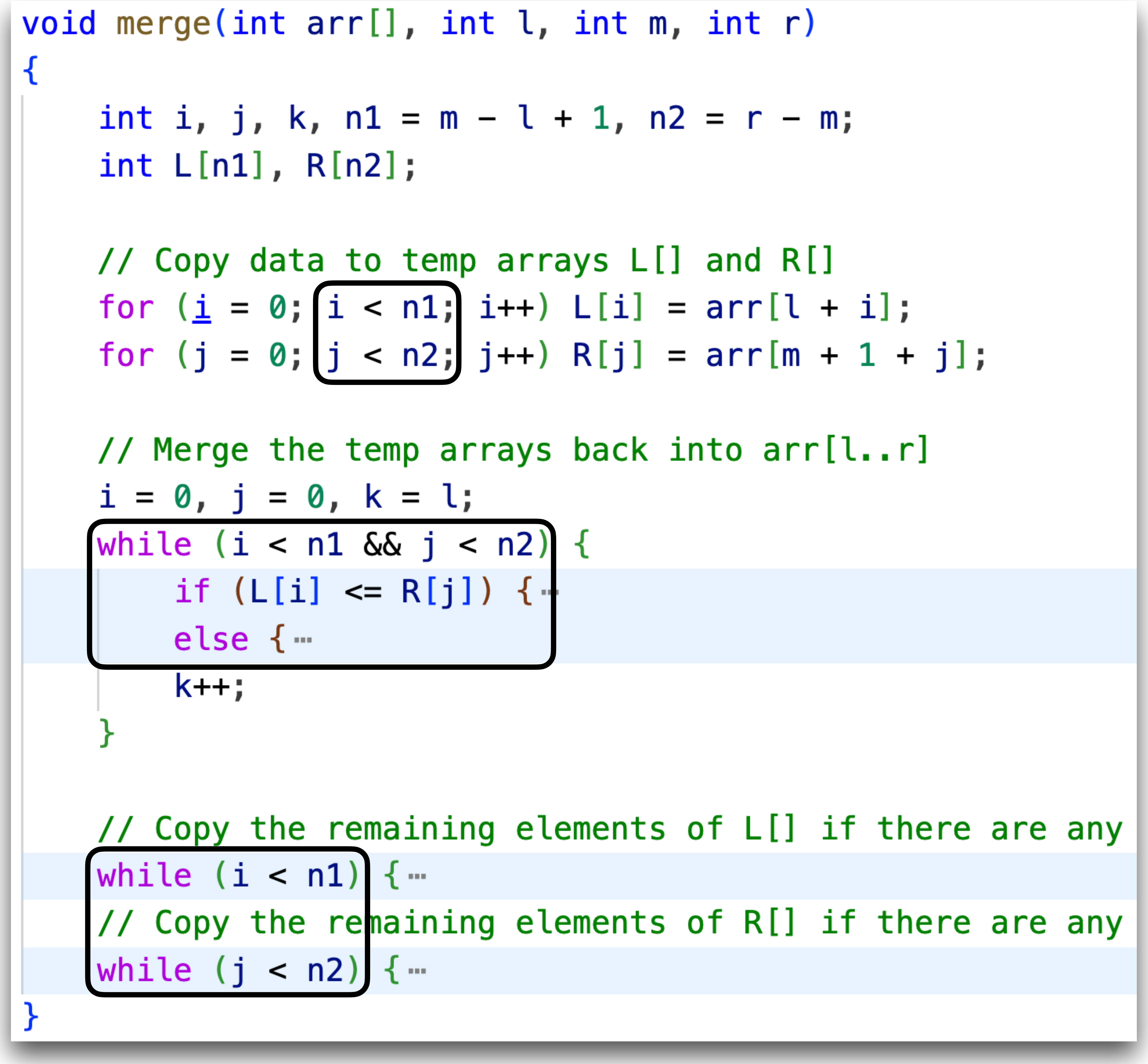
My Generic Toolchains: Being capable of *any real-world* computation



#Gate ↑ Cost ↑

Branching if C_0 else C_1

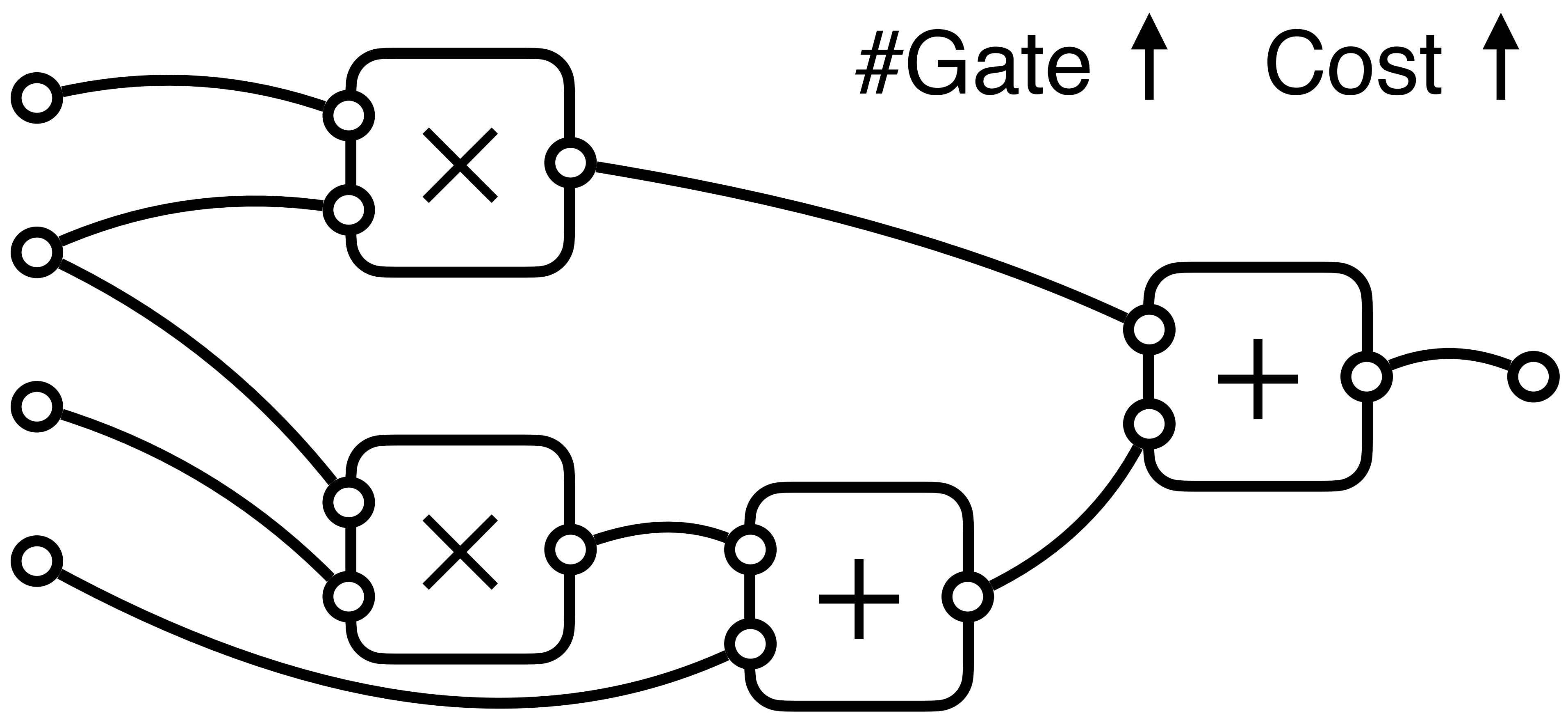
$$C = (1-b)C_0 + bC_1 \quad |C| \approx |C_0| + |C_1|$$



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Existing Generic Methods: Being capable of *any* computation

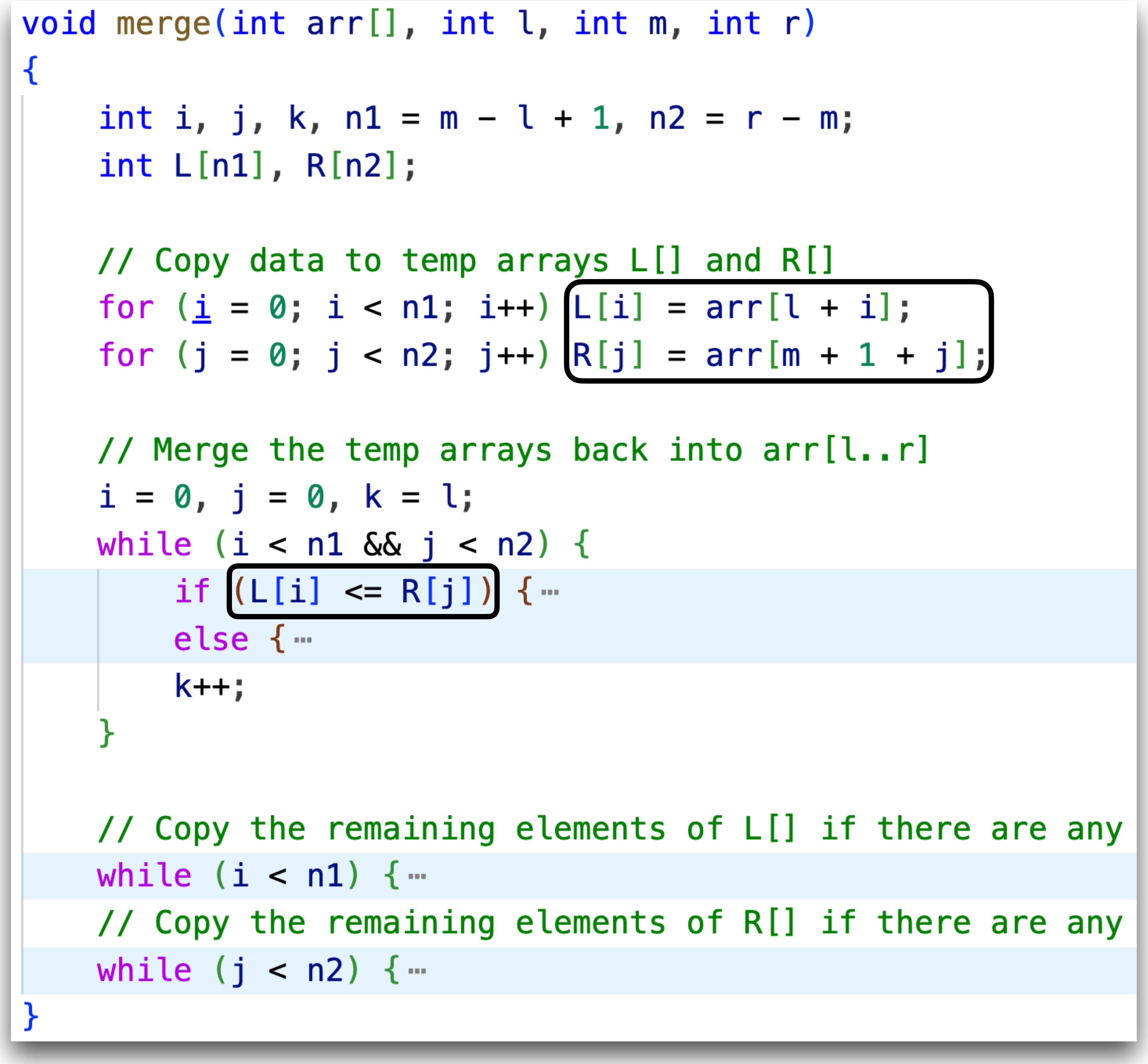
My Generic Toolchains: Being capable of *any real-world* computation



Branching if C_0 else C_1

$$C = (1 - b)C_0 + bC_1 \quad |C| \approx |C_0| + |C_1|$$

Memory $M[i]$

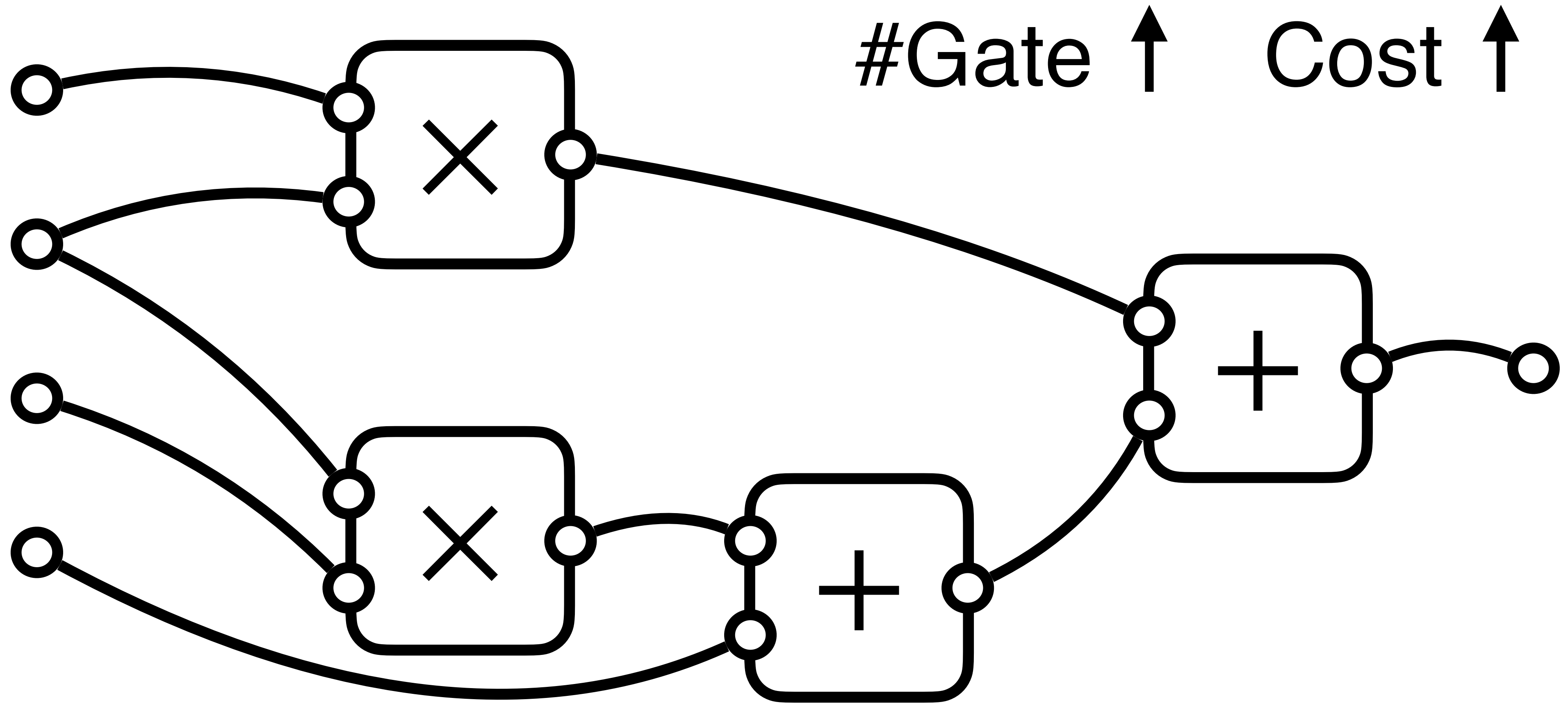


```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Existing Generic Methods: Being capable of *any* computation

←

My Generic Toolchains: Being capable of *any real-world* computation



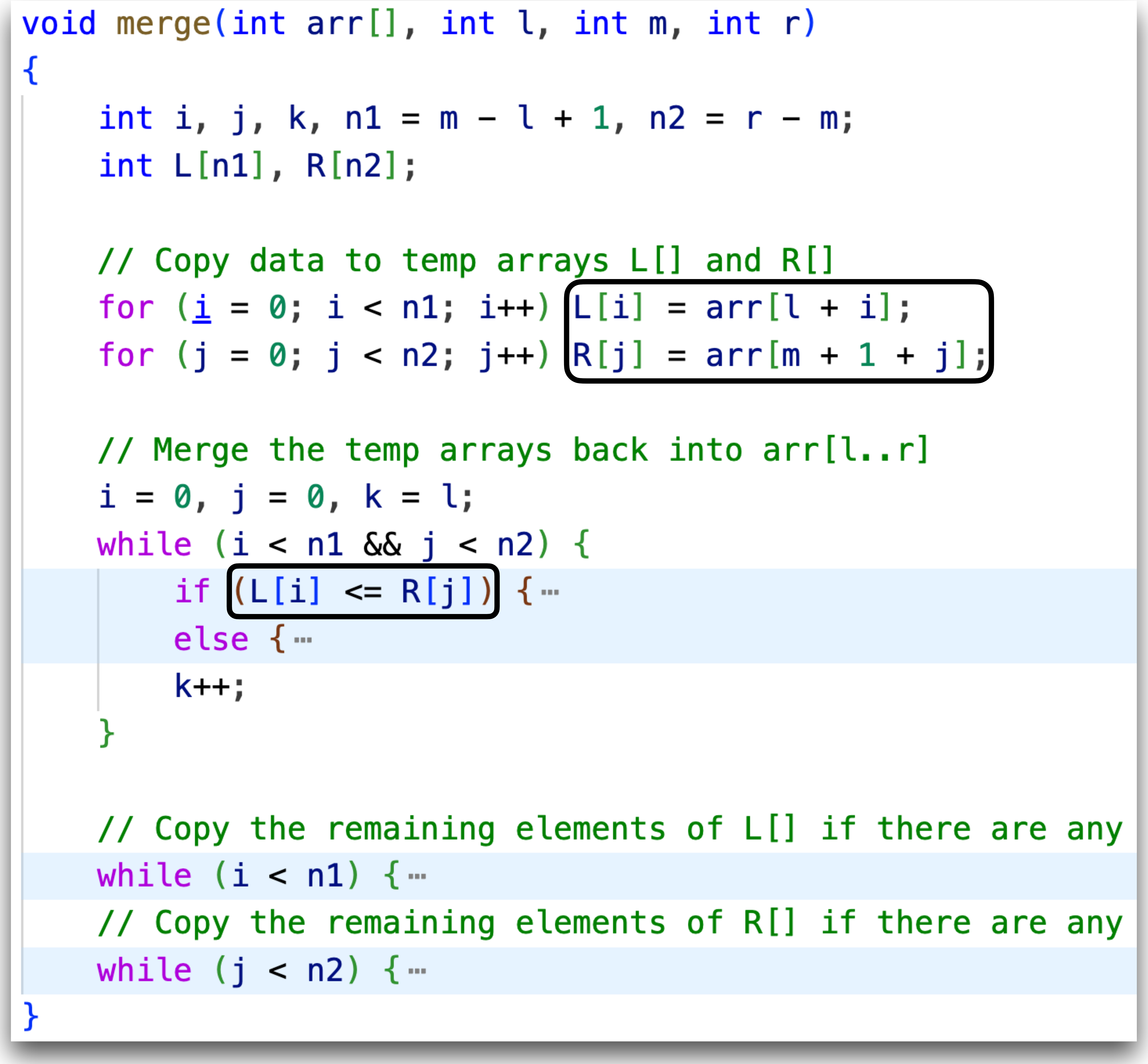
#Gate ↑ Cost ↑

Branching if C_0 else C_1

$$C = (1-b)C_0 + bC_1 \quad |C| \approx |C_0| + |C_1|$$

Memory $M[i]$

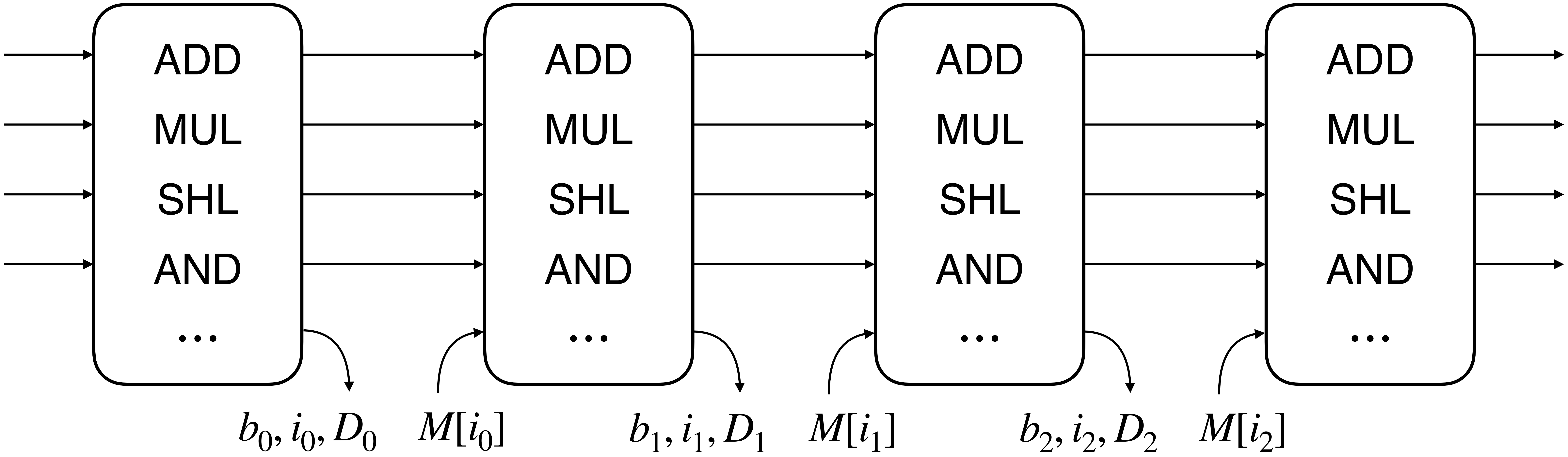
$$C = \sum_{j=1}^{N} (i \overset{?}{=} j) \cdot M[j] \qquad |C| \approx N$$



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

INTEL® CORE™ i9
i9-14900KF
SRN49
X340M715 e4

2GB PC2-5300U-555
swissbit®
MEU25664D6BC2EP-30R
0840
Made in Germany



ADD

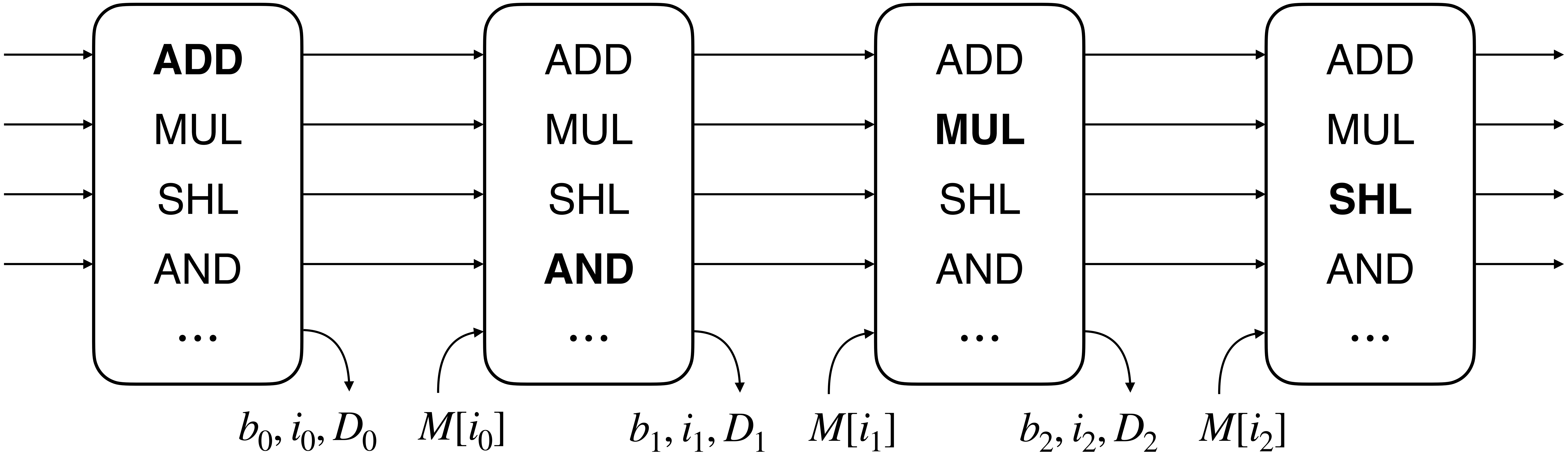
MUL

SHL

AND

…

b_0, i_0, D_0 $M[i_0]$ b_1, i_1, D_1 $M[i_1]$ b_2, i_2, D_2 $M[i_2]$



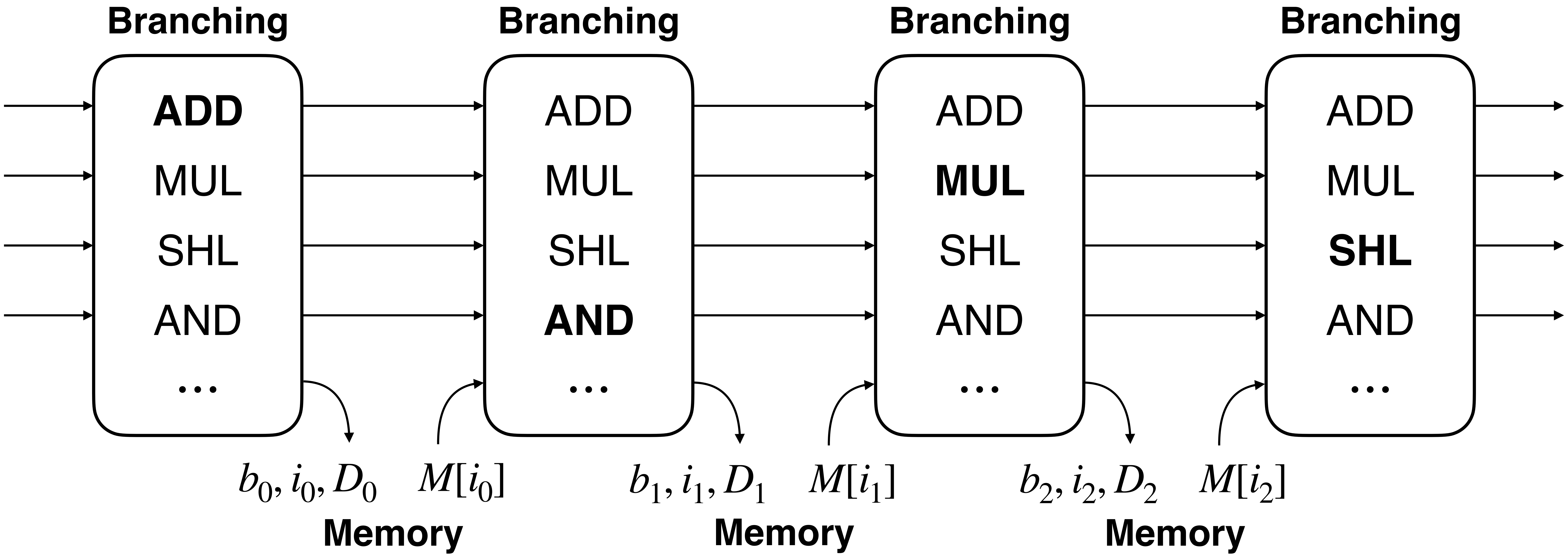
ADD
MUL
SHL
AND
…

ADD
MUL
SHL
AND
…

ADD
MUL
SHL
AND
…

ADD
MUL
SHL
AND
…

b_0, i_0, D_0 $M[i_0]$ b_1, i_1, D_1 $M[i_1]$ b_2, i_2, D_2 $M[i_2]$



Branching

ADD
MUL
SHL
AND
…

b_0, i_0, D_0 $M[i_0]$

Memory

Branching

ADD
MUL
SHL
AND
…

b_1, i_1, D_1 $M[i_1]$

Memory

Branching

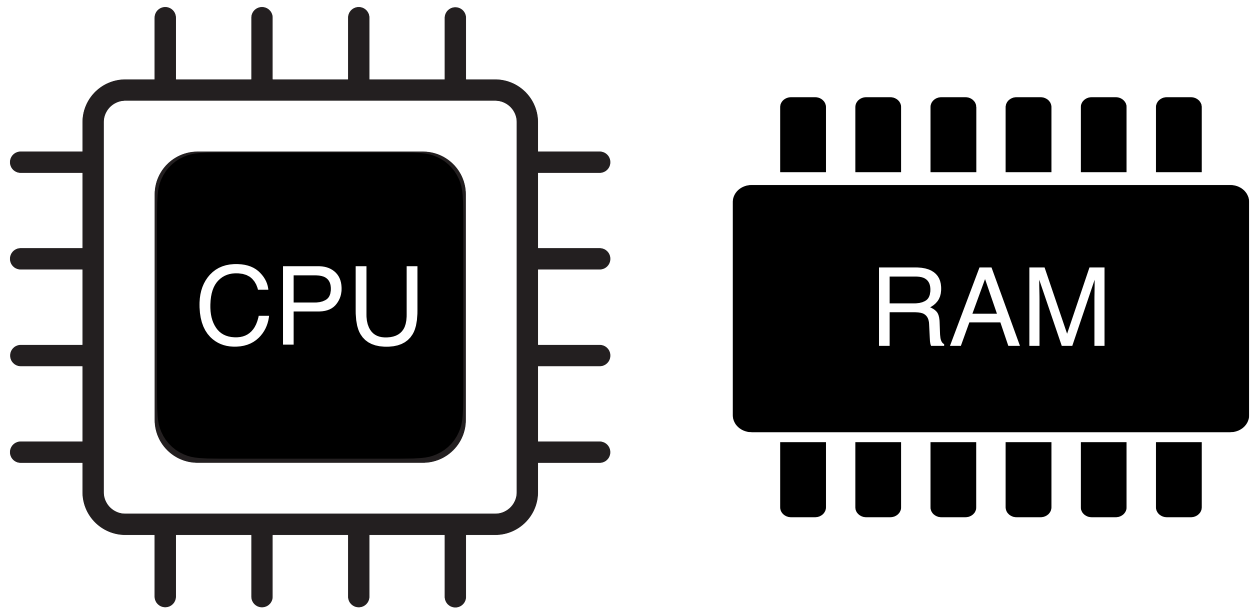
ADD
MUL
SHL
AND
…

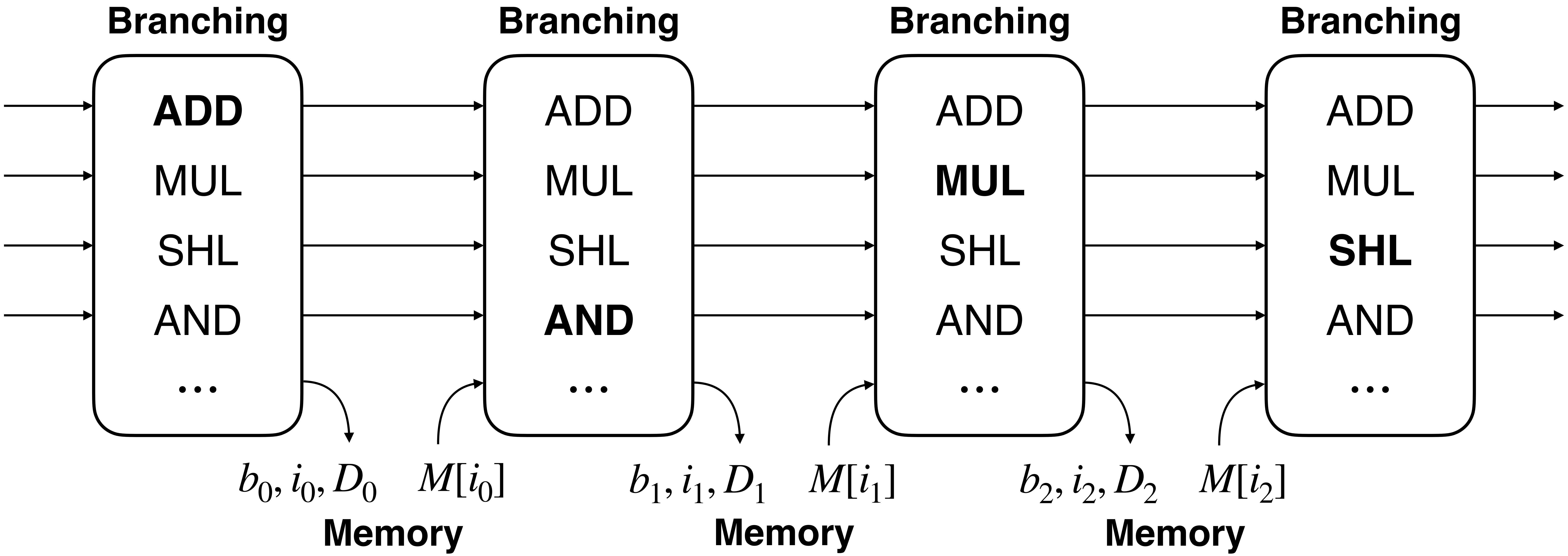
b_2, i_2, D_2 $M[i_2]$

Memory

Branching

ADD
MUL
SHL
AND
…





Branching | Branching | Branching | Branching

ADD MUL SHL AND …

ADD MUL SHL **AND** …

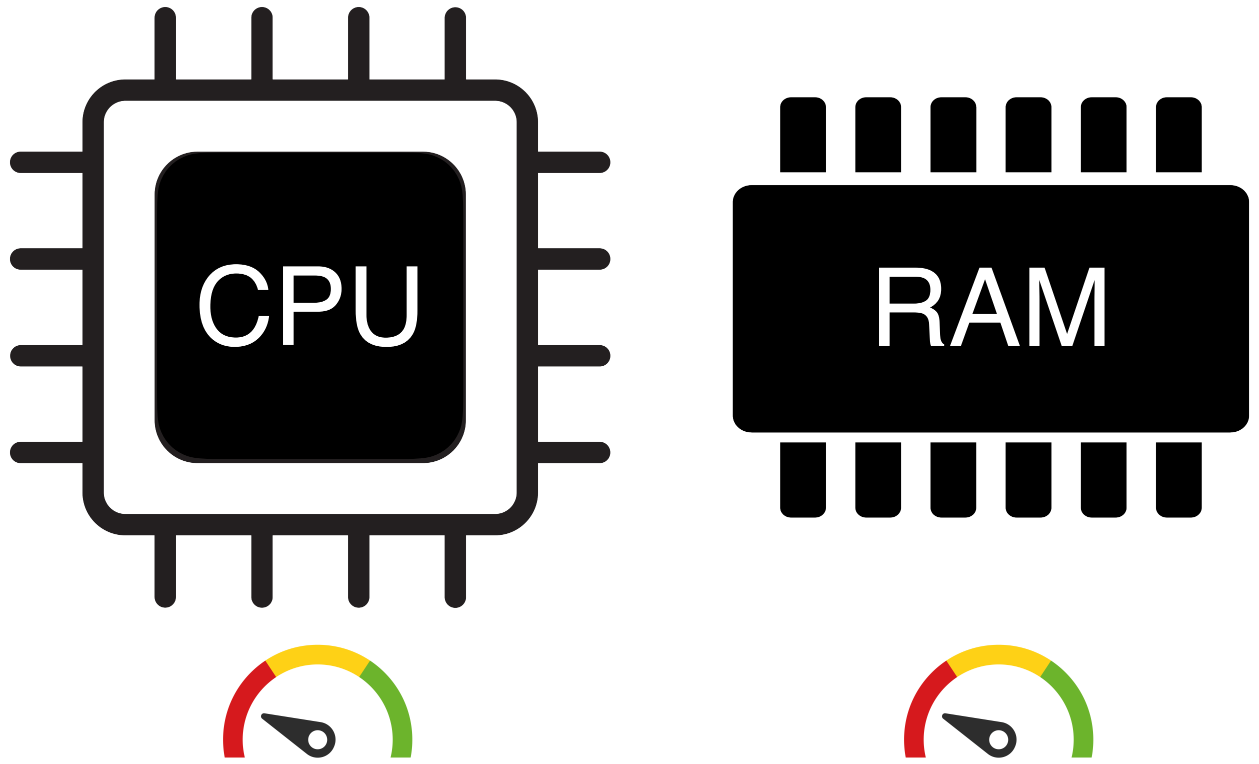
ADD **MUL** SHL AND …

ADD MUL **SHL** AND …

b_0, i_0, D_0 $M[i_0]$ **Memory**

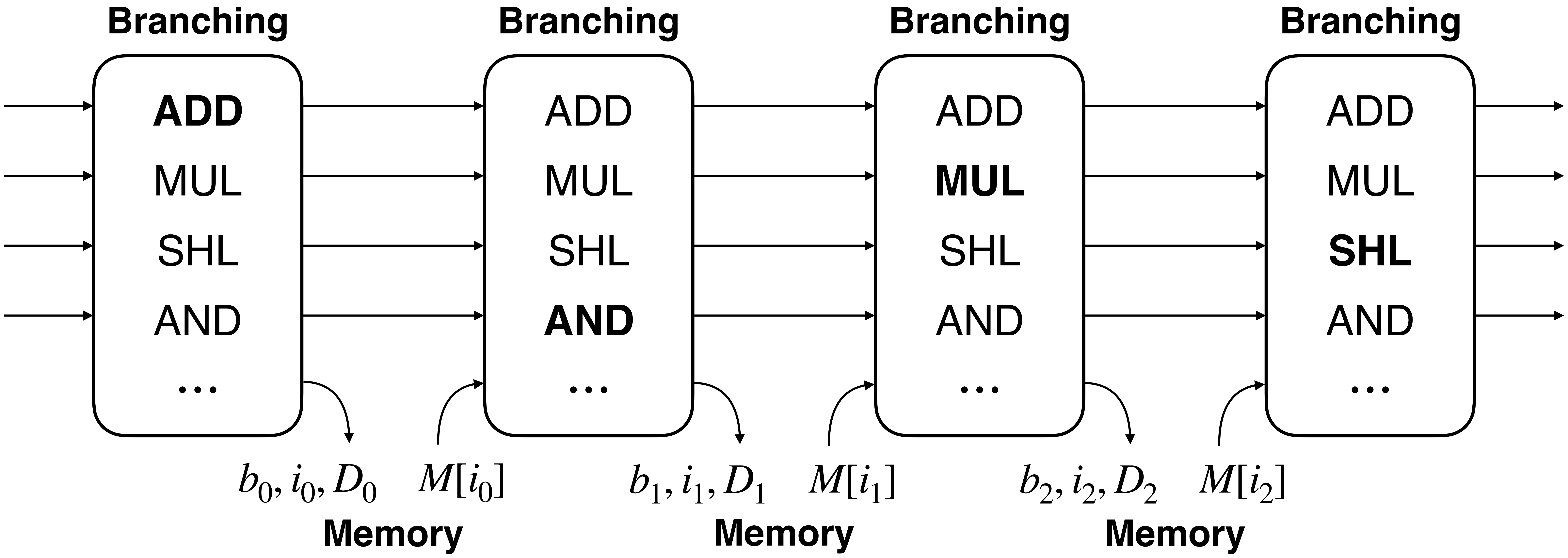
b_1, i_1, D_1 $M[i_1]$ **Memory**

b_2, i_2, D_2 $M[i_2]$ **Memory**



Prior ZK VMs [BCTV14b, BCTV14a, BCG+13]

“<10Hz”!



Branching

ADD

MUL

SHL

AND

…

Branching

ADD

MUL

SHL

AND

…

Branching

ADD

MUL

SHL

AND

…

Branching

ADD

MUL

SHL

AND

…

b_0, i_0, D_0 $M[i_0]$

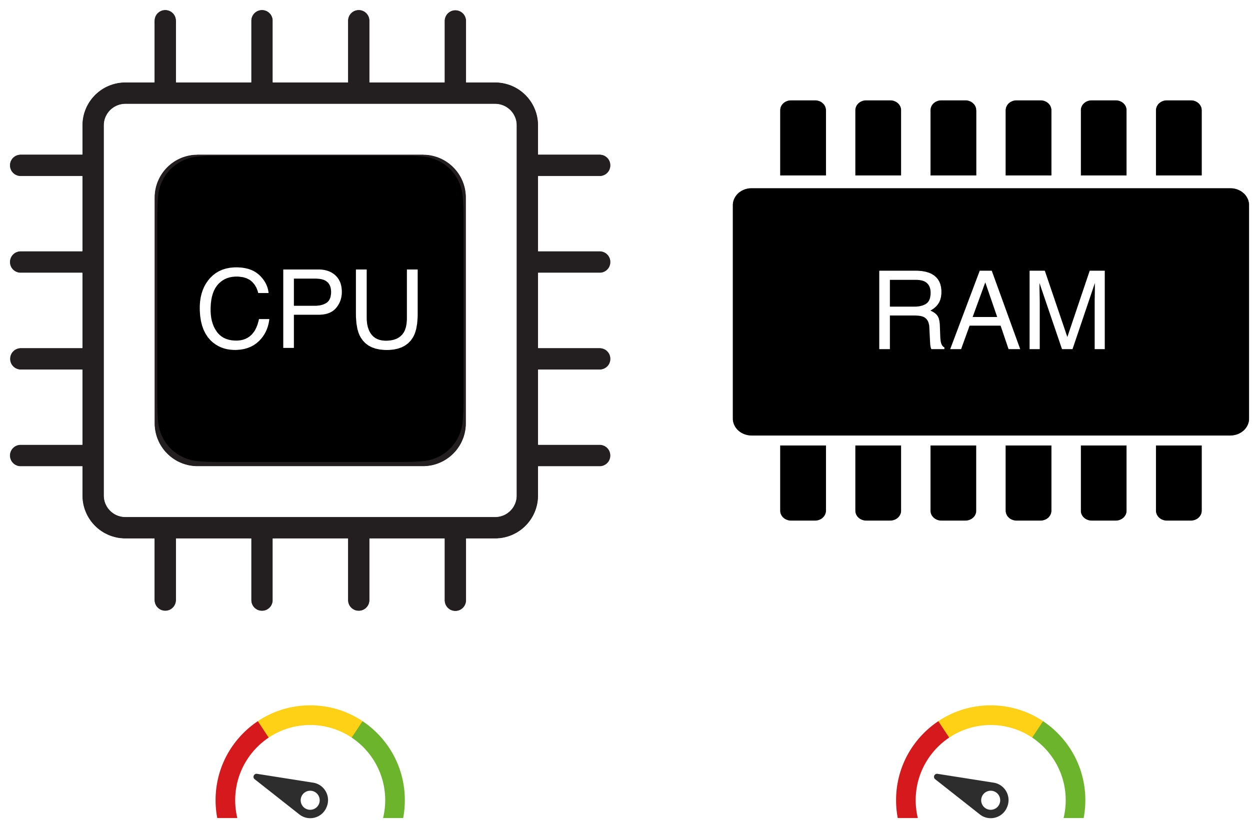
Memory

b_1, i_1, D_1 $M[i_1]$

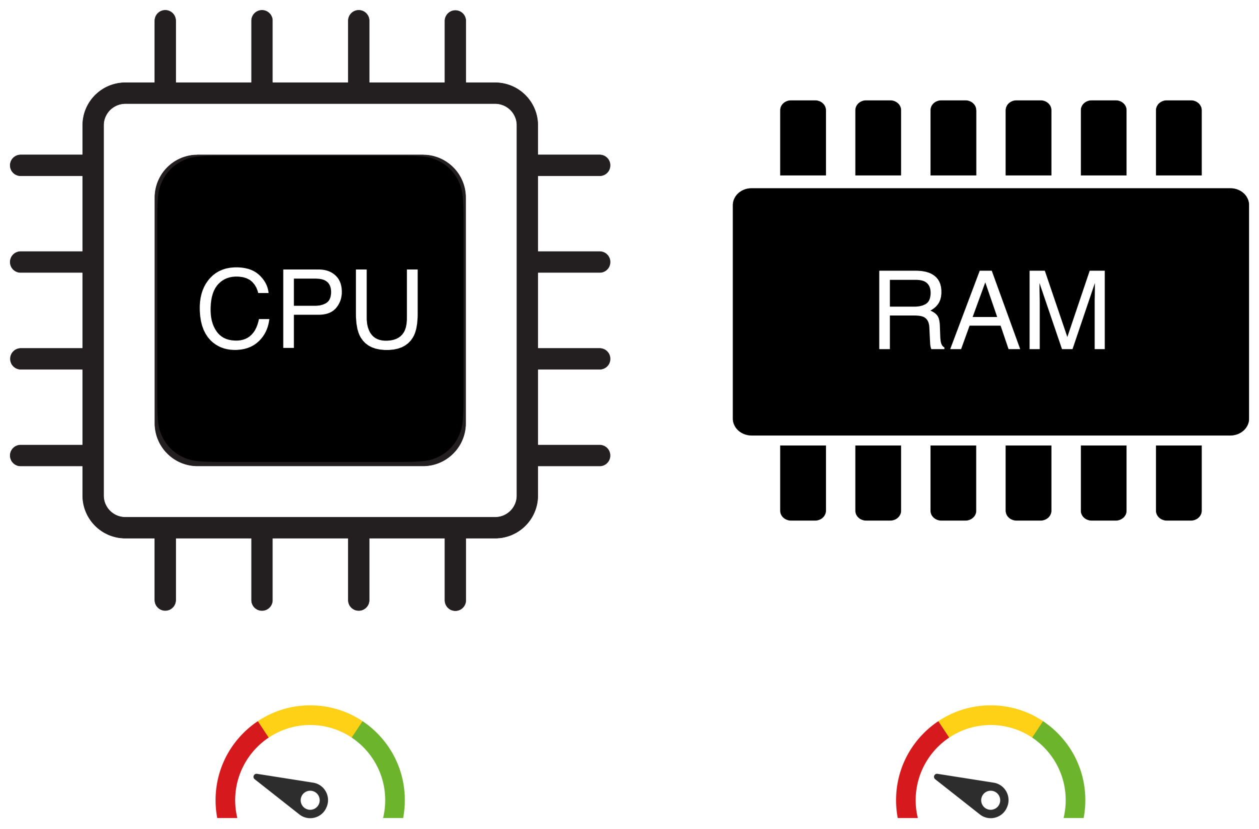
Memory

b_2, i_2, D_2 $M[i_2]$

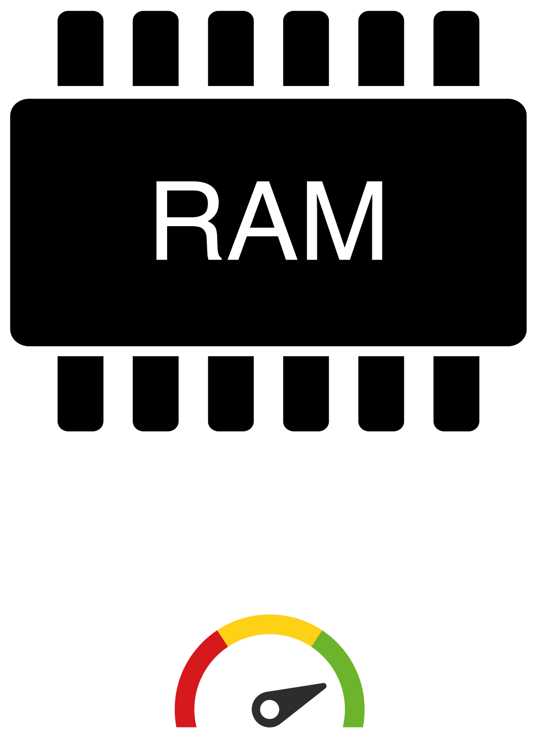
Memory



Prior Work

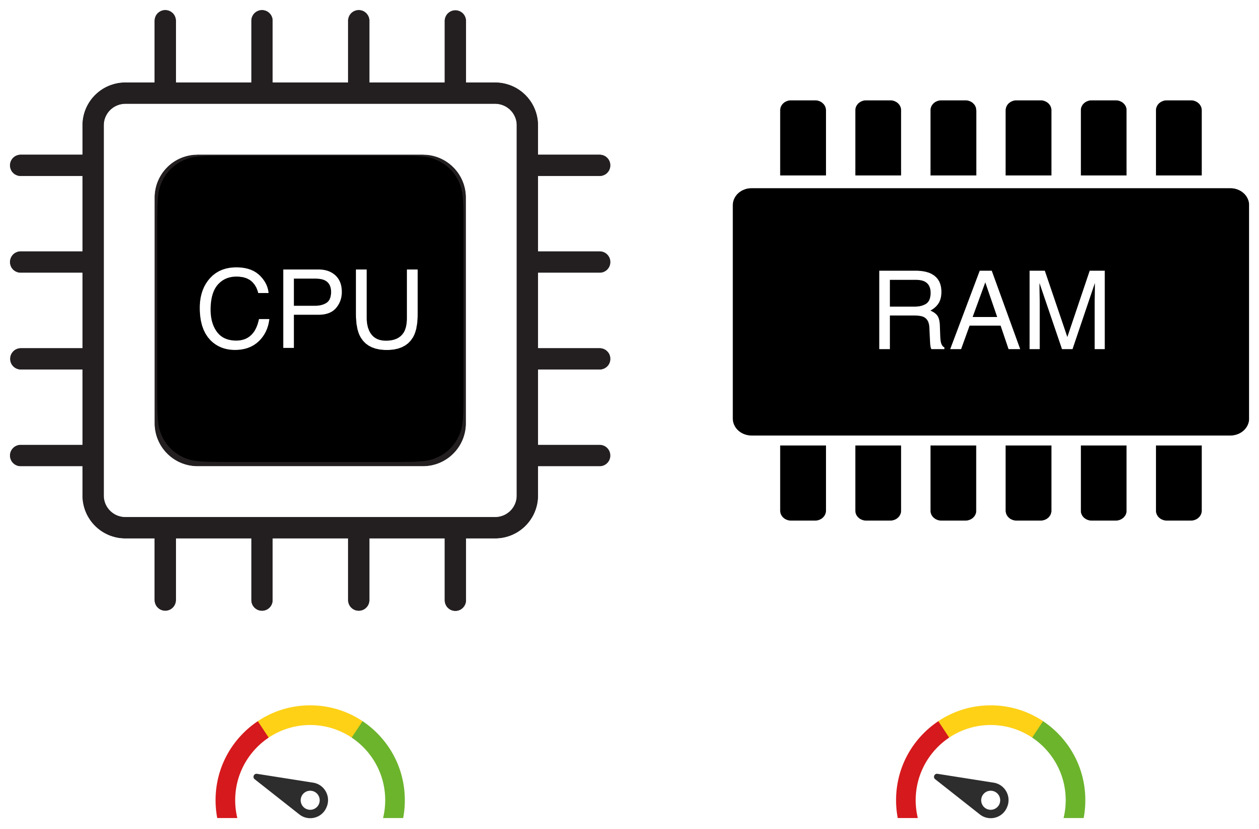


Prior Work

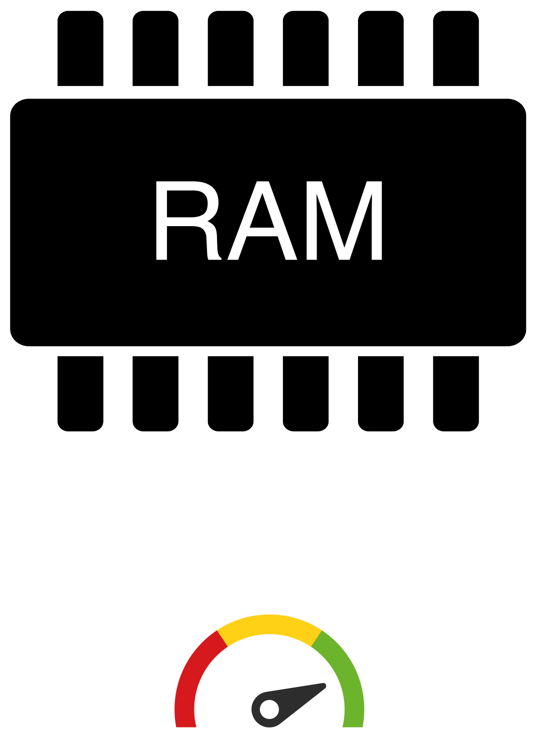


[H[1]**Y**[1]DK S&P’21] [**Y**HKD EuroS&P’22] [**Y**H Security’24]

[1] Co-first Authorship

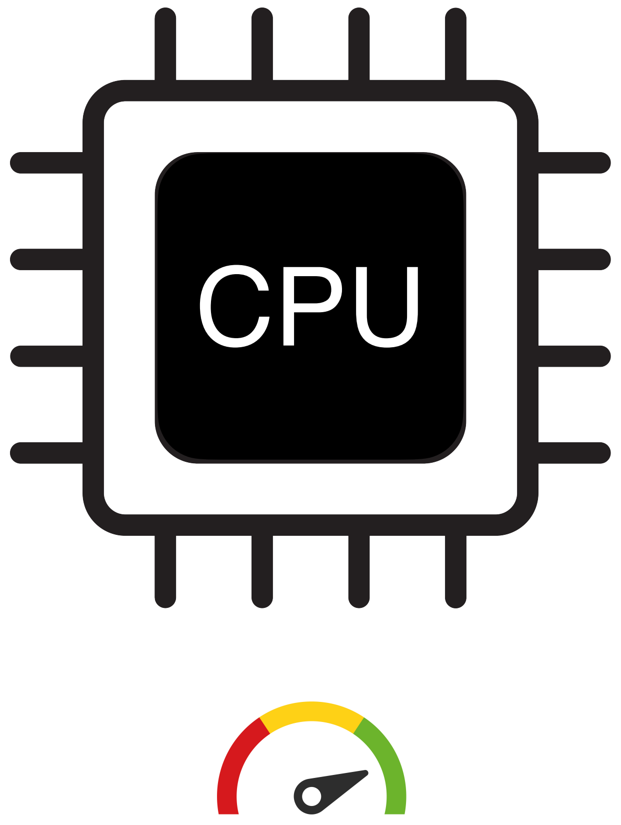


Prior Work



[H[1]**Y**[1]DK S&P’21] [**Y**HKD EuroS&P’22] [**Y**H Security’24]

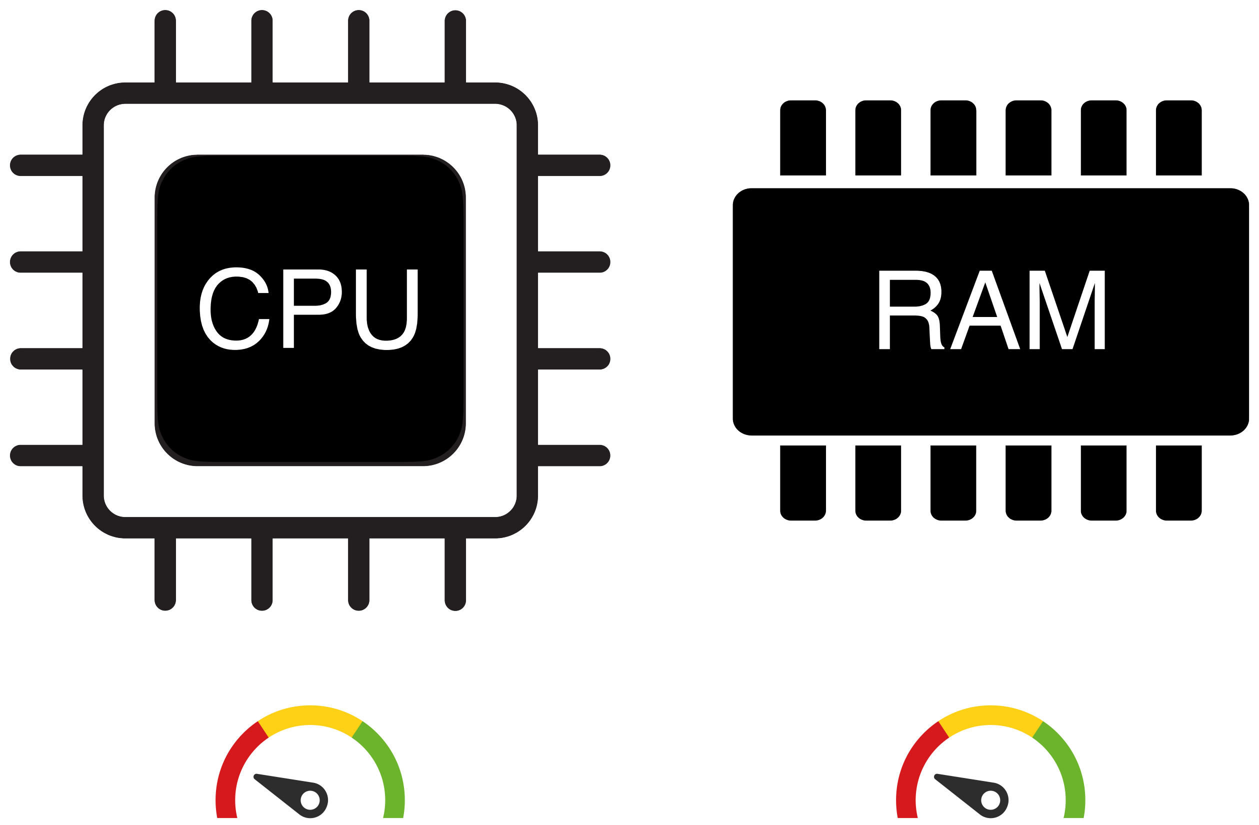
[1] Co-first Authorship



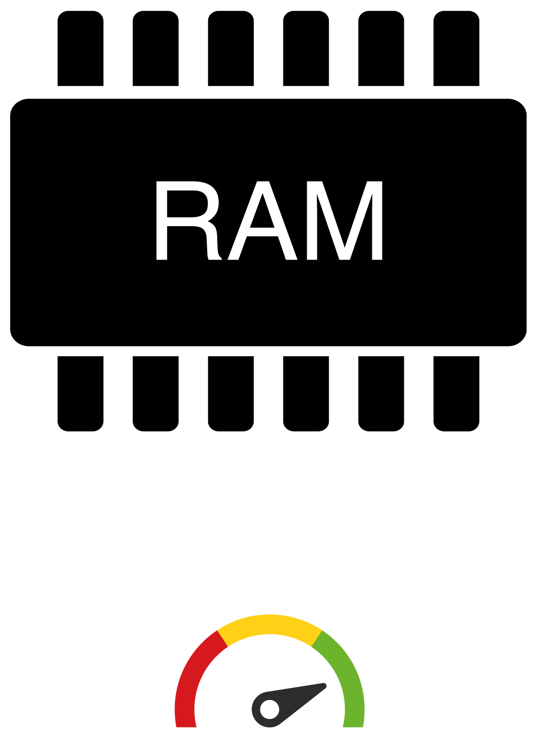
[**Y**HHKV CCS’23]🏆 [HHKV**Y** Asiacrypt’24]⇩ [**Yang** ePrint’25]

⇩ Alphabetic Order

🏆 Distinguished Paper Award

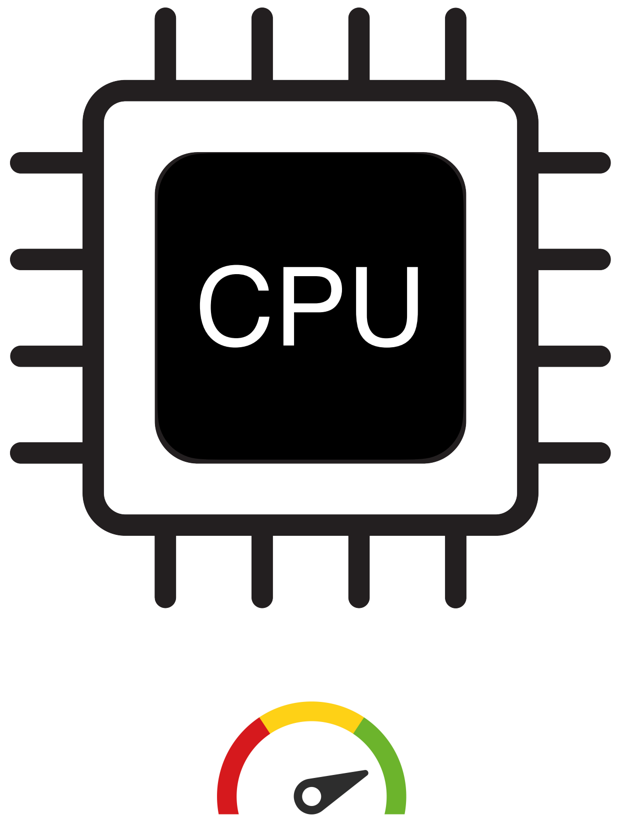


Prior Work

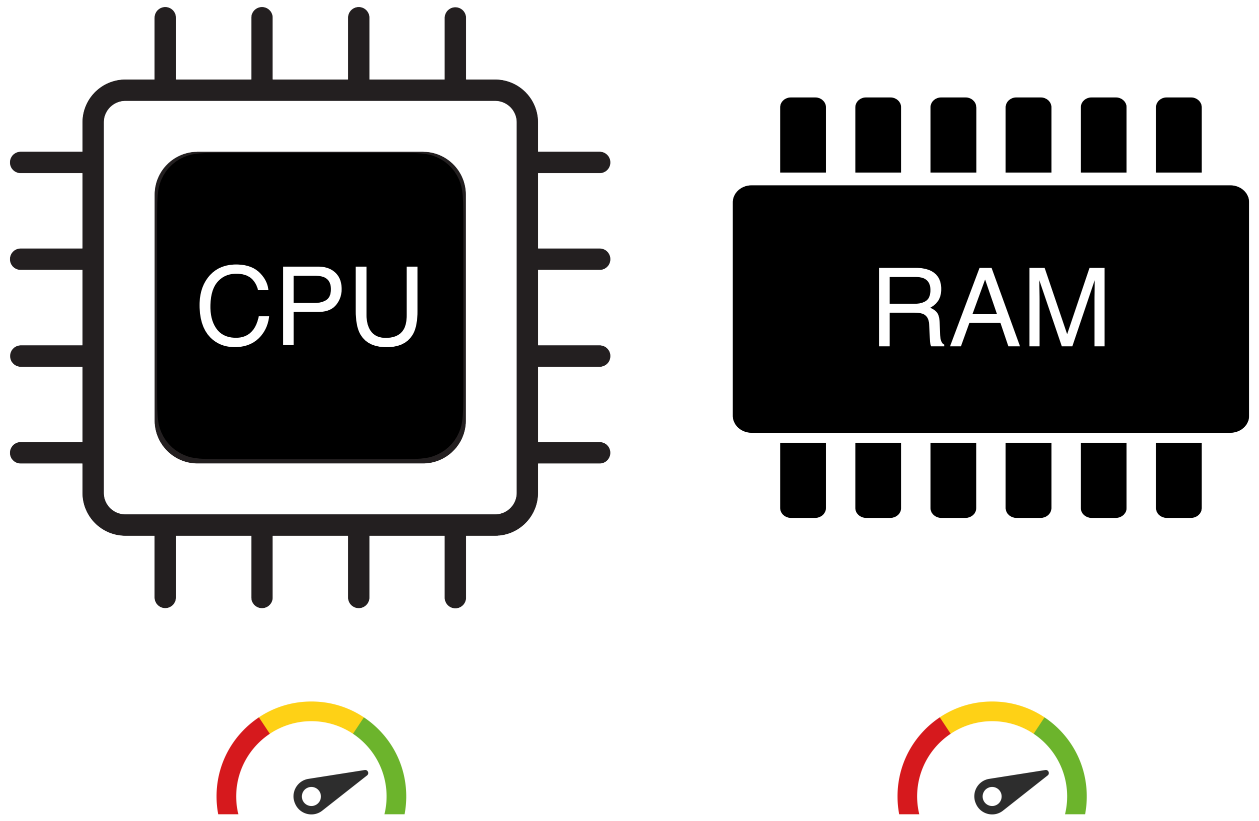


[H[1]**Y**[1]DK S&P’21] [**Y**HKD EuroS&P’22] [**Y**H Security’24]

[1] Co-first Authorship



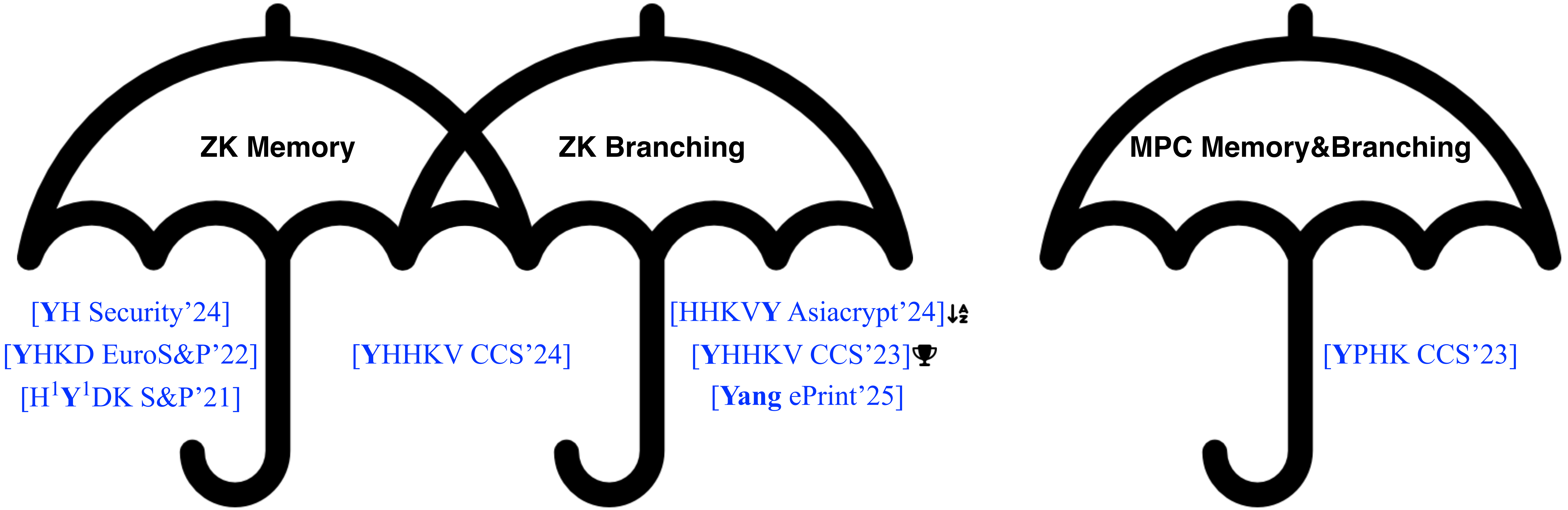
[**Y**HHKV CCS’23]🏆 [HHKV**Y** Asiacrypt’24]↓AZ [**Yang** ePrint’25]



[**Y**PHK CCS’23] [**Y**HHKV CCS’24]

↓AZ Alphabetic Order

🏆 Distinguished Paper Award



ZK Memory

[**Y**H Security'24]

[**Y**HKD EuroS&P'22]

[H[1]**Y**[1]DK S&P'21]

[**Y**HHKV CCS'24]

ZK Branching

[HHKV**Y** Asiacrypt'24]↓AZ

[**Y**HHKV CCS'23]🏆

[**Yang** ePrint'25]

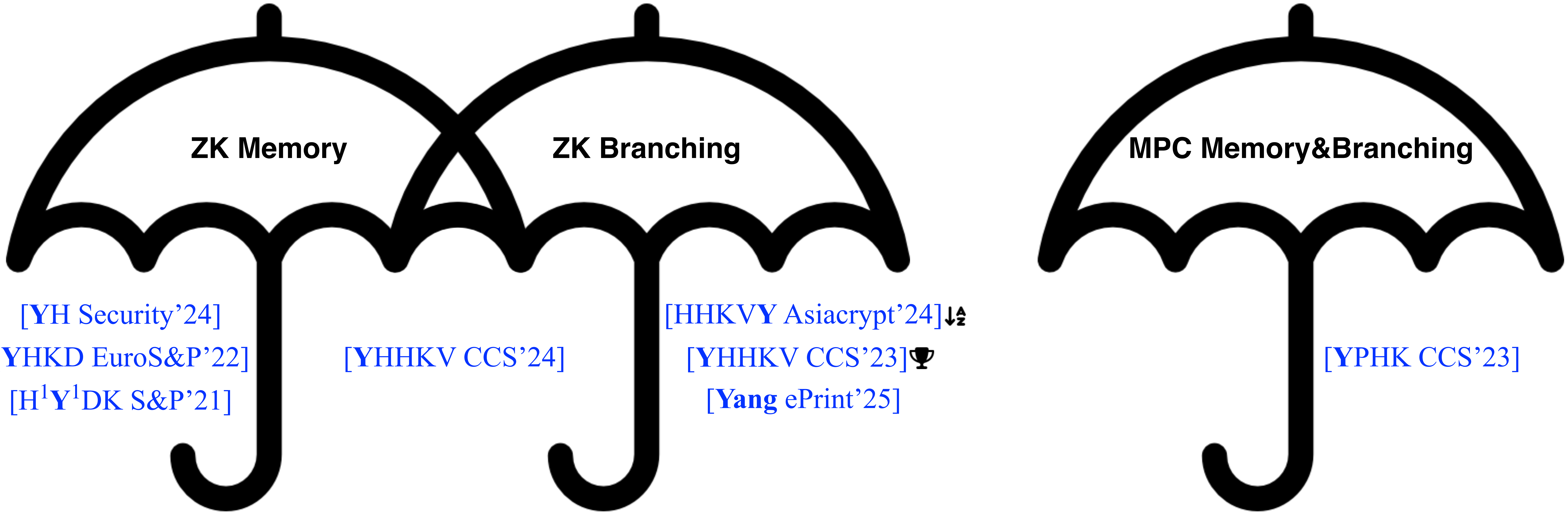
MPC Memory&Branching

[**Y**PHK CCS'23]

↓AZ Alphabetic Order

🏆 Distinguished Paper Award

[1] Co-first Authorship

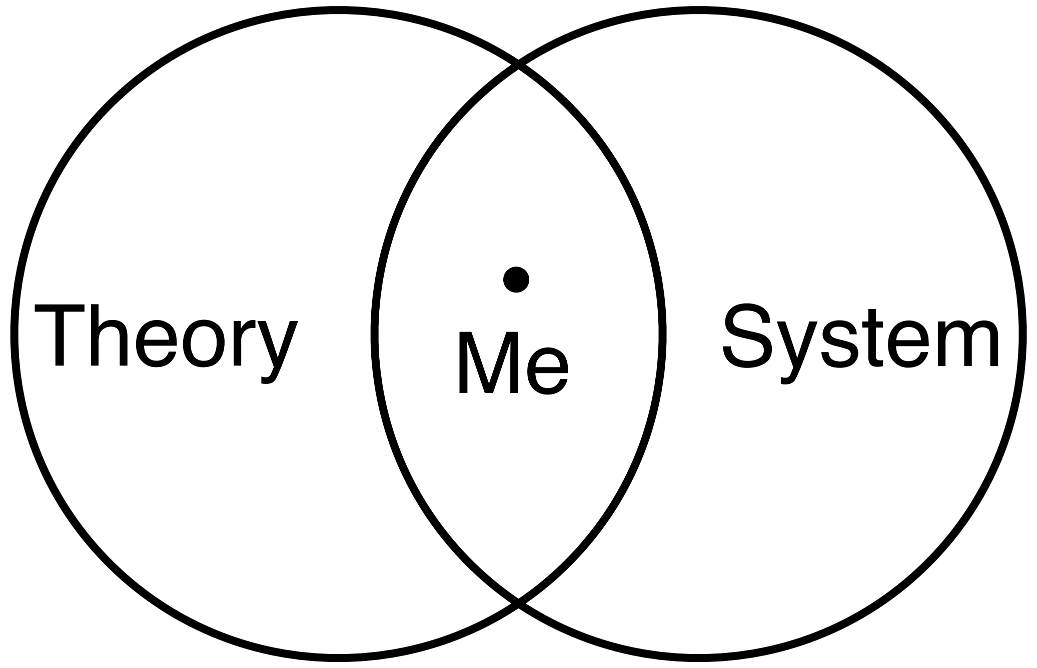


[**Y**H Security’24]
[**Y**HKD EuroS&P’22]
[H[1]**Y**[1]DK S&P’21]

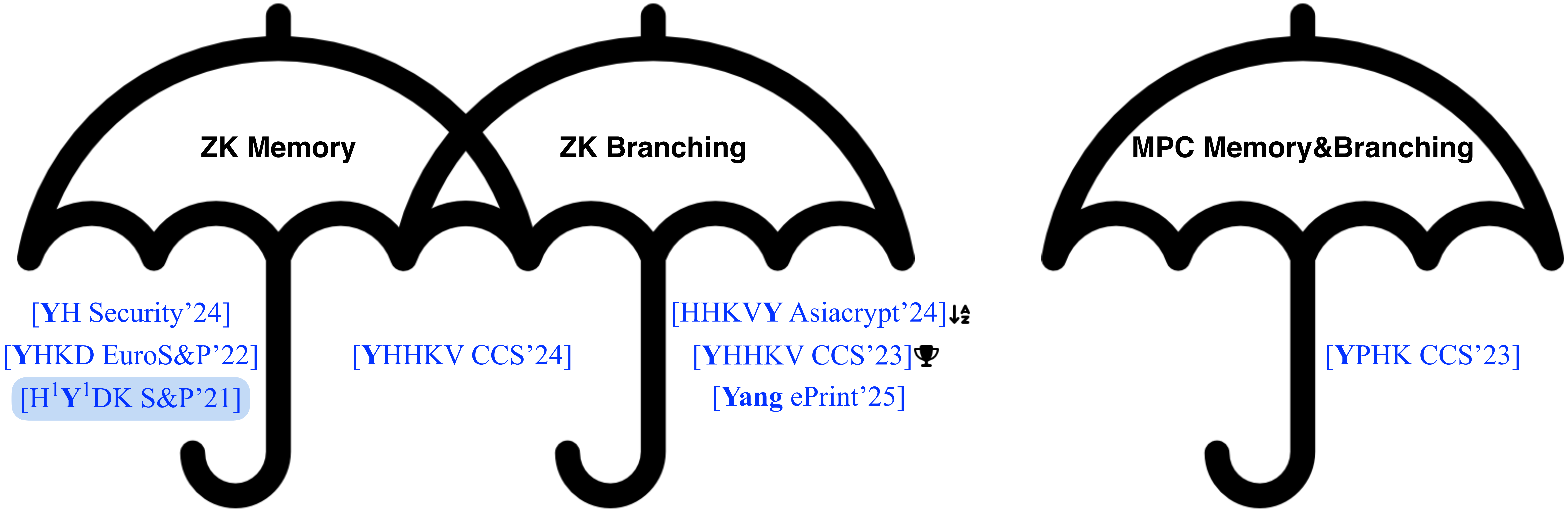
[**Y**HHKV CCS’24]

[HHKV**Y** Asiacrypt’24]↓AZ
[**Y**HHKV CCS’23]🏆
[**Yang** ePrint’25]

[**Y**PHK CCS’23]



↓AZ Alphabetic Order
🏆 Distinguished Paper Award
1 Co-first Authorship



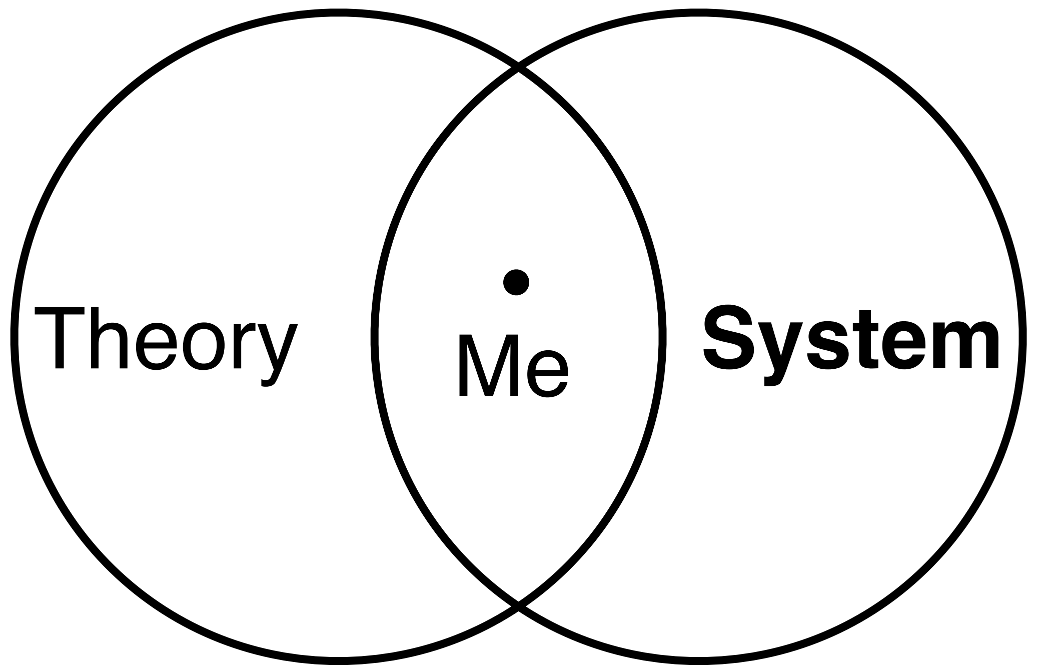
[**Y**H Security’24]
[**Y**HKD EuroS&P’22]
[H^1**Y**^{1}DK S&P’21]

[**Y**HHKV CCS’24]

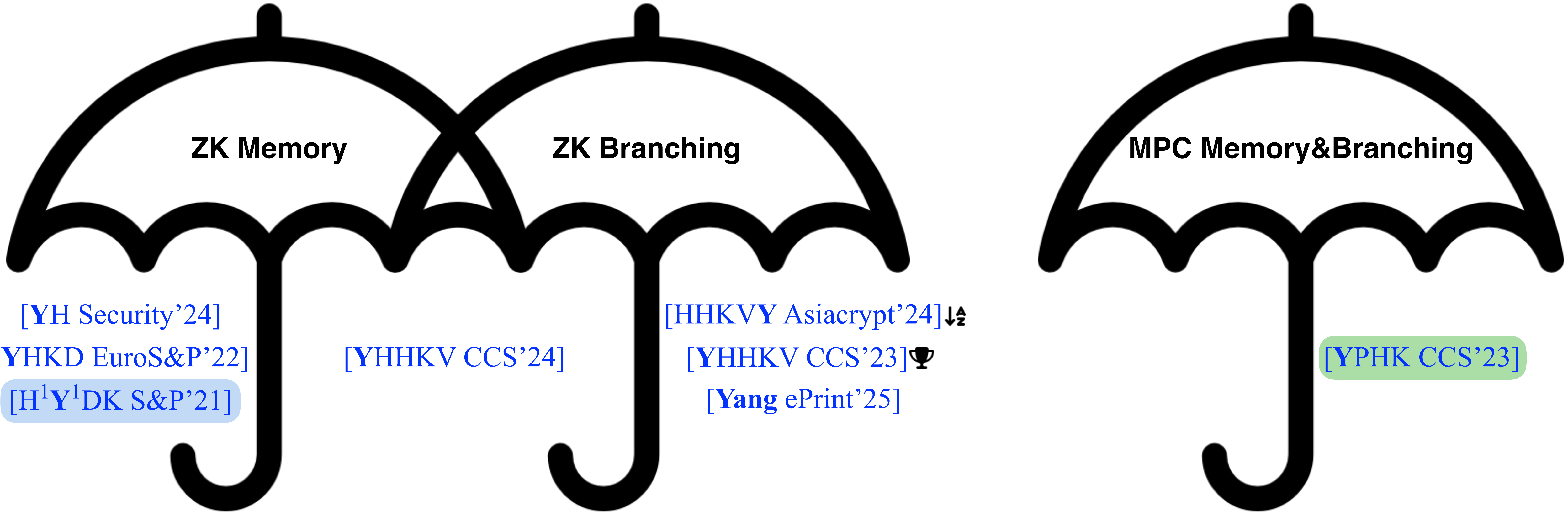
[HHKV**Y** Asiacrypt’24]↓AZ
[**Y**HHKV CCS’23]🏆
[**Yang** ePrint’25]

[**Y**PHK CCS’23]

- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)



↓AZ Alphabetic Order
🏆 Distinguished Paper Award
1 Co-first Authorship



ZK Memory

[**Y**H Security'24]

[**Y**HKD EuroS&P'22]

[H^1**Y**^{1}DK S&P'21]

[**Y**HHKV CCS'24]

ZK Branching

[HHKV**Y** Asiacrypt'24]↓AZ

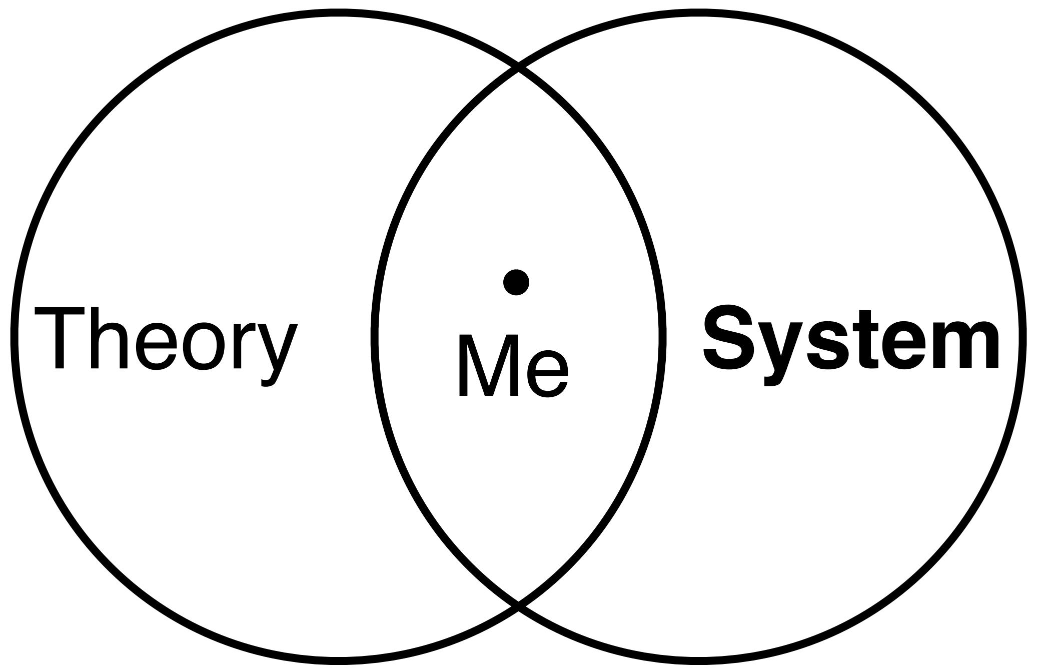
[**Y**HHKV CCS'23]🏆

[**Yang** ePrint'25]

MPC Memory&Branching

[**Y**PHK CCS'23]

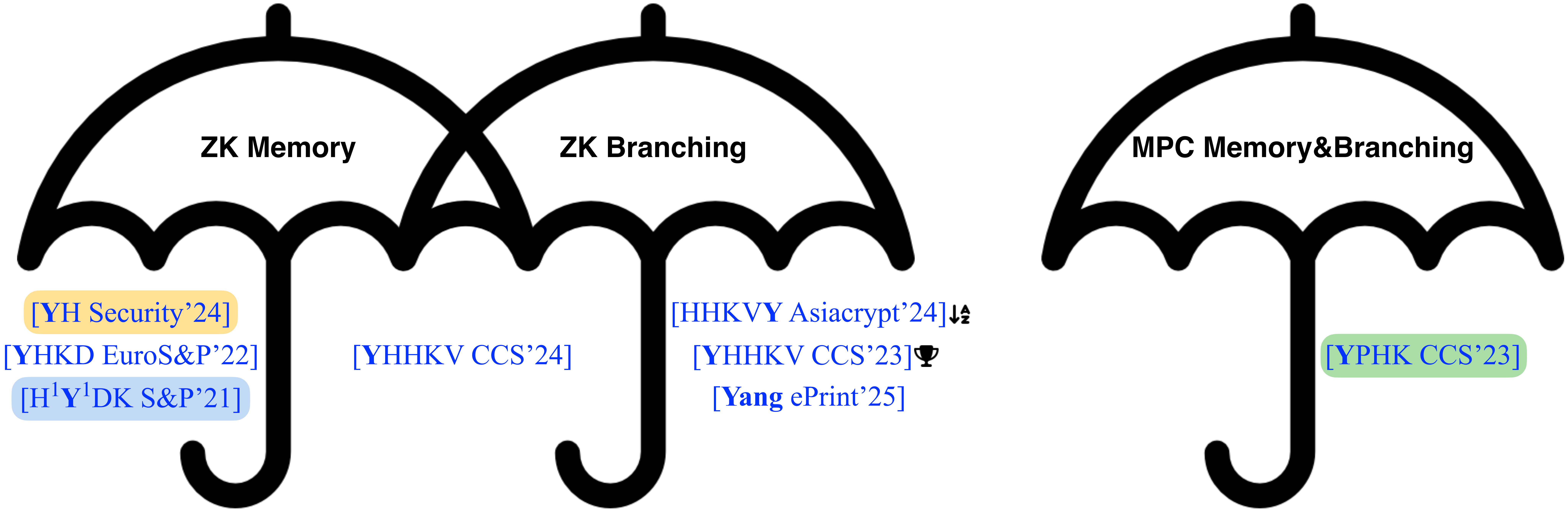
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈ 10KHz ($\approx$**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈ 1KHz ($\approx$**1000x**)



↓AZ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



[**Y**H Security’24]

[**Y**HKD EuroS&P’22]

[H^1**Y**^{1}DK S&P’21]

[**Y**HHKV CCS’24]

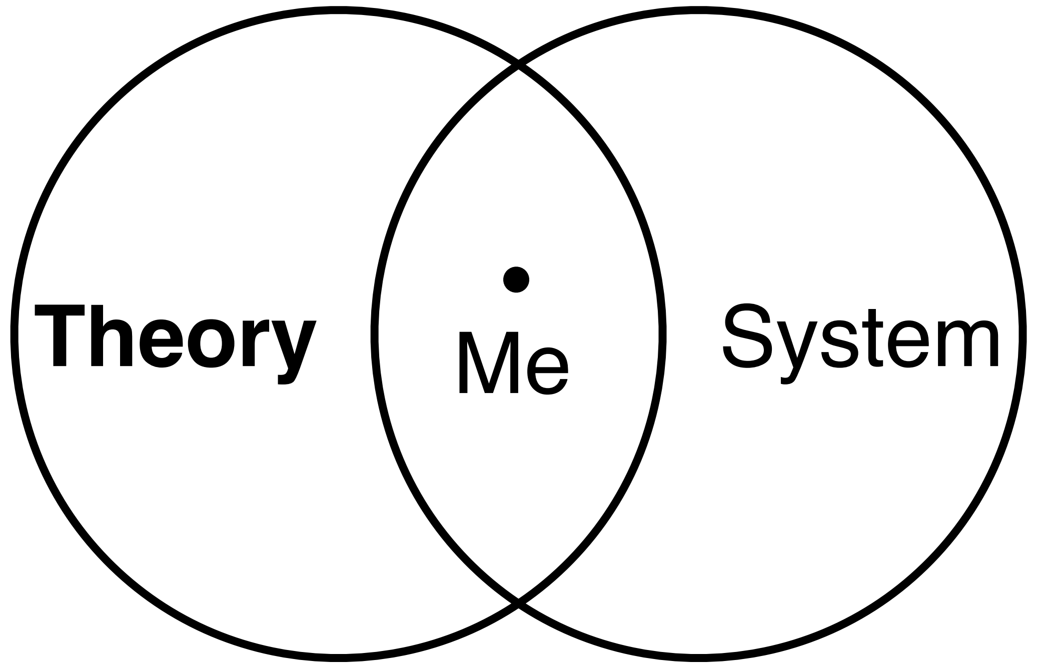
[HHKV**Y** Asiacrypt’24]↓AZ

[**Y**HHKV CCS’23]🏆

[**Yang** ePrint’25]

[**Y**PHK CCS’23]

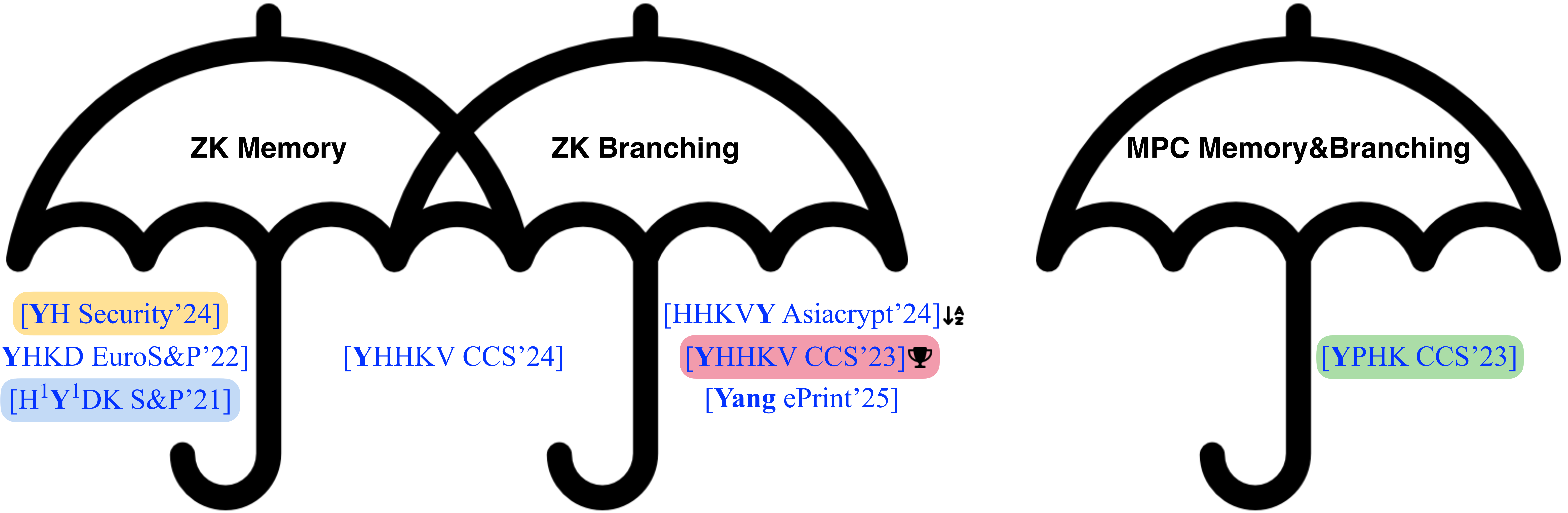
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈1KHz (≈**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity



↓AZ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



ZK Memory

[**Y**H Security’24]

[**Y**HKD EuroS&P’22]

[H[1]**Y**[1]DK S&P’21]

[**Y**HHKV CCS’24]

ZK Branching

[HHKV**Y** Asiacrypt’24]↓AZ

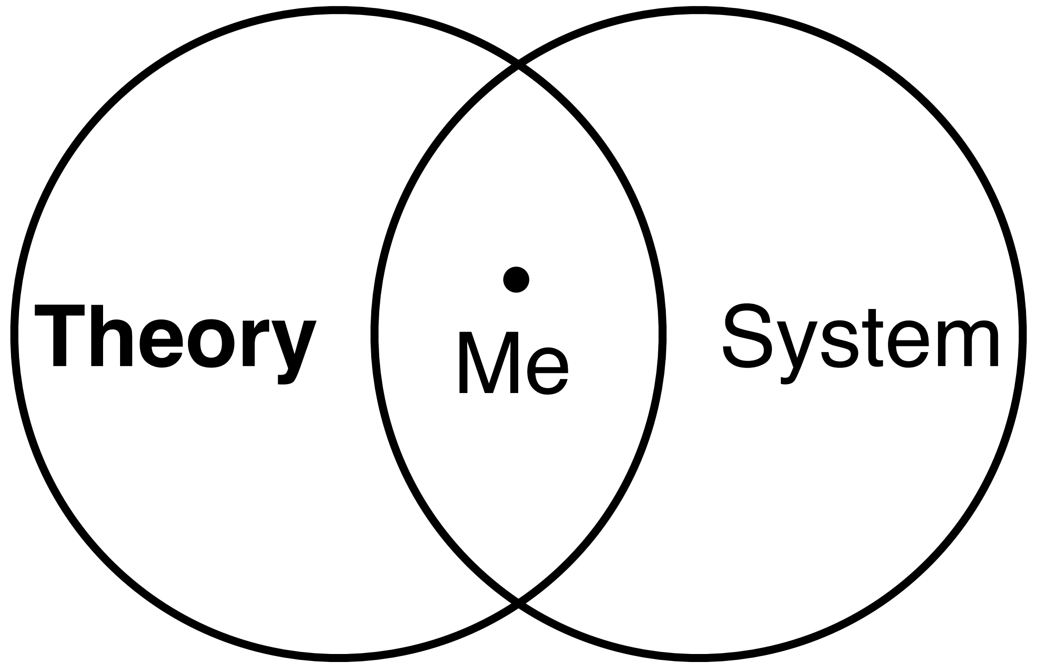
[**Y**HHKV CCS’23]🏆

[**Yang** ePrint’25]

MPC Memory&Branching

[**Y**PHK CCS’23]

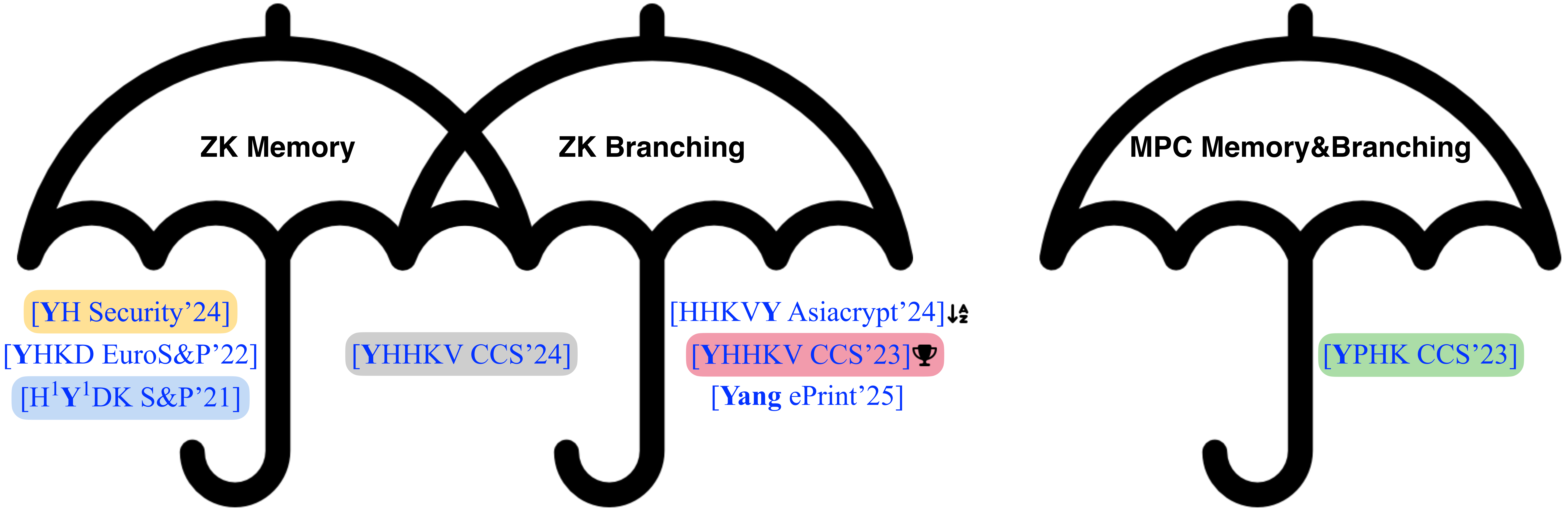
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈1KHz (≈**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity



↓AZ Alphabetic Order

🏆 Distinguished Paper Award

[1] Co-first Authorship



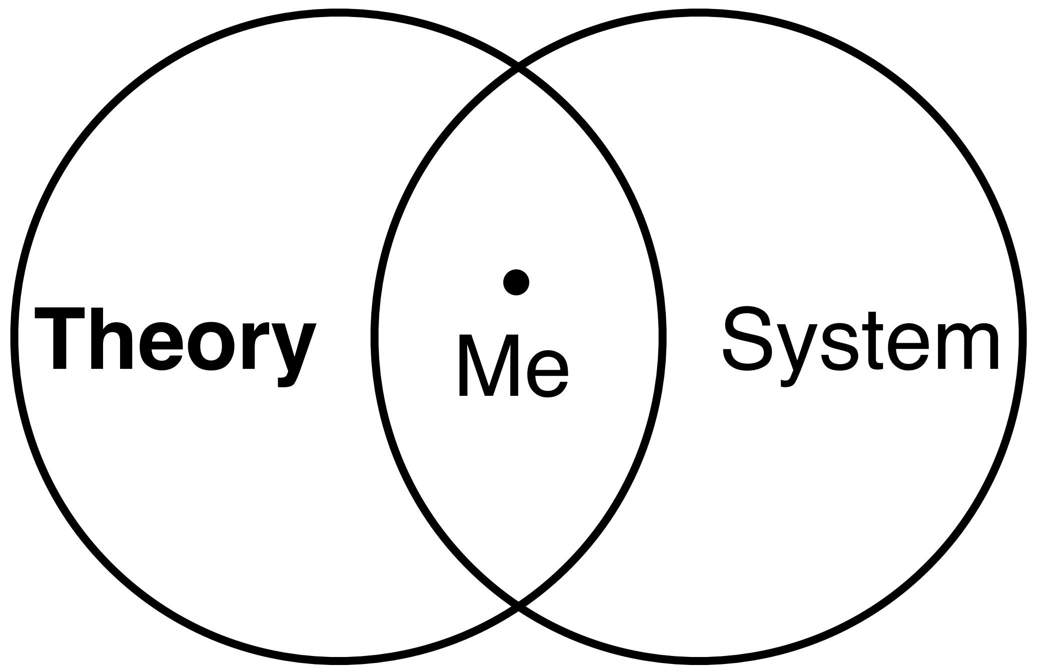
[**Y**H Security'24]
[**Y**HKD EuroS&P'22]
[H^1**Y**^{1}DK S&P'21]

[**Y**HHKV CCS'24]

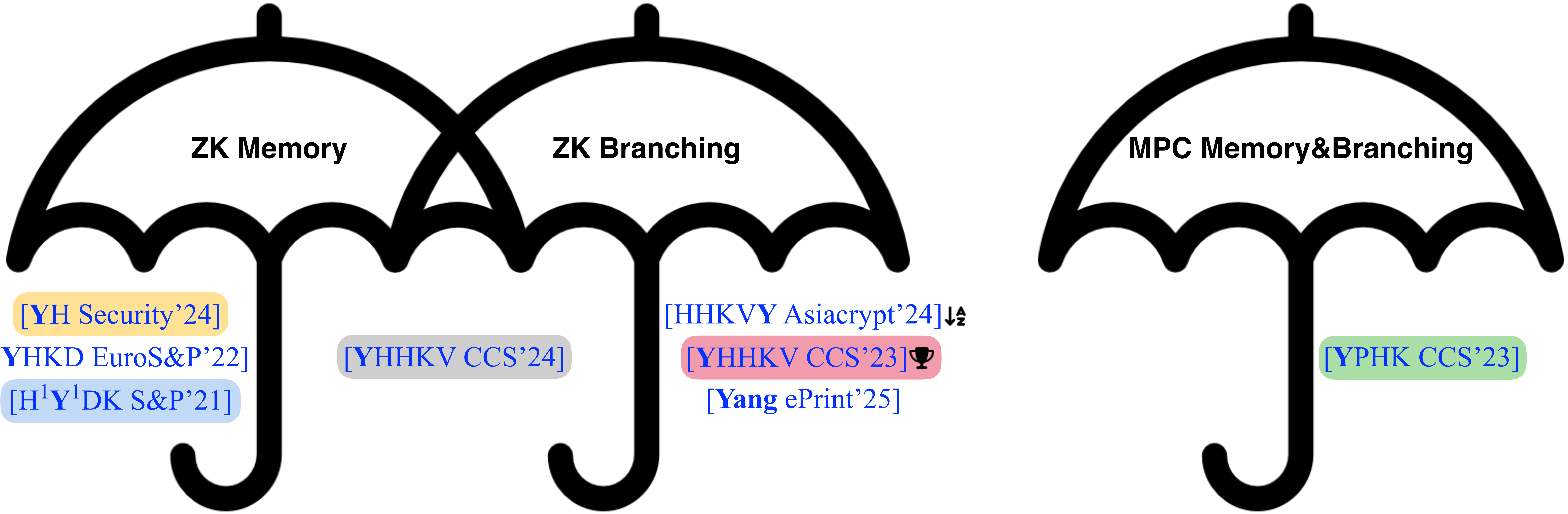
[HHKV**Y** Asiacrypt'24]↓AZ
[**Y**HHKV CCS'23]🏆
[**Yang** ePrint'25]

[**Y**PHK CCS'23]

- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈1KHz (≈**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity
- A zero-knowledge (ZK) CPU+RAM achieving optimal complexity (≈**100x**)



↓AZ Alphabetic Order
🏆 Distinguished Paper Award
1 Co-first Authorship



ZK Memory

ZK Branching

MPC Memory&Branching

[**Y**H Security’24]

[**Y**HKD EuroS&P’22]

[H^1**Y**^{1}DK S&P’21]

[**Y**HHKV CCS’24]

[HHKV**Y** Asiacrypt’24]↓AZ

[**Y**HHKV CCS’23]🏆

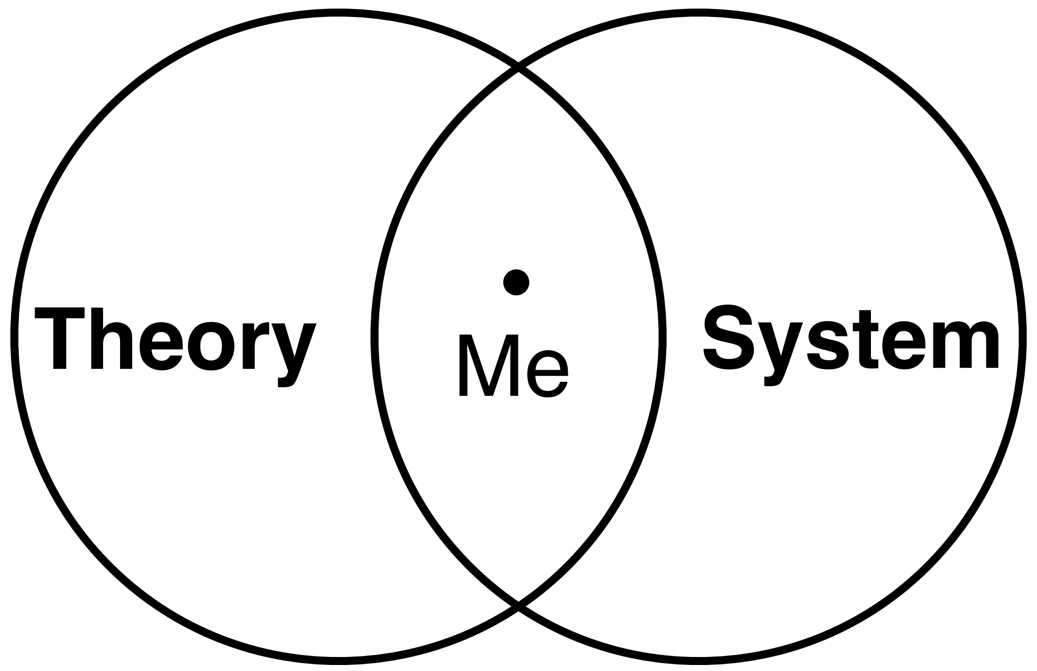
[**Yang** ePrint’25]

[**Y**PHK CCS’23]

1. A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)

2. A zero-knowledge (ZK) read-write memory achieving optimal complexity

3. A zero-knowledge (ZK) branching protocol achieving optimal complexity



Theory

Me

System

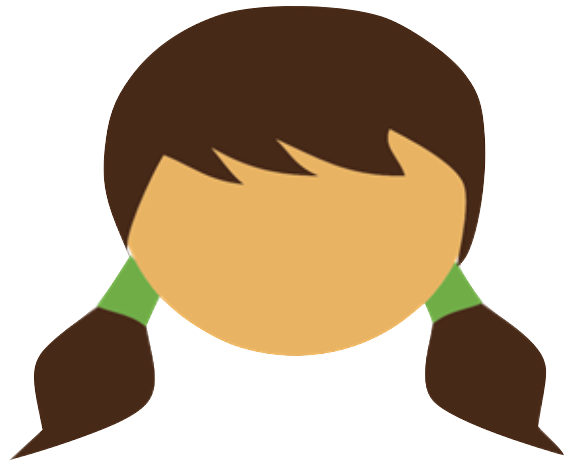
↓AZ Alphabetic Order

🏆 Distinguished Paper Award

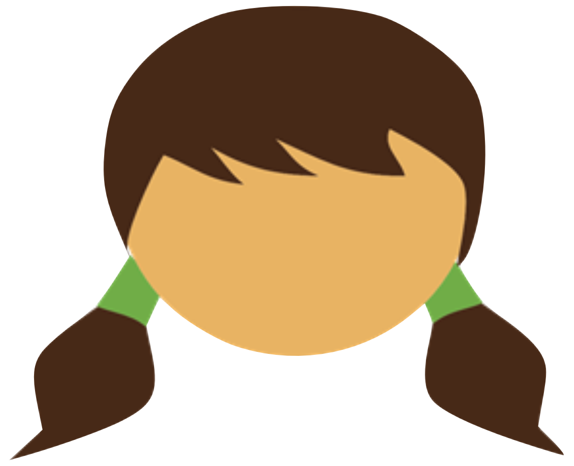
1 Co-first Authorship

Notation

Notation



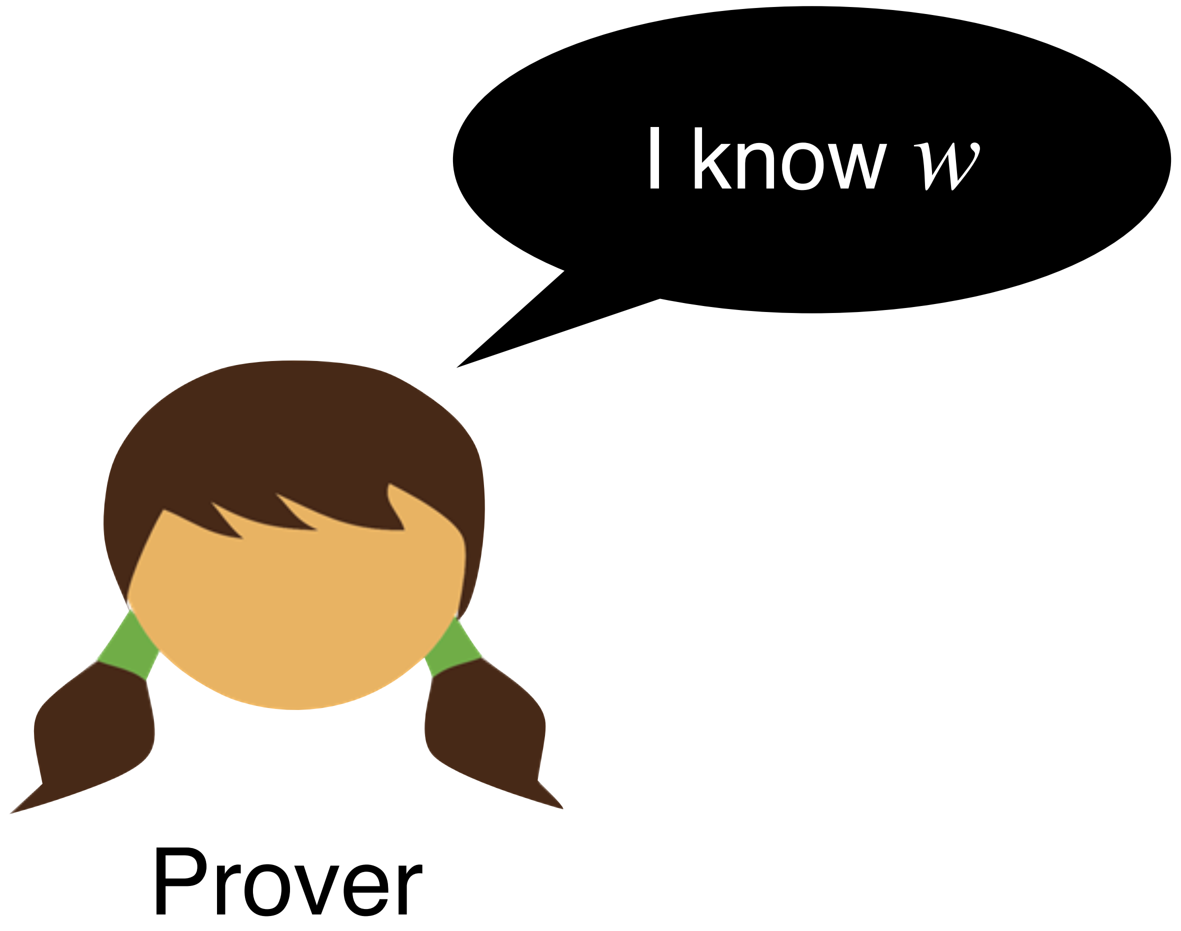
Notation



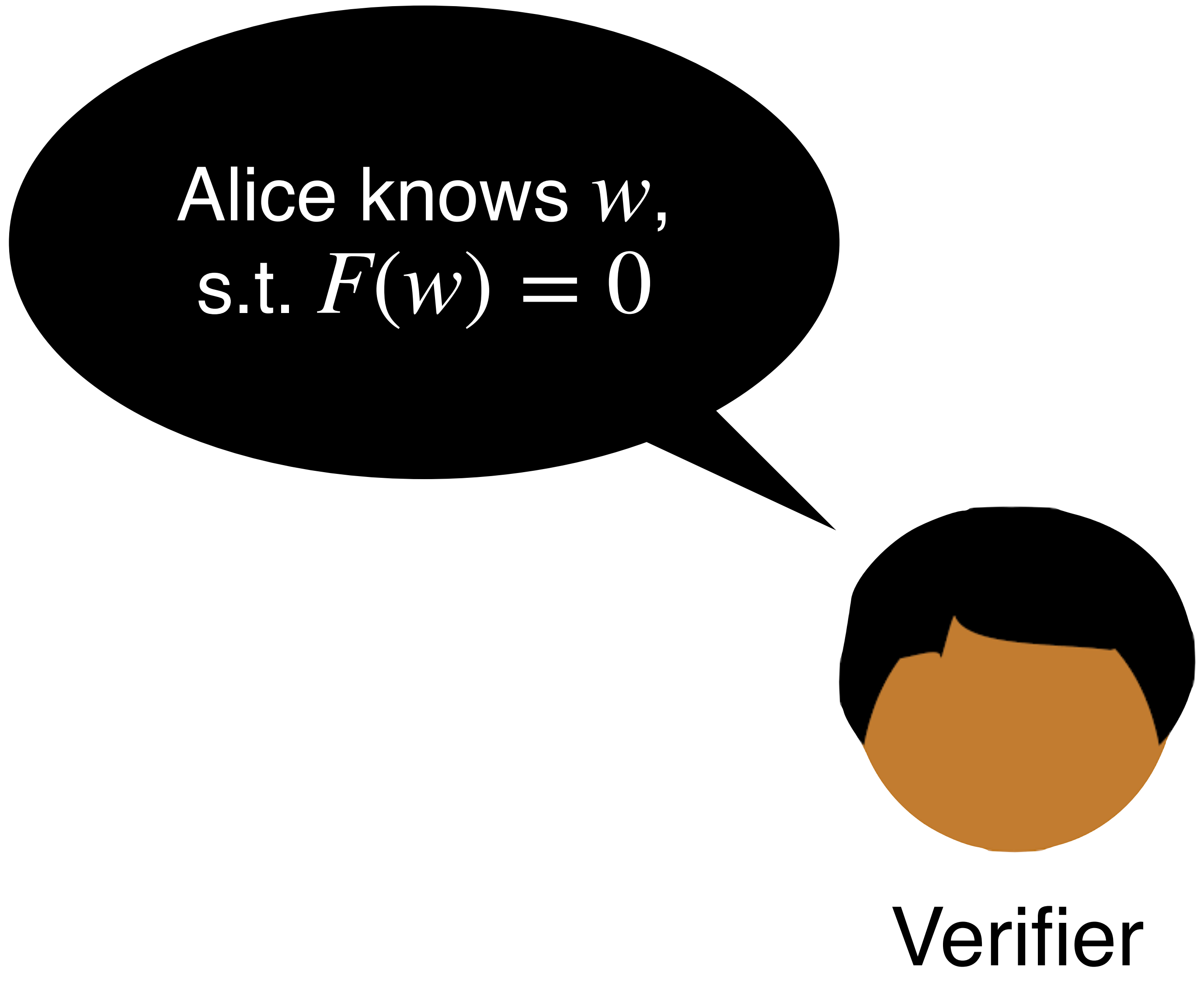
Prover

Verifier

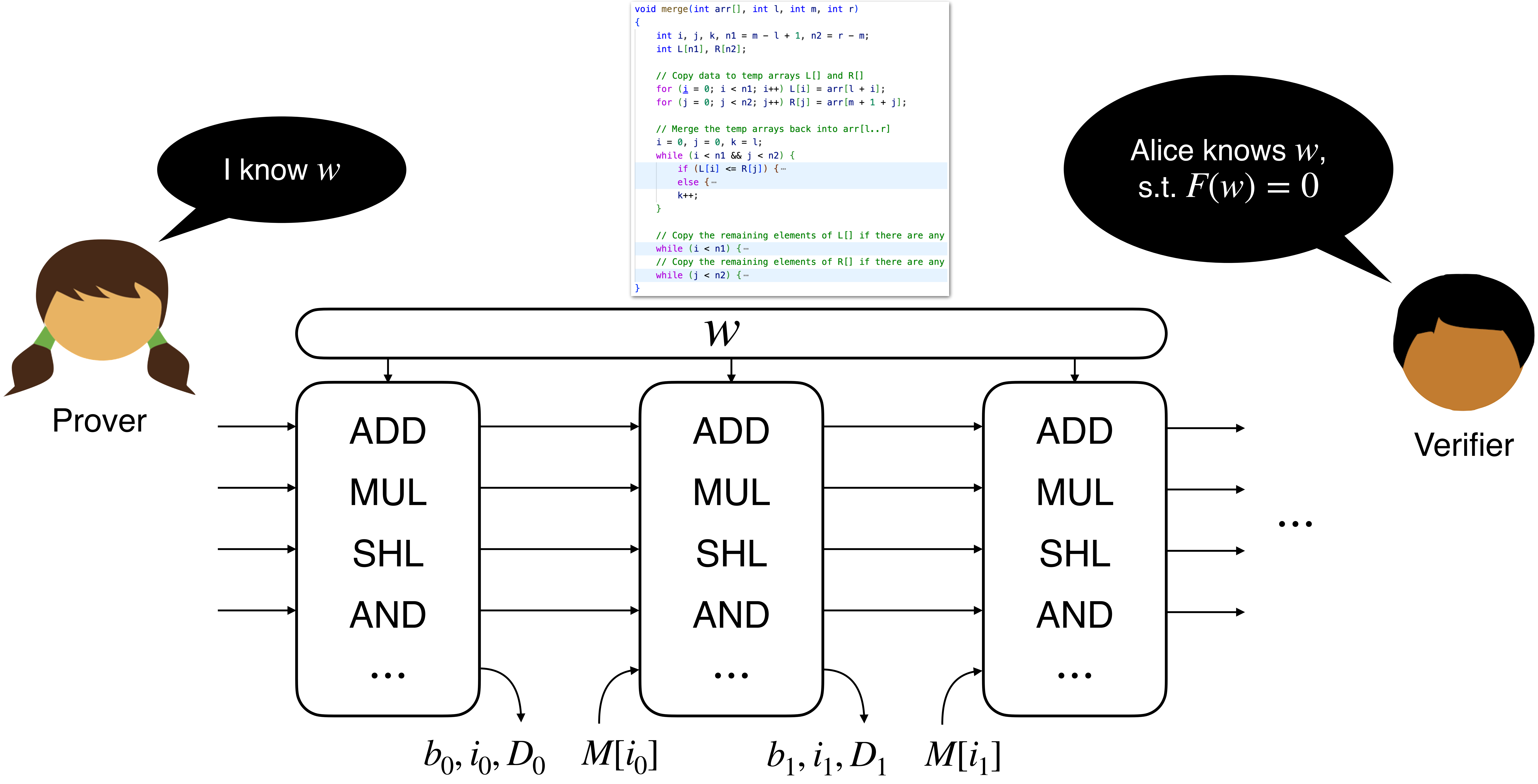
Notation



$$F(\,\cdot\,)$$



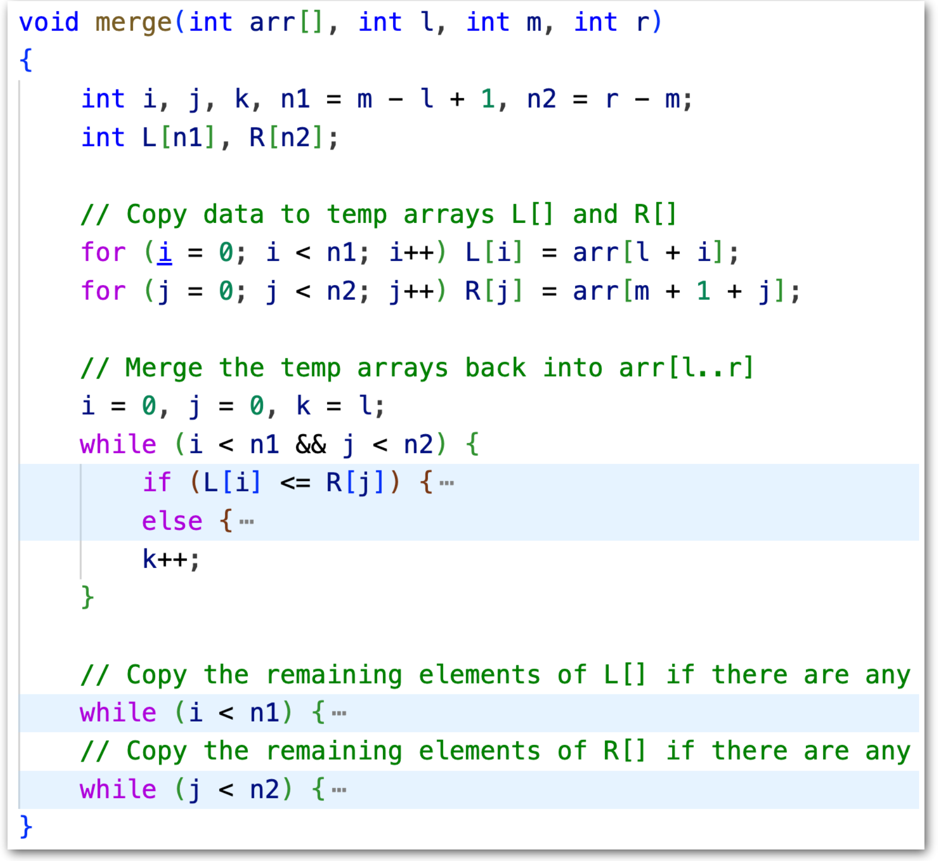
Notation



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

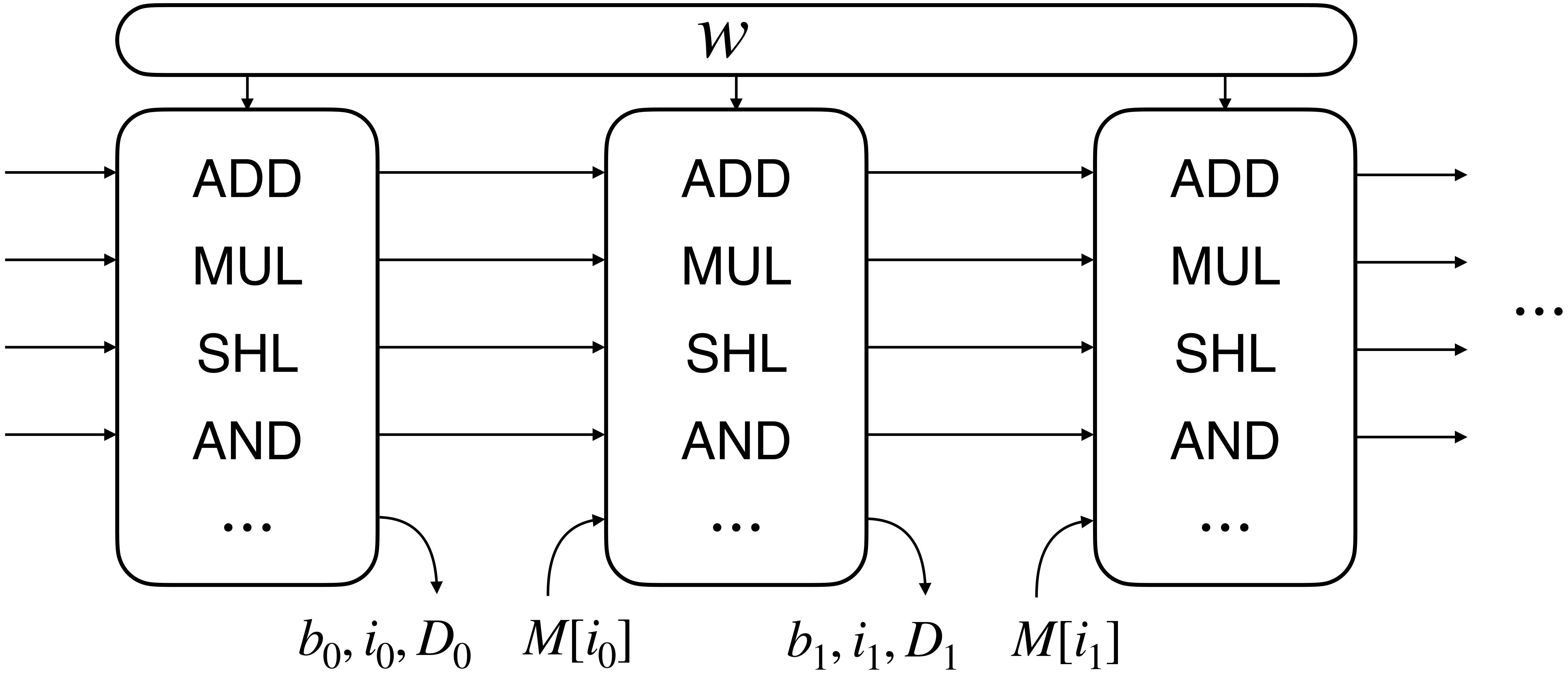
Notation

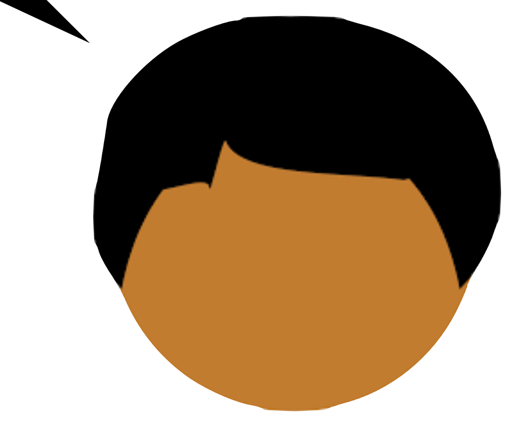
Soundness



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Prover

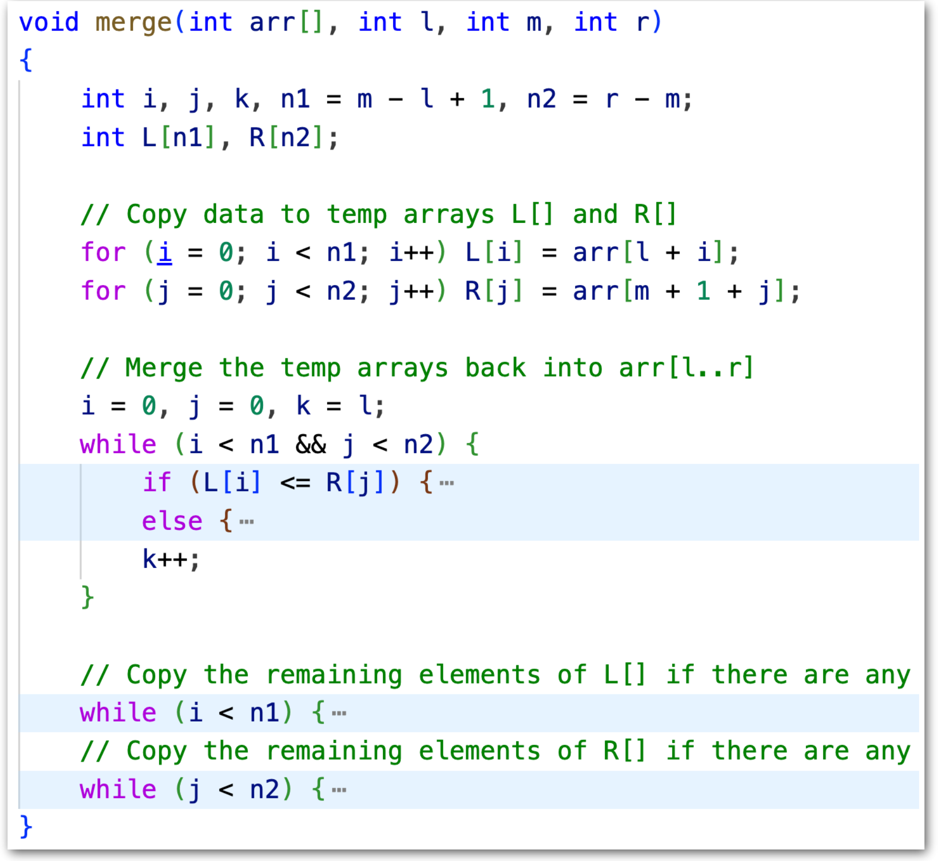




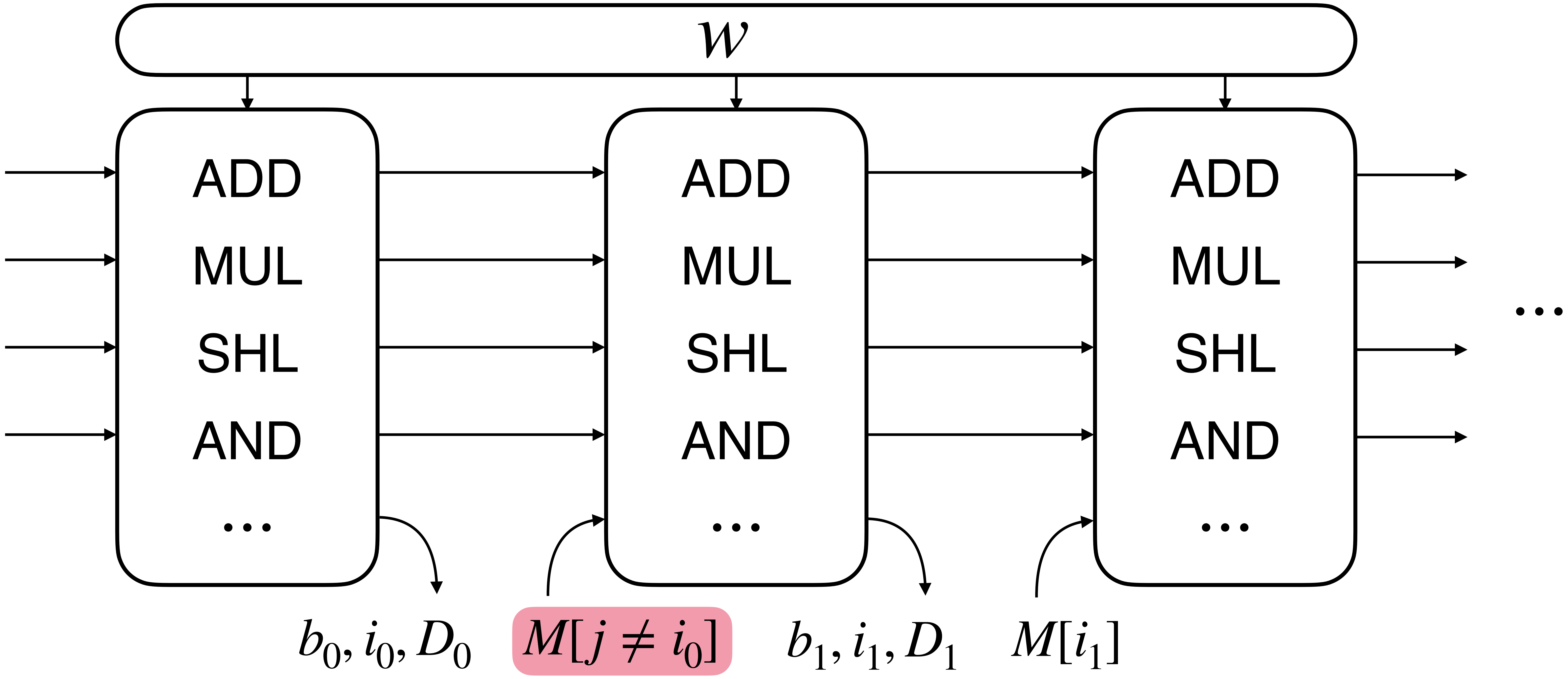
Verifier

Notation

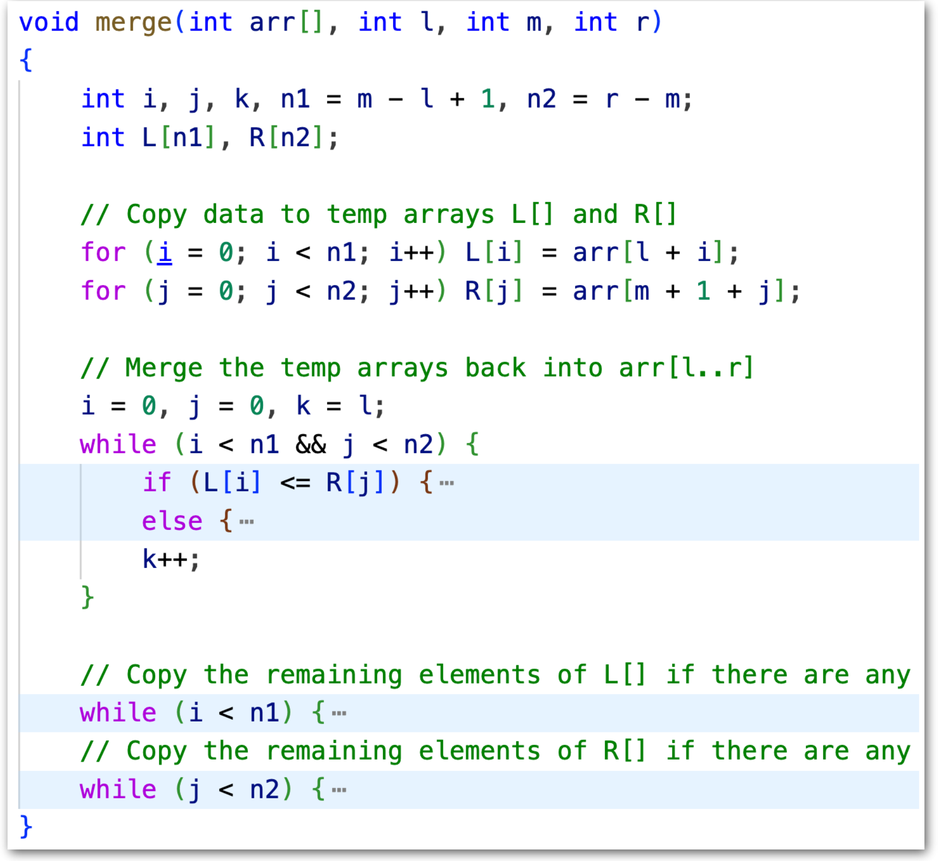
Soundness



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```



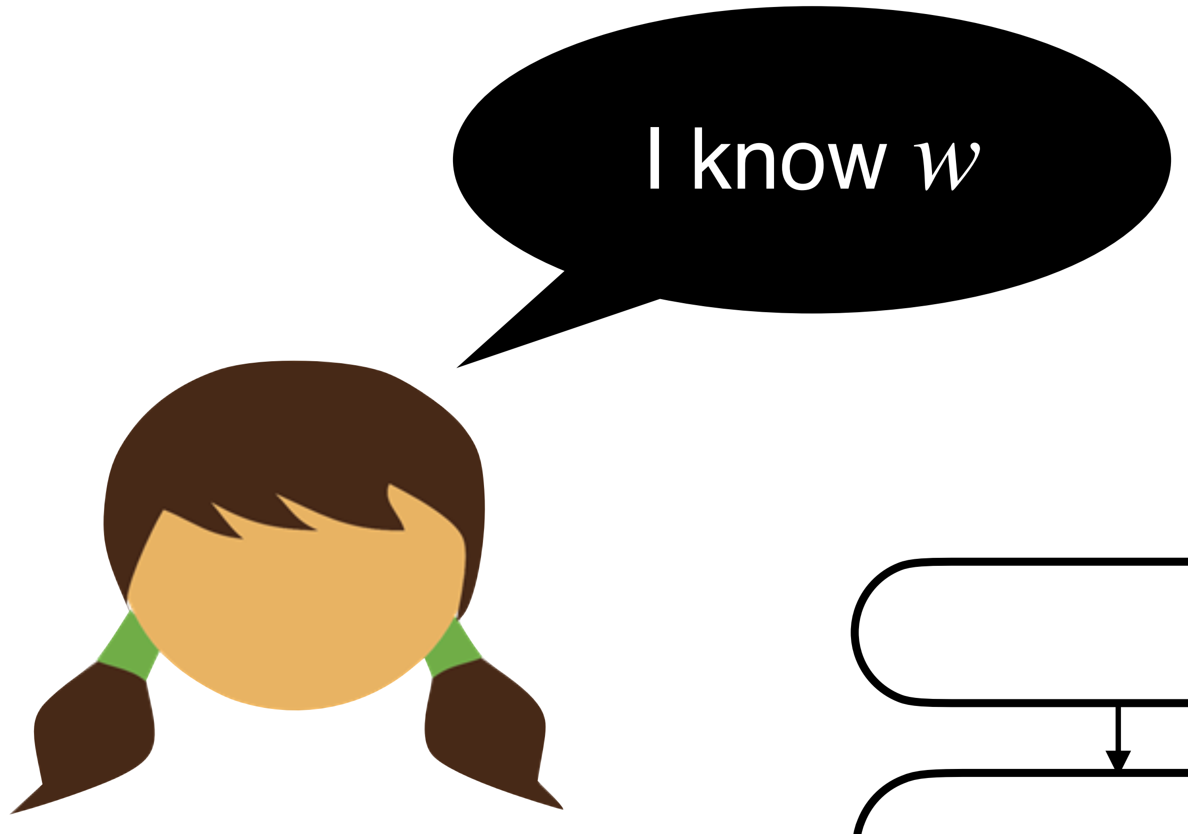
Notation



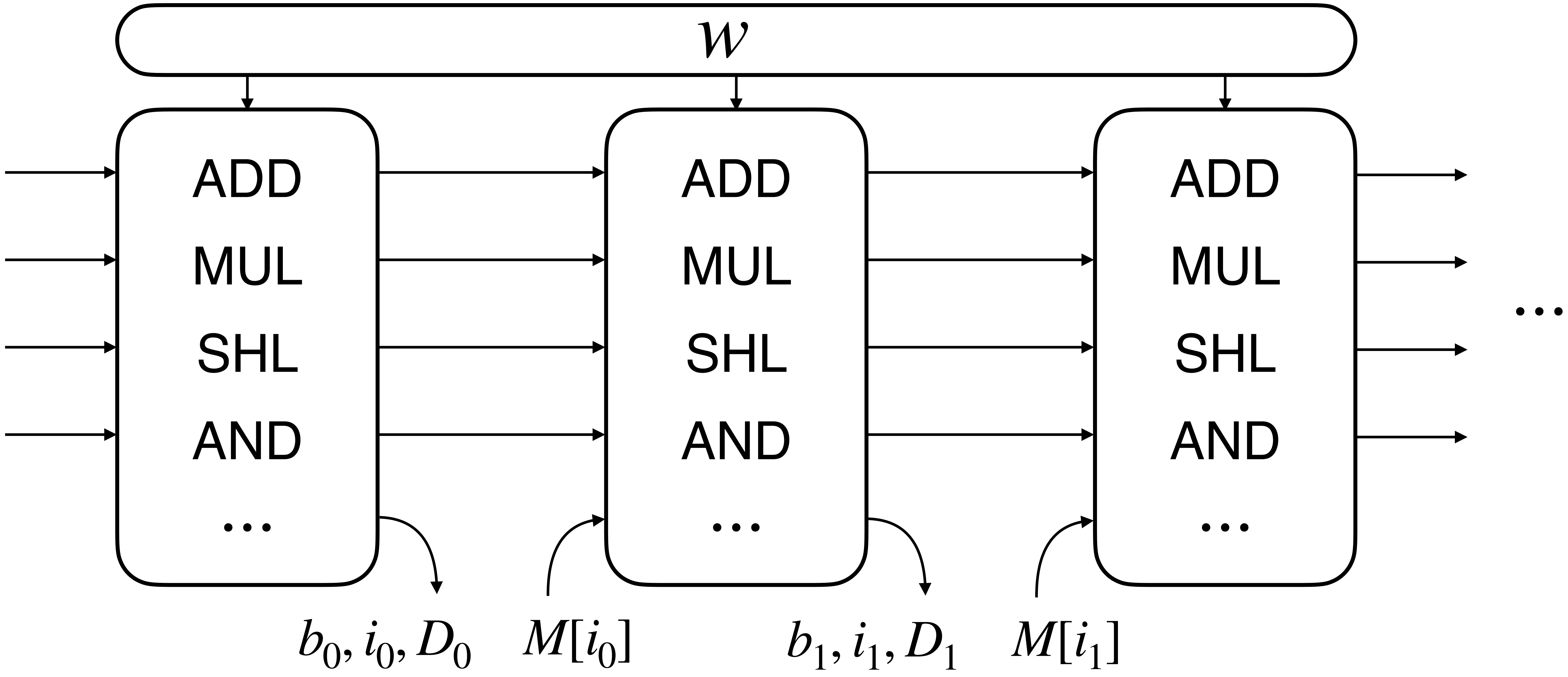
```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Soundness

Zero Knowledge



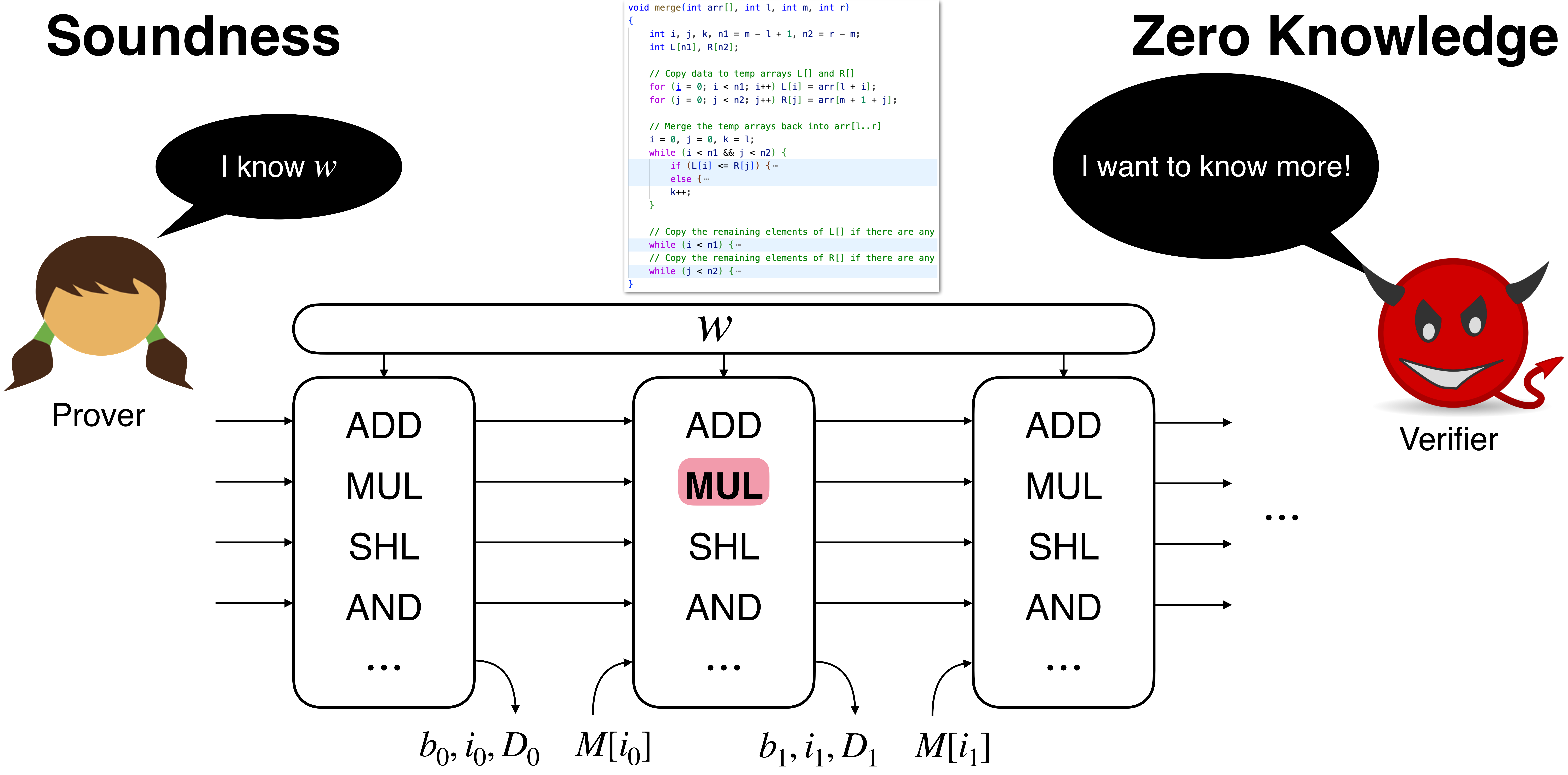
I want to know more!



Notation

Soundness

Zero Knowledge

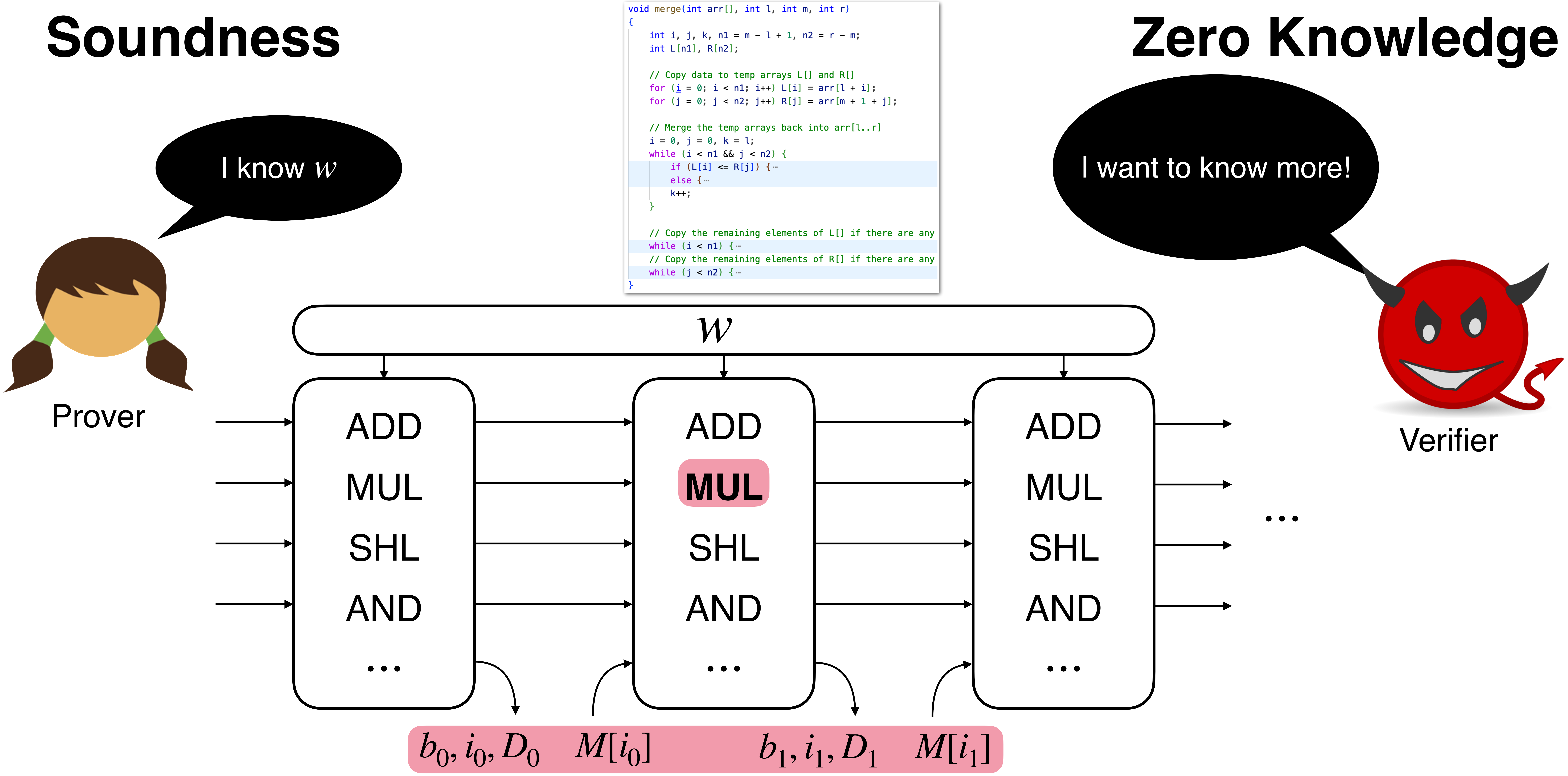


```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Notation

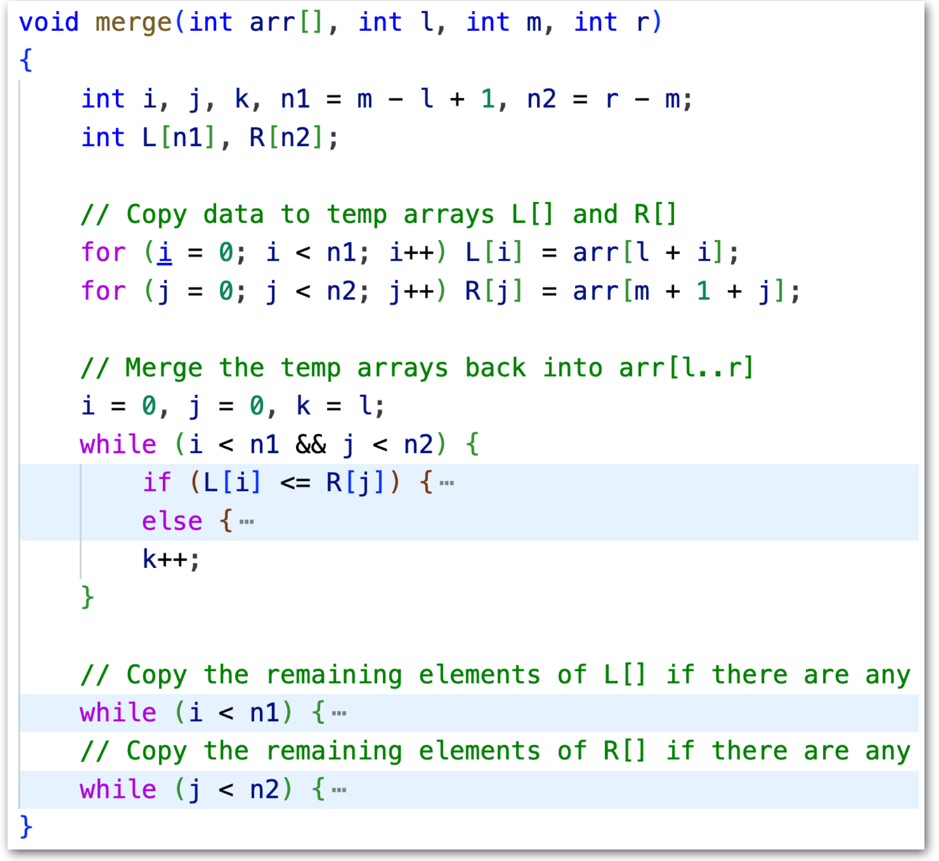
Soundness

Zero Knowledge



```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

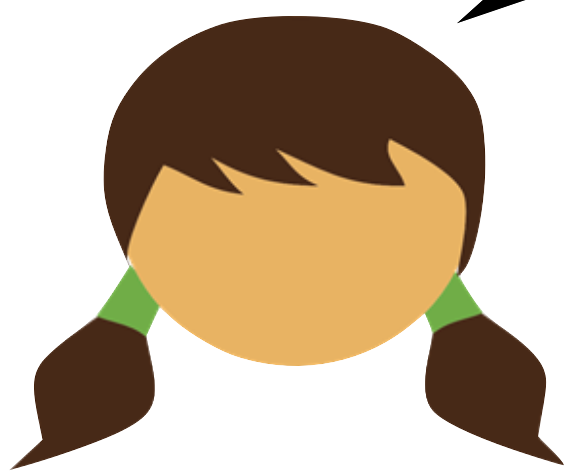
Notation

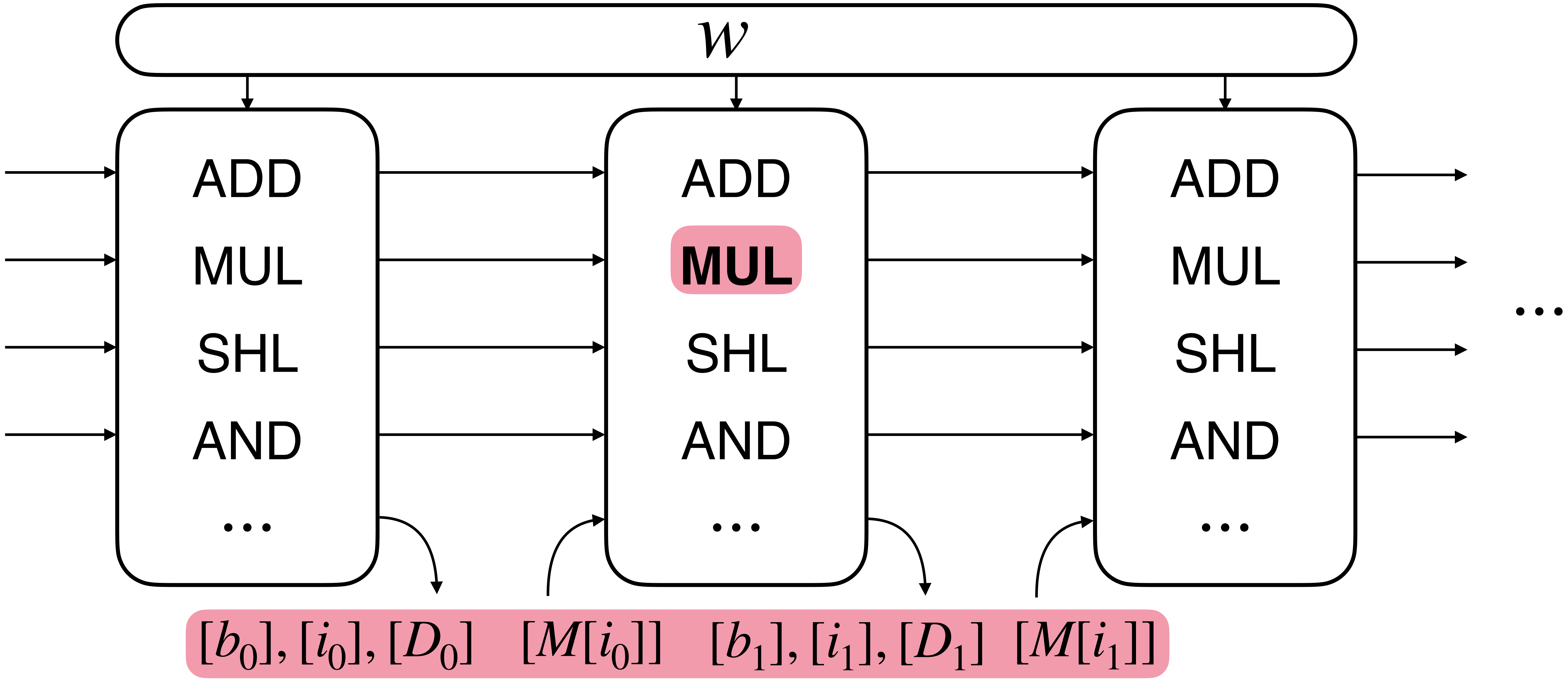


```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Soundness

Zero Knowledge



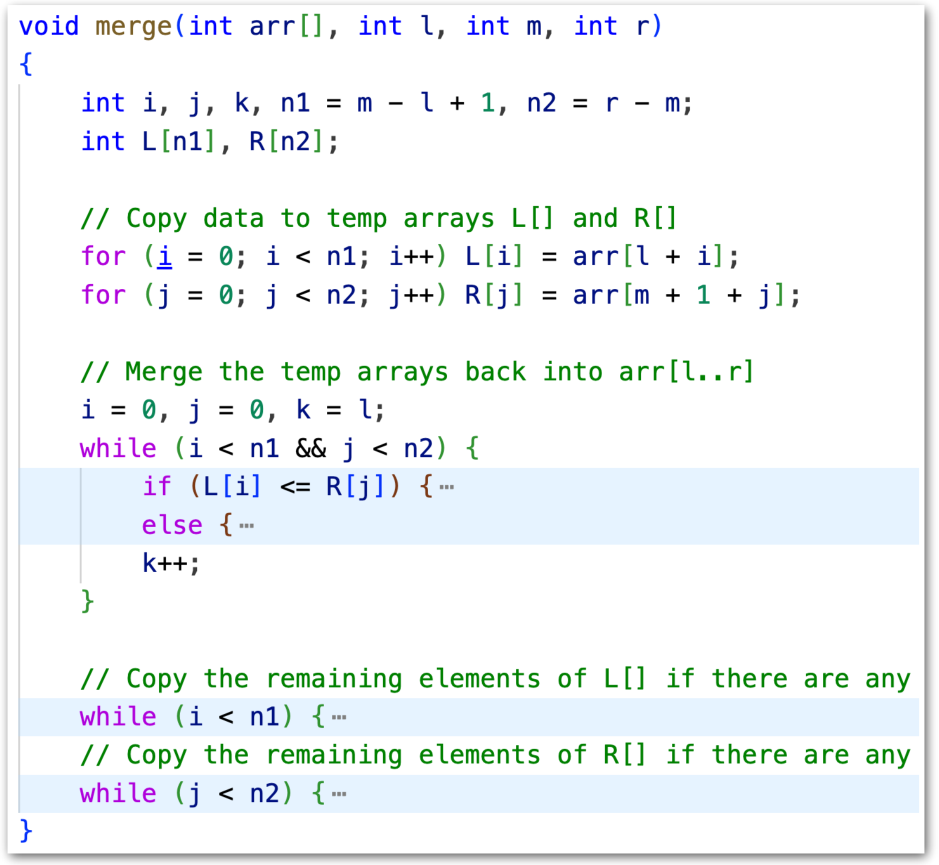


Notation

Soundness

I know w

Prover

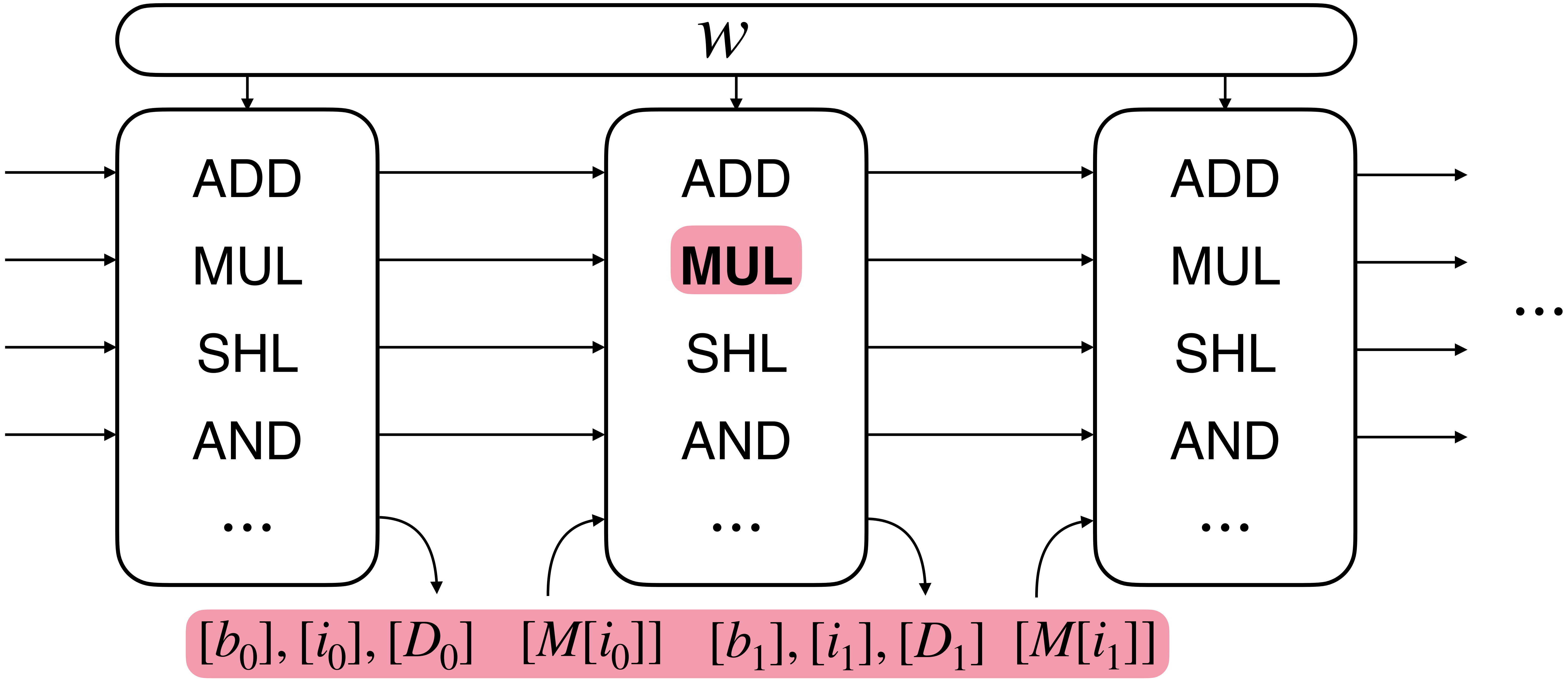


```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {⋯
        else {⋯
        k++;
    }

    // Copy the remaining elements of L[] if there are any
    while (i < n1) {⋯
    // Copy the remaining elements of R[] if there are any
    while (j < n2) {⋯
}
```

Zero Knowledge

I want to know more!

Verifier



w

ADD

MUL

SHL

AND

…

ADD

MUL

SHL

AND

…

ADD

MUL

SHL

AND

…

…

$[b_0], [i_0], [D_0]$ $[M[i_0]]$ $[b_1], [i_1], [D_1]$ $[M[i_1]]$

Each costs $O(1)$:

$[x] + [y] = [x + y]$

$[x] \cdot [y] = [x \cdot y]$

test_zero($[x]$)

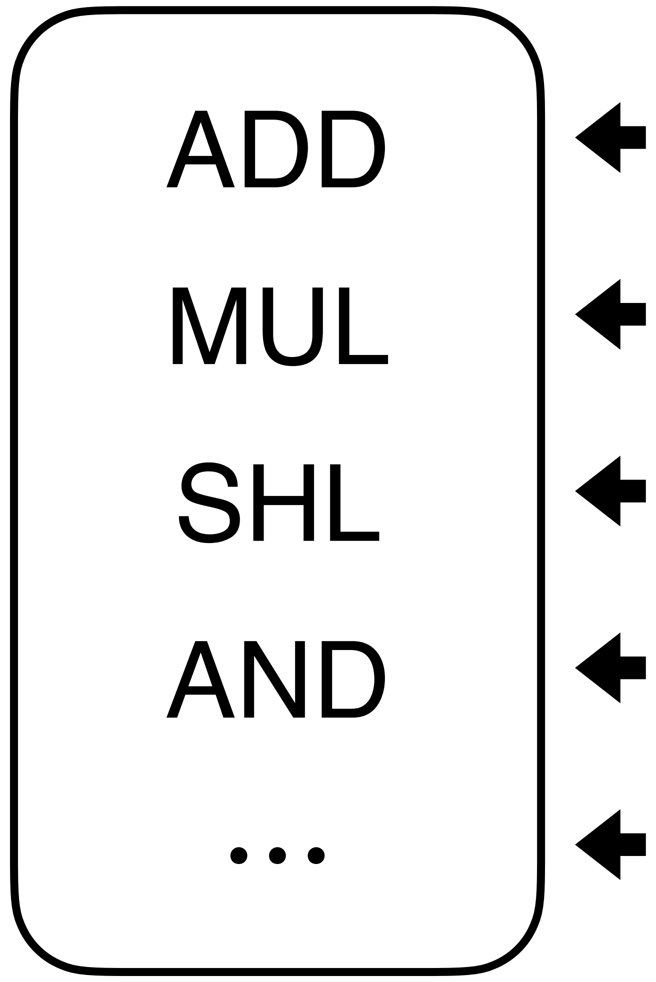
Challenges and Techniques

Challenges and Techniques

Verifier

Challenges and Techniques

Verifier



ADD

MUL

SHL

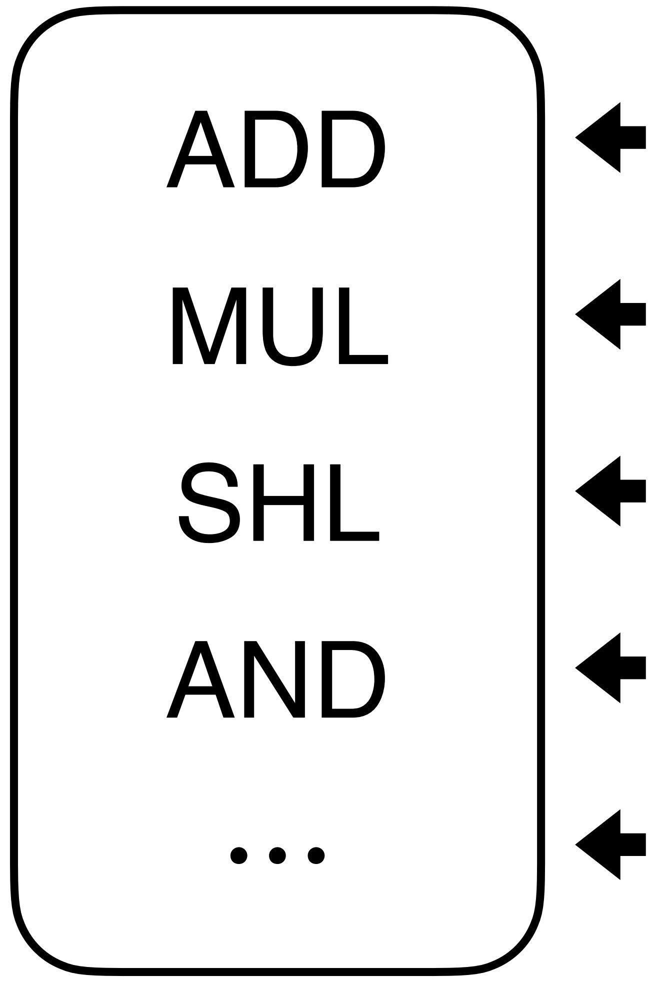
AND

…

Verifier needs to read every instruction;
otherwise, the unread one is not executed

Challenges and Techniques

Verifier



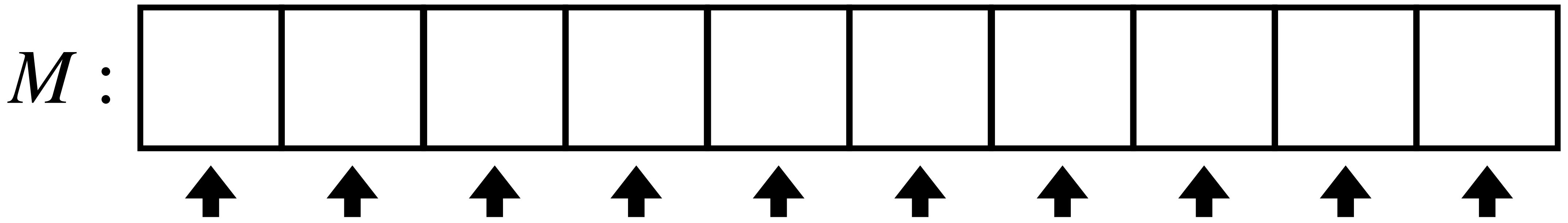
ADD

MUL

SHL

AND

…



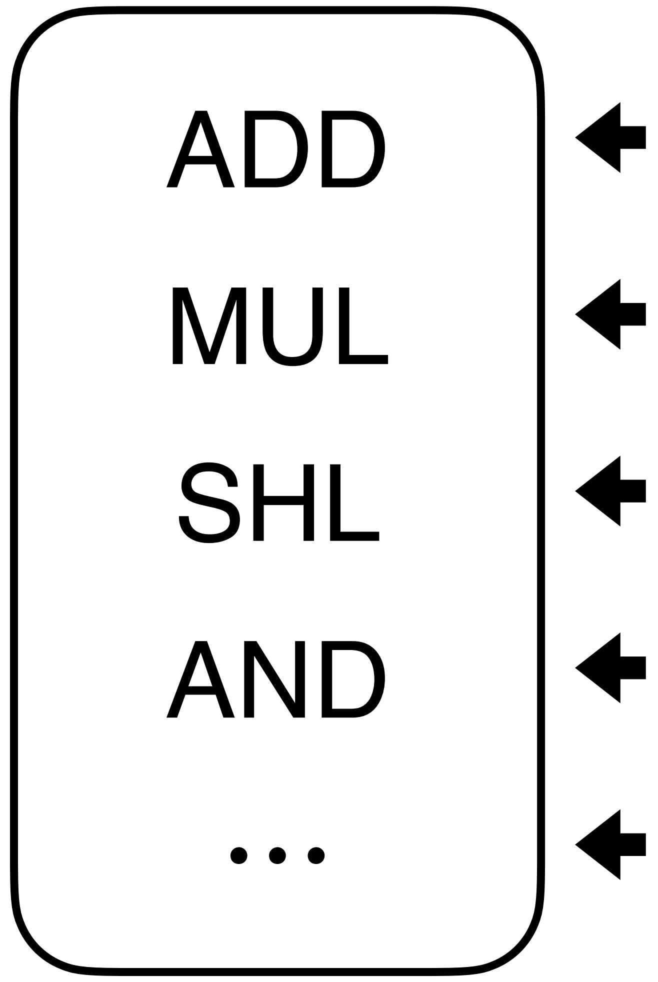
$M:$

Verifier needs to read every instruction;
otherwise, the unread one is not executed

Verifier needs to read every slot;
otherwise, the unread one is not executed

Challenges and Techniques

Verifier



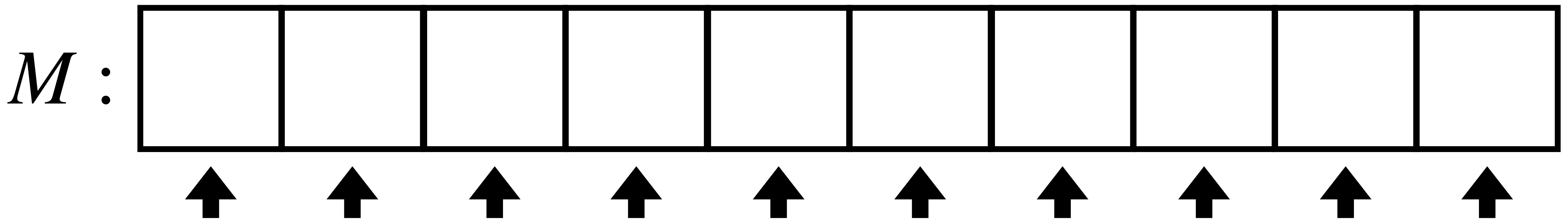
ADD

MUL

SHL

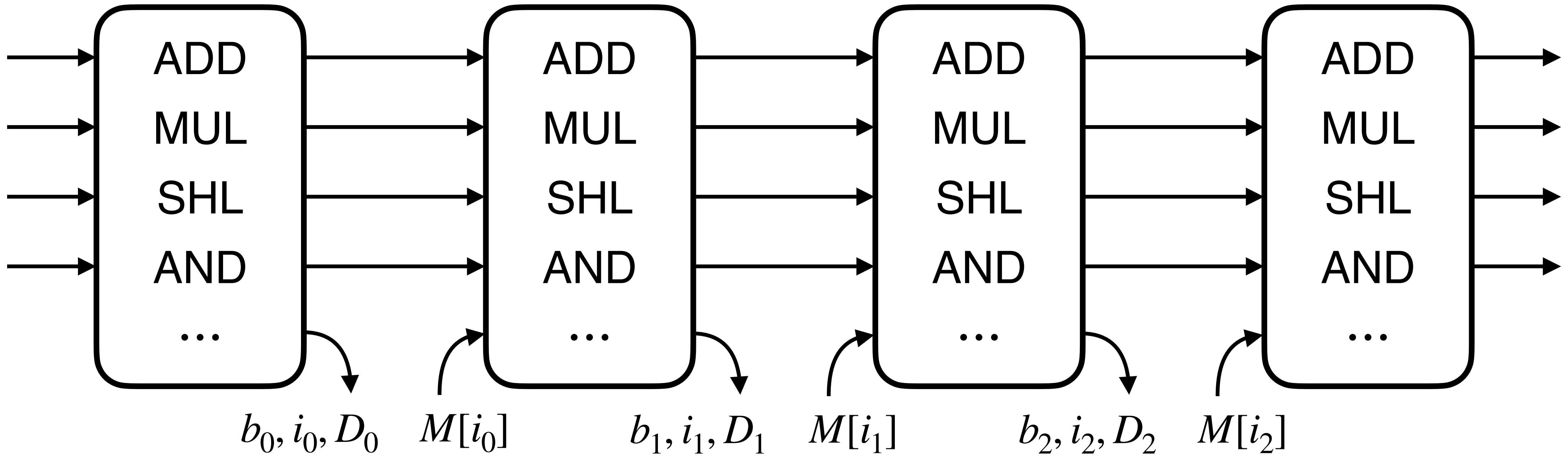
AND

…



Verifier needs to read every instruction;
otherwise, the unread one is not executed

Verifier needs to read every slot;
otherwise, the unread one is not executed

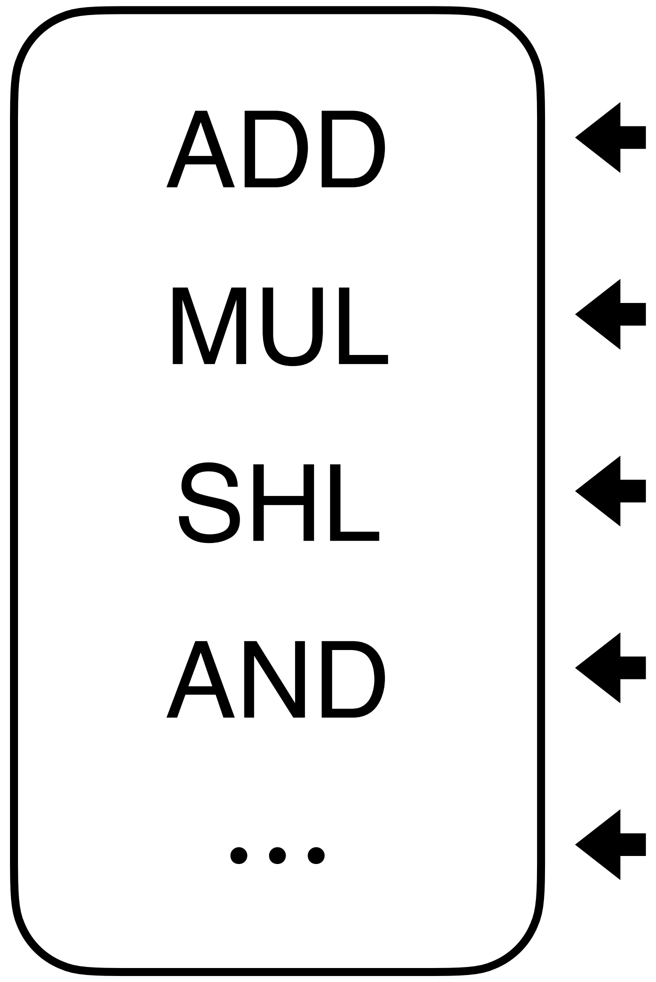


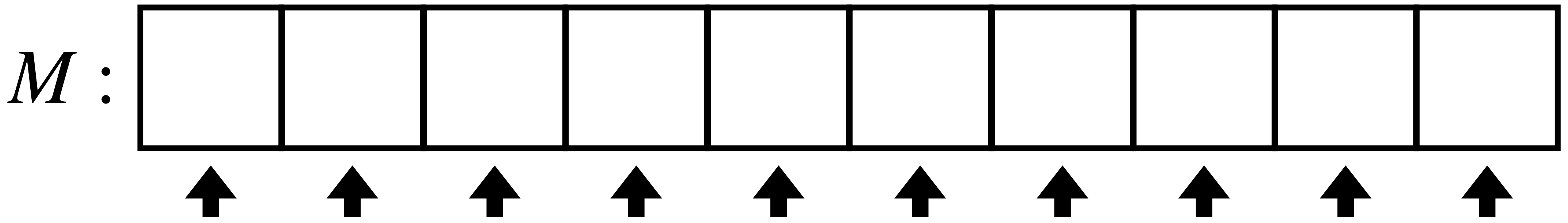
ADD MUL SHL AND … → ADD MUL SHL AND … → ADD MUL SHL AND … → ADD MUL SHL AND …

b_0, i_0, D_0 $M[i_0]$ b_1, i_1, D_1 $M[i_1]$ b_2, i_2, D_2 $M[i_2]$

Challenges and Techniques

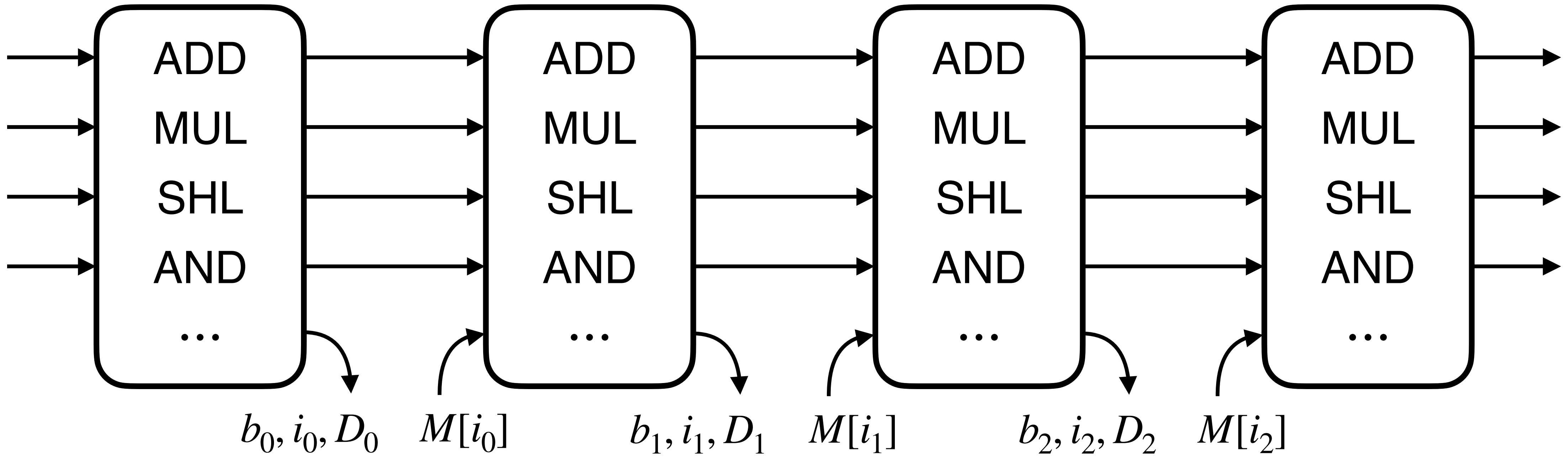
Verifier





Verifier needs to read every instruction;
otherwise, the unread one is not executed

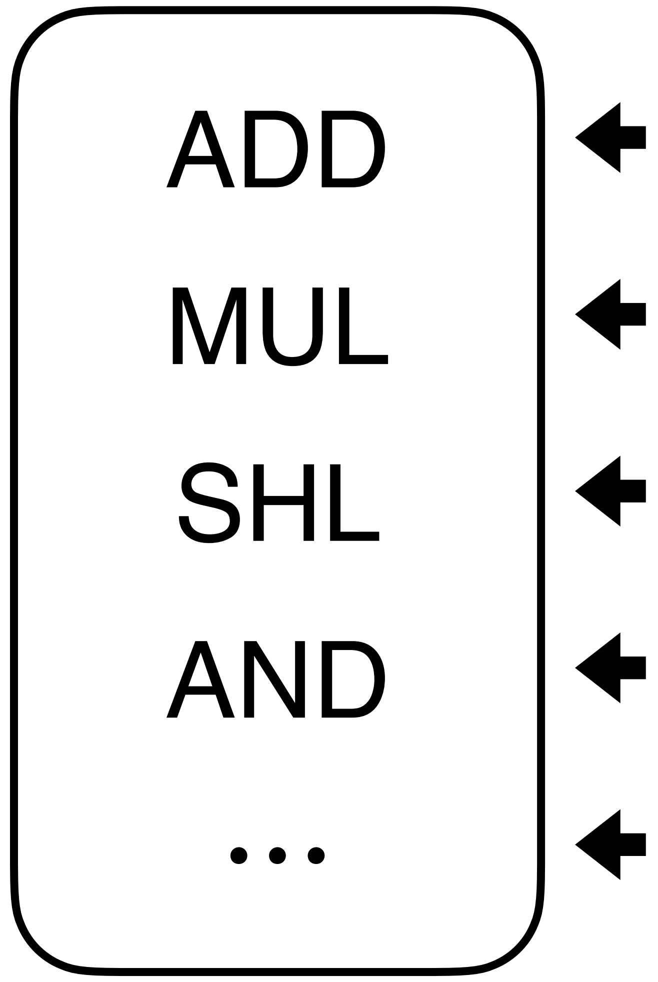
Verifier needs to read every slot;
otherwise, the unread one is not executed

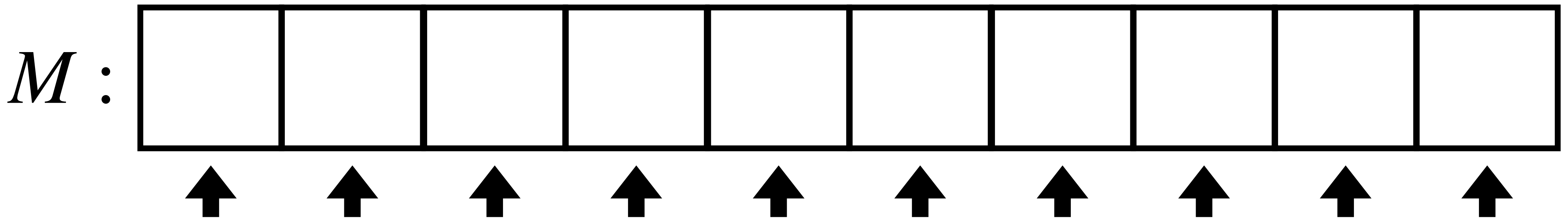


1. **We _repeatedly_ use the same branching or memory, the linear cost can be effectively amortized over multiple accesses**

Challenges and Techniques

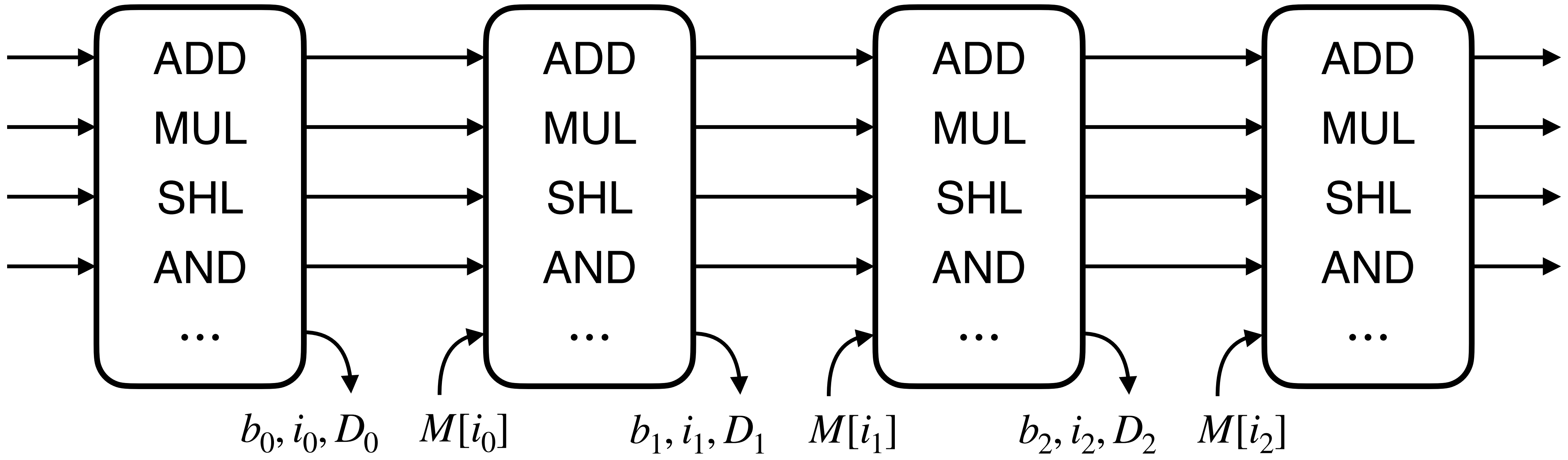
Verifier





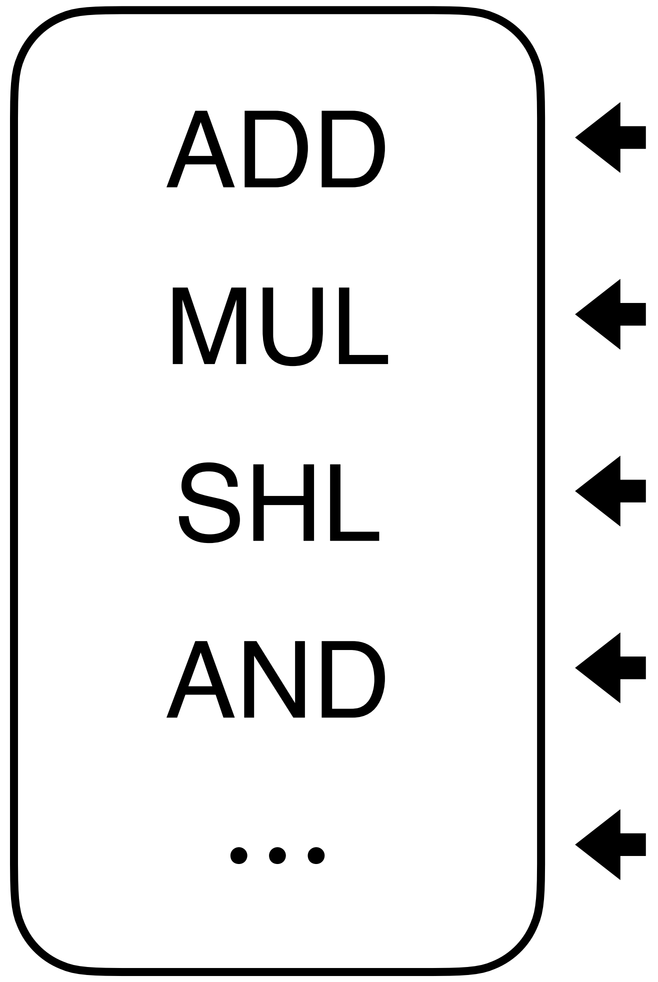
Verifier needs to read every instruction;
otherwise, the unread one is not executed

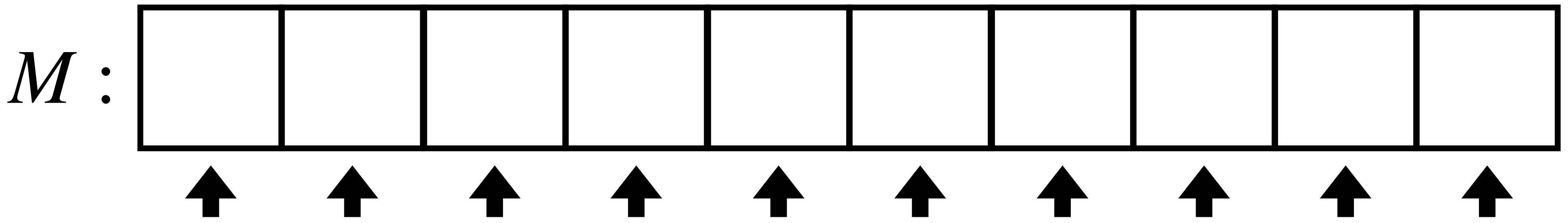
Verifier needs to read every slot;
otherwise, the unread one is not executed



1. **We *repeatedly* use the same branching or memory, the linear cost can be effectively amortized over multiple accesses**
2. **👧 knows *everything*, 👦🏾 only needs to verify (Remark: 👧 can still cheat!)**

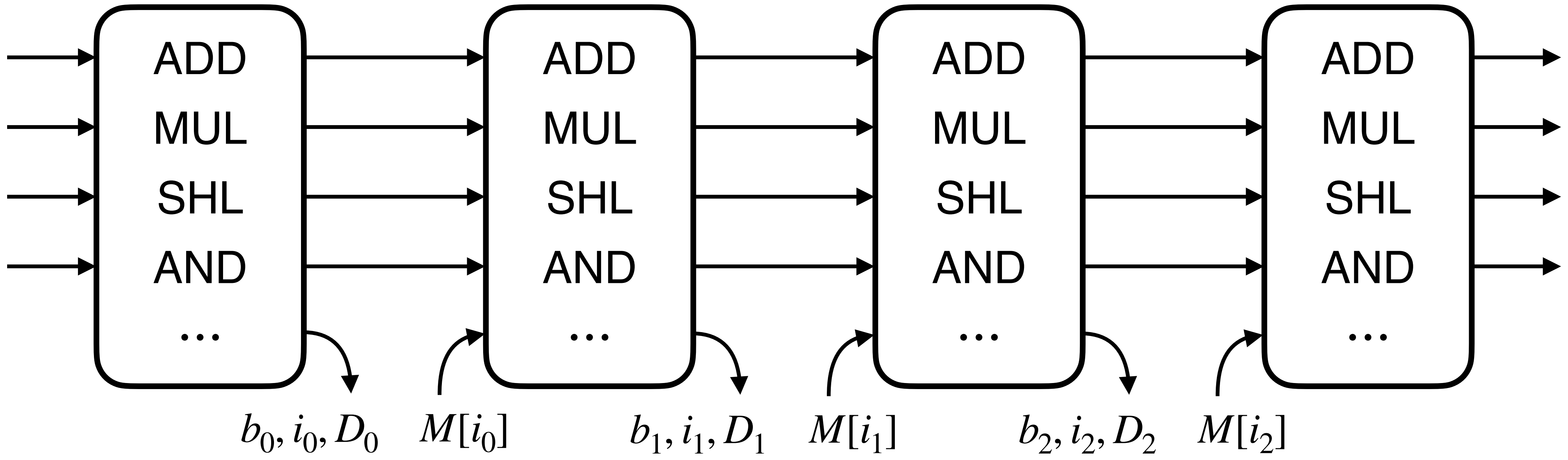
Challenges and Techniques





Verifier needs to read every instruction;
otherwise, the unread one is not executed

Verifier needs to read every slot;
otherwise, the unread one is not executed



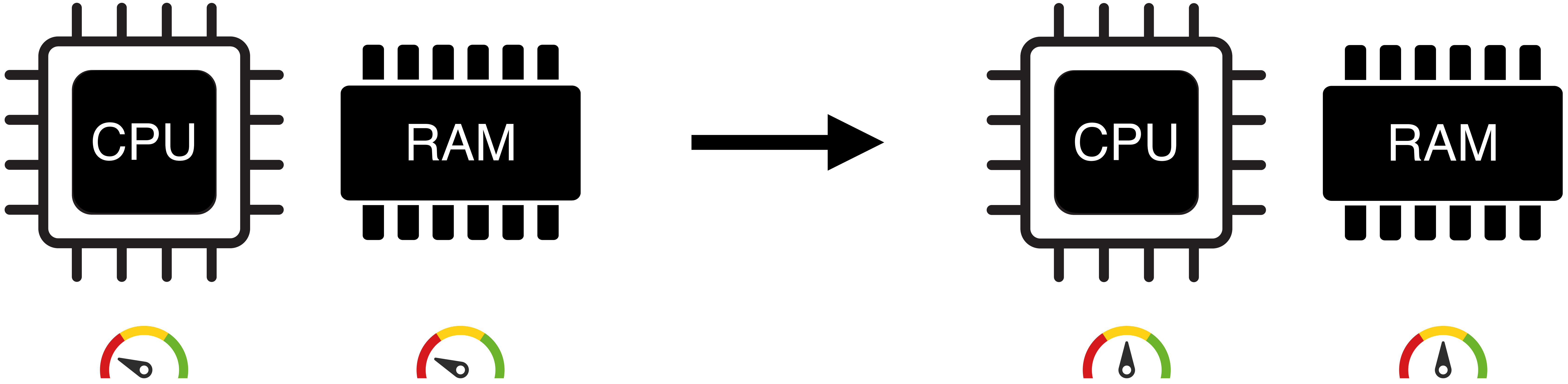
Tech. 1: reuse and amortize

Tech. 2: P knows and helps

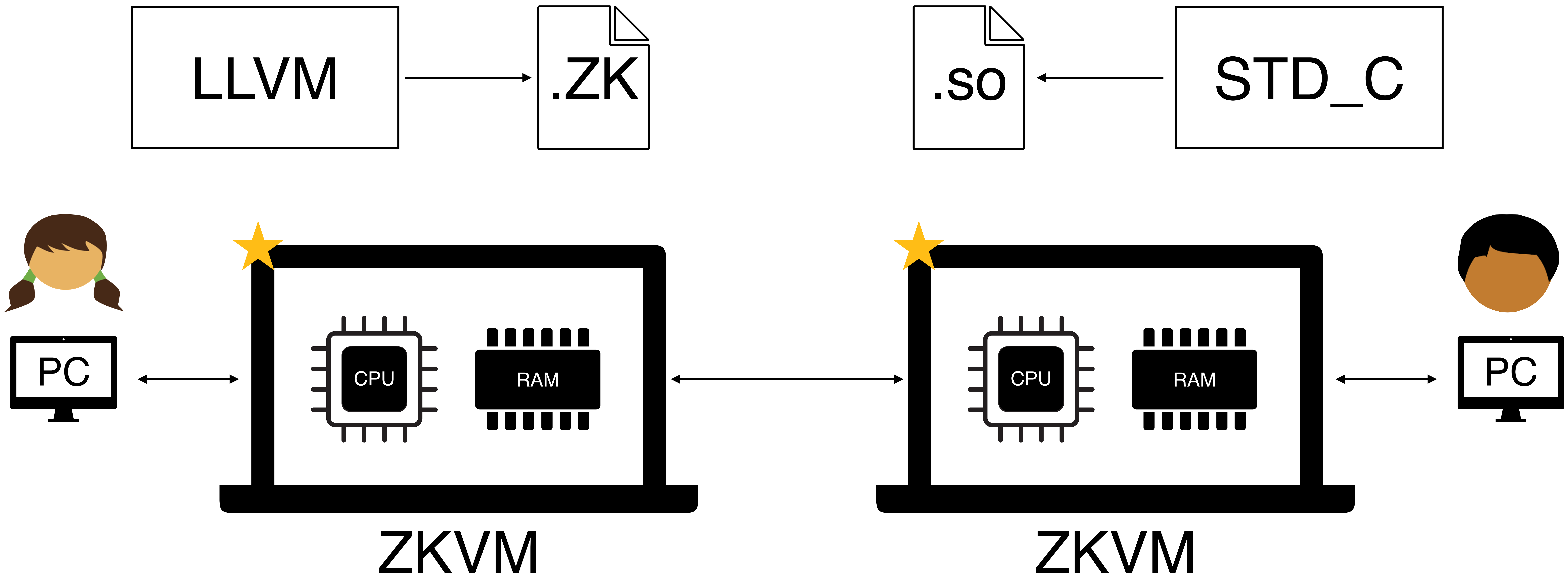
IEEE S&P 2021

Zero Knowledge for Everything and Everyone:
Fast ZK Processor with Cached ORAM for ANSI C Programs

David Heath* Yibin Yang* David Devecsery Vladimir Kolesnikov
Georgia Institute of Technology
Email: {heath.davidanthony,yyang811,ddevec,kolesnikov}@gatech.edu
*Authors contributed equally



A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈ 10KHz ($\approx \mathbf{1000}$**x**)



LLVM

.ZK

.so

STD_C

PC

CPU

RAM

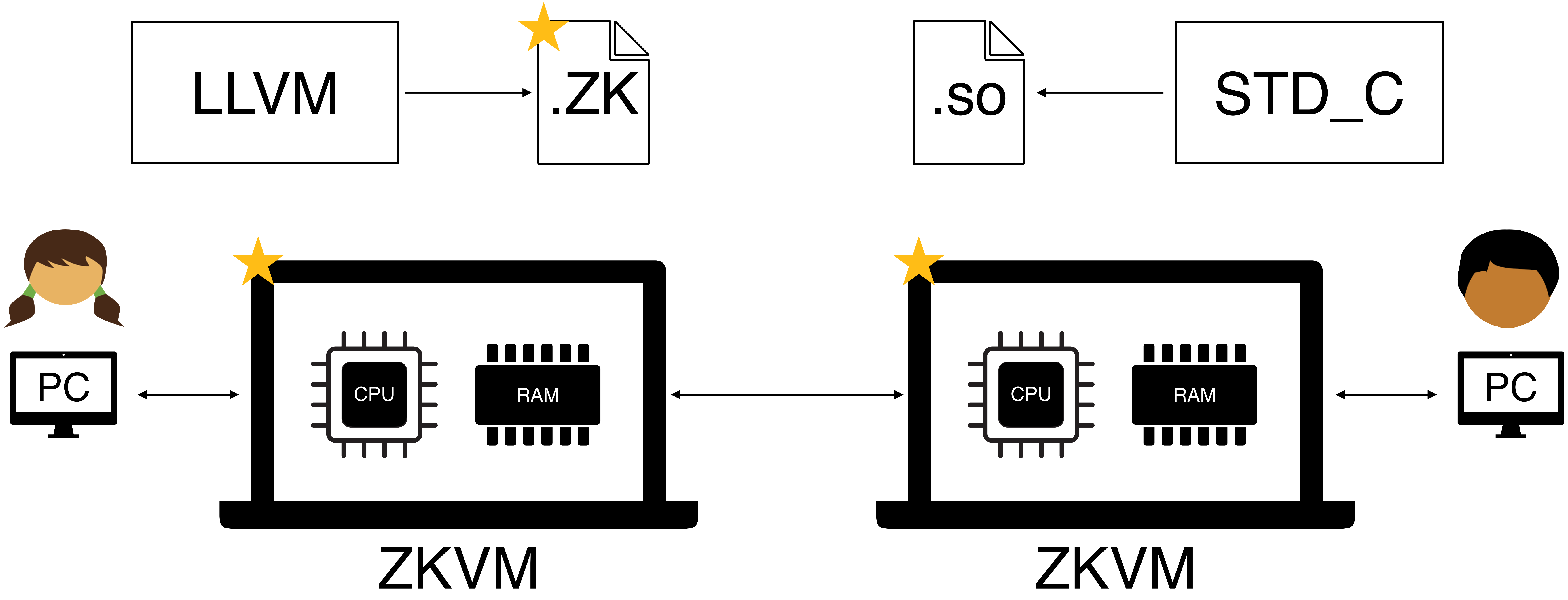
ZKVM

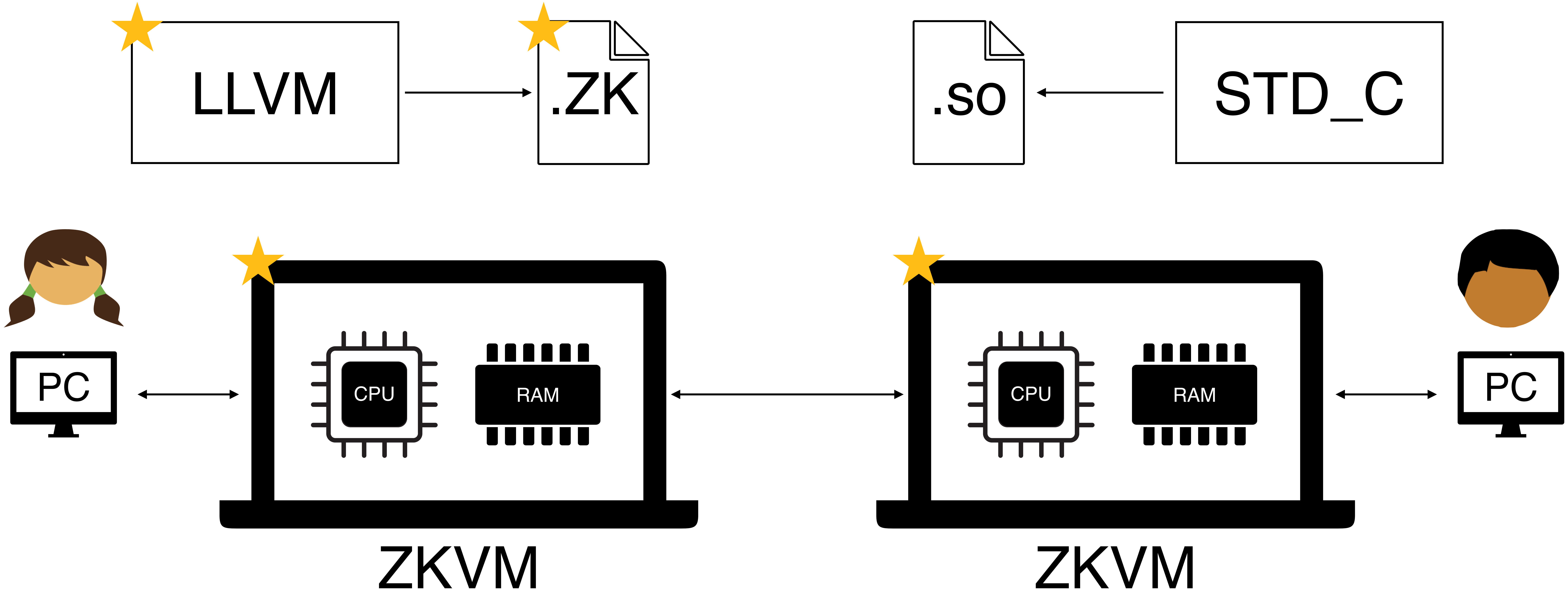
CPU

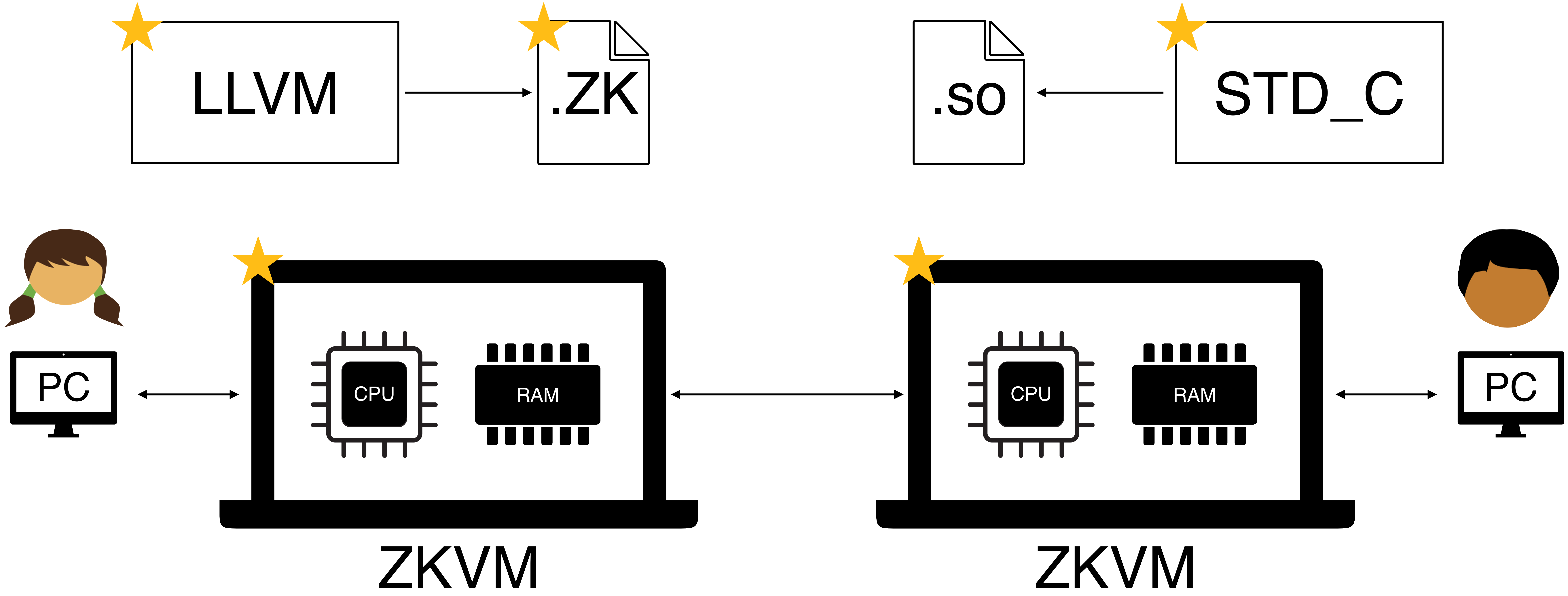
RAM

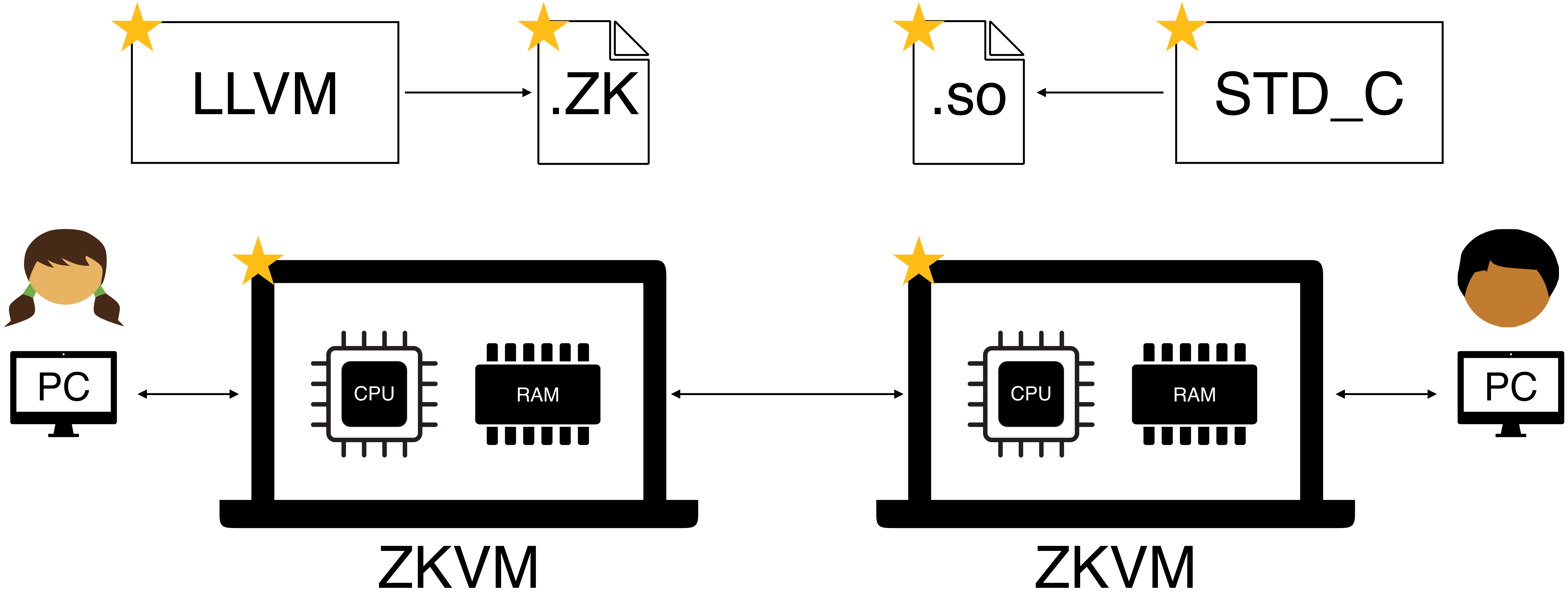
ZKVM

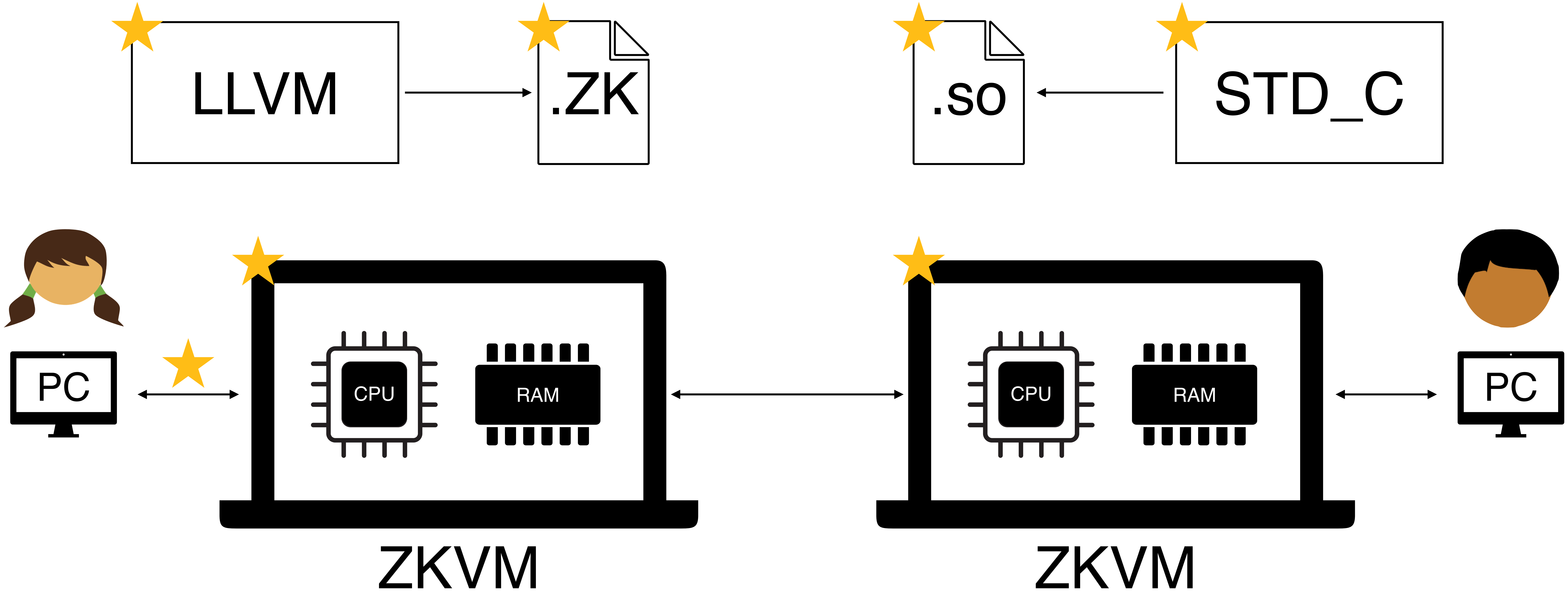
PC


LLVM
.ZK
.so
STD_C
PC
CPU
RAM
ZKVM
CPU
RAM
ZKVM
PC

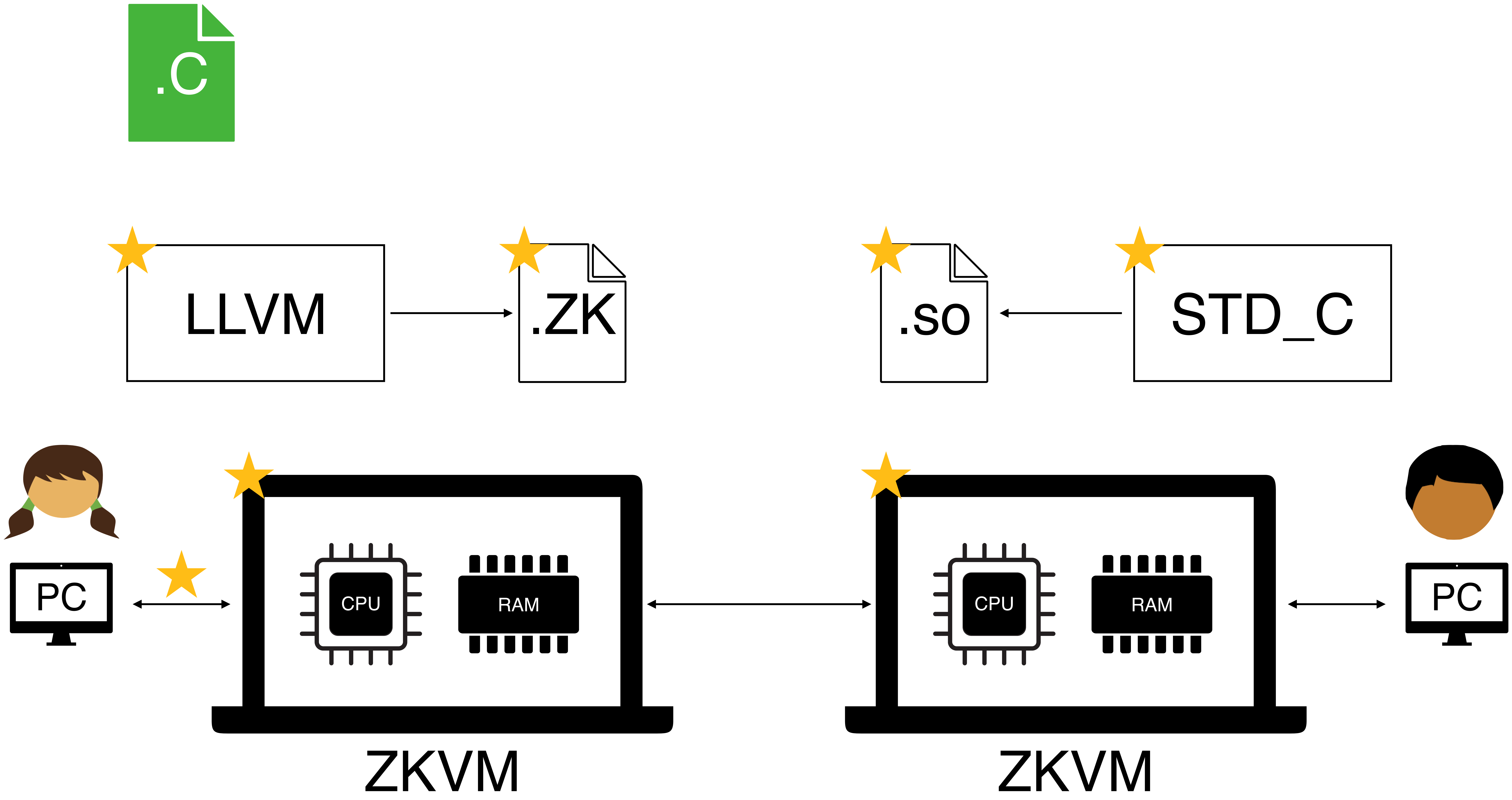

LLVM
.ZK
.so
STD_C
PC
CPU
RAM
ZKVM
CPU
RAM
ZKVM
PC


LLVM
.ZK
.so
STD_C
PC
CPU
RAM
ZKVM
CPU
RAM
ZKVM
PC


LLVM
.ZK
.so
STD_C
PC
CPU
RAM
ZKVM
CPU
RAM
ZKVM
PC


LLVM
.ZK
.so
STD_C
PC
CPU
RAM
ZKVM
CPU
RAM
ZKVM
PC

Design



.C

LLVM

.ZK

.so

STD_C

PC

CPU

RAM

ZKVM

CPU

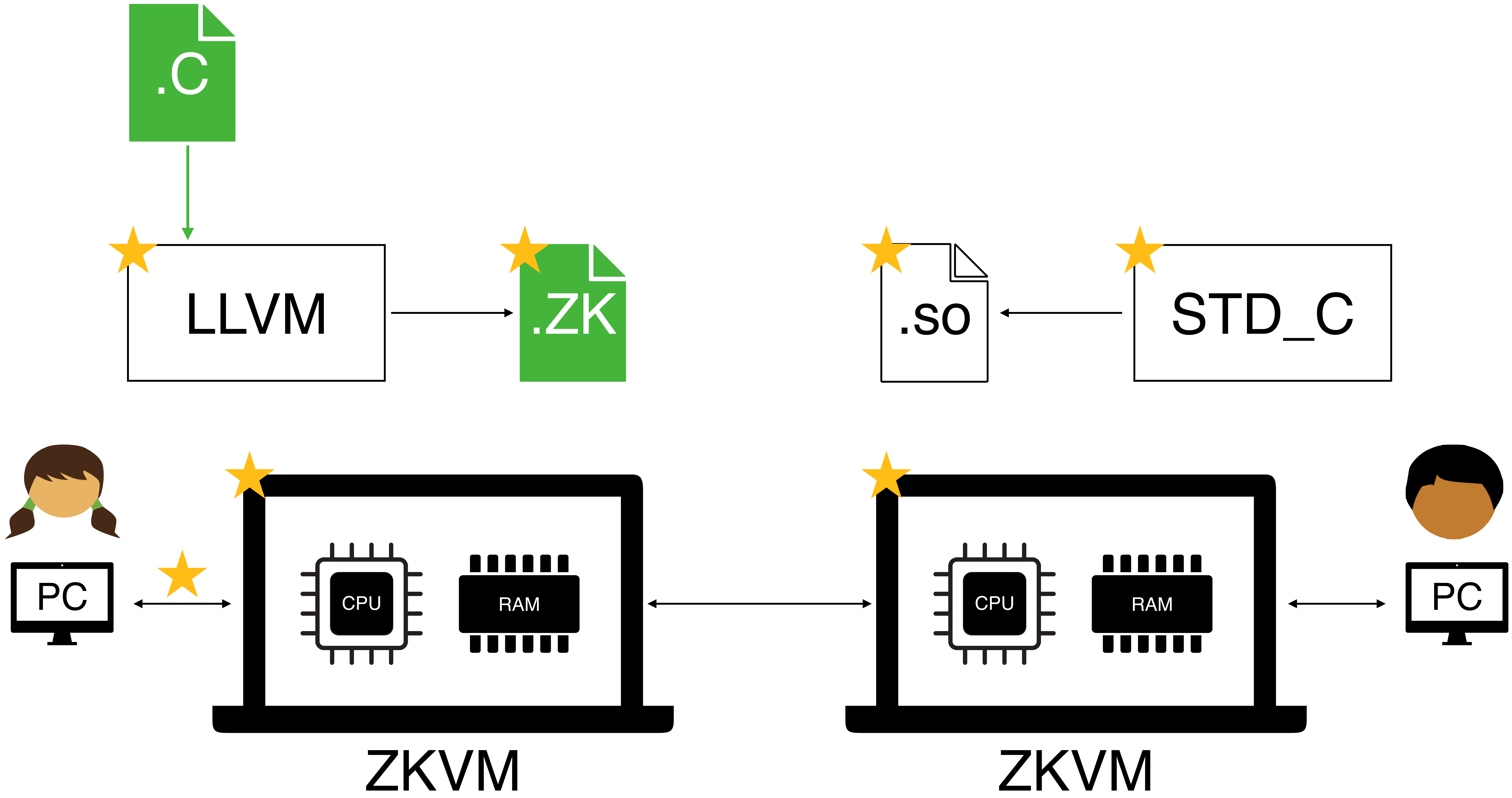
RAM

ZKVM

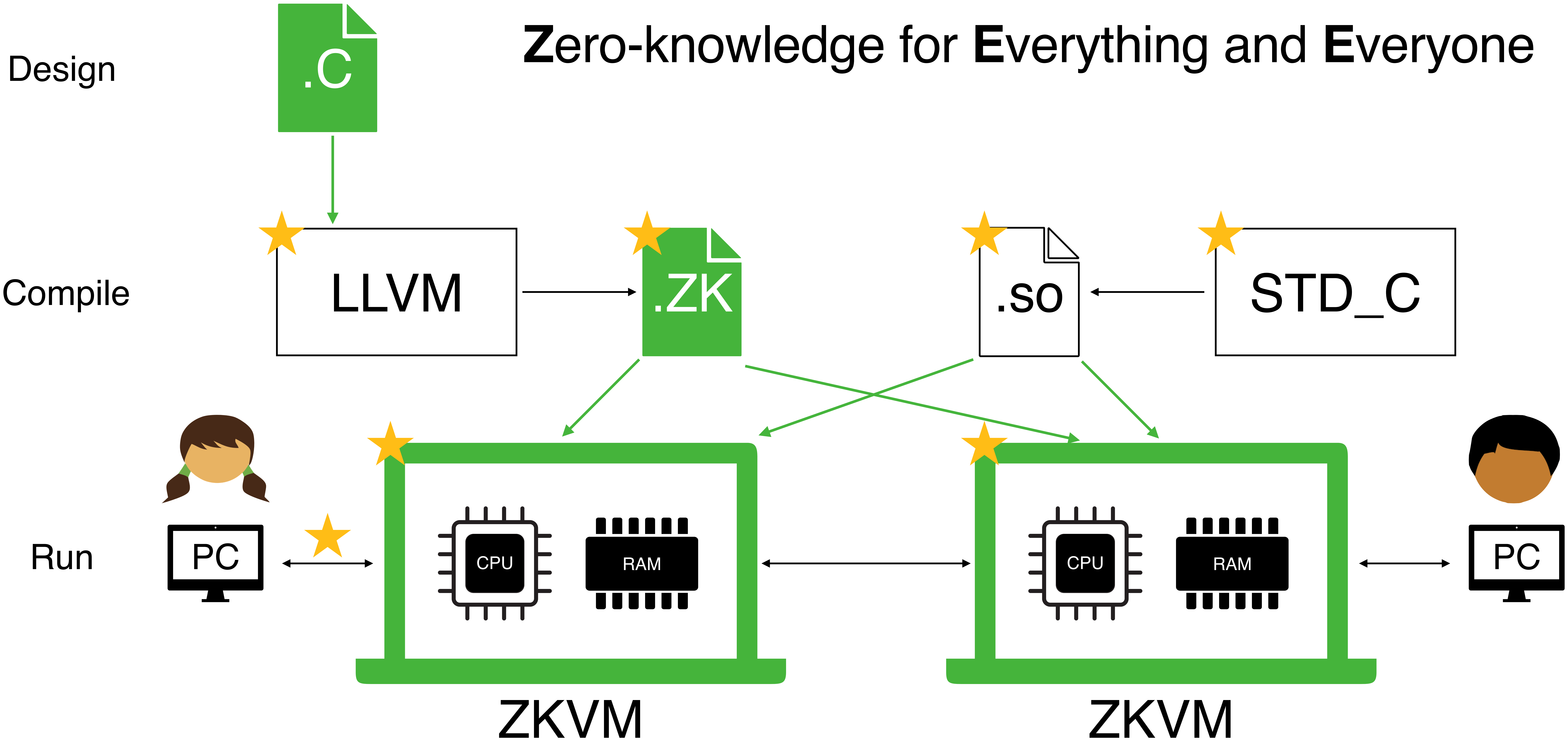
PC

Design

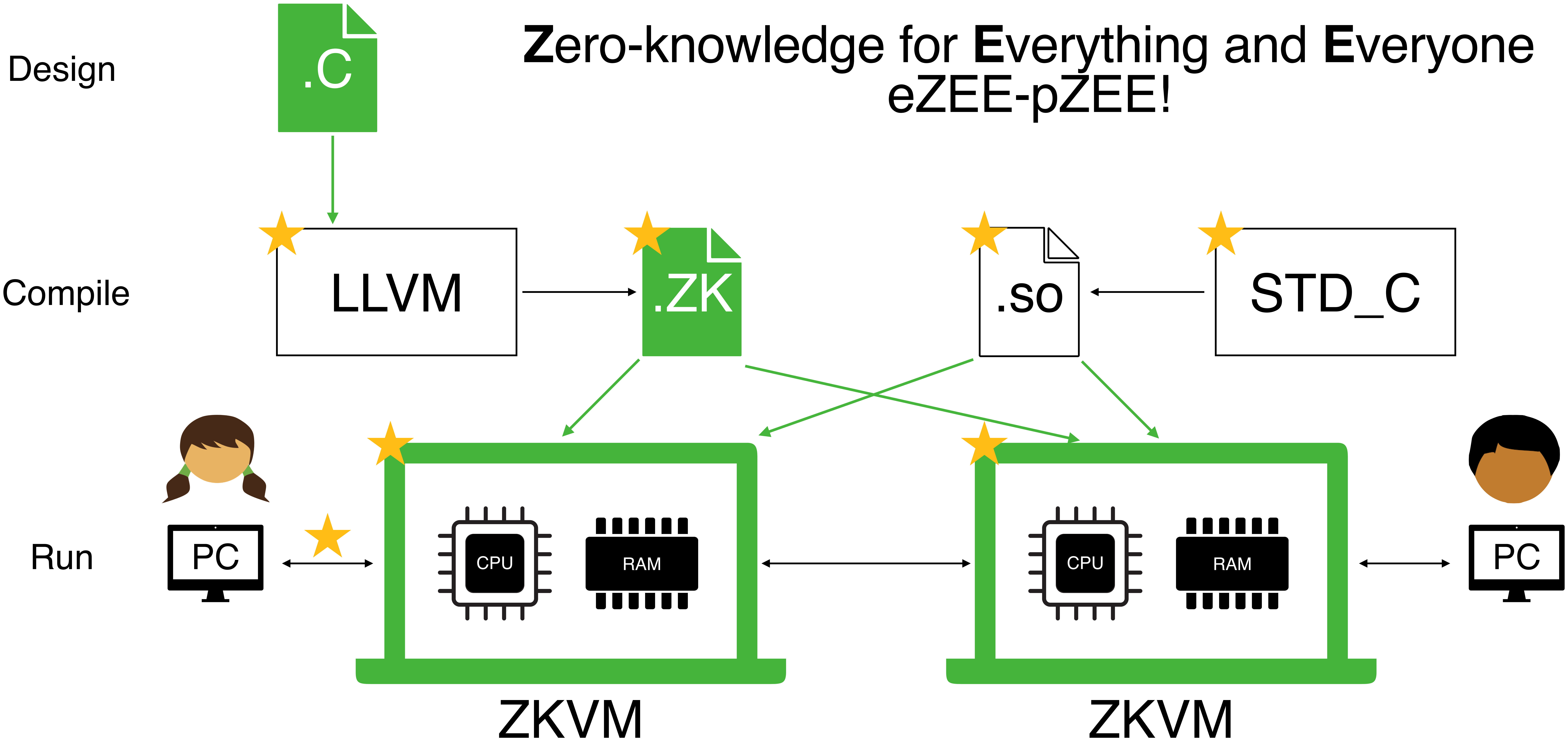
Compile



Zero-knowledge for **E**verything and **E**veryone

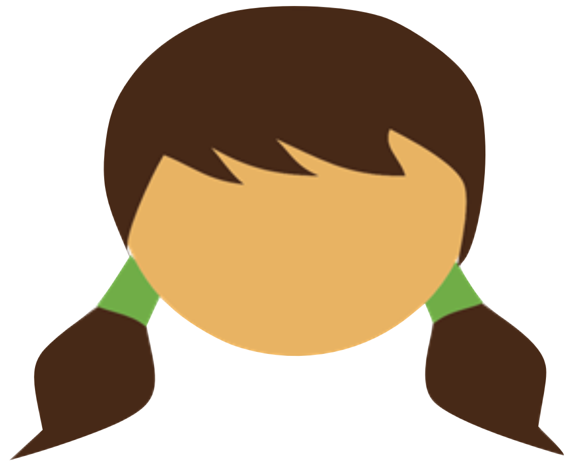


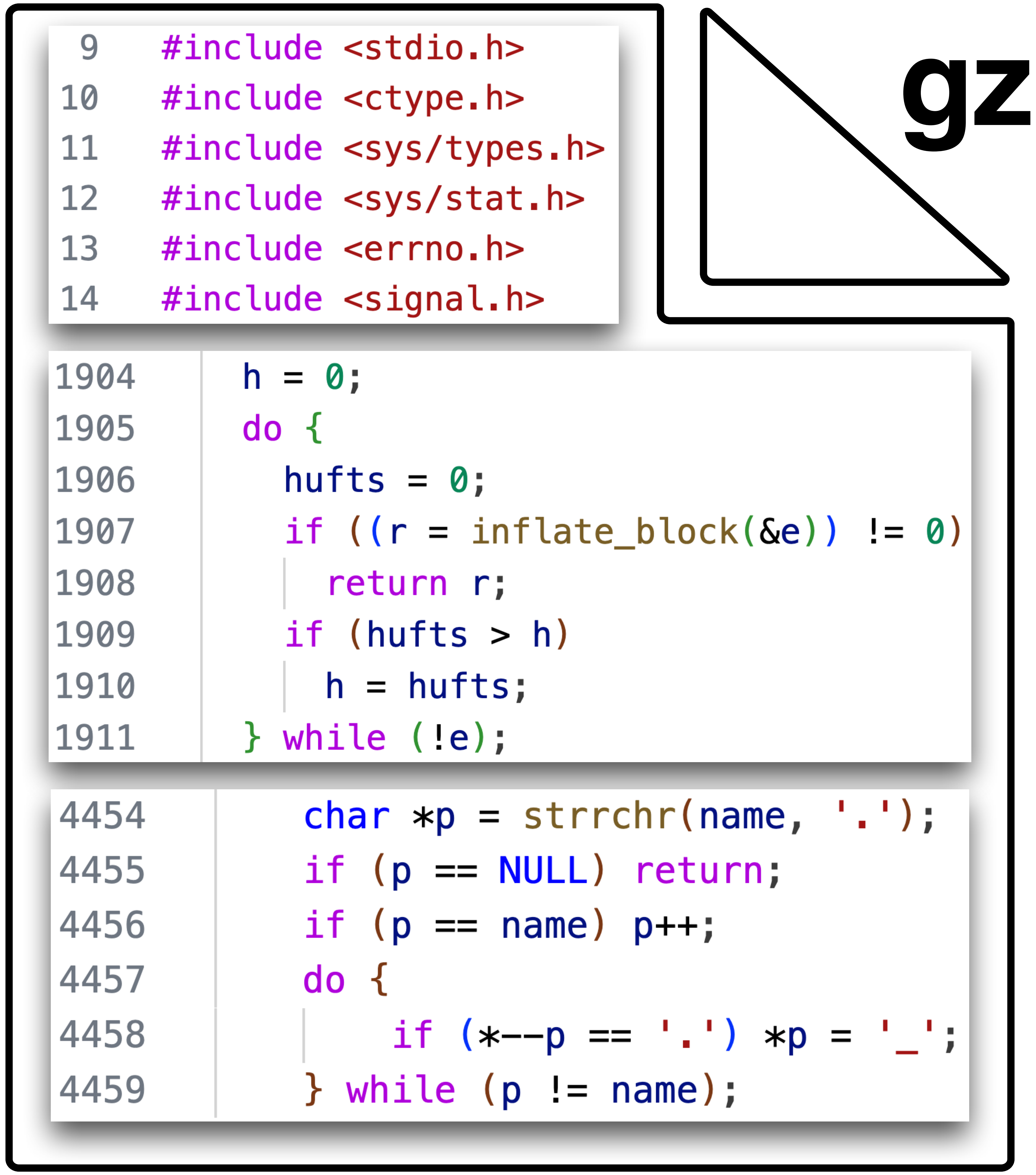
Zero-knowledge for **E**verything and **E**veryone
eZEE-pZEE!



Demo

Demo





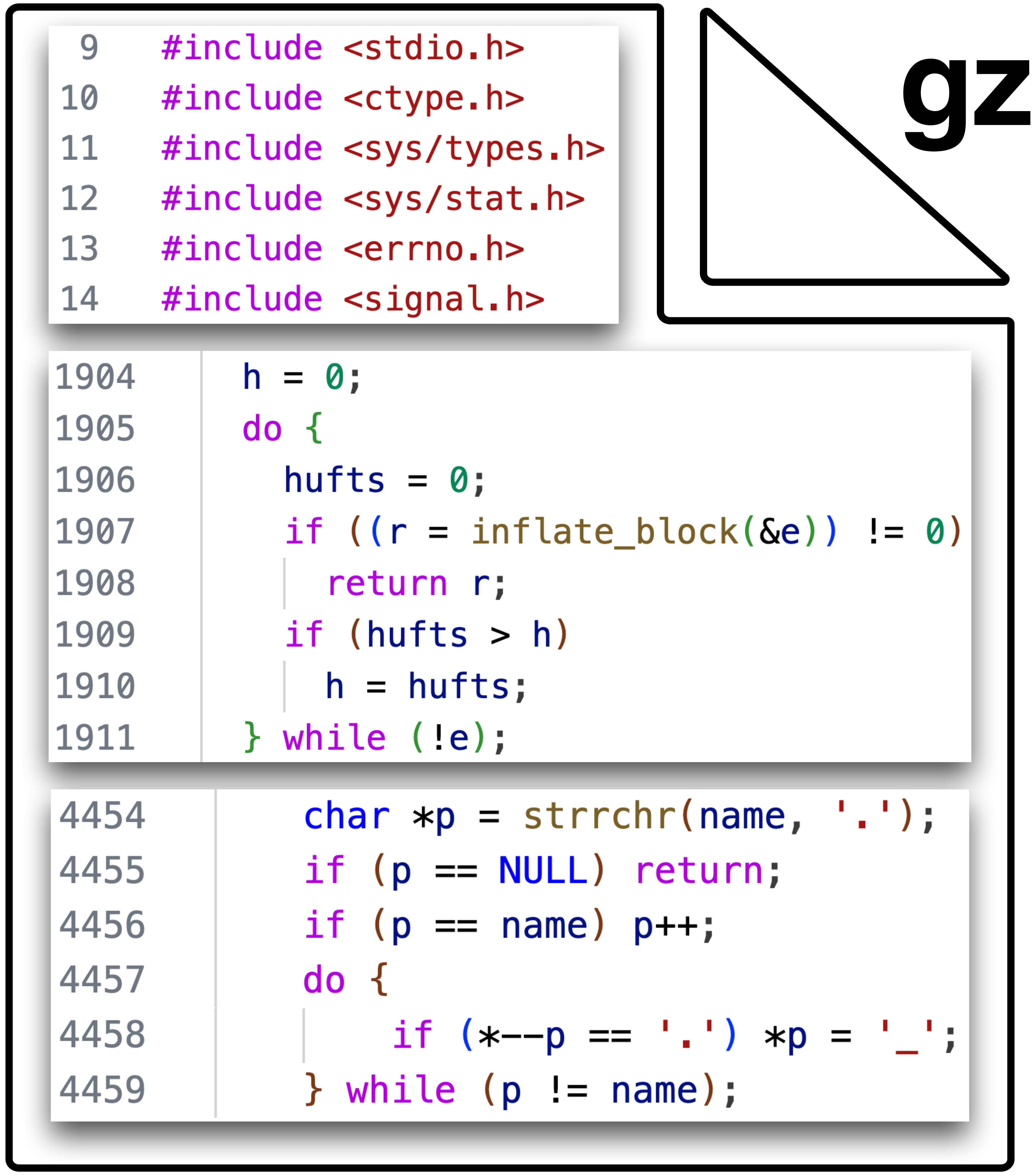
gzip1.3

```c
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <signal.h>
```

```c
h = 0;
do {
  hufts = 0;
  if ((r = inflate_block(&e)) != 0)
    return r;
  if (hufts > h)
    h = hufts;
} while (!e);
```

```c
char *p = strrchr(name, '.');
if (p == NULL) return;
if (p == name) p++;
do {
    if (*--p == '.') *p = '_';
} while (p != name);
```

Demo

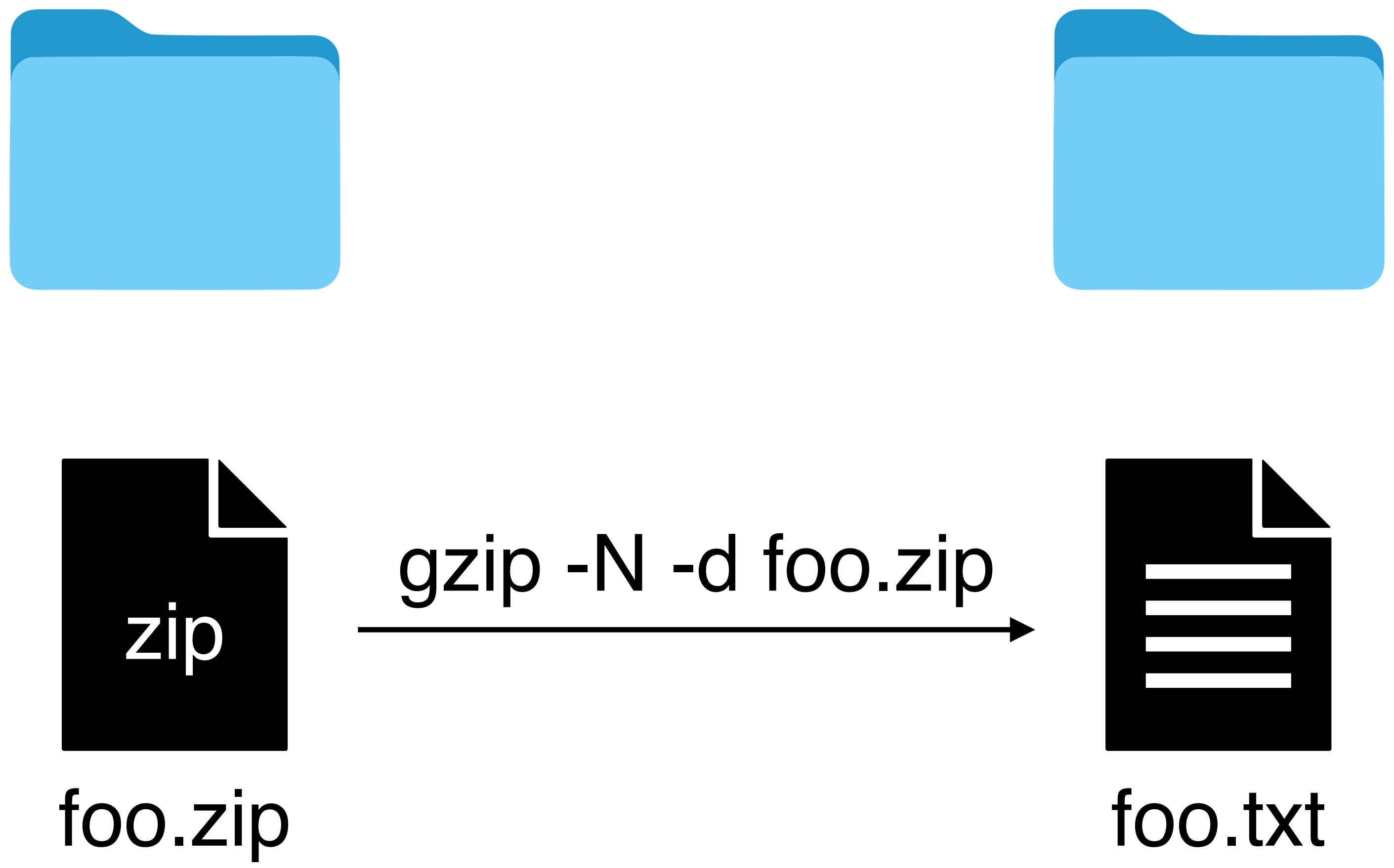


gzip1.3

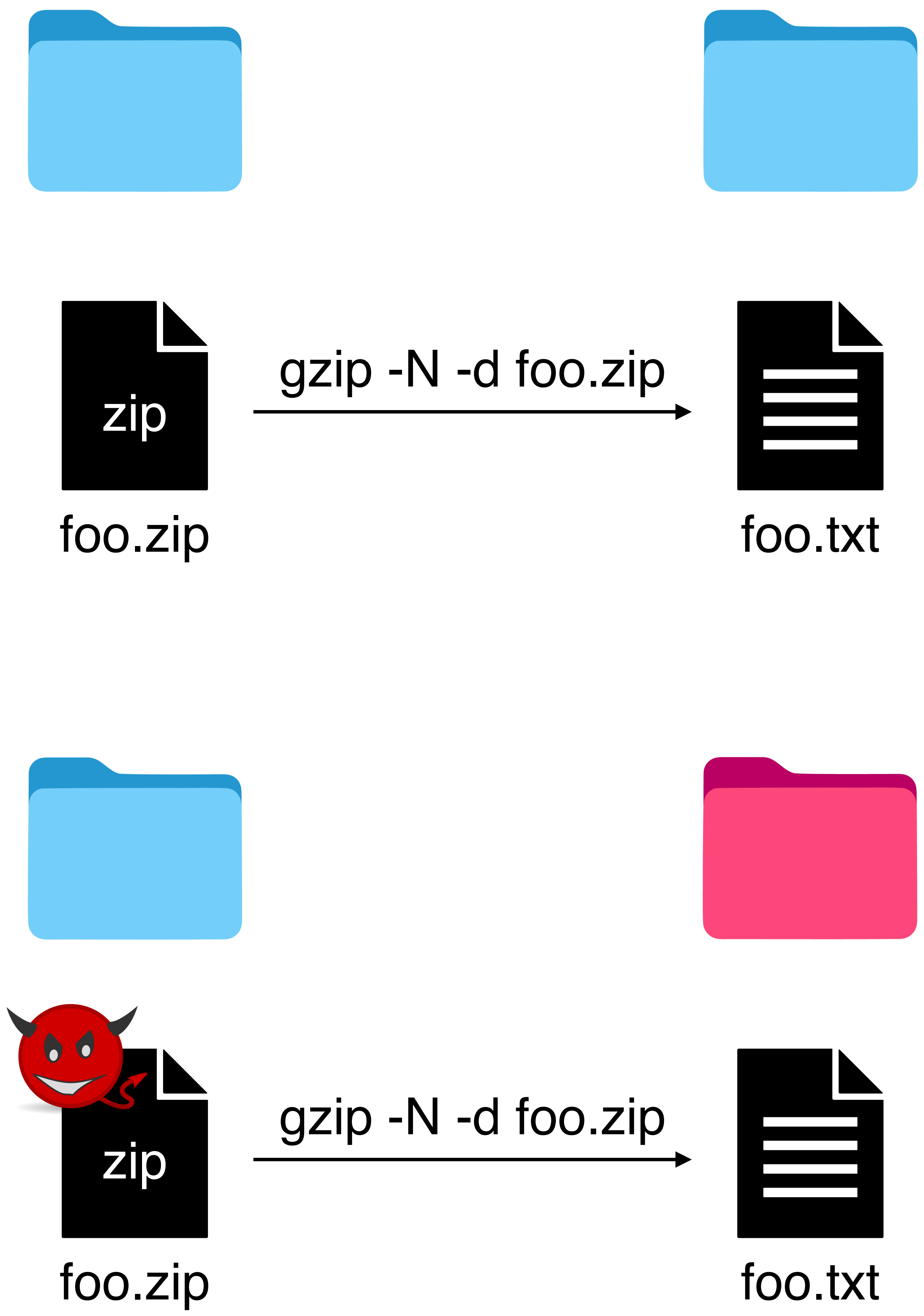
```c
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <signal.h>
```

```c
h = 0;
do {
  hufts = 0;
  if ((r = inflate_block(&e)) != 0)
    return r;
  if (hufts > h)
    h = hufts;
} while (!e);
```

```c
char *p = strrchr(name, '.');
if (p == NULL) return;
if (p == name) p++;
do {
    if (*--p == '.') *p = '_';
} while (p != name);
```



```
gzip -N -d foo.zip
```



zip

foo.zip

gzip -N -d foo.zip

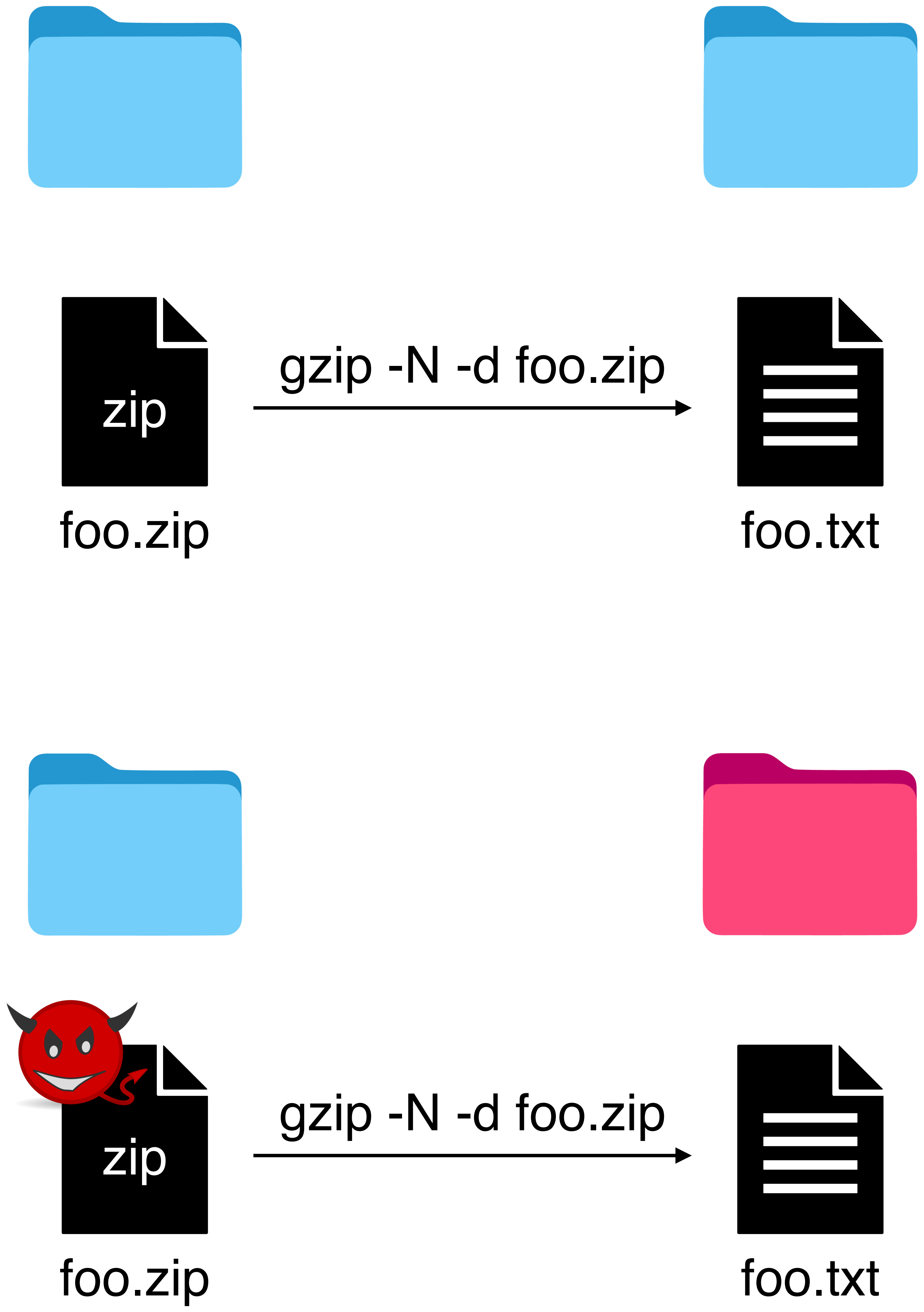
foo.txt

zip

foo.zip

gzip -N -d foo.zip

foo.txt



zip

foo.zip

gzip -N -d foo.zip

foo.txt

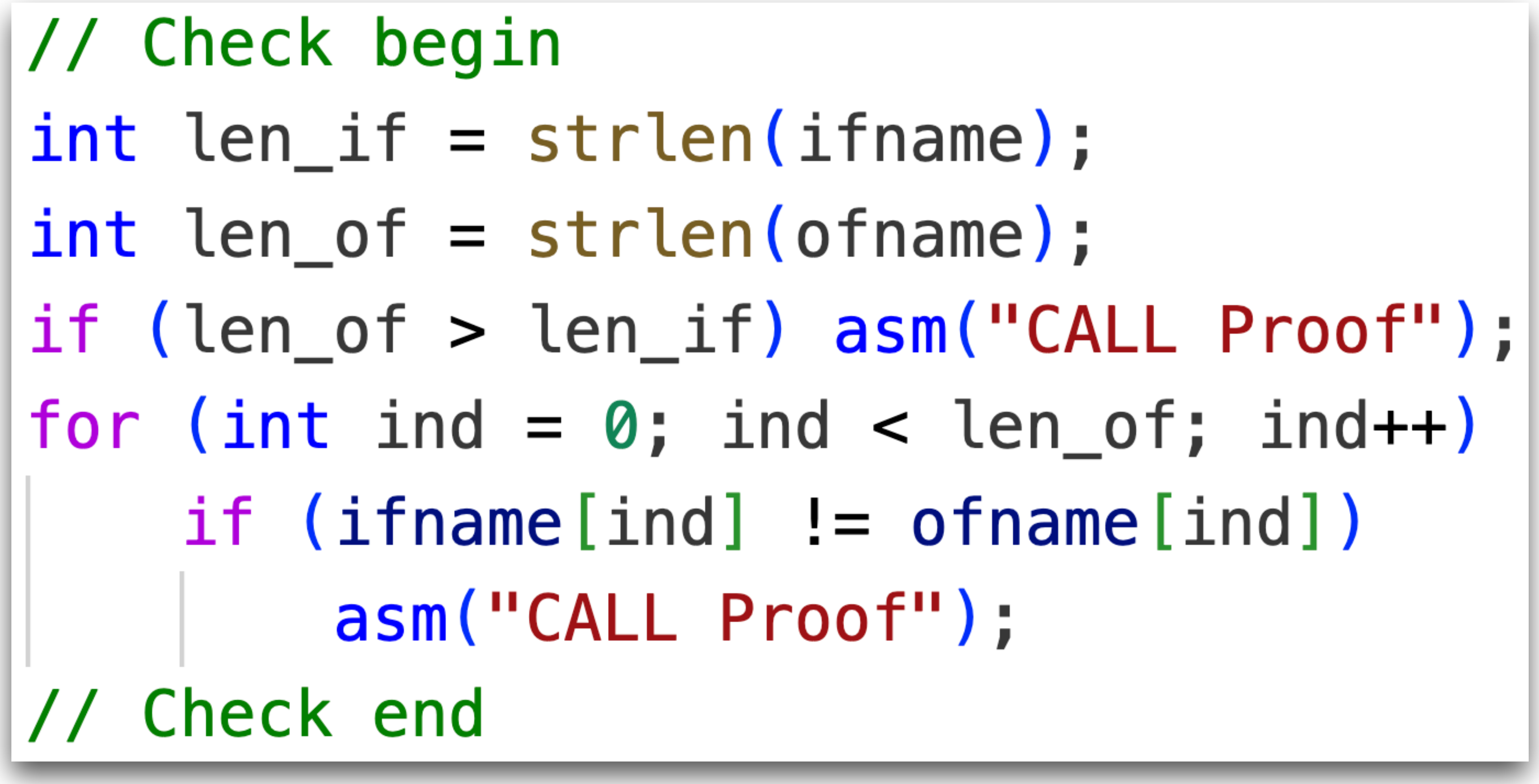
zip

foo.zip

gzip -N -d foo.zip

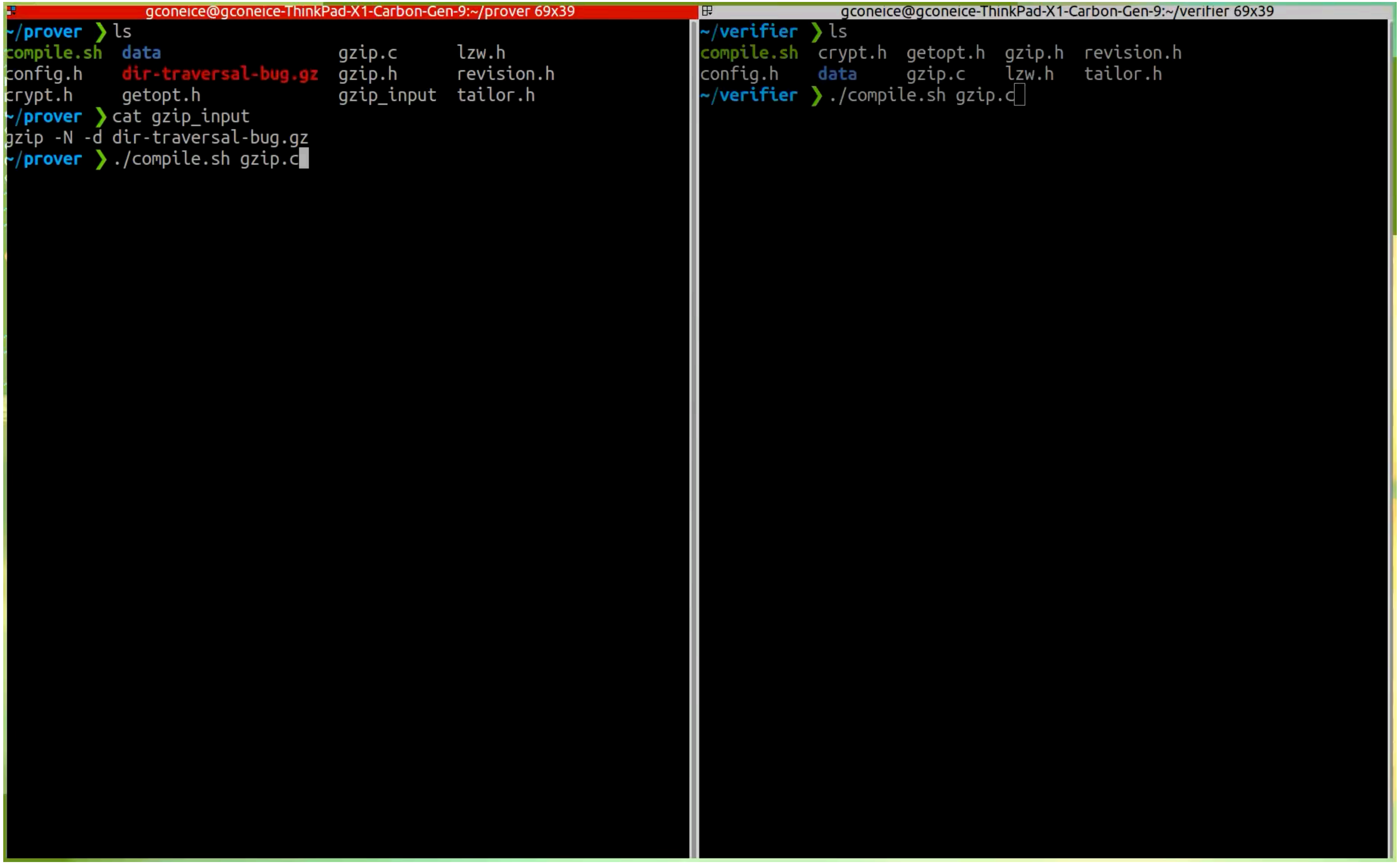
foo.txt

gzip1.3



```
// Check begin
int len_if = strlen(ifname);
int len_of = strlen(ofname);
if (len_of > len_if) asm("CALL Proof");
for (int ind = 0; ind < len_of; ind++)
    if (ifname[ind] != ofname[ind])
        asm("CALL Proof");
// Check end
```

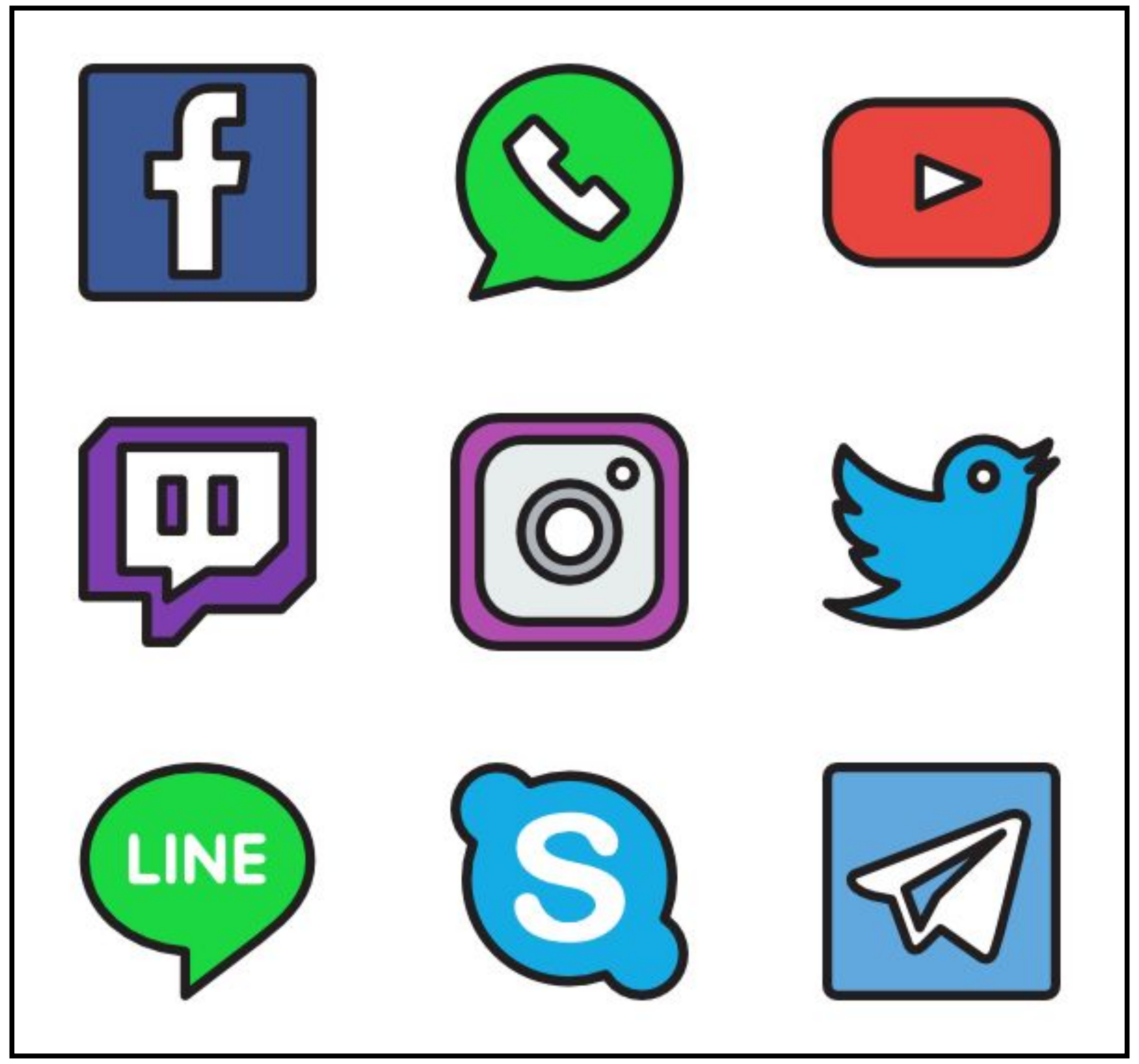
Check that the output path is different


gconeice@gconeice-ThinkPad-X1-Carbon-Gen-9:~/prover 69x39

```
~/prover ❯ ls
compile.sh  data                 gzip.c      lzw.h
config.h    dir-traversal-bug.gz gzip.h      revision.h
crypt.h     getopt.h             gzip_input  tailor.h
~/prover ❯ cat gzip_input
gzip -N -d dir-traversal-bug.gz
~/prover ❯ ./compile.sh gzip.c
```

gconeice@gconeice-ThinkPad-X1-Carbon-Gen-9:~/verifier 69x39

```
~/verifier ❯ ls
compile.sh  crypt.h  getopt.h  gzip.h  revision.h
config.h    data     gzip.c    lzw.h   tailor.h
~/verifier ❯ ./compile.sh gzip.c
```

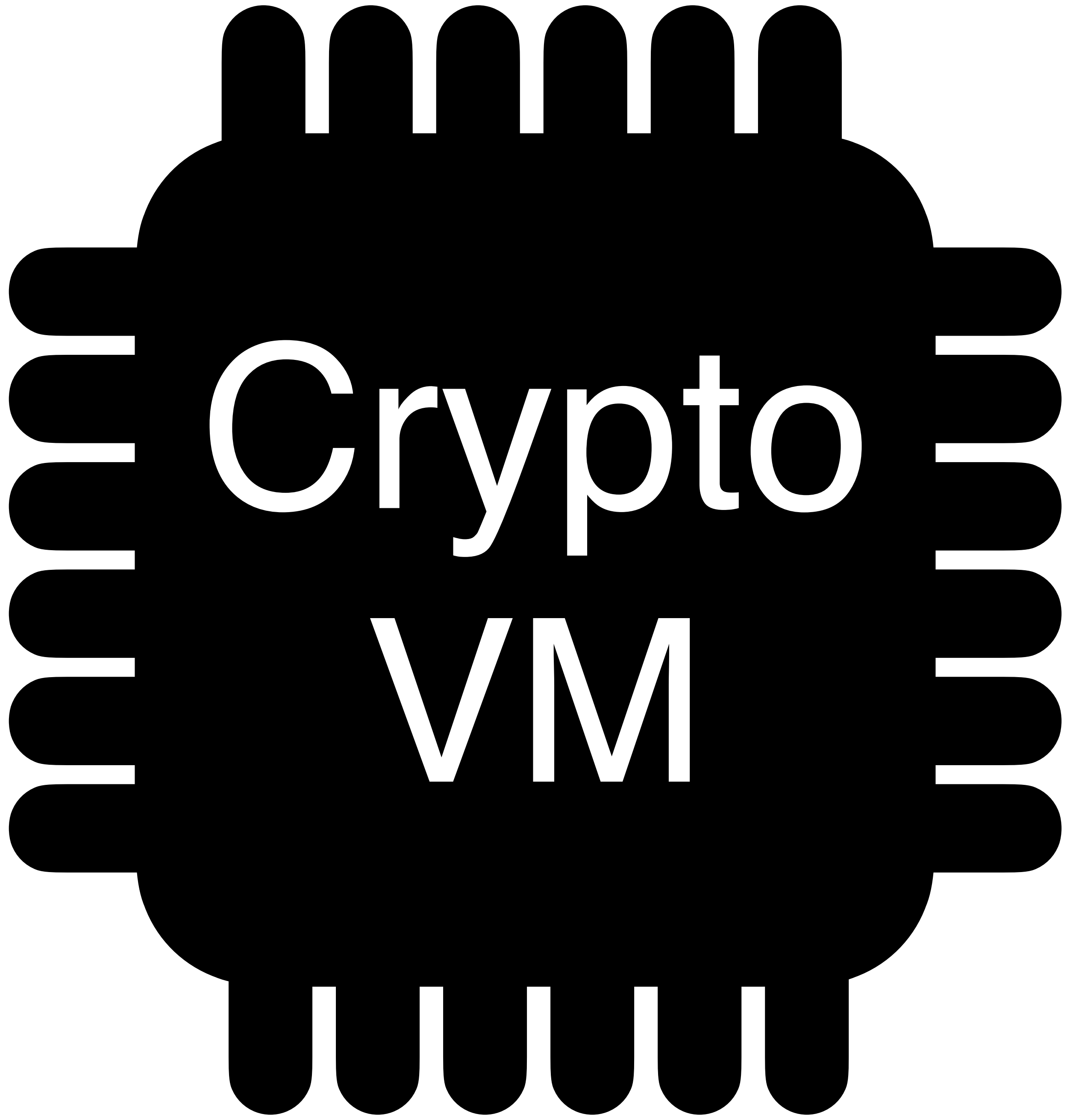


🔒 Secure

⏱ Efficient

Is ≈ 10KHz Fast Enough?

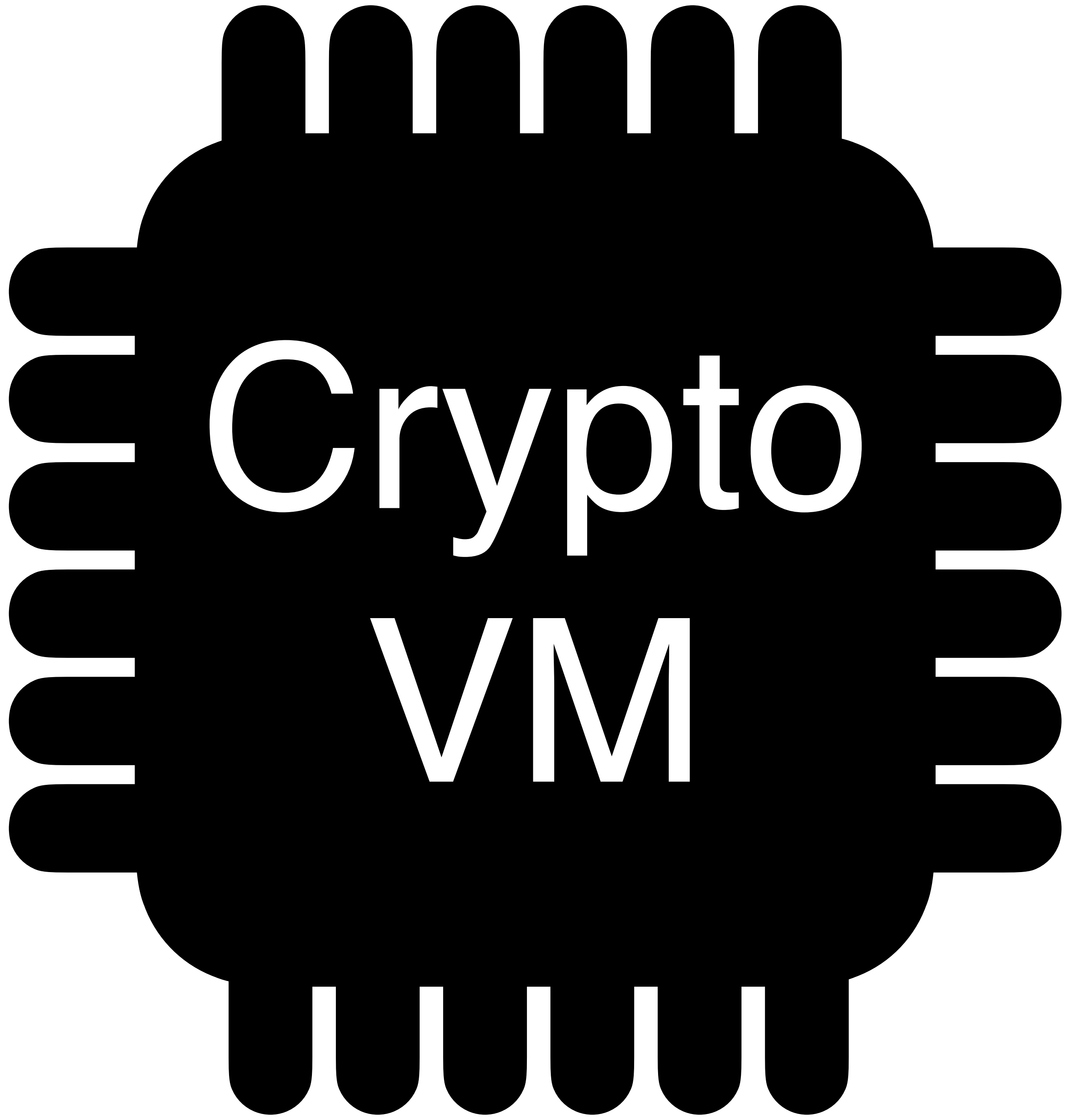
Is ≈ 10KHz Fast Enough?



Crypto VM

v.s.

Is ≈ 10KHz Fast Enough?



Crypto VM

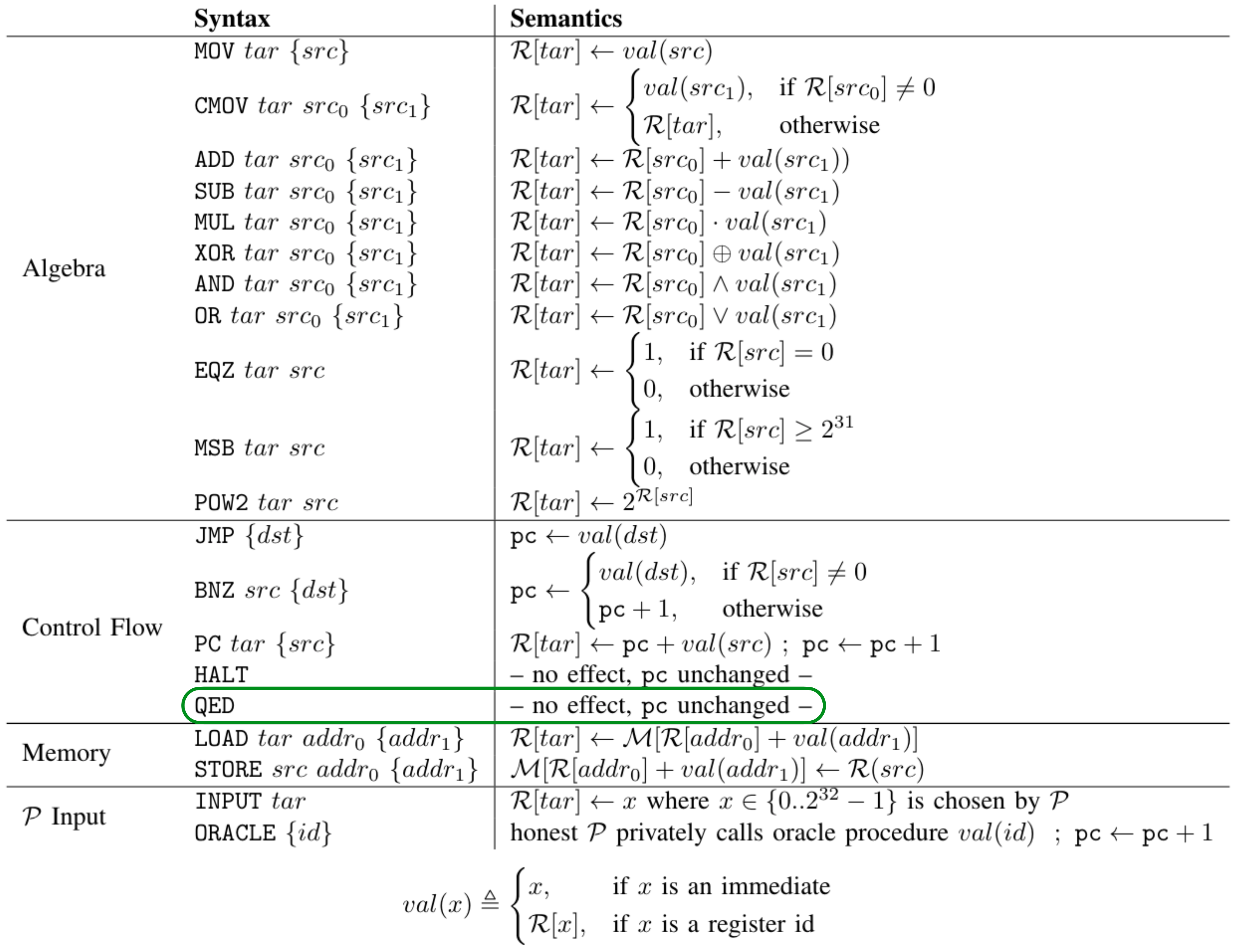
v.s.

We can run Linux programs gzip/sed/bzip, and prove the existence of CVE bugs in <20s

ZK Branching

ZK Branching

We carefully choose the instruction set, resulting in a relatively small CPU “unit” circuit

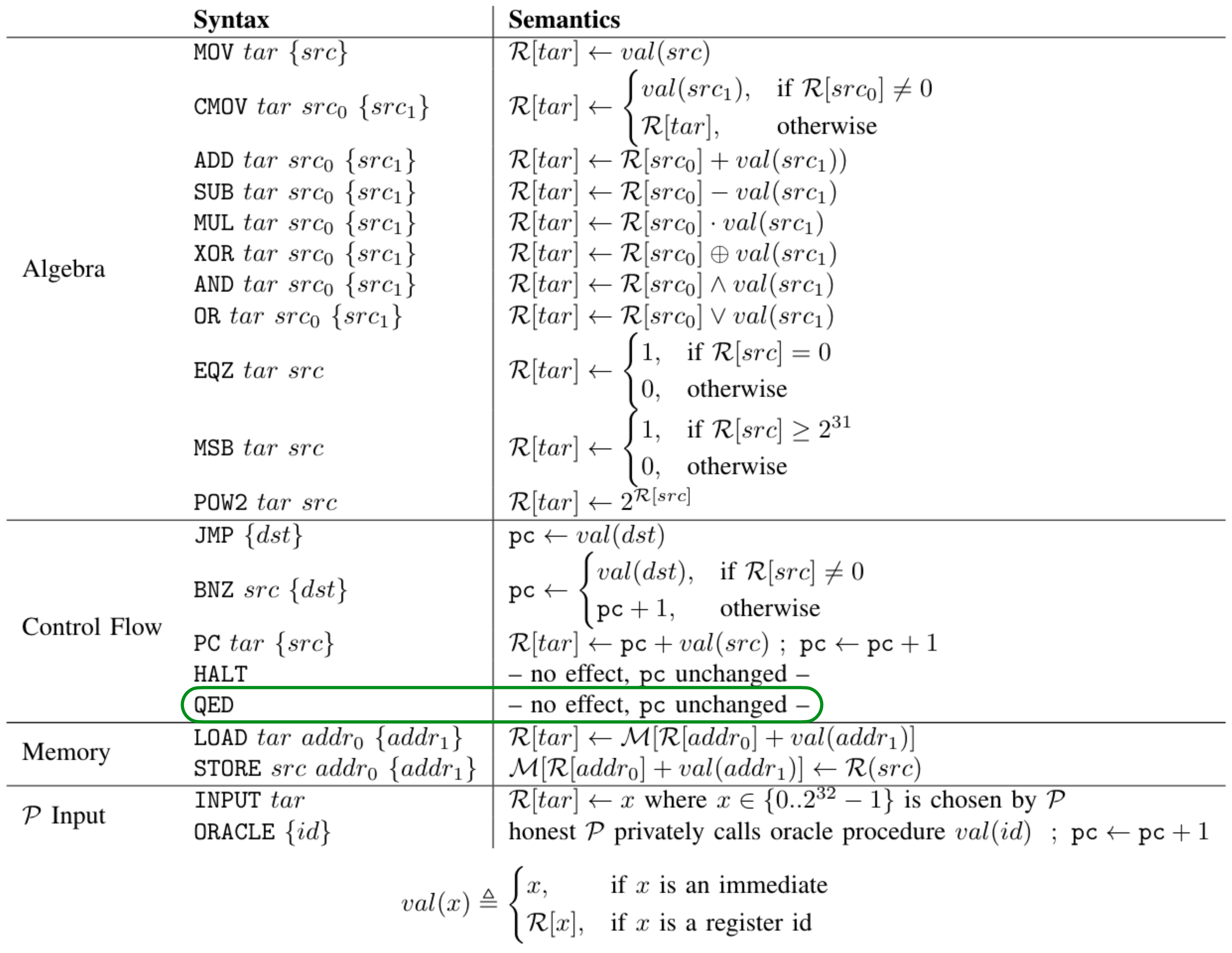


	Syntax	Semantics
Algebra	MOV tar $\{src\}$	$\mathcal{R}[tar] \leftarrow val(src)$
	CMOV tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \begin{cases} val(src_1), & \text{if } \mathcal{R}[src_0] \neq 0 \\ \mathcal{R}[tar], & \text{otherwise} \end{cases}$
	ADD tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] + val(src_1))$
	SUB tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] - val(src_1)$
	MUL tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \cdot val(src_1)$
	XOR tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \oplus val(src_1)$
	AND tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \wedge val(src_1)$
	OR tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \vee val(src_1)$
	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 $\{dst\}$	$\texttt{pc} \leftarrow val(dst)$
	BNZ src $\{dst\}$	$\texttt{pc} \leftarrow \begin{cases} val(dst), & \text{if } \mathcal{R}[src] \neq 0 \\ \texttt{pc} + 1, & \text{otherwise} \end{cases}$
	PC tar $\{src\}$	$\mathcal{R}[tar] \leftarrow \texttt{pc} + val(src)$; $\texttt{pc} \leftarrow \texttt{pc} + 1$
	HALT	– no effect, pc unchanged –
	QED	– no effect, pc unchanged –
Memory	LOAD tar $addr_0$ $\{addr_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{M}[\mathcal{R}[addr_0] + val(addr_1)]$
	STORE src $addr_0$ $\{addr_1\}$	$\mathcal{M}[\mathcal{R}[addr_0] + val(addr_1)] \leftarrow \mathcal{R}(src)$
$\mathcal{P}$ 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 $val(id)$; $\texttt{pc} \leftarrow \texttt{pc} + 1$

$$val(x) \triangleq \begin{cases} x, & \text{if } x \text{ is an immediate} \\ \mathcal{R}[x], & \text{if } x \text{ is a register id} \end{cases}$$

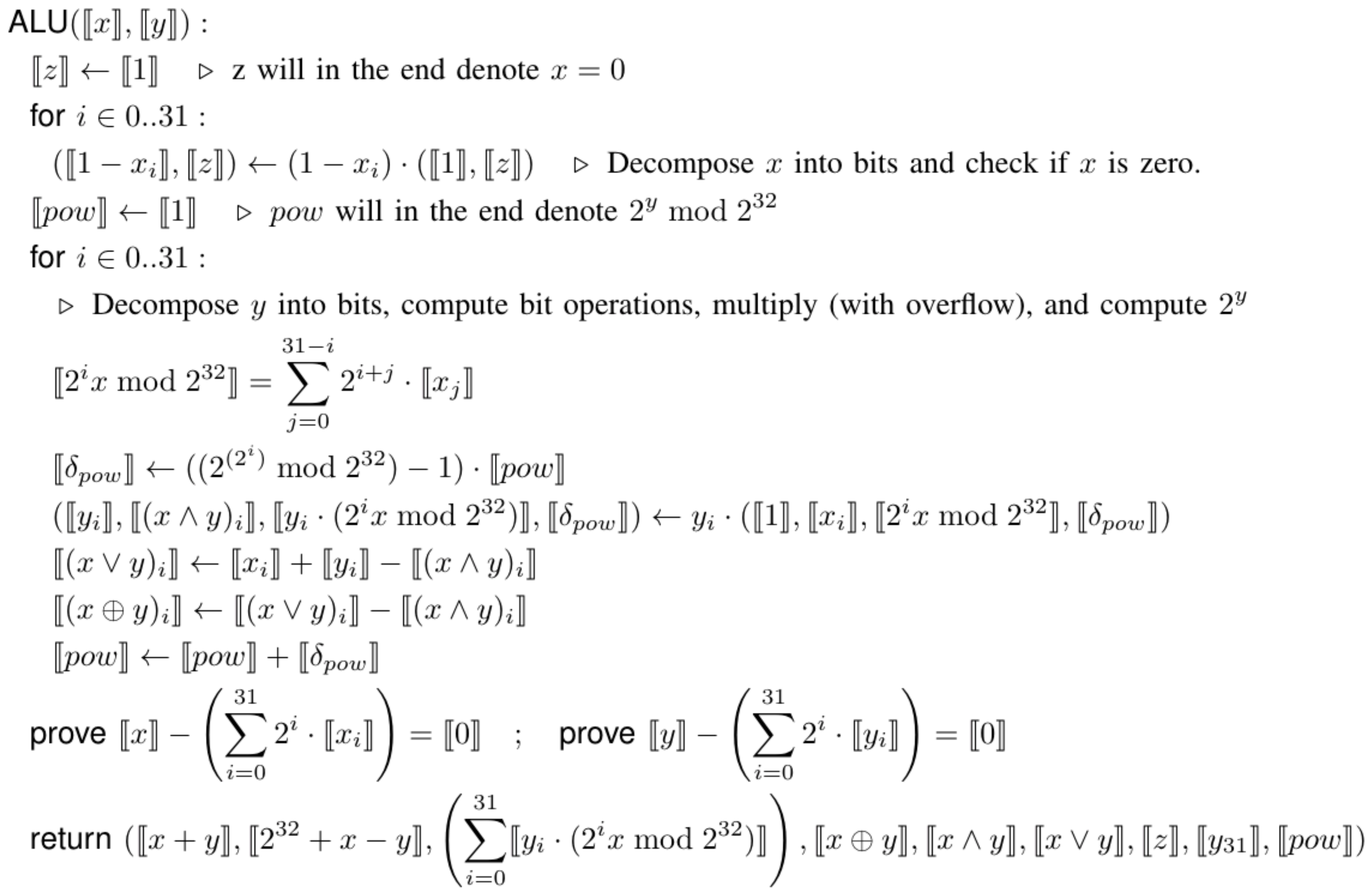
ZK Branching

We carefully choose the instruction set, resulting in a relatively small CPU “unit” circuit



	Syntax	Semantics
Algebra	`MOV` $tar\ \{src\}$	$\mathcal{R}[tar] \leftarrow val(src)$
	`CMOV` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \begin{cases} val(src_1), & \text{if } \mathcal{R}[src_0] \neq 0 \\ \mathcal{R}[tar], & \text{otherwise} \end{cases}$
	`ADD` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] + val(src_1))$
	`SUB` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] - val(src_1)$
	`MUL` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \cdot val(src_1)$
	`XOR` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \oplus val(src_1)$
	`AND` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \wedge val(src_1)$
	`OR` $tar\ src_0\ \{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \vee val(src_1)$
	`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` $\{dst\}$	$\texttt{pc} \leftarrow val(dst)$
	`BNZ` $src\ \{dst\}$	$\texttt{pc} \leftarrow \begin{cases} val(dst), & \text{if } \mathcal{R}[src] \neq 0 \\ \texttt{pc}+1, & \text{otherwise} \end{cases}$
	`PC` $tar\ \{src\}$	$\mathcal{R}[tar] \leftarrow \texttt{pc} + val(src)\ ;\ \texttt{pc} \leftarrow \texttt{pc}+1$
	`HALT`	– no effect, `pc` unchanged –
	`QED`	– no effect, `pc` unchanged –
Memory	`LOAD` $tar\ addr_0\ \{addr_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{M}[\mathcal{R}[addr_0] + val(addr_1)]$
	`STORE` $src\ addr_0\ \{addr_1\}$	$\mathcal{M}[\mathcal{R}[addr_0] + val(addr_1)] \leftarrow \mathcal{R}(src)$
$\mathcal{P}$ 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 $val(id)\ ;\ \texttt{pc} \leftarrow \texttt{pc}+1$

$$val(x) \triangleq \begin{cases} x, & \text{if } x \text{ is an immediate} \\ \mathcal{R}[x], & \text{if } x \text{ is a register id} \end{cases}$$



```
ALU($\llbracket x \rrbracket, \llbracket y \rrbracket$) :
  $\llbracket z \rrbracket \leftarrow \llbracket 1 \rrbracket$    ▷ z will in the end denote $x = 0$
  for $i \in 0..31$ :
    $(\llbracket 1 - x_i \rrbracket, \llbracket z \rrbracket) \leftarrow (1 - x_i) \cdot (\llbracket 1 \rrbracket, \llbracket z \rrbracket)$    ▷ Decompose $x$ into bits and check if $x$ is zero.
  $\llbracket pow \rrbracket \leftarrow \llbracket 1 \rrbracket$    ▷ $pow$ will in the end denote $2^y \bmod 2^{32}$
  for $i \in 0..31$ :
    ▷ Decompose $y$ into bits, compute bit operations, multiply (with overflow), and compute $2^y$
    $\llbracket 2^i x \bmod 2^{32} \rrbracket = \sum_{j=0}^{31-i} 2^{i+j} \cdot \llbracket x_j \rrbracket$
    $\llbracket \delta_{pow} \rrbracket \leftarrow ((2^{(2^i)} \bmod 2^{32}) - 1) \cdot \llbracket pow \rrbracket$
    $(\llbracket y_i \rrbracket, \llbracket (x \wedge y)_i \rrbracket, \llbracket y_i \cdot (2^i x \bmod 2^{32}) \rrbracket, \llbracket \delta_{pow} \rrbracket) \leftarrow y_i \cdot (\llbracket 1 \rrbracket, \llbracket x_i \rrbracket, \llbracket 2^i x \bmod 2^{32} \rrbracket, \llbracket \delta_{pow} \rrbracket)$
    $\llbracket (x \vee y)_i \rrbracket \leftarrow \llbracket x_i \rrbracket + \llbracket y_i \rrbracket - \llbracket (x \wedge y)_i \rrbracket$
    $\llbracket (x \oplus y)_i \rrbracket \leftarrow \llbracket (x \vee y)_i \rrbracket - \llbracket (x \wedge y)_i \rrbracket$
    $\llbracket pow \rrbracket \leftarrow \llbracket pow \rrbracket + \llbracket \delta_{pow} \rrbracket$
  prove $\llbracket x \rrbracket - \left( \sum_{i=0}^{31} 2^i \cdot \llbracket x_i \rrbracket \right) = \llbracket 0 \rrbracket$   ;   prove $\llbracket y \rrbracket - \left( \sum_{i=0}^{31} 2^i \cdot \llbracket y_i \rrbracket \right) = \llbracket 0 \rrbracket$
  return $(\llbracket x + y \rrbracket, \llbracket 2^{32} + x - y \rrbracket, \left( \sum_{i=0}^{31} \llbracket y_i \cdot (2^i x \bmod 2^{32}) \rrbracket \right), \llbracket x \oplus y \rrbracket, \llbracket x \wedge y \rrbracket, \llbracket x \vee y \rrbracket, \llbracket z \rrbracket, \llbracket y_{31} \rrbracket, \llbracket pow \rrbracket)$
```

ZK Memory

ZK Memory

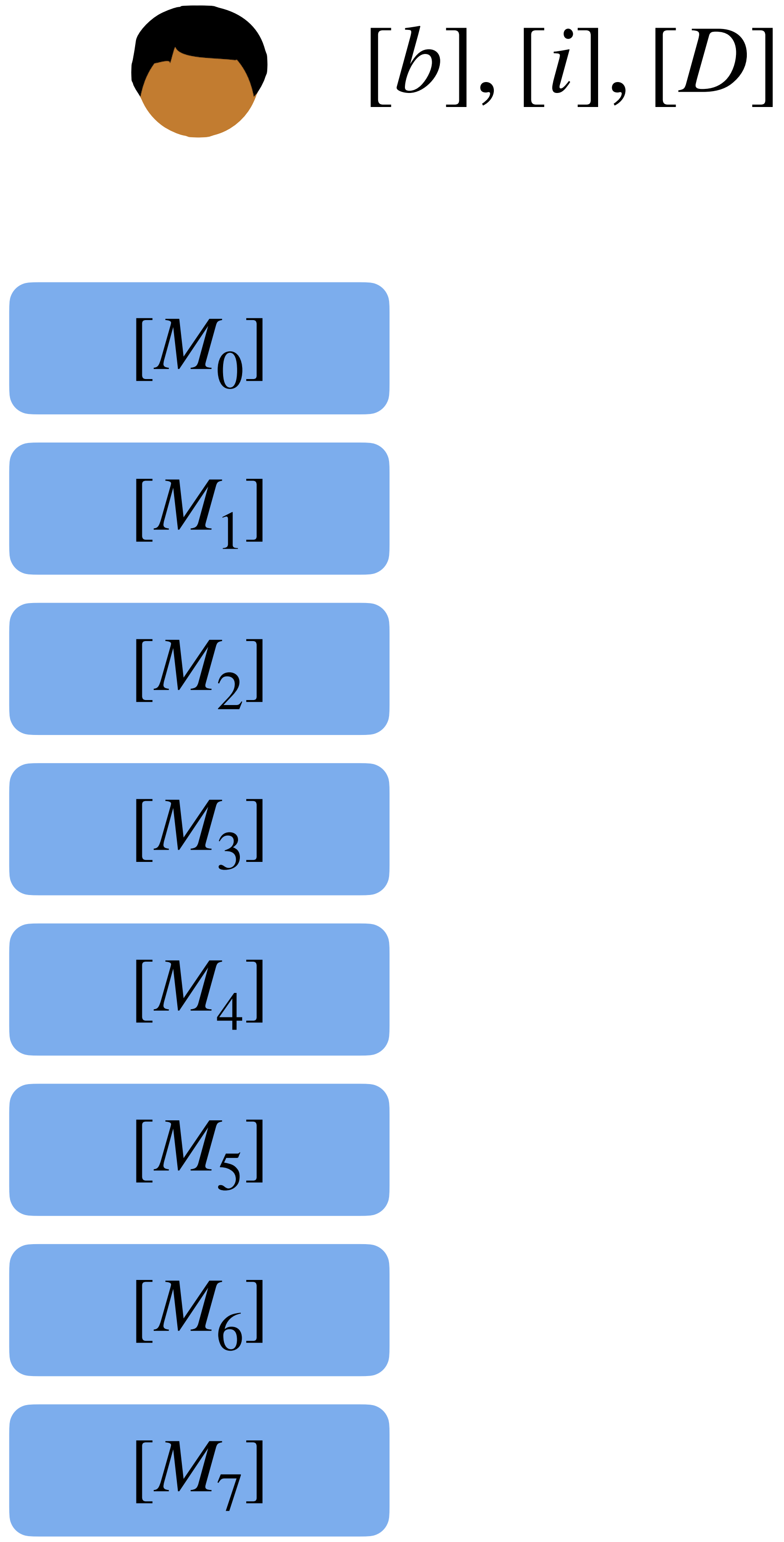
Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b], [i], [D]$

$[M_0]$

$[M_1]$

$[M_2]$

$[M_3]$

$[M_4]$

$[M_5]$

$[M_6]$

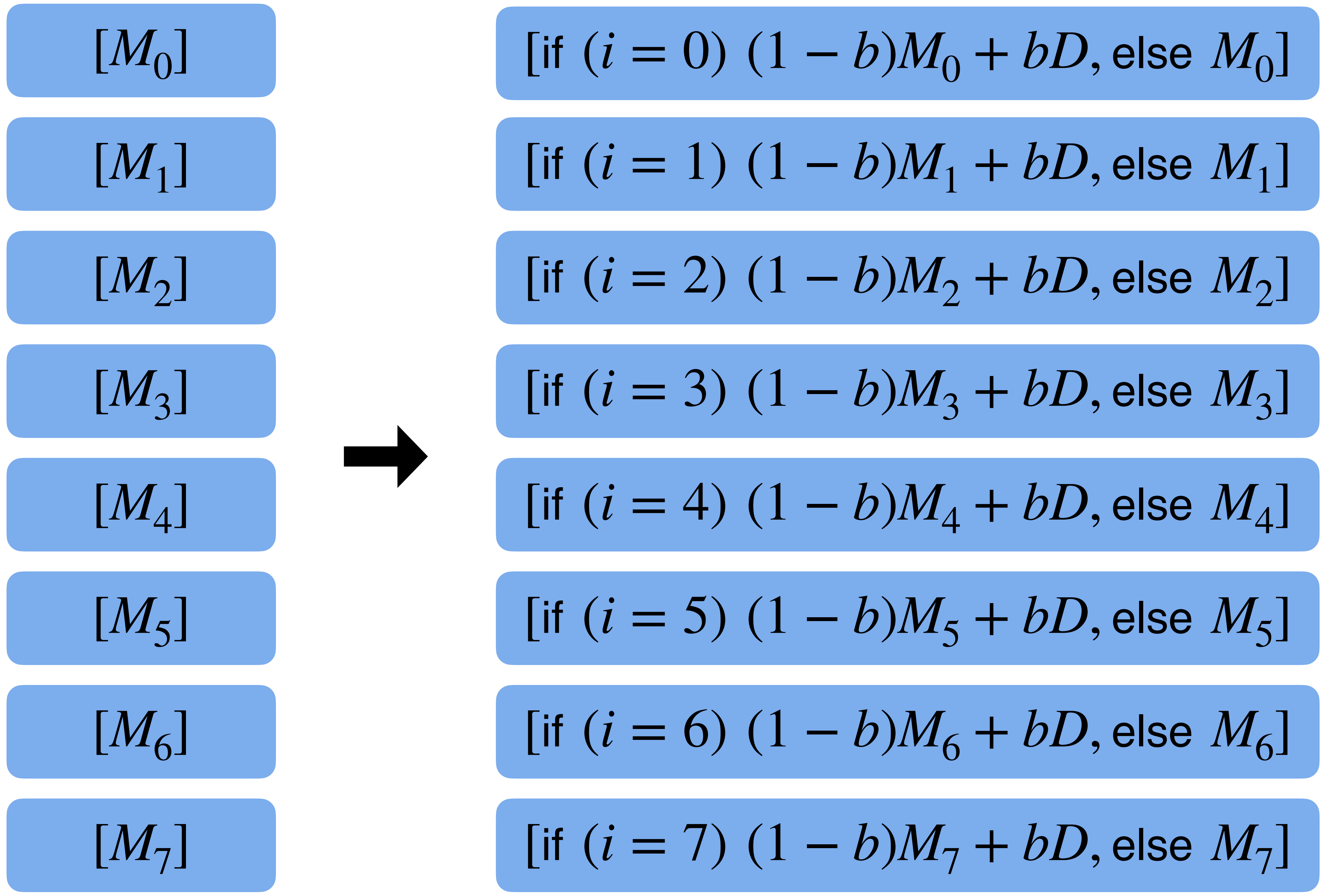
$[M_7]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access

$[b], [i], [D]$ → $[M_i]$



$[M_0]$

$[M_1]$

$[M_2]$

$[M_3]$

$[M_4]$

$[M_5]$

$[M_6]$

$[M_7]$

→

[if $(i = 0)$ $(1-b)M_0 + bD$, else M_0]

[if $(i = 1)$ $(1-b)M_1 + bD$, else M_1]

[if $(i = 2)$ $(1-b)M_2 + bD$, else M_2]

[if $(i = 3)$ $(1-b)M_3 + bD$, else M_3]

[if $(i = 4)$ $(1-b)M_4 + bD$, else M_4]

[if $(i = 5)$ $(1-b)M_5 + bD$, else M_5]

[if $(i = 6)$ $(1-b)M_6 + bD$, else M_6]

[if $(i = 7)$ $(1-b)M_7 + bD$, else M_7]

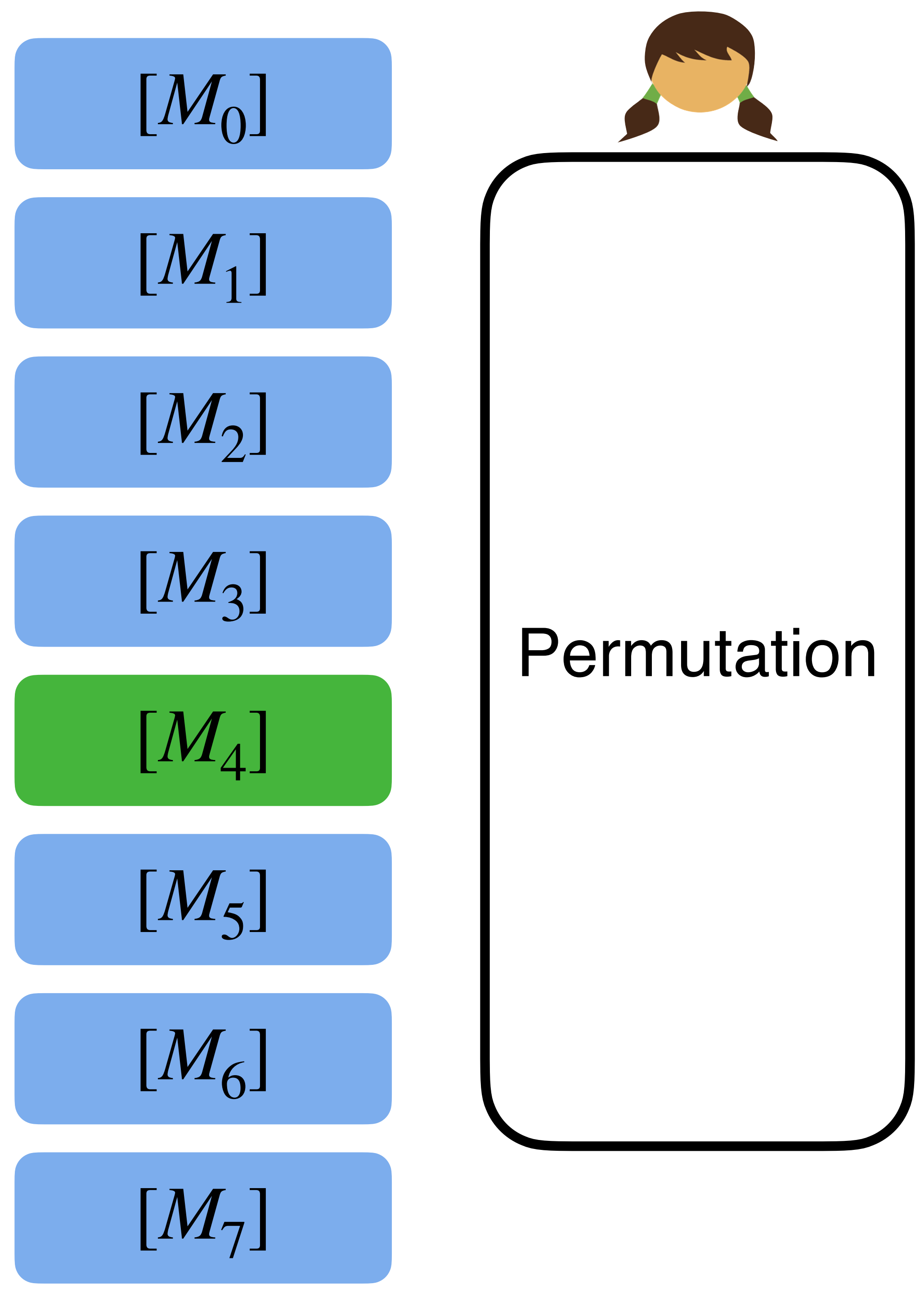
ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access

$[b], [4], [D]$

Tech. 2: P knows and helps



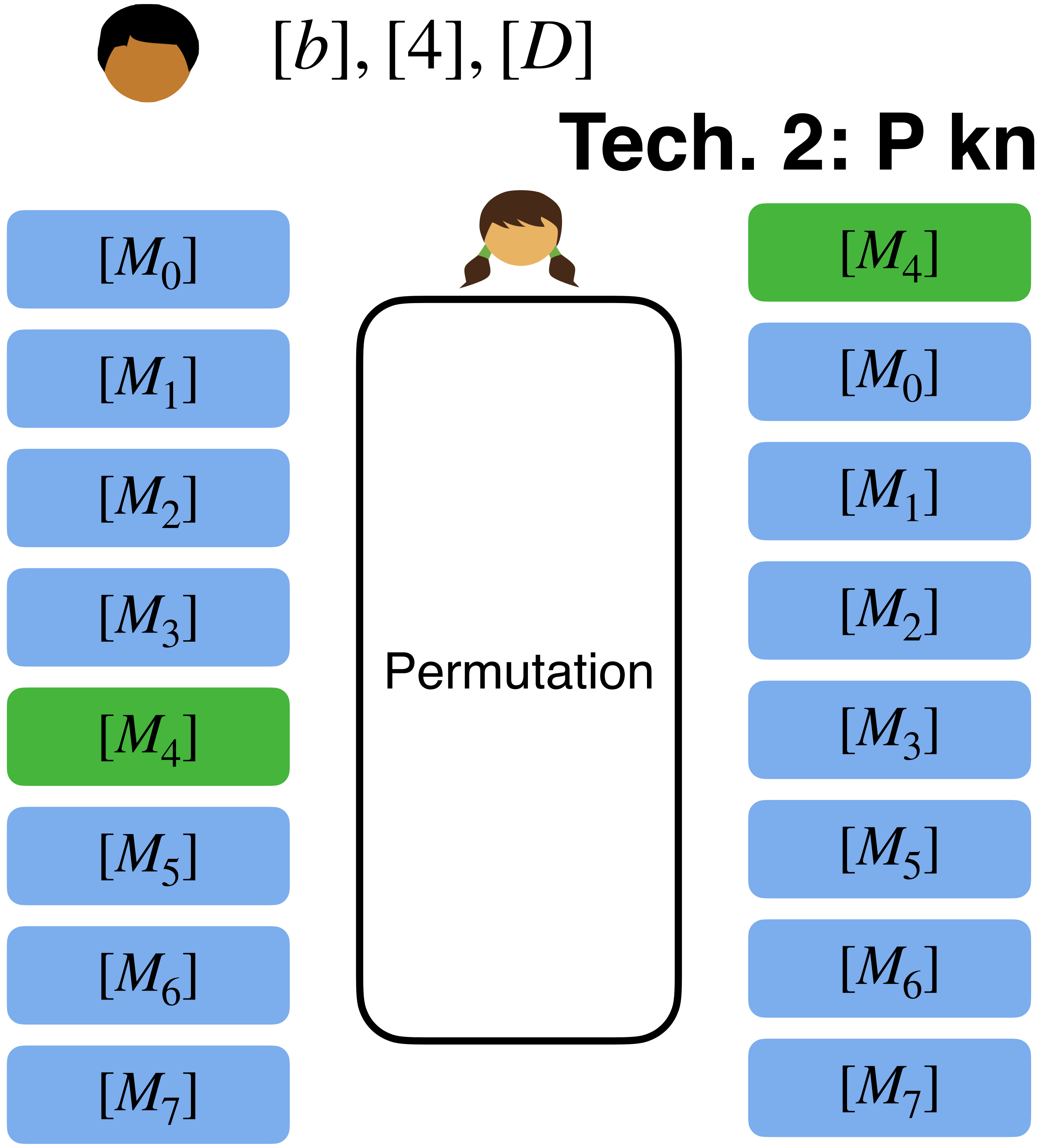
ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access

$[b], [4], [D]$

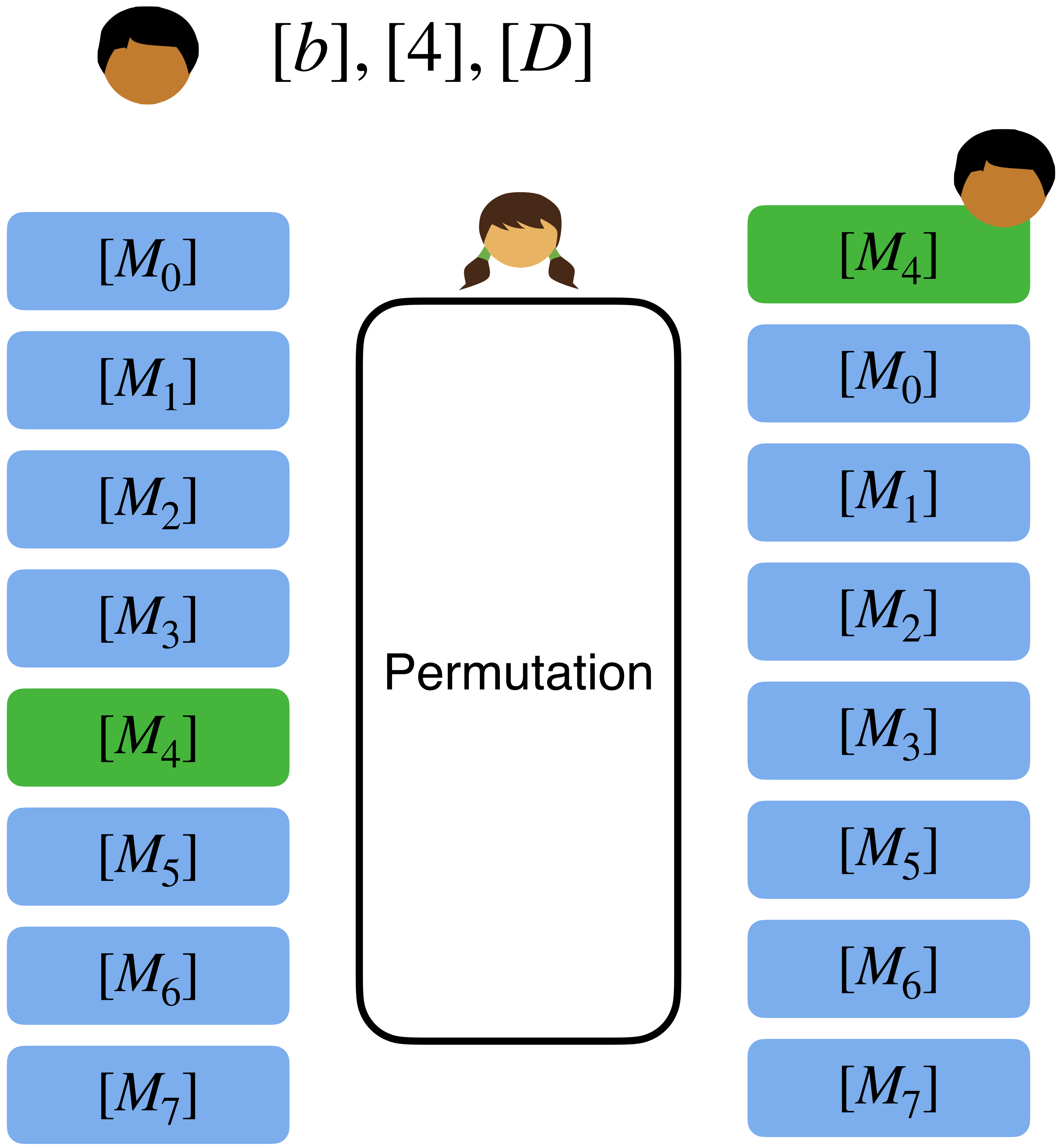
Tech. 2: P knows and helps



ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b], [4], [D]$

$[M_0]$ $[M_1]$ $[M_2]$ $[M_3]$ $[M_4]$ $[M_5]$ $[M_6]$ $[M_7]$

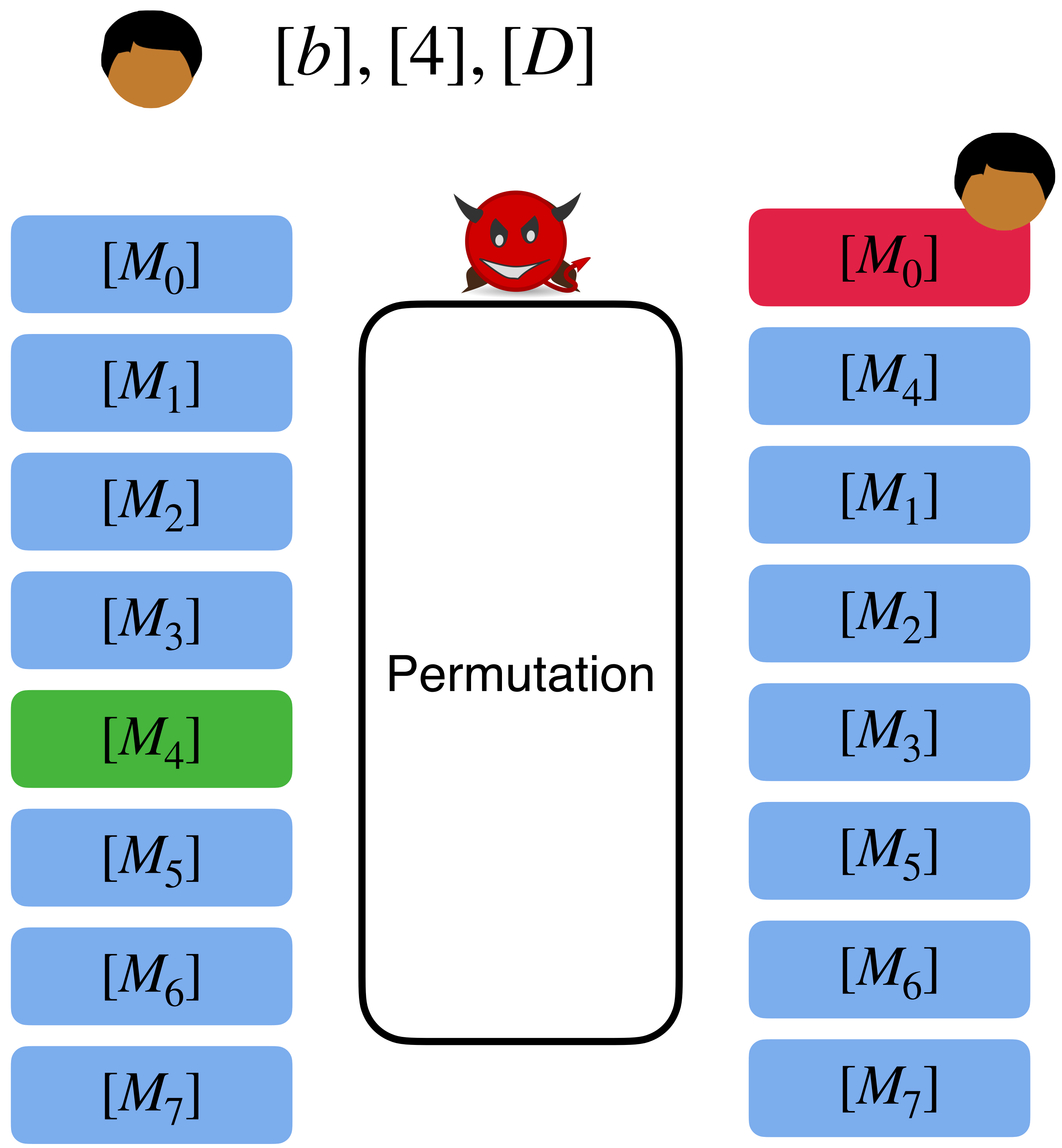
Permutation

$[M_4]$ $[M_0]$ $[M_1]$ $[M_2]$ $[M_3]$ $[M_5]$ $[M_6]$ $[M_7]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

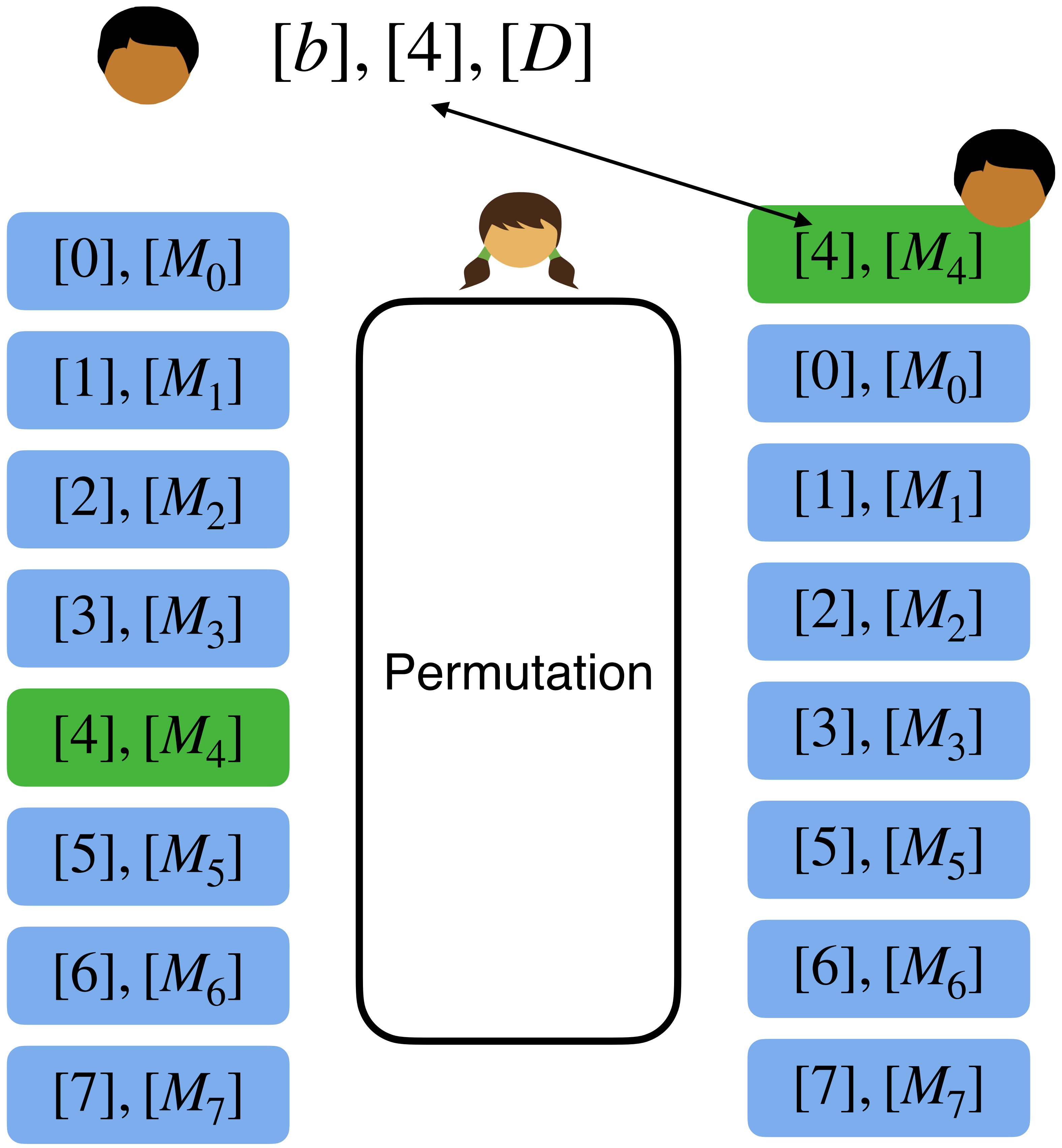
$O(N) \Rightarrow O(\log N)$ per access



ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b], [4], [D]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation

$[4], [M_4]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

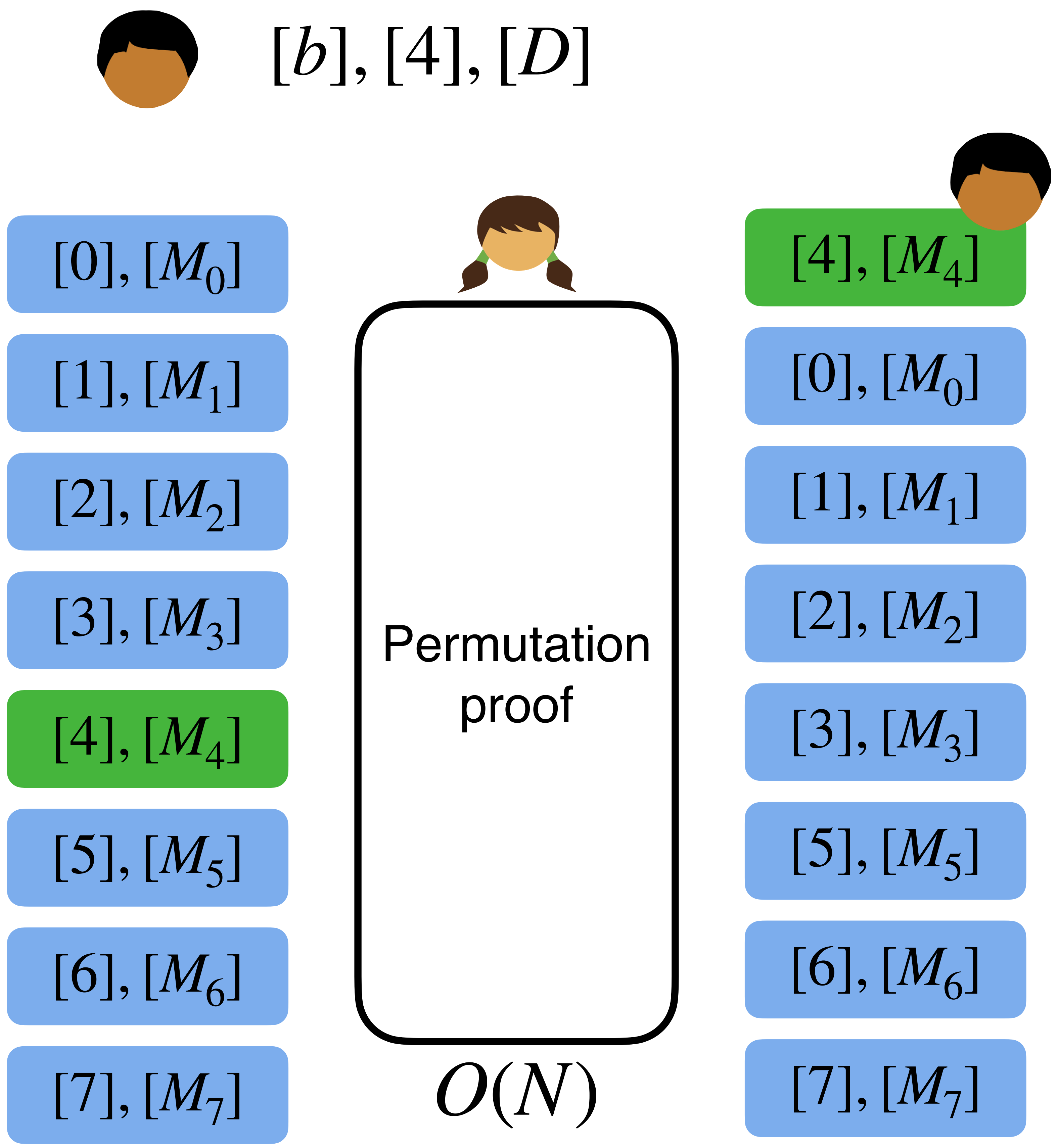
$[6], [M_6]$

$[7], [M_7]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b], [4], [D]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

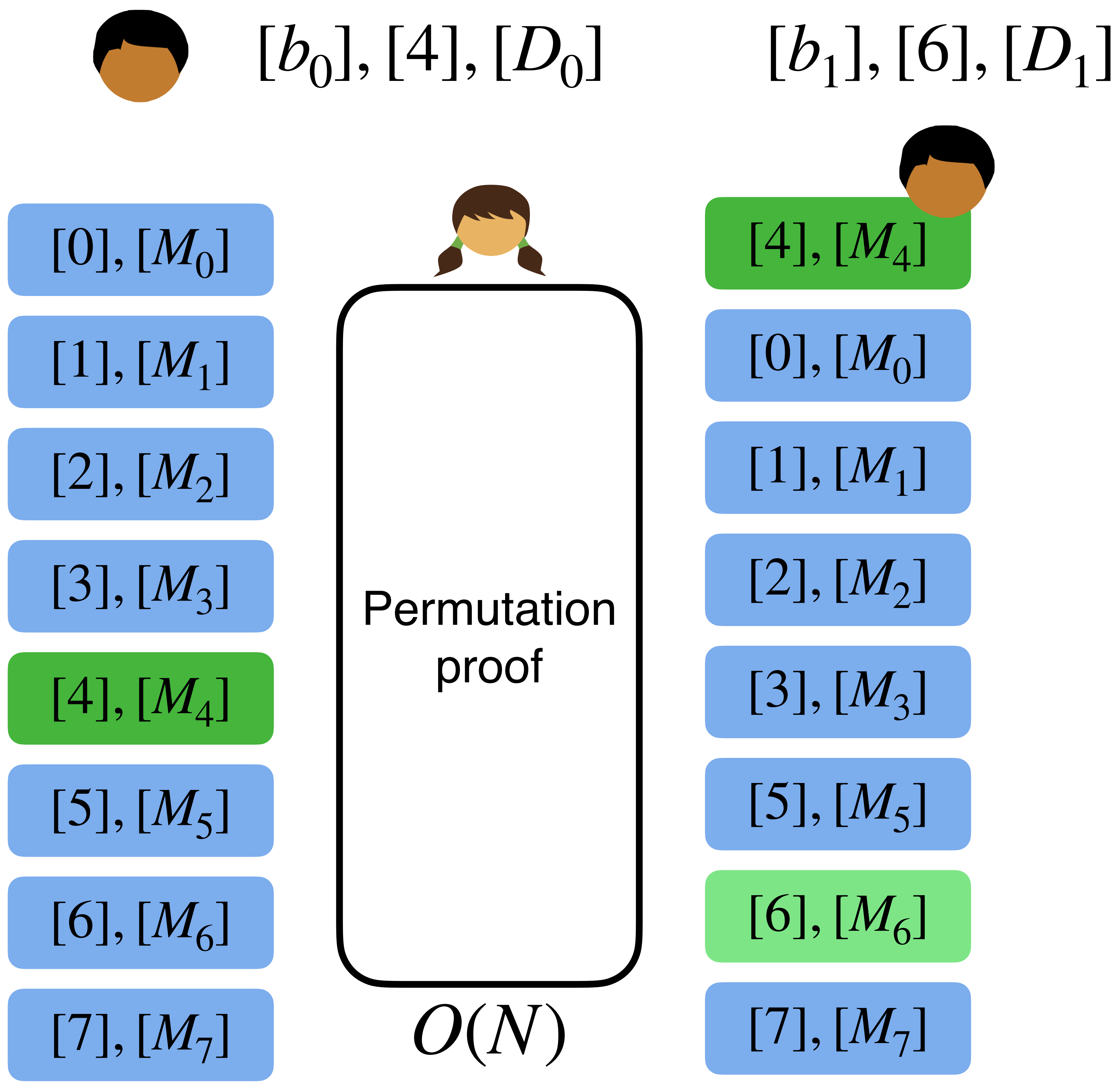
$[6], [M_6]$

$[7], [M_7]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$

$[b_1], [6], [D_1]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

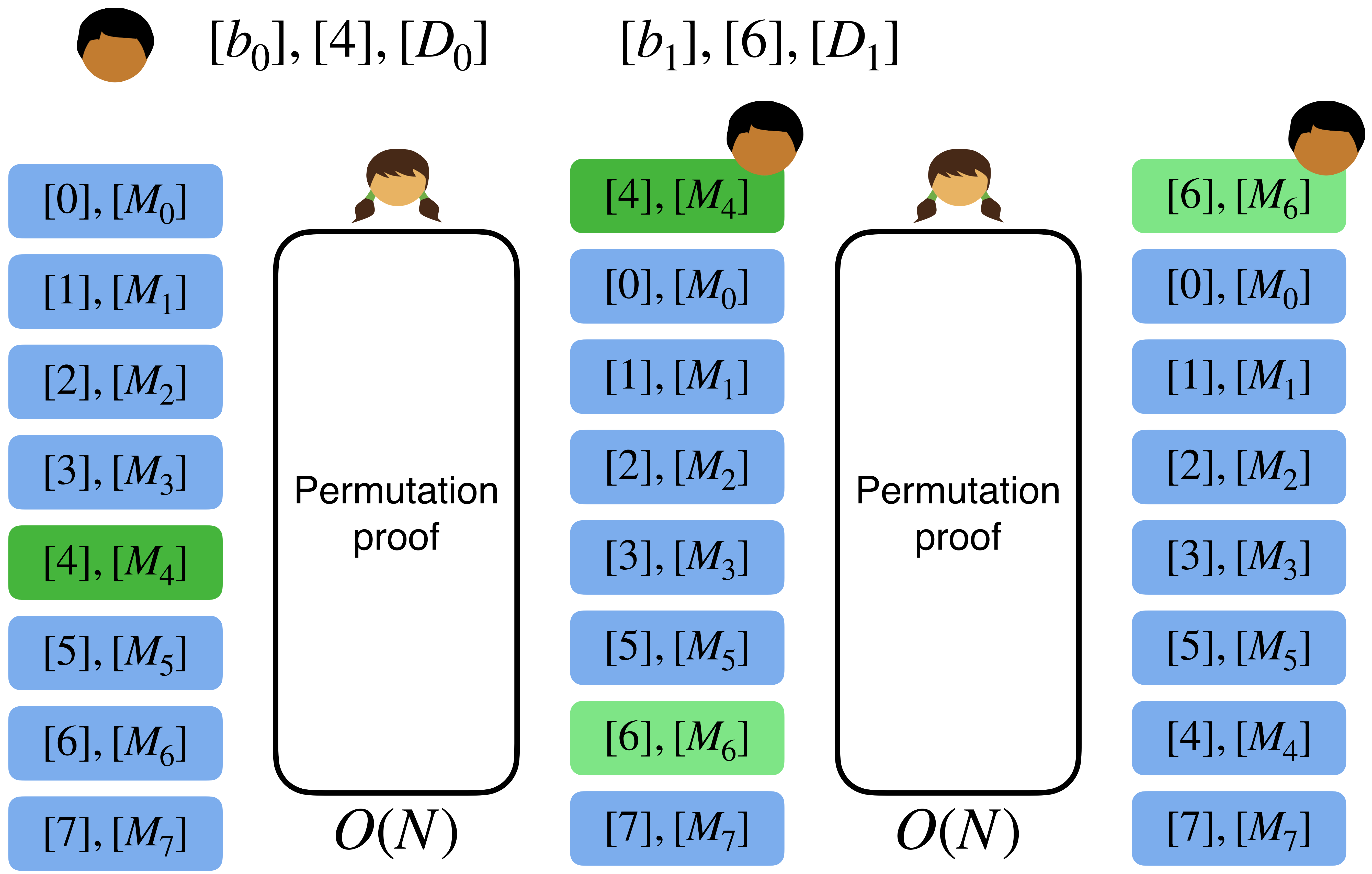
$[6], [M_6]$

$[7], [M_7]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

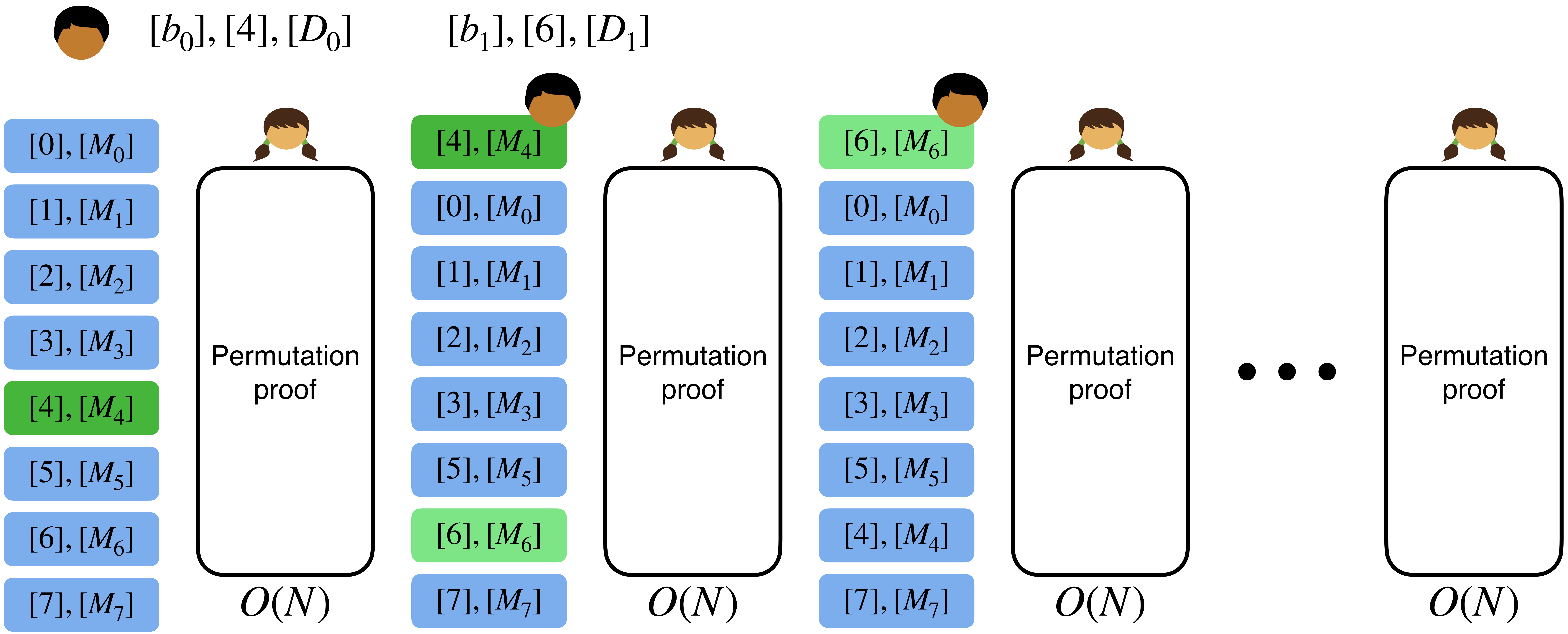
$O(N) \Rightarrow O(\log N)$ per access



ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$

$[b_1], [6], [D_1]$

$[0], [M_0]$
$[1], [M_1]$
$[2], [M_2]$
$[3], [M_3]$
$[4], [M_4]$
$[5], [M_5]$
$[6], [M_6]$
$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$
$[0], [M_0]$
$[1], [M_1]$
$[2], [M_2]$
$[3], [M_3]$
$[5], [M_5]$
$[6], [M_6]$
$[7], [M_7]$

Permutation proof

$O(N)$

$[6], [M_6]$
$[0], [M_0]$
$[1], [M_1]$
$[2], [M_2]$
$[3], [M_3]$
$[5], [M_5]$
$[4], [M_4]$
$[7], [M_7]$

Permutation proof

$O(N)$

• • •

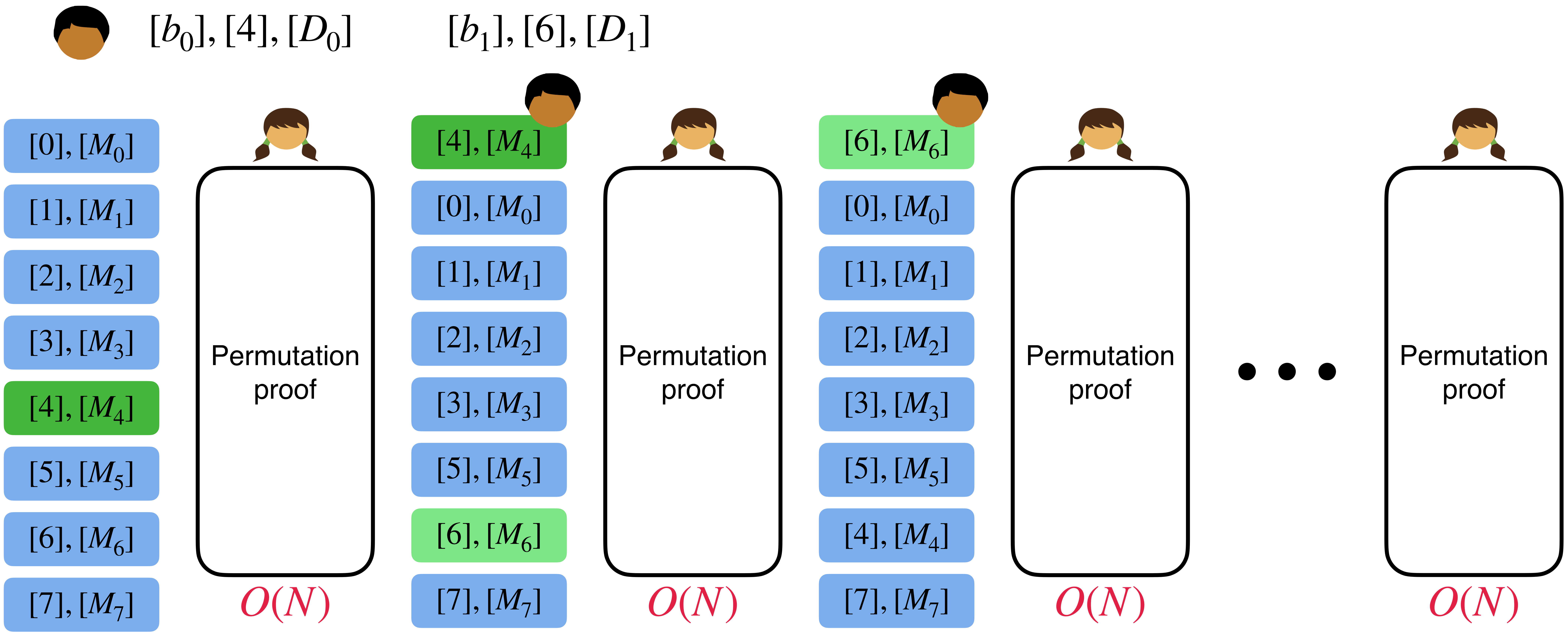
Permutation proof

$O(N)$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$

$[b_1], [6], [D_1]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[6], [M_6]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

$[4], [M_4]$

$[7], [M_7]$

Permutation proof

$O(N)$

• • •

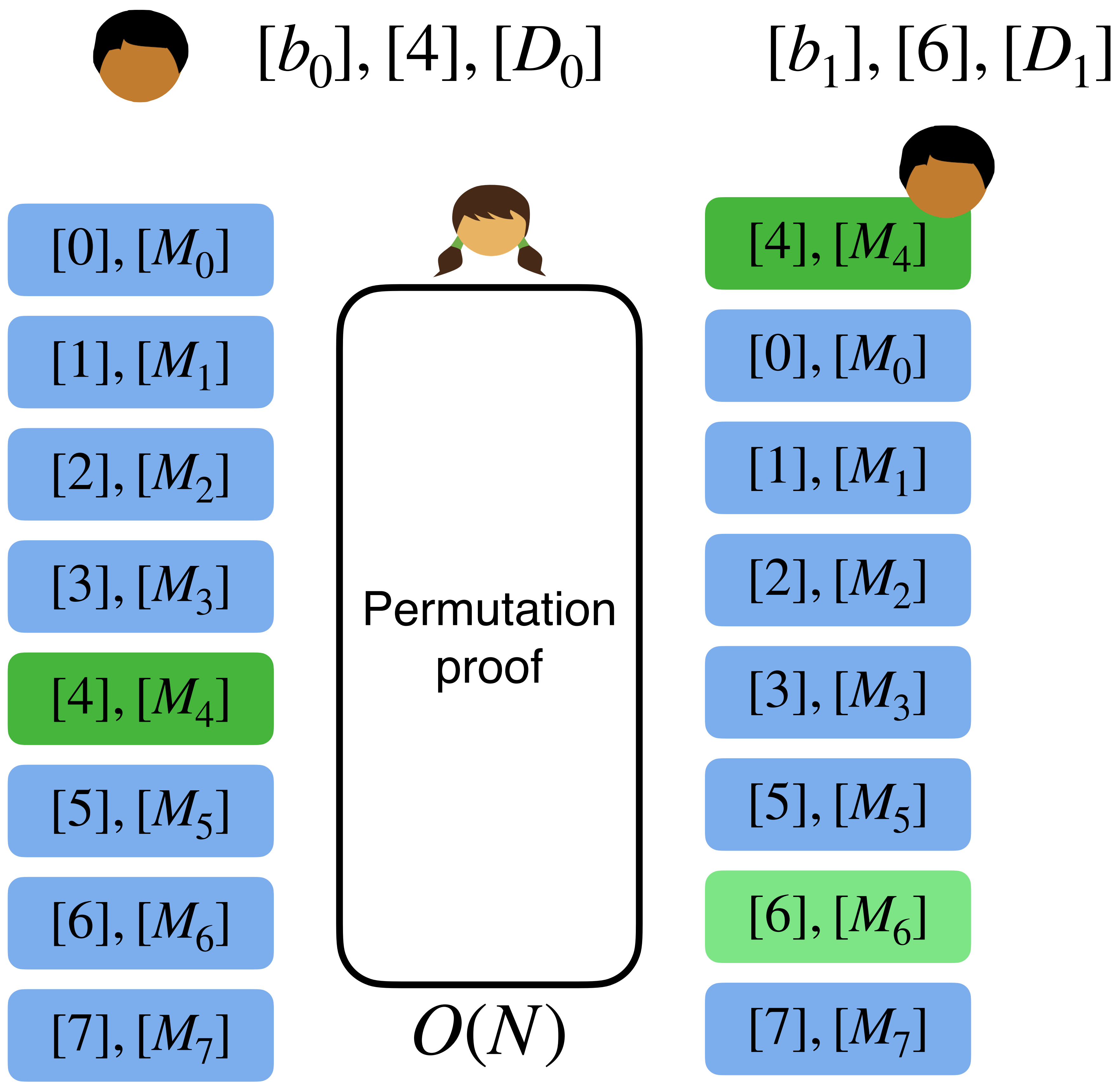
Permutation proof

$O(N)$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$

$[b_1], [6], [D_1]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

$[6], [M_6]$

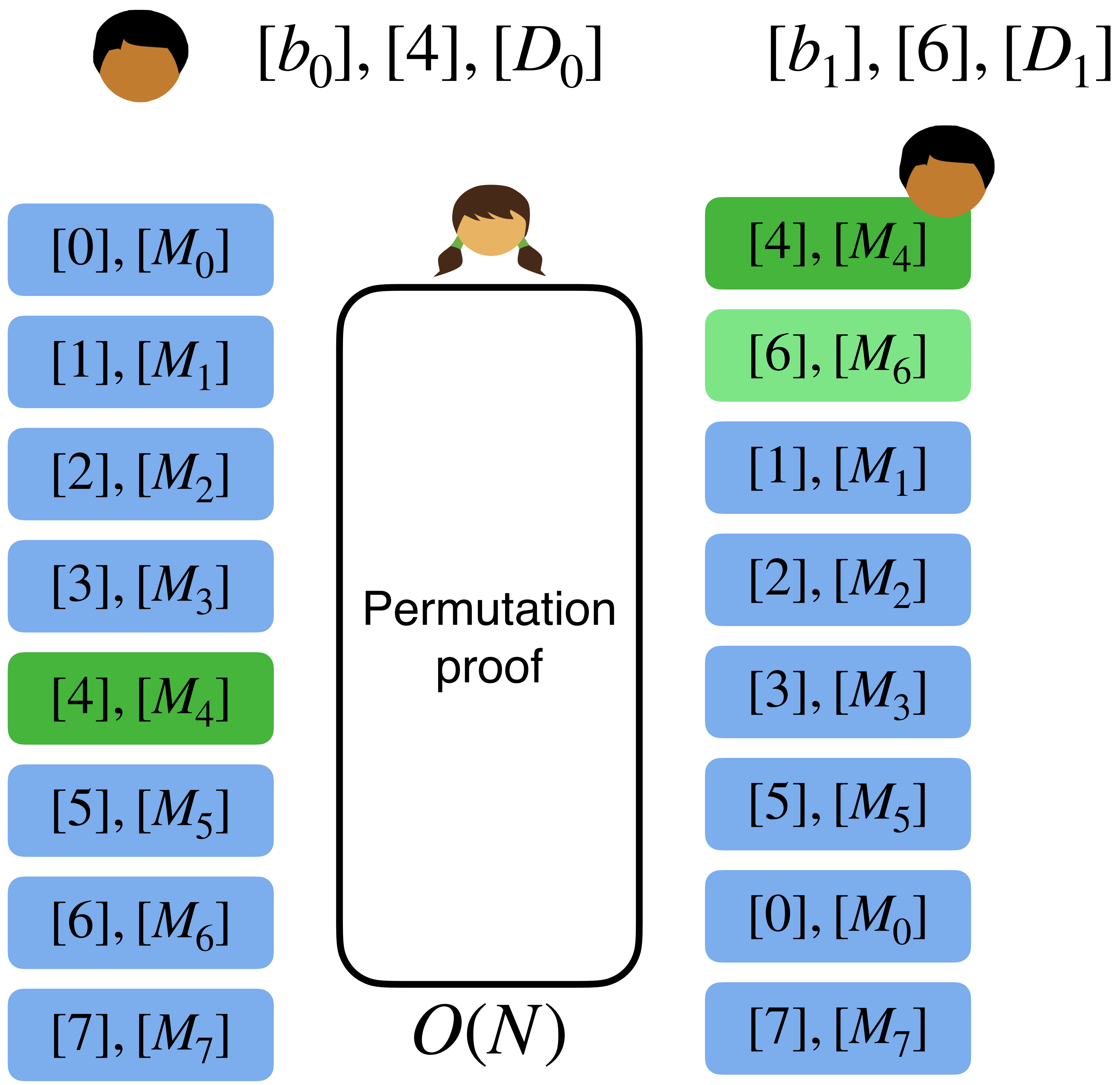
$[7], [M_7]$

Tech. 1: reuse and amortize

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access

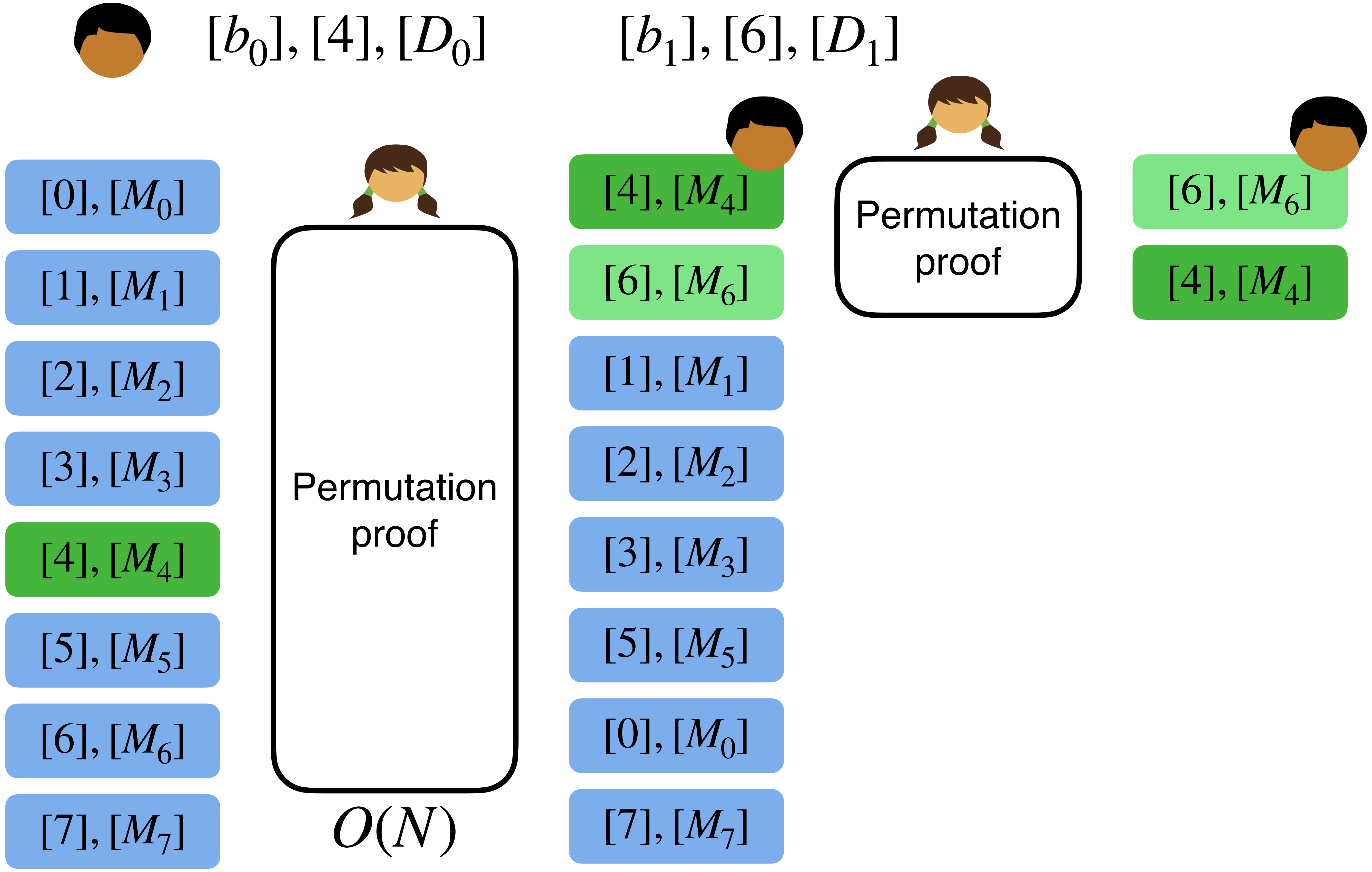


Tech. 1: reuse and amortize

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

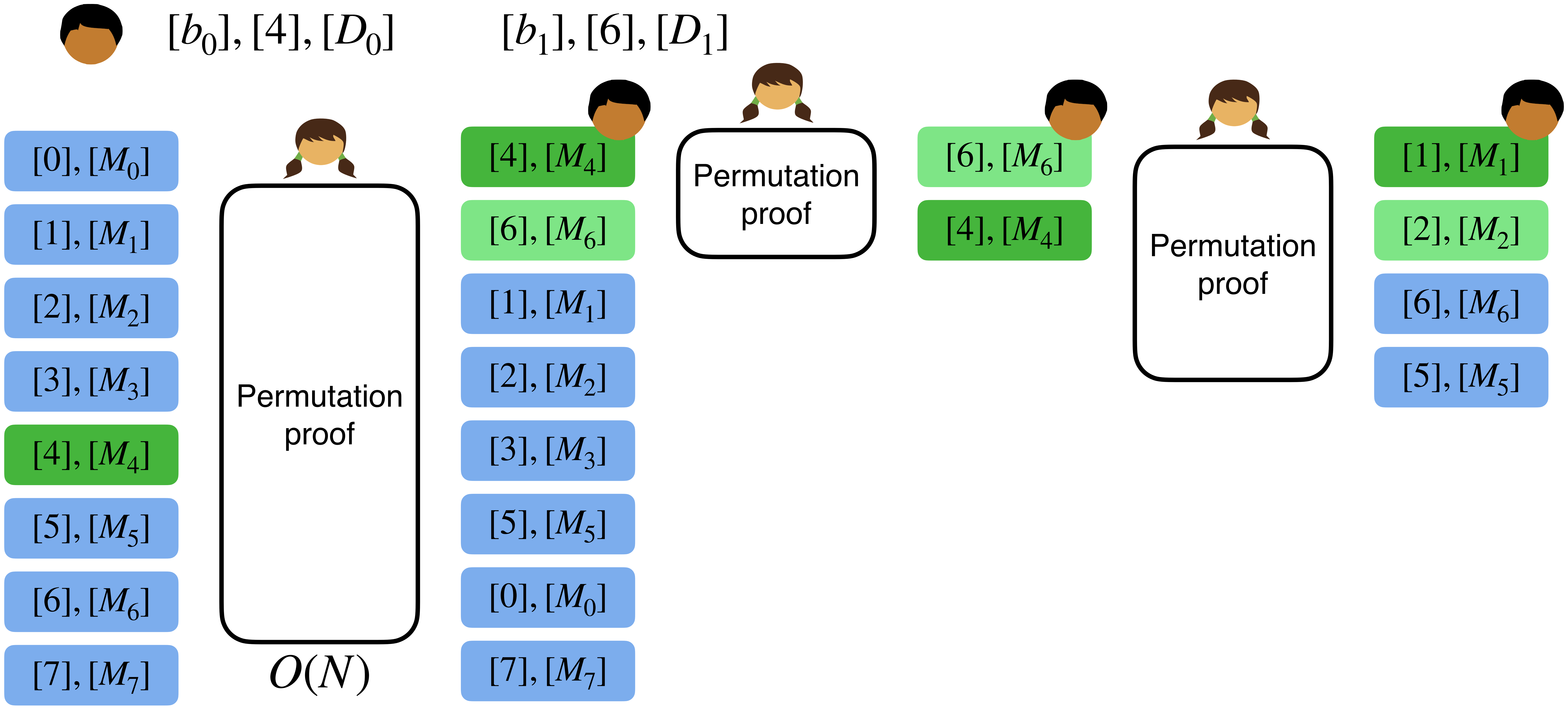
$O(N) \Rightarrow O(\log N)$ per access



ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$

$[b_1], [6], [D_1]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$

$[6], [M_6]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

$[0], [M_0]$

$[7], [M_7]$

Permutation proof

$[6], [M_6]$

$[4], [M_4]$

Permutation proof

$[1], [M_1]$

$[2], [M_2]$

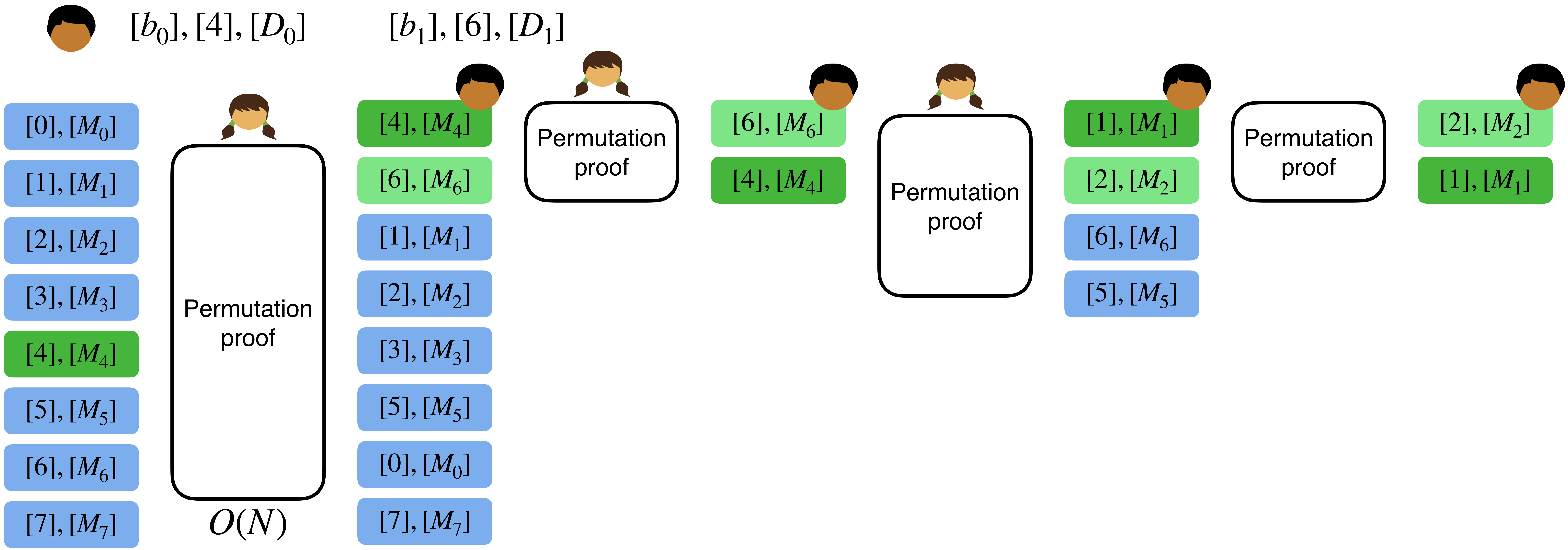
$[6], [M_6]$

$[5], [M_5]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$ $[b_1], [6], [D_1]$

$[0], [M_0]$
$[1], [M_1]$
$[2], [M_2]$
$[3], [M_3]$
$[4], [M_4]$
$[5], [M_5]$
$[6], [M_6]$
$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$
$[6], [M_6]$
$[1], [M_1]$
$[2], [M_2]$
$[3], [M_3]$
$[5], [M_5]$
$[0], [M_0]$
$[7], [M_7]$

Permutation proof

$[6], [M_6]$
$[4], [M_4]$

Permutation proof

$[1], [M_1]$
$[2], [M_2]$
$[6], [M_6]$
$[5], [M_5]$

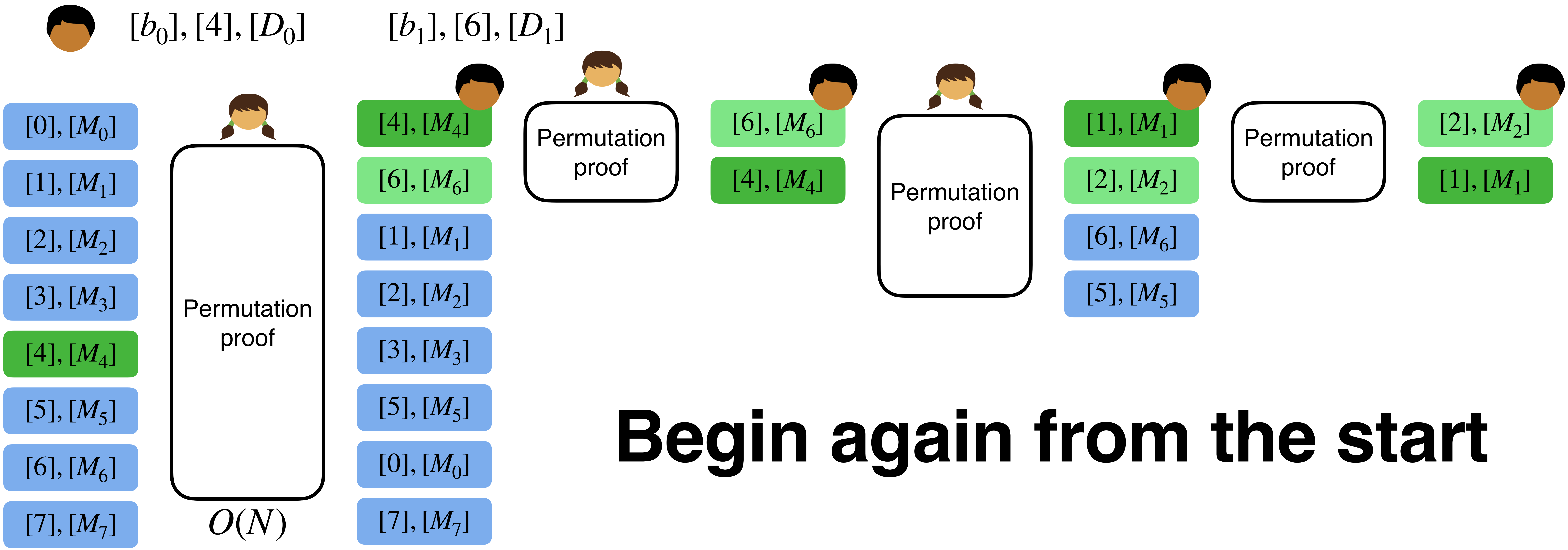
Permutation proof

$[2], [M_2]$
$[1], [M_1]$

ZK Memory

Assuming a read-write memory with N slots, we propose BubbleCache:

$O(N) \Rightarrow O(\log N)$ per access



$[b_0], [4], [D_0]$

$[b_1], [6], [D_1]$

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

Permutation proof

$O(N)$

$[4], [M_4]$

$[6], [M_6]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

$[0], [M_0]$

$[7], [M_7]$

Permutation proof

$[6], [M_6]$

$[4], [M_4]$

Permutation proof

$[1], [M_1]$

$[2], [M_2]$

$[6], [M_6]$

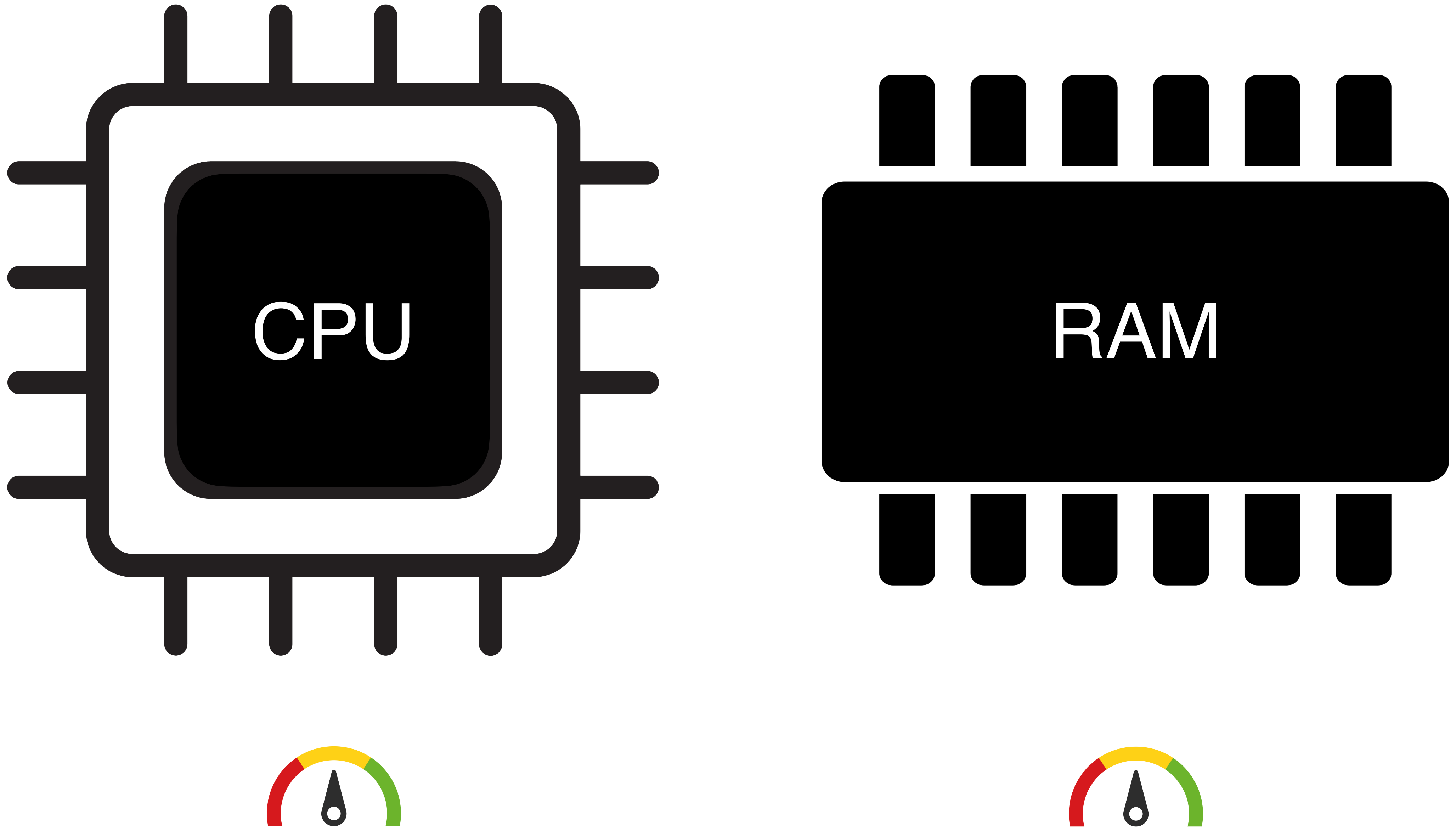
$[5], [M_5]$

Permutation proof

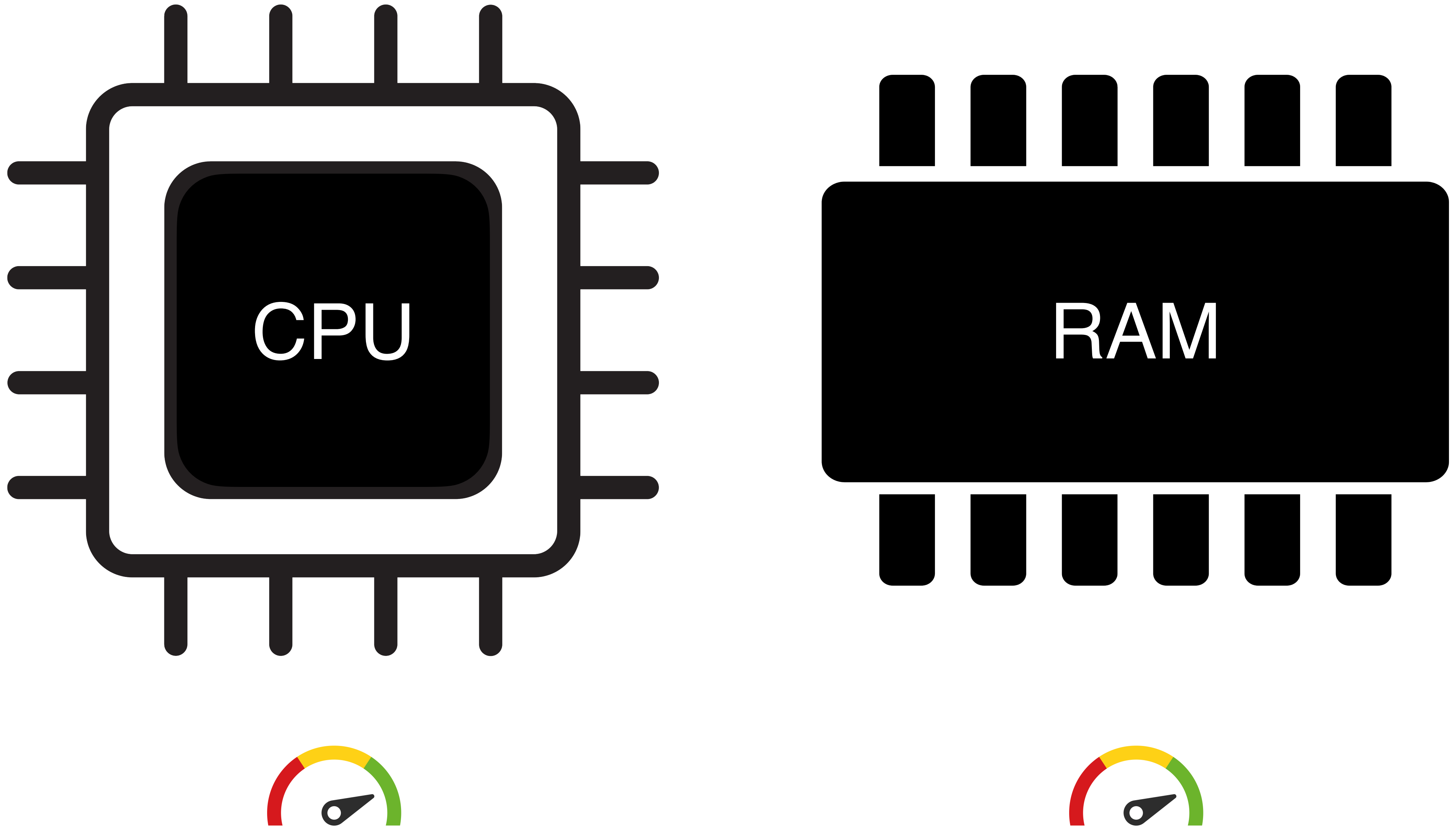
$[2], [M_2]$

$[1], [M_1]$

Begin again from the start



CPU
RAM



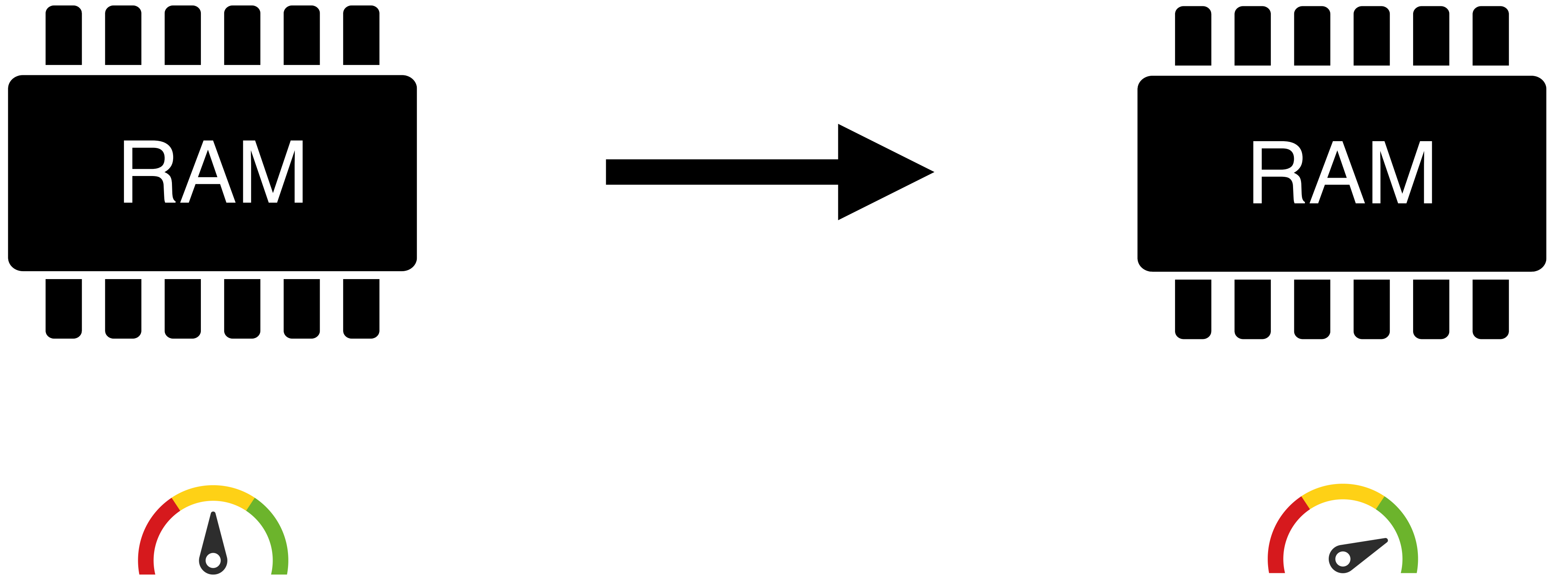
Continue!

USENIX Security 2024

Two Shuffles Make a RAM: Improved Constant Overhead Zero Knowledge RAM

Yibin Yang
Georgia Institute of Technology

David Heath
University of Illinois Urbana-Champaign



Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

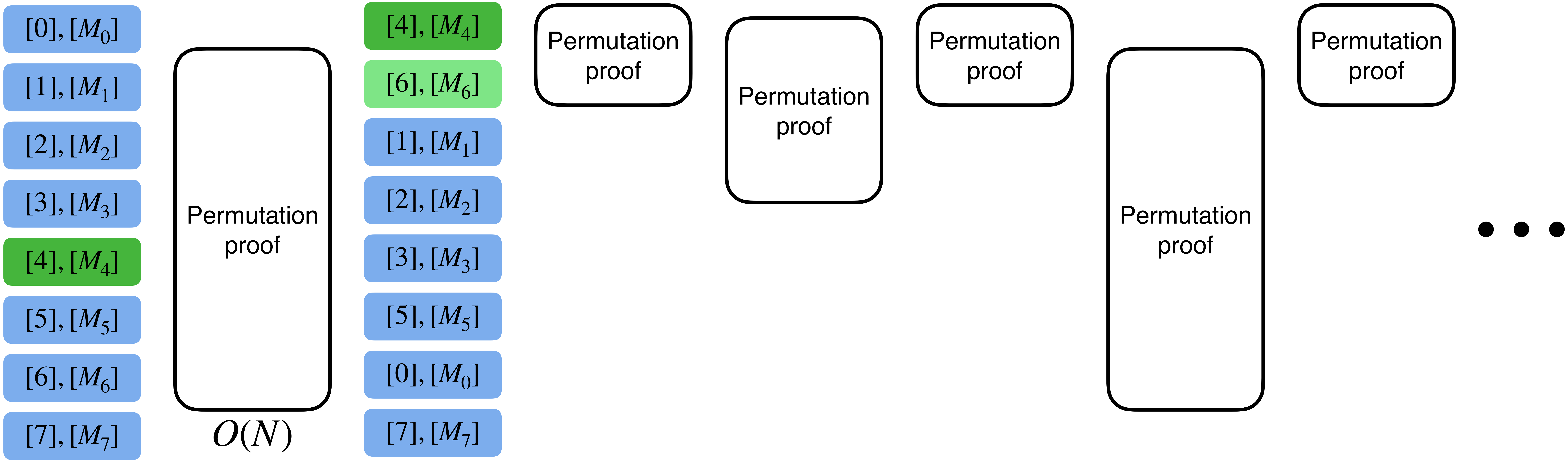
“Two Shuffles (i.e., Permutations) Make a RAM”

Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

“Two Shuffles (i.e., Permutations) Make a RAM”

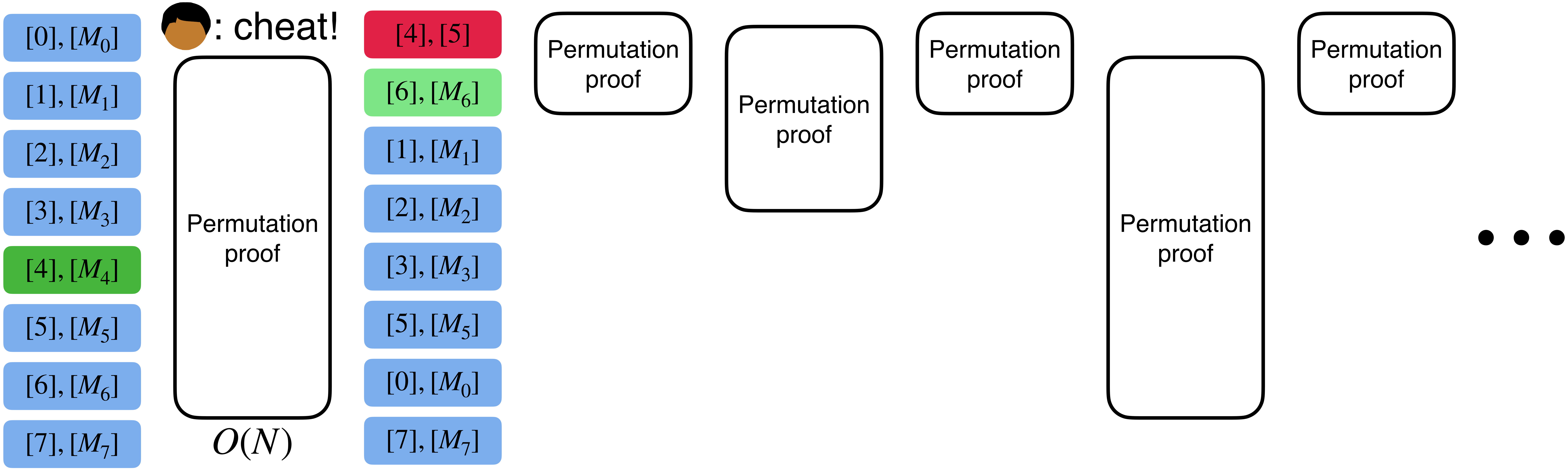


Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

"Two Shuffles (i.e., Permutations) Make a RAM"



$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

: cheat!

Permutation proof

$O(N)$

$[4], [5]$

$[6], [M_6]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[5], [M_5]$

$[0], [M_0]$

$[7], [M_7]$

Permutation proof

Permutation proof

Permutation proof

Permutation proof

Permutation proof

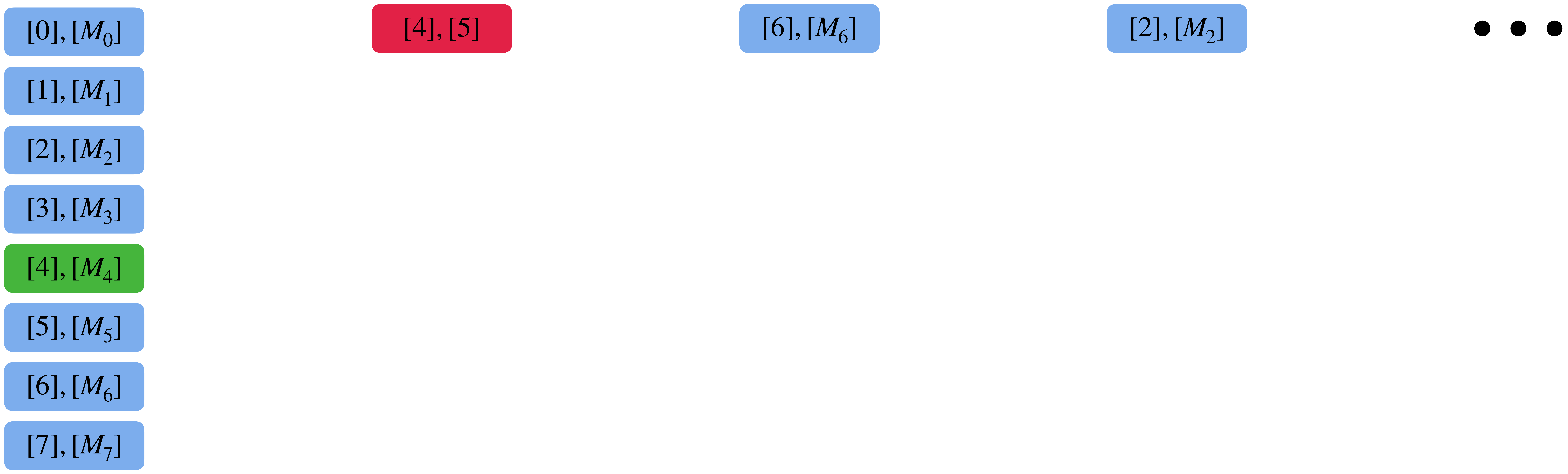
• • •

Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

“Two Shuffles (i.e., Permutations) Make a RAM”



$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$

$[4], [5]$

$[6], [M_6]$

$[2], [M_2]$

• • •

Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

“Two Shuffles (i.e., Permutations) Make a RAM”

$[0], [M_0]$

$[1], [M_1]$

$[2], [M_2]$

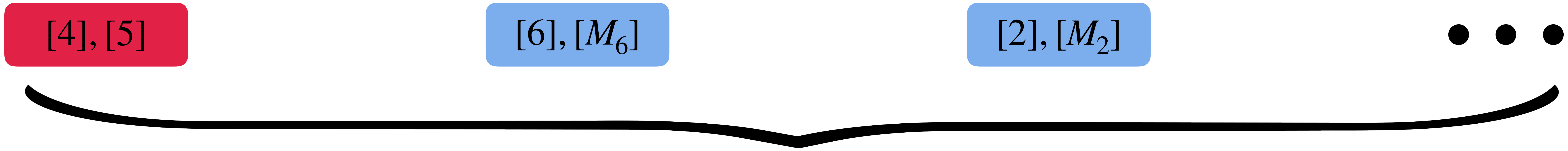
$[3], [M_3]$

$[4], [M_4]$

$[5], [M_5]$

$[6], [M_6]$

$[7], [M_7]$



Global final check

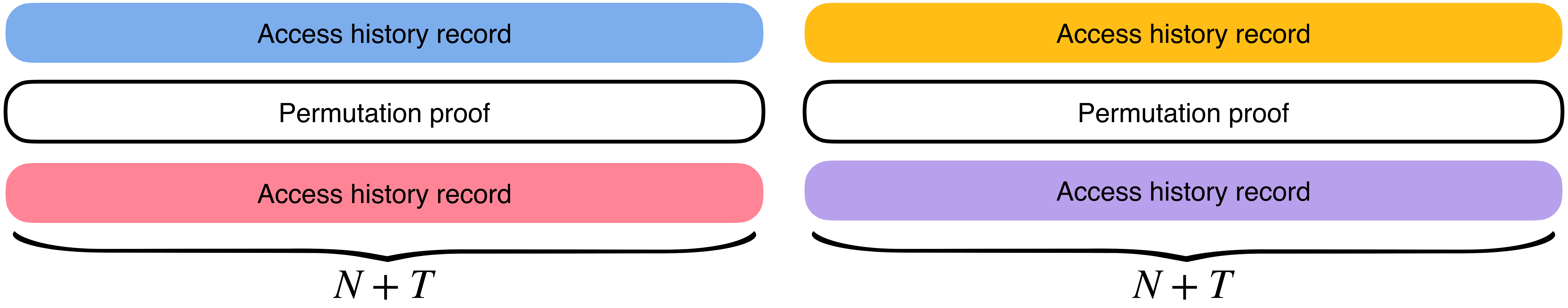
: cheat!

Constant-Overhead ZK Memory

Assuming a read-write memory with N slots, we propose a ZK memory:

$O(N) \Rightarrow O(\log N) \Rightarrow O(1)$ per access

"<u>Two</u> Shuffles (i.e., Permutations) Make a RAM"



(T denotes the time of accesses)

Simplifications:

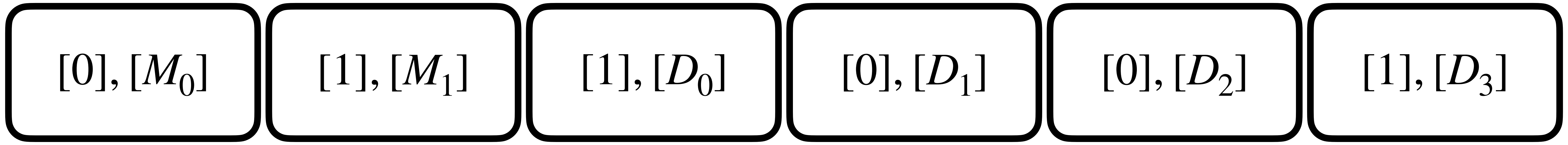
Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)



$[0], [M_0]$ $[1], [M_1]$ $[1], [D_0]$ $[0], [D_1]$ $[0], [D_2]$ $[1], [D_3]$

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

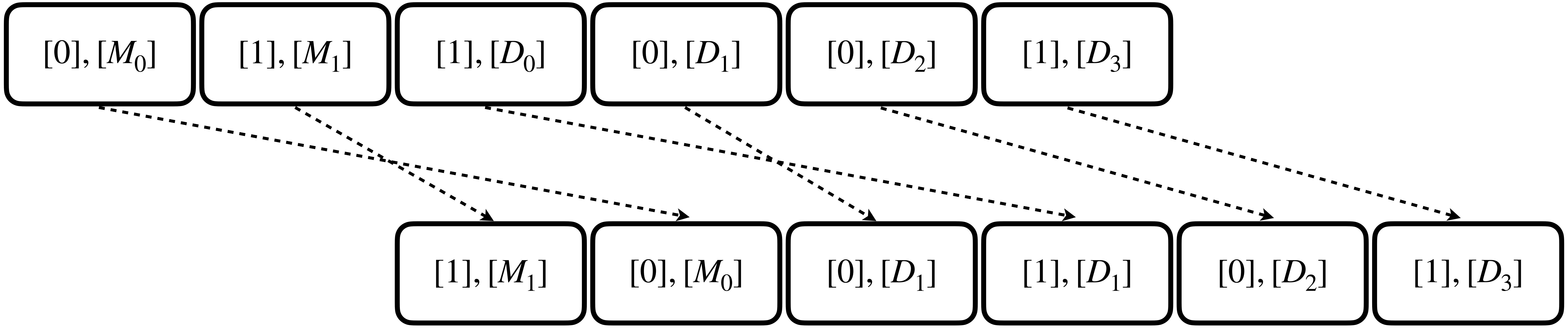
Tech. 2: P knows and helps

$[0], [M_0]$	$[1], [M_1]$	$[1], [D_0]$	$[0], [D_1]$	$[0], [D_2]$	$[1], [D_3]$

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

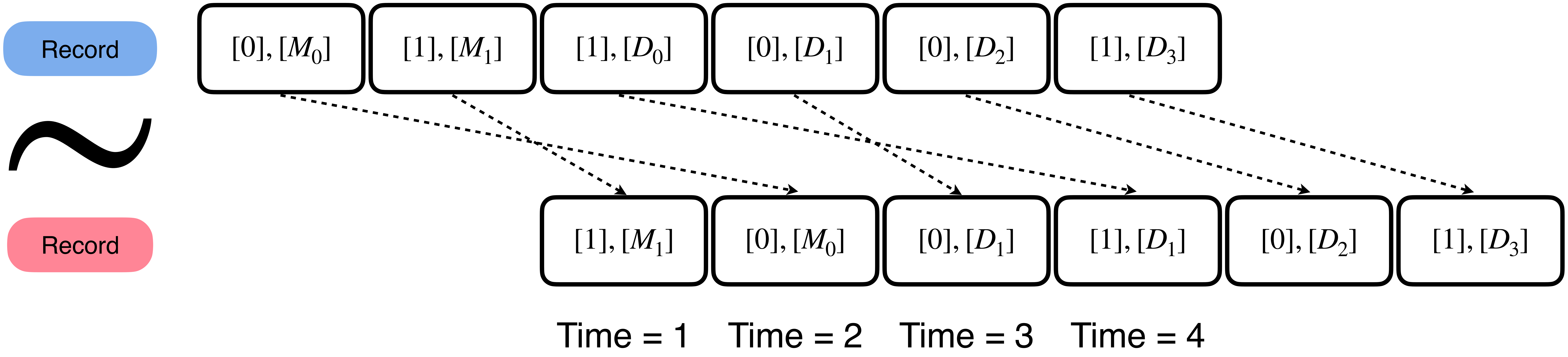


Time = 1 Time = 2 Time = 3 Time = 4

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps



Record

$[0], [M_0]$ $[1], [M_1]$ $[1], [D_0]$ $[0], [D_1]$ $[0], [D_2]$ $[1], [D_3]$

$\sim$

Record

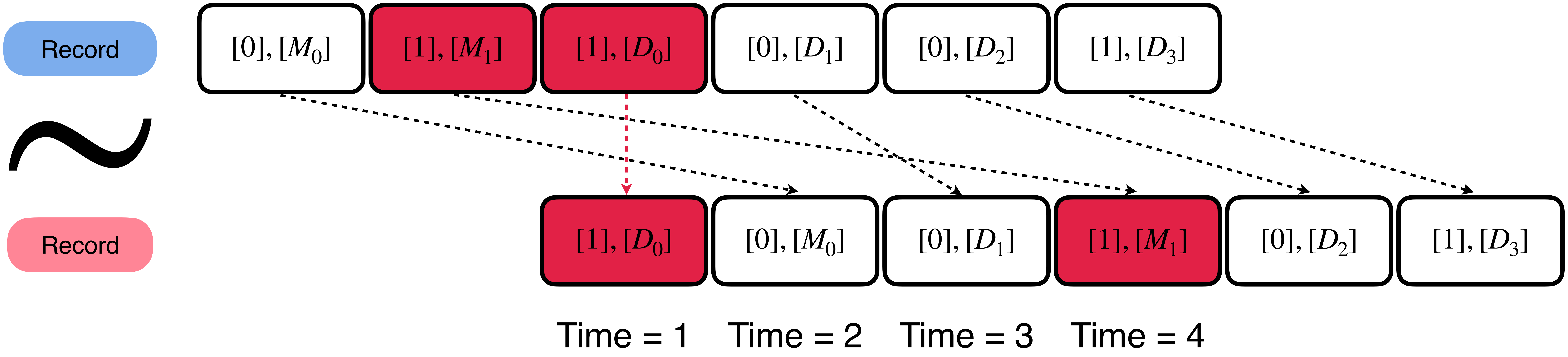
$[1], [M_1]$ $[0], [M_0]$ $[0], [D_1]$ $[1], [D_1]$ $[0], [D_2]$ $[1], [D_3]$

Time = 1 Time = 2 Time = 3 Time = 4

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

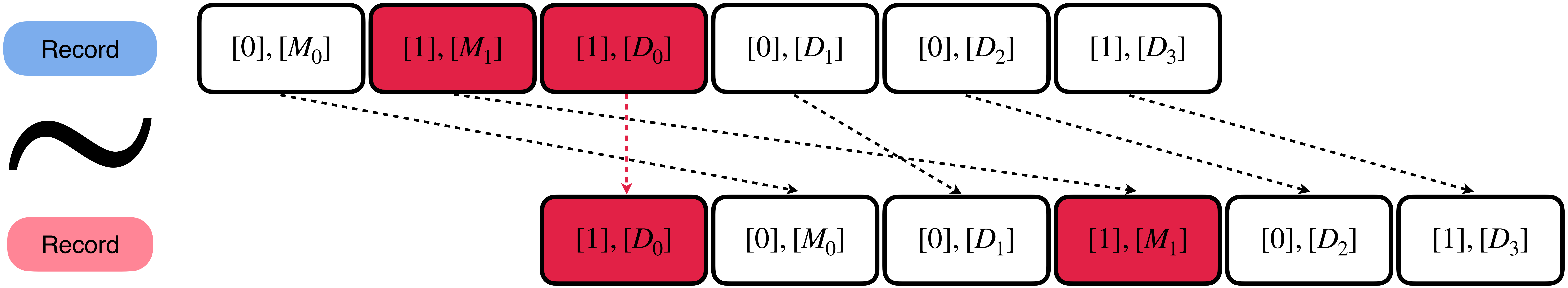


Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)



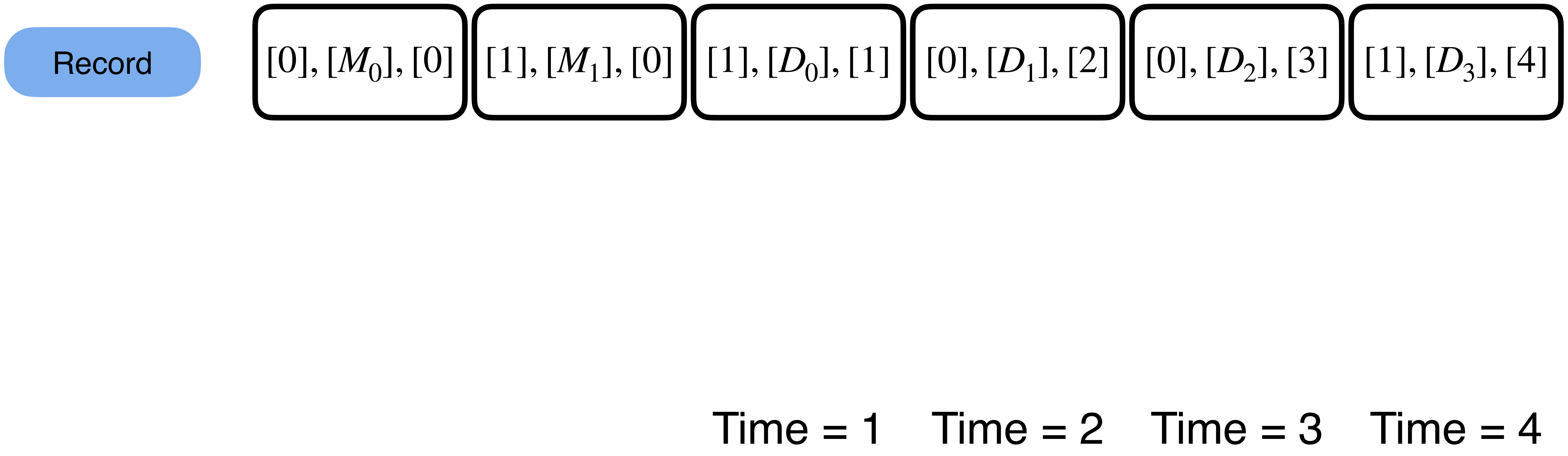
Time = 1 Time = 2 Time = 3 Time = 4

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)



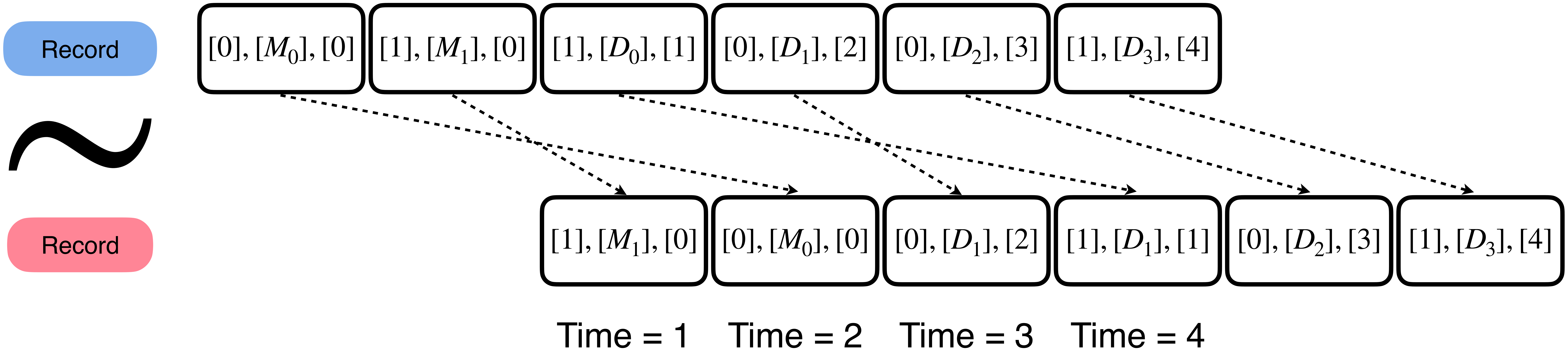
Record	$[0], [M_0], [0]$	$[1], [M_1], [0]$	$[1], [D_0], [1]$	$[0], [D_1], [2]$	$[0], [D_2], [3]$	$[1], [D_3], [4]$
			Time = 1	Time = 2	Time = 3	Time = 4

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)



Record

$[0], [M_0], [0]$ | $[1], [M_1], [0]$ | $[1], [D_0], [1]$ | $[0], [D_1], [2]$ | $[0], [D_2], [3]$ | $[1], [D_3], [4]$

$\sim$

Record

$[1], [M_1], [0]$ | $[0], [M_0], [0]$ | $[0], [D_1], [2]$ | $[1], [D_1], [1]$ | $[0], [D_2], [3]$ | $[1], [D_3], [4]$

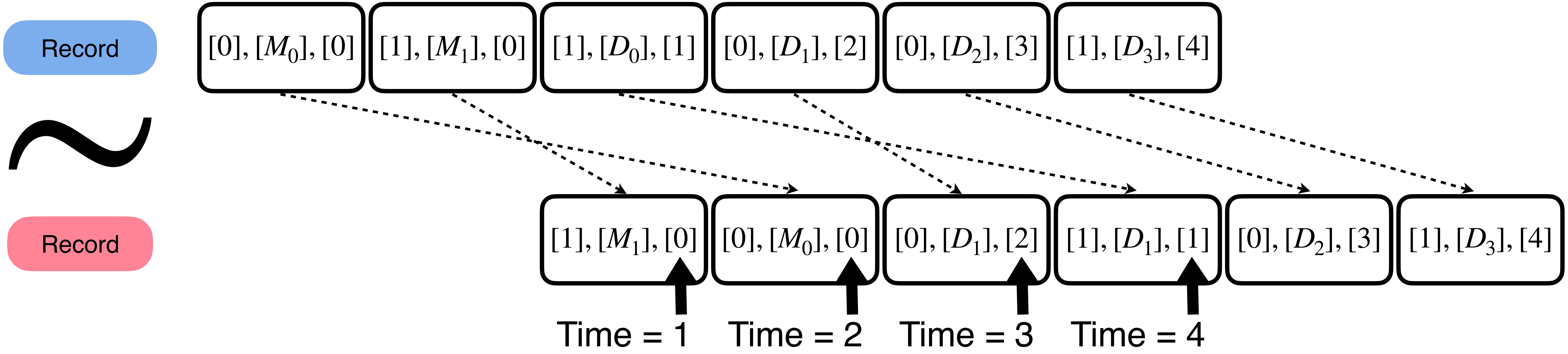
Time = 1 Time = 2 Time = 3 Time = 4

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)

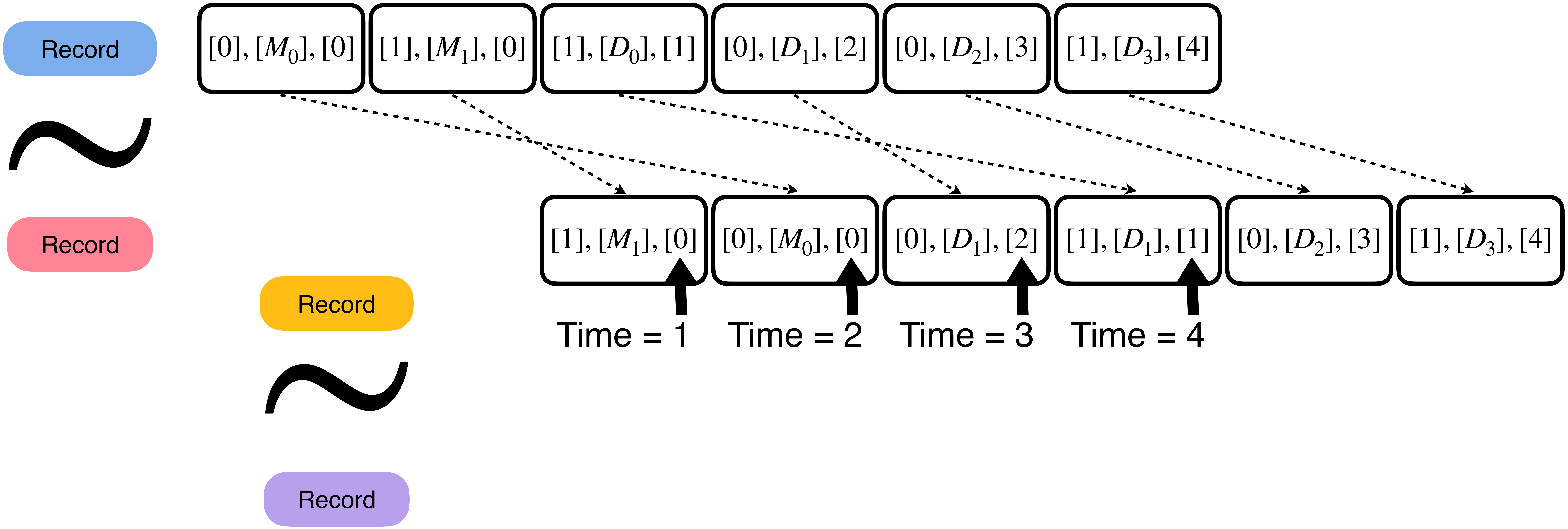


Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)



Record

$\sim$

Record

$[0], [M_0], [0]$ $[1], [M_1], [0]$ $[1], [D_0], [1]$ $[0], [D_1], [2]$ $[0], [D_2], [3]$ $[1], [D_3], [4]$

$[1], [M_1], [0]$ $[0], [M_0], [0]$ $[0], [D_1], [2]$ $[1], [D_1], [1]$ $[0], [D_2], [3]$ $[1], [D_3], [4]$

Time = 1 Time = 2 Time = 3 Time = 4

Record

$\sim$

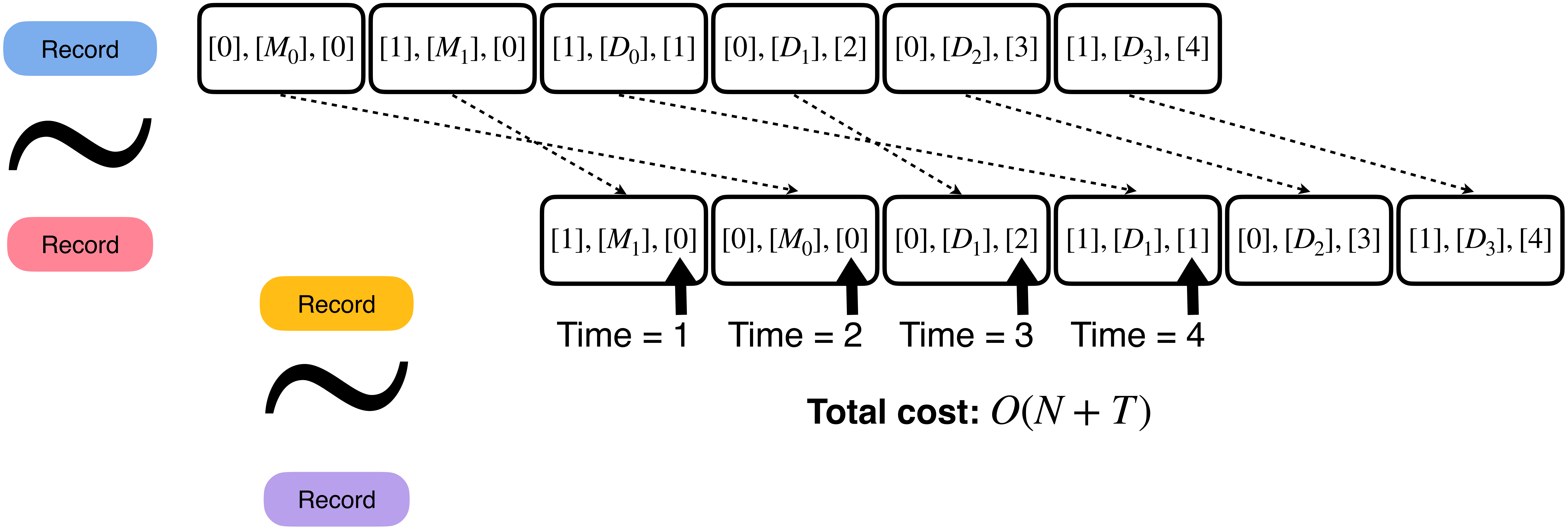
Record

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)



$[0], [M_0], [0]$ | $[1], [M_1], [0]$ | $[1], [D_0], [1]$ | $[0], [D_1], [2]$ | $[0], [D_2], [3]$ | $[1], [D_3], [4]$

$[1], [M_1], [0]$ | $[0], [M_0], [0]$ | $[0], [D_1], [2]$ | $[1], [D_1], [1]$ | $[0], [D_2], [3]$ | $[1], [D_3], [4]$

Time = 1 Time = 2 Time = 3 Time = 4

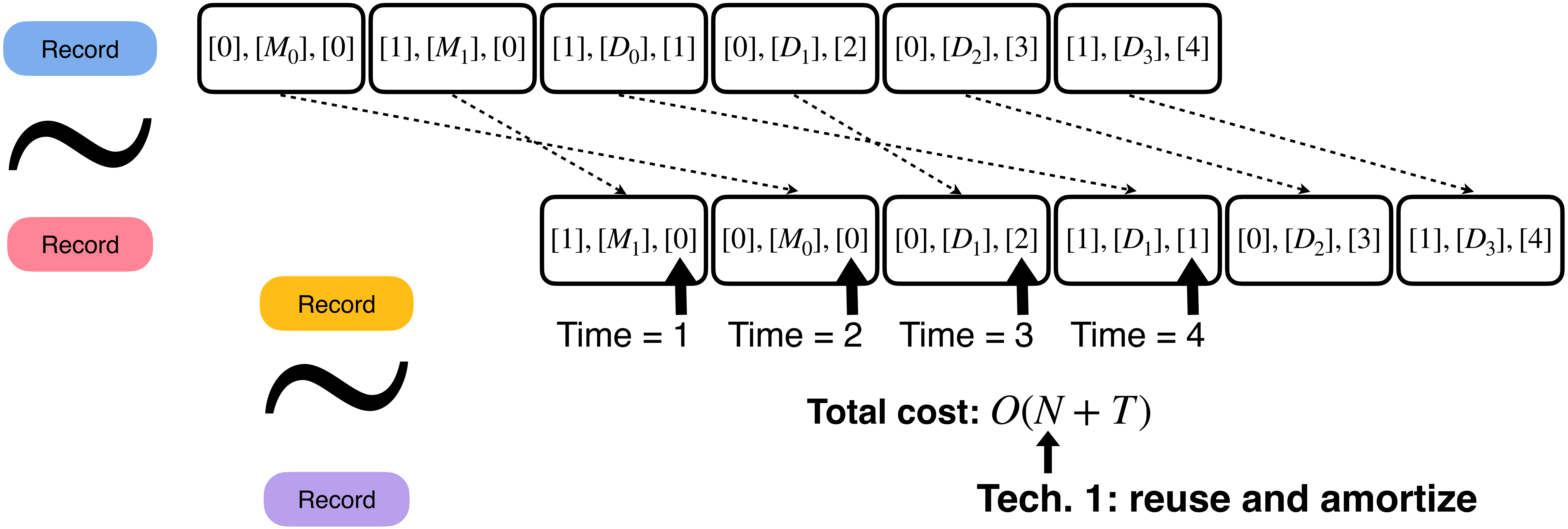
Total cost: $O(N + T)$

Simplifications: Perform $T = 4$ accesses to a ZK RAM with $N = 2$ slots

Read-Write($[1]$,$[D_0]$) Read-Write($[0]$,$[D_1]$) Read-Write($[0]$,$[D_2]$) Read-Write($[1]$,$[D_3]$)

Tech. 2: P knows and helps

Maintain triples: (index, value, timestamp)



Total cost: $O(N + T)$

$\uparrow$

Tech. 1: reuse and amortize

Concrete improvements over BubbleCache: $12–160\times$, depending on the network settings

Concrete improvements over BubbleCache:
$12{-}160\times$, depending on the network settings

“One Shuffle (i.e., Permutation) Makes a ROM”

Concrete improvements over BubbleCache:
$12\text{--}160\times$, depending on the network settings

“One Shuffle (i.e., Permutation) Makes a ROM”

ZK RAM and ROM over vectors

Total cost: $O(Nm + Tm)$ for T accesses and length-m vectors

ACM CCS 2023

Batchman and Robin:
Batched and Non-batched Branching for Interactive ZK

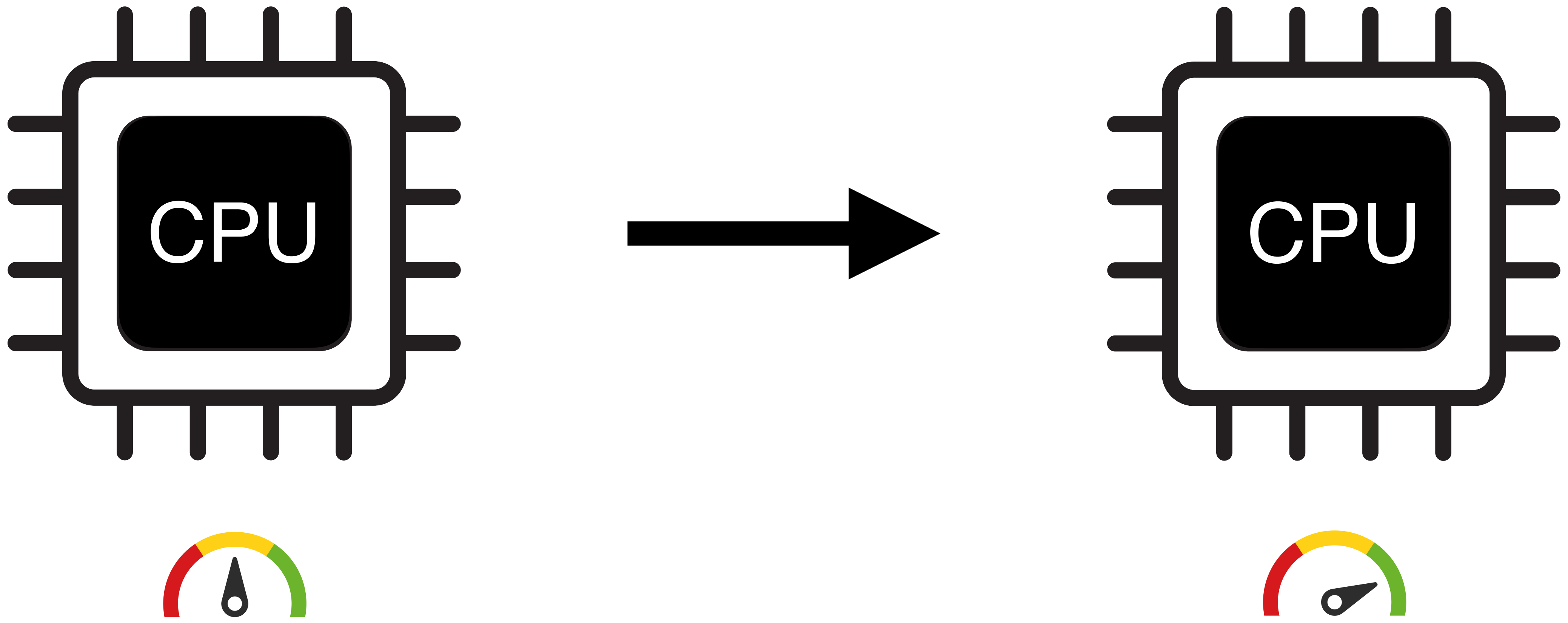
Yibin Yang
Georgia Institute of Technology, USA
yyang811@gatech.edu

David Heath
University of Illinois
Urbana-Champaign, USA
daheath@illinois.edu

Carmit Hazay
Bar-Ilan University, Ligero Inc., Israel
Carmit.Hazay@biu.ac.il

Vladimir Kolesnikov
Georgia Institute of Technology, USA
kolesnikov@gatech.edu

Muthuramakrishnan
Venkitasubramaniam
Ligero Inc., USA
muthu@ligero-inc.com

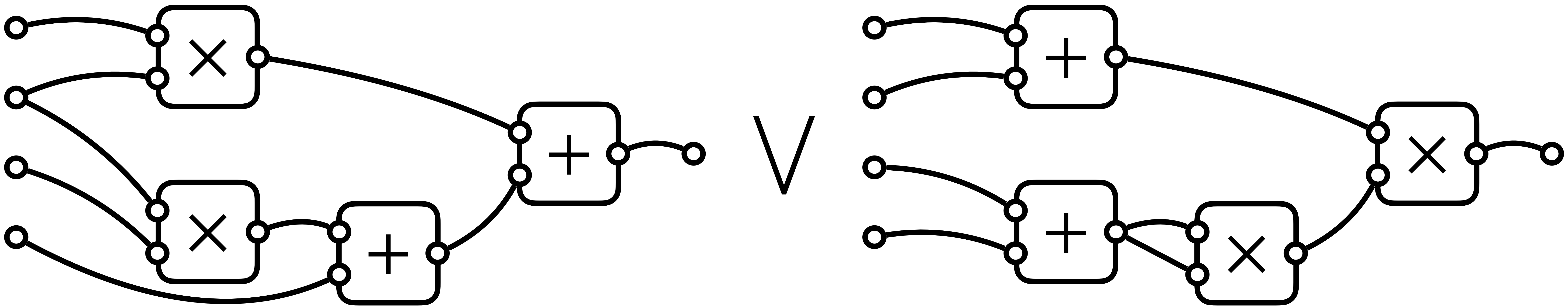


🏆 Distinguished Paper Award

Batched Disjunctions

Batched Disjunctions

Containing 2 Branches and 1 Repetition



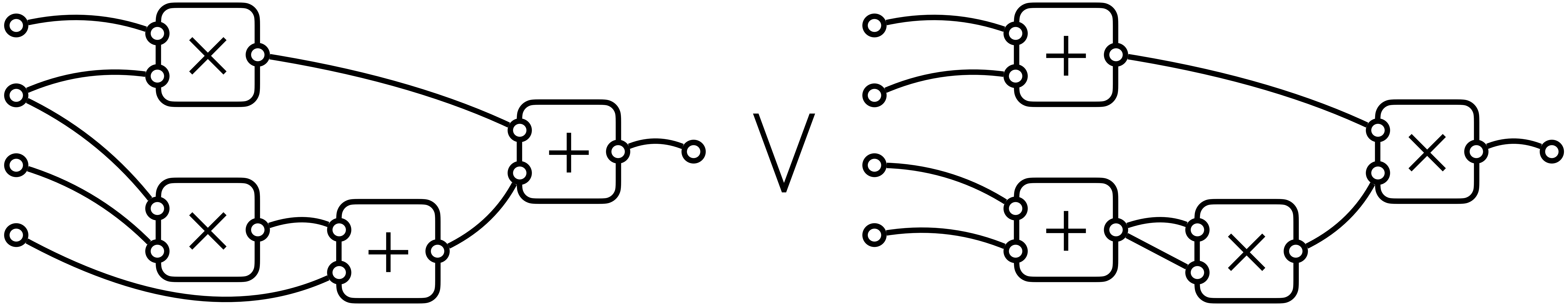
I know the inputs that make **one of these two** circuits to produce an output of zero

Batched Disjunctions

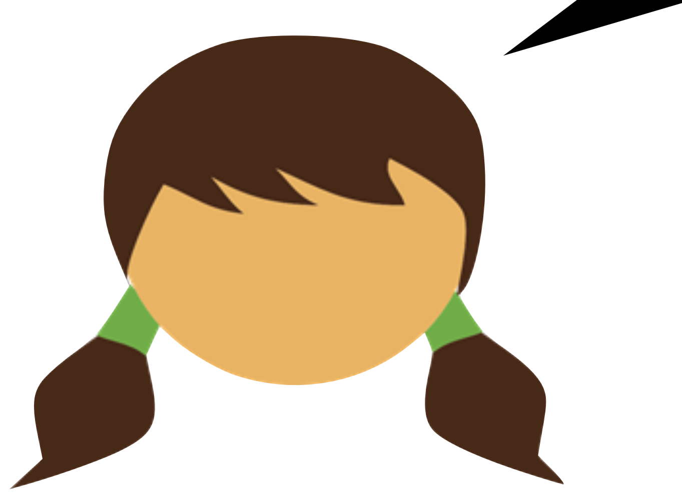
Containing 2 Branches and 1 Repetition

SHL

AND



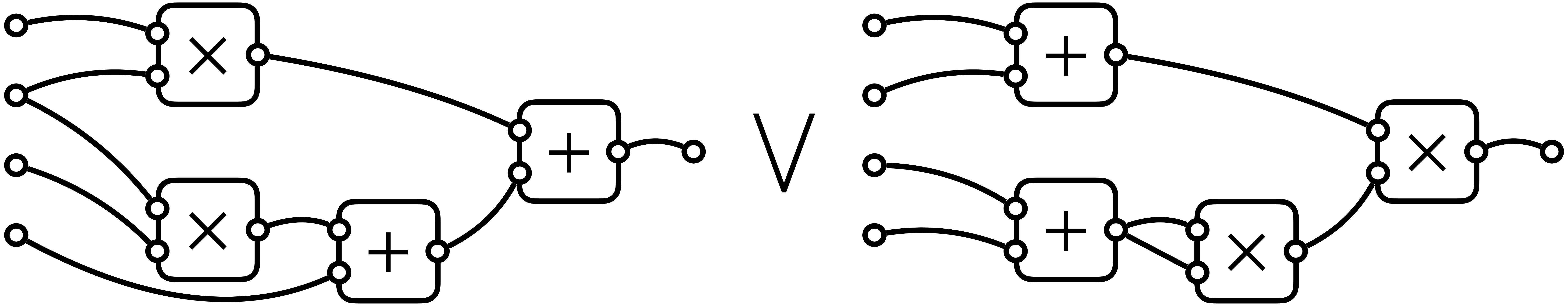
I know the inputs that make **one of these two** circuits to produce an output of zero

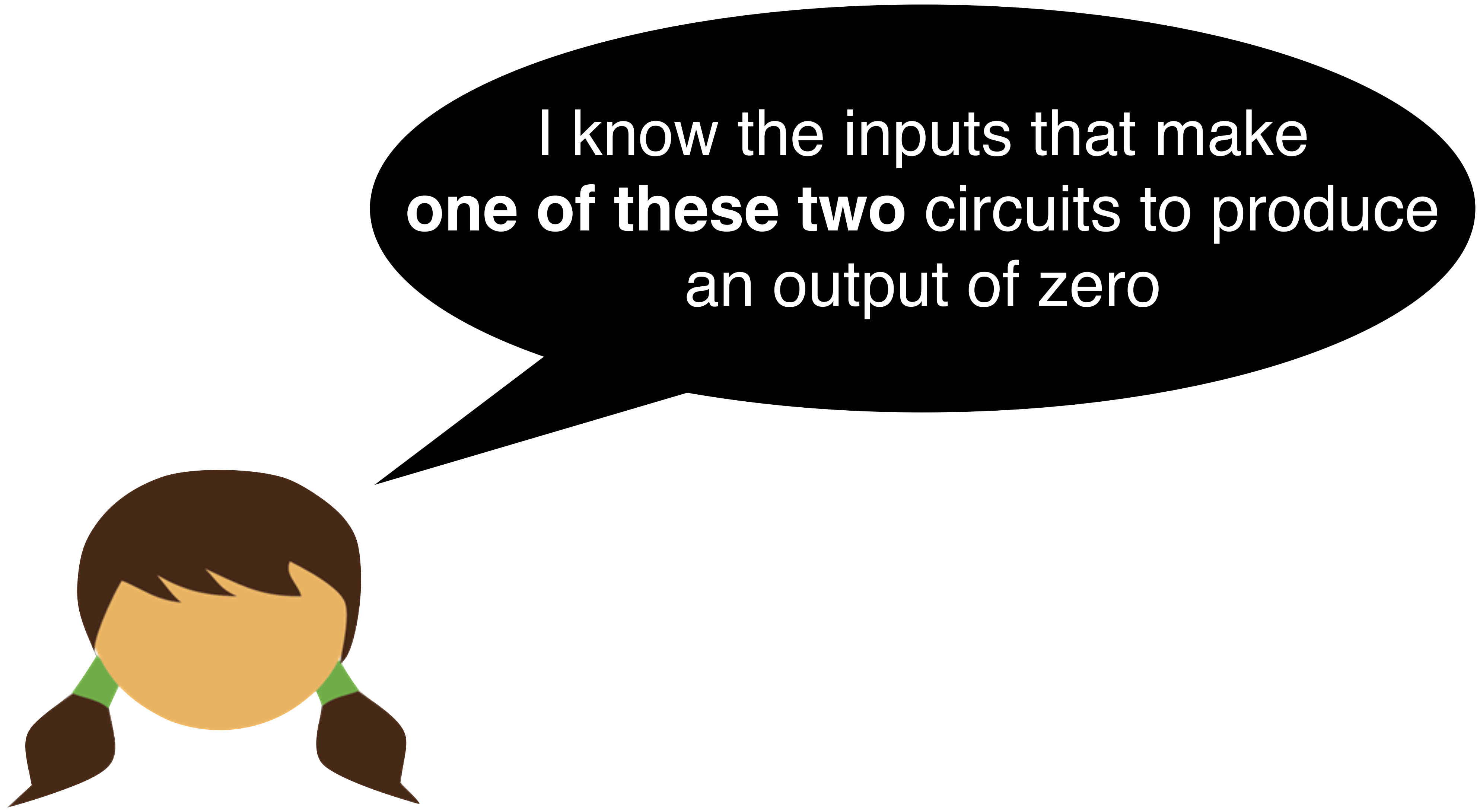


Batched Disjunctions

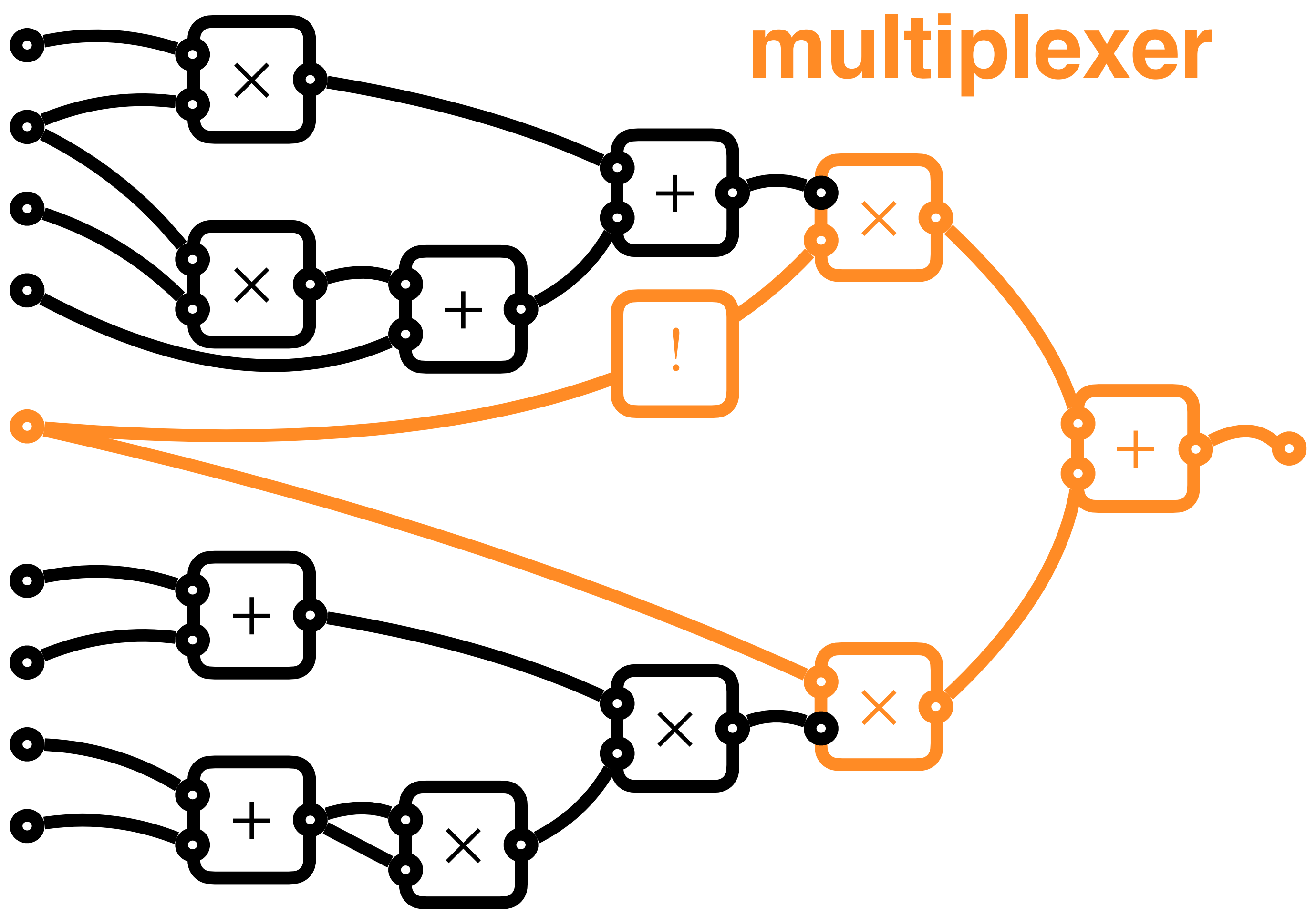
Containing 2 Branches and 1 Repetition

SHL
AND




I know the inputs that make **one of these two** circuits to produce an output of zero

Straightforward-but-expensive approach:

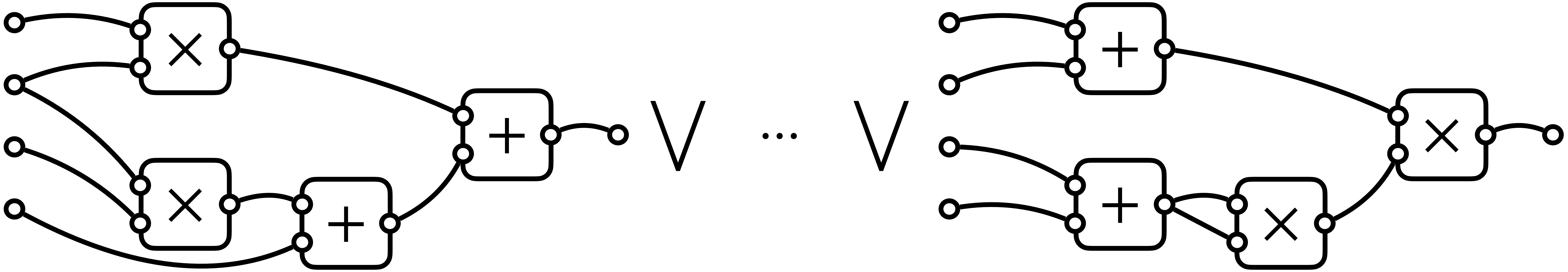


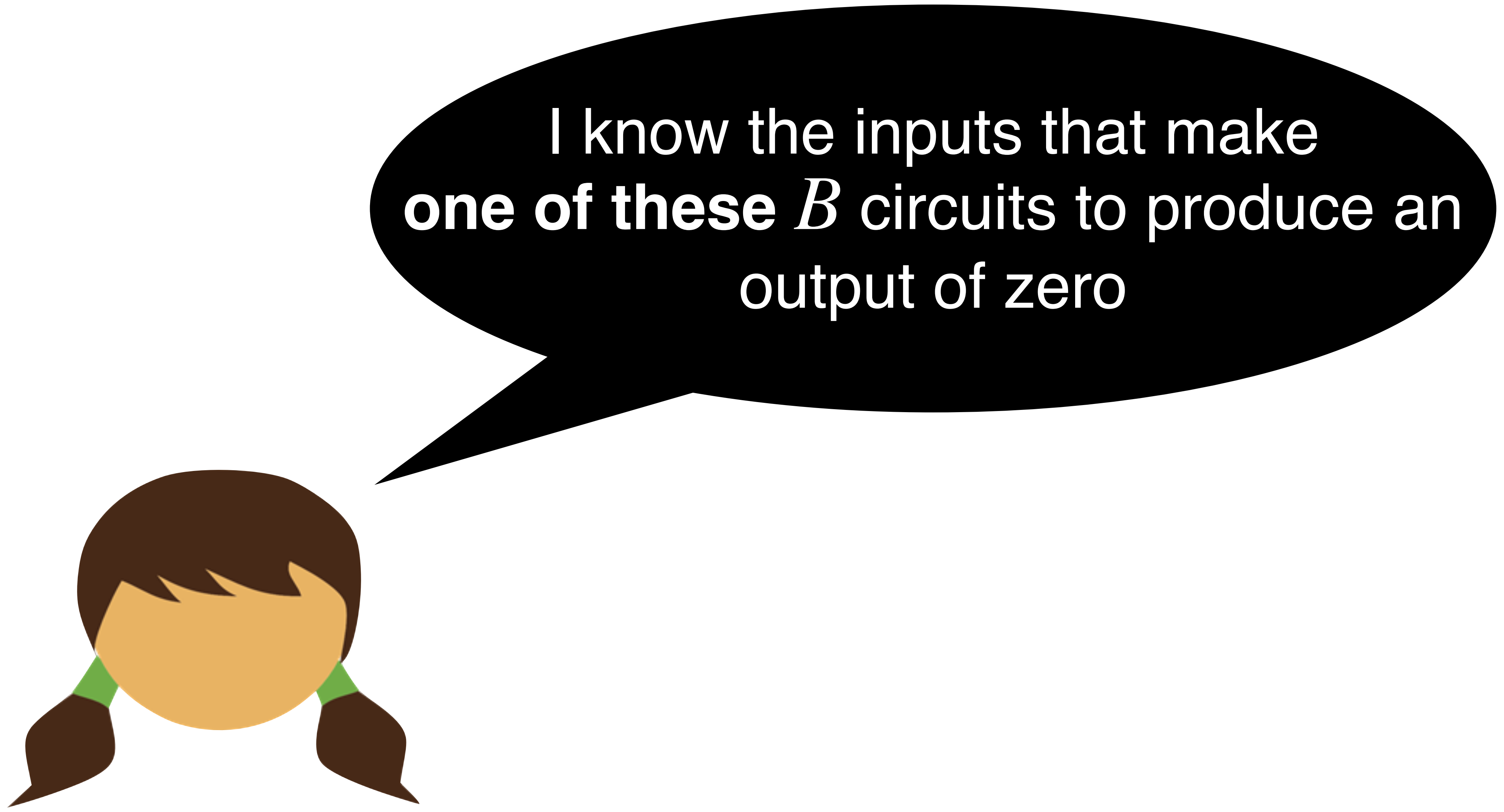
Cost: $>2|C|$

Batched Disjunctions

Containing B Branches and 1 Repetition

SHL
AND
LT
XOR
...



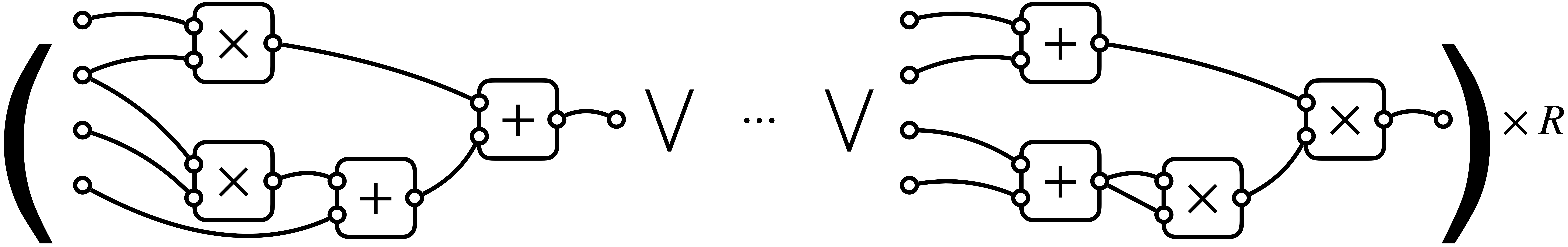

I know the inputs that make **one of these** B circuits to produce an output of zero

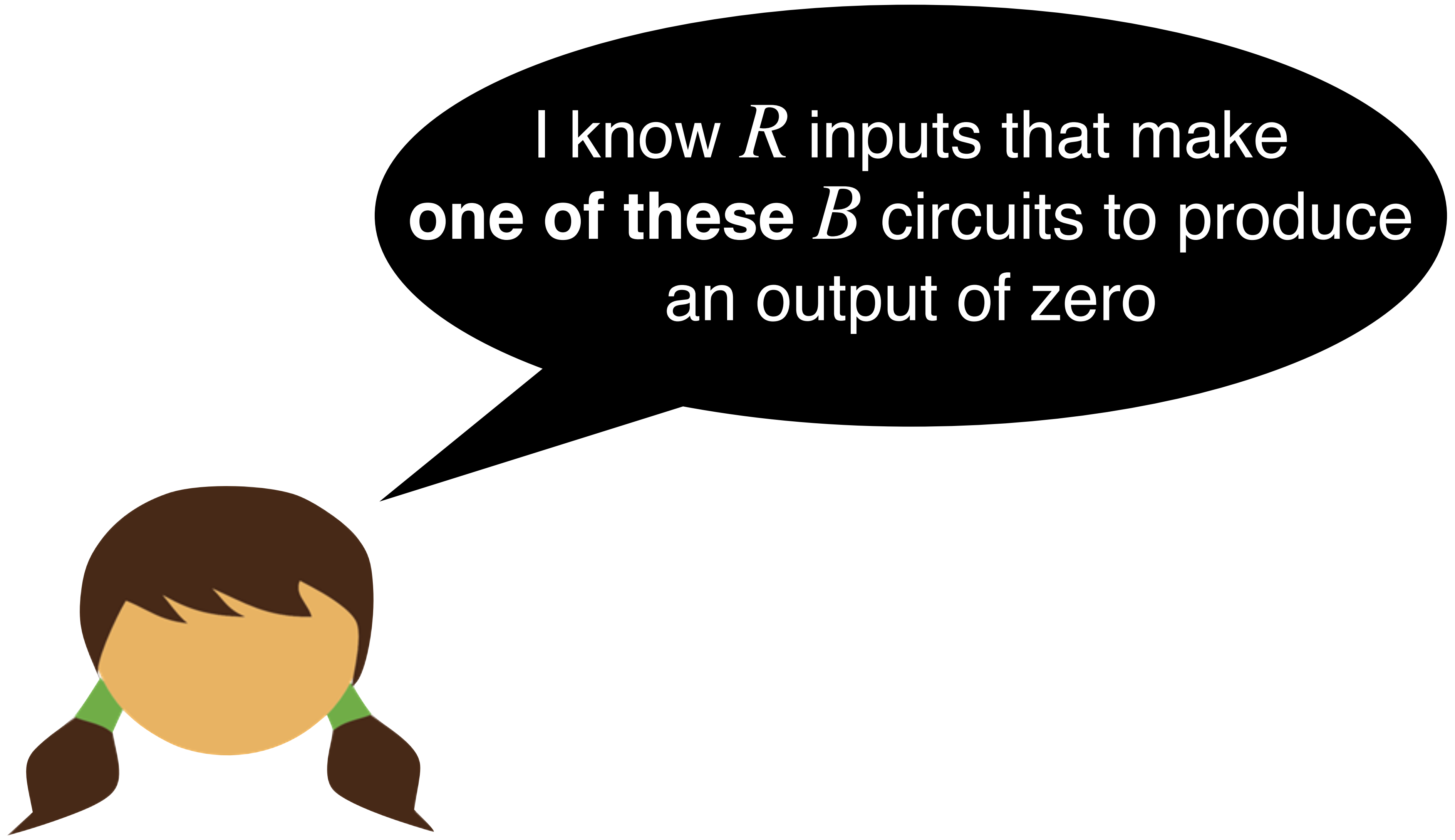
$$O(B|C|)$$

Batched Disjunctions

Containing B Branches and R Repetition

SHL
AND
LT
XOR
…
$\times R$





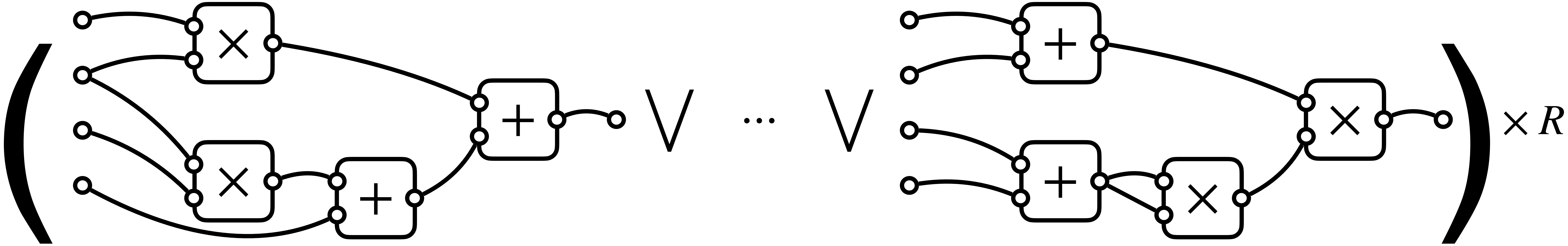
I know R inputs that make **one of these** B circuits to produce an output of zero

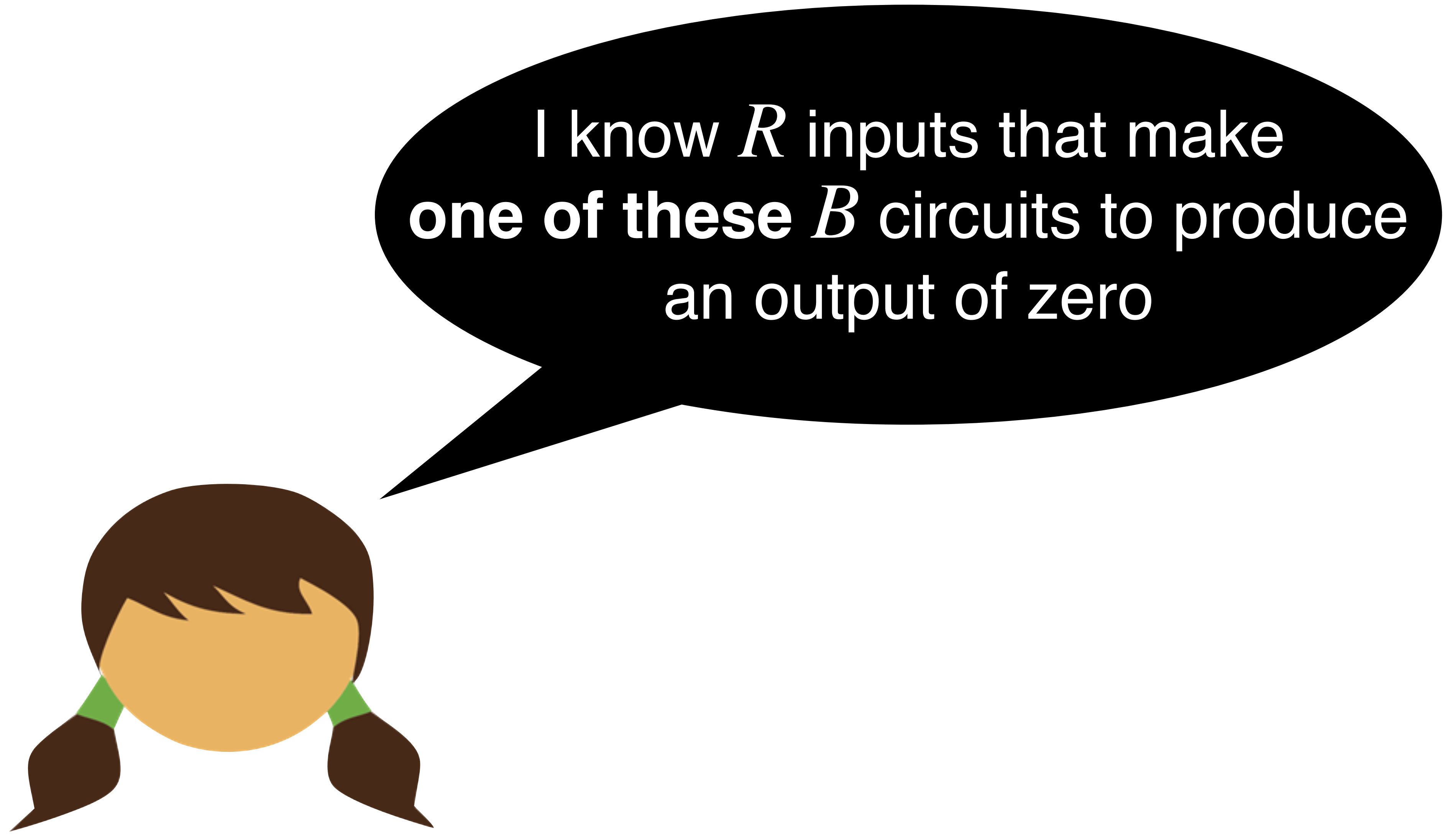
$$O(RB\,|\,C\,|\,)$$

Batched Disjunctions

Containing B Branches and R Repetition

SHL
AND
LT
XOR
…
$\times R$



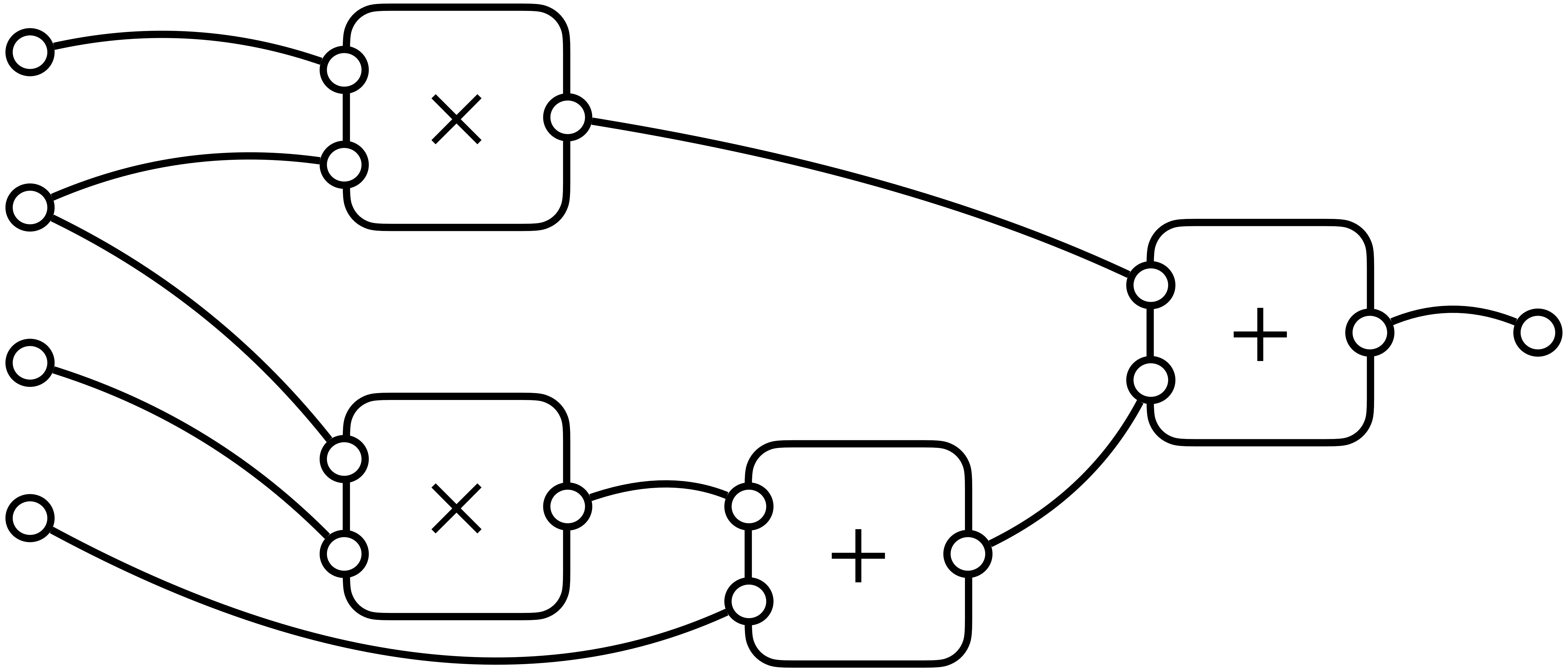


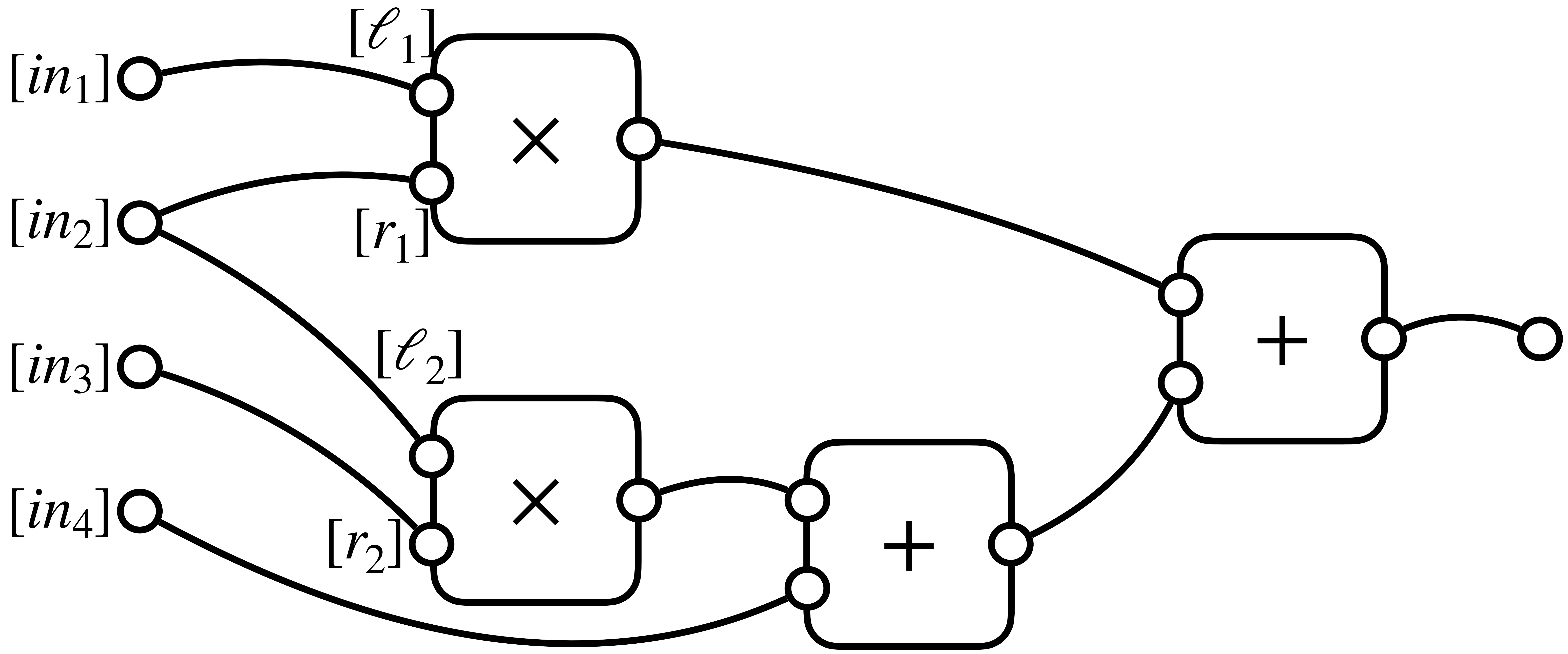
I know R inputs that make **one of these** B circuits to produce an output of zero

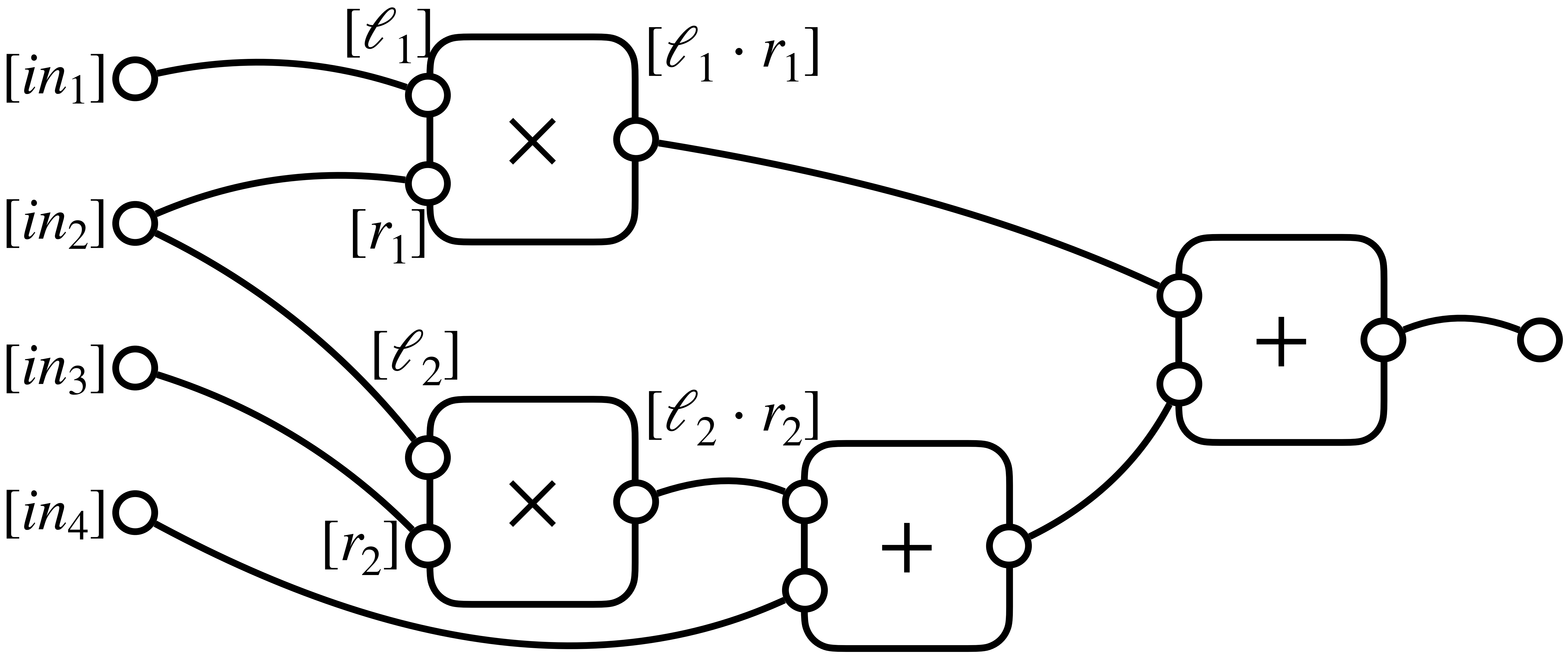
$$O(RB|C|)$$

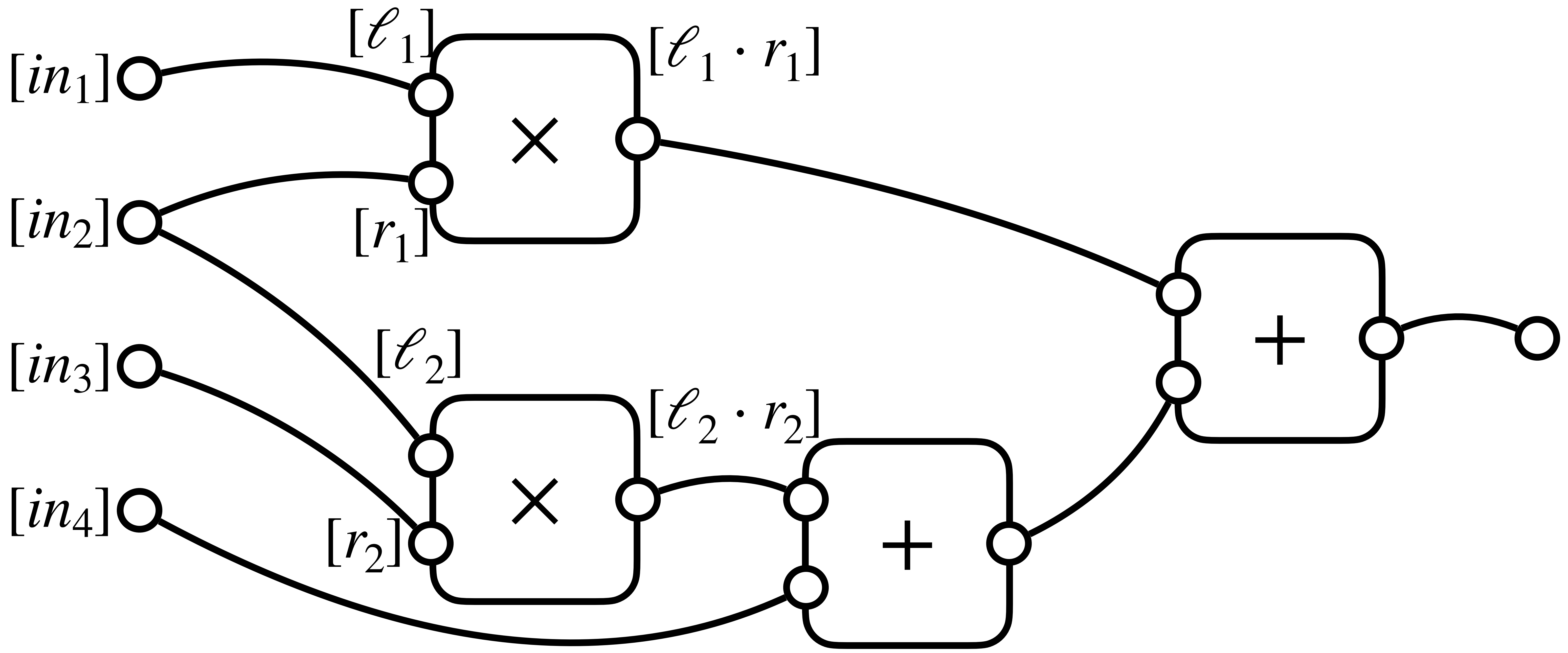
↓

$$O(B|C| + R|C|)$$



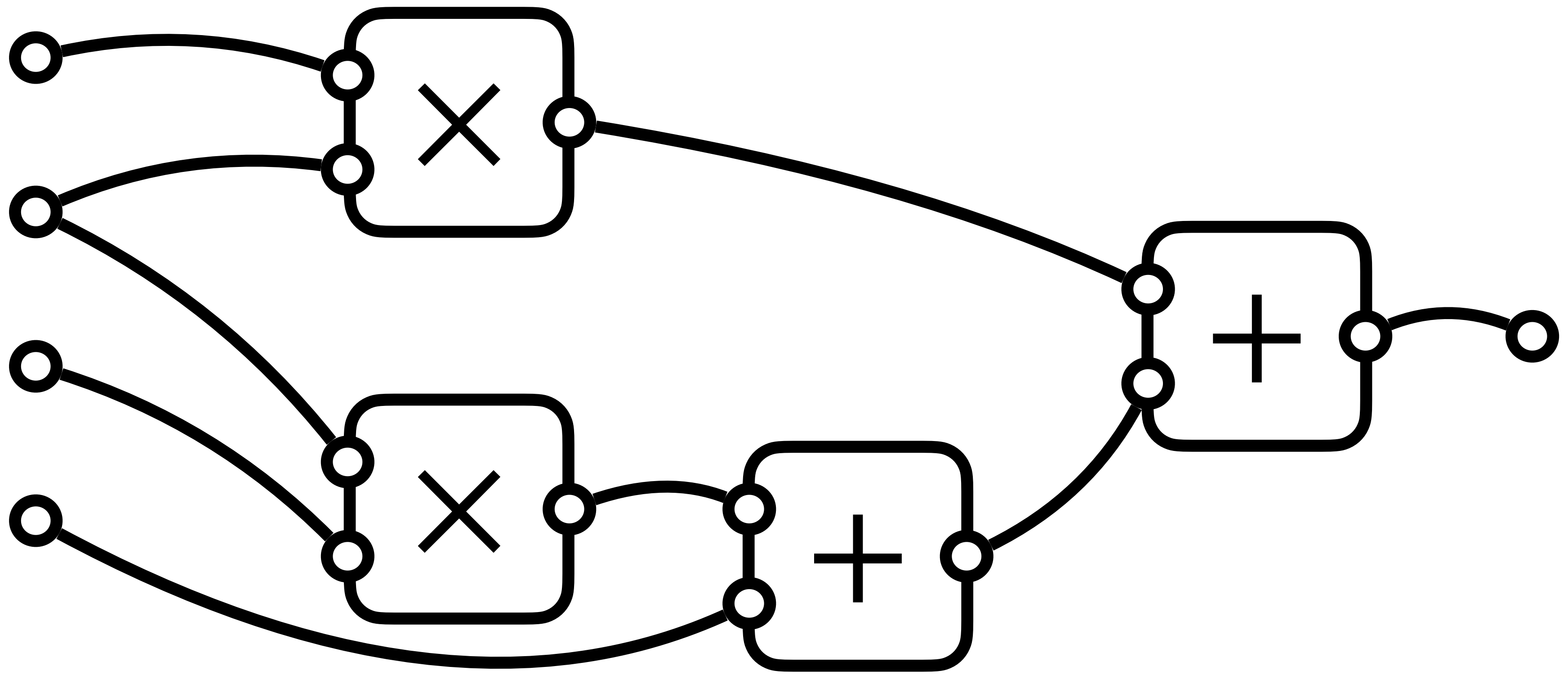

$[in_1]$
$[\ell_1]$
$\times$
$[in_2]$
$[r_1]$
$[in_3]$
$[\ell_2]$
$[in_4]$
$[r_2]$
$\times$
$+$
$+$


$[in_1]$
$[in_2]$
$[in_3]$
$[in_4]$
$[\ell_1]$
$[r_1]$
$[\ell_2]$
$[r_2]$
$\times$
$\times$
$[\ell_1 \cdot r_1]$
$[\ell_2 \cdot r_2]$
$+$
$+$



Topology vector: public and determined by the circuit

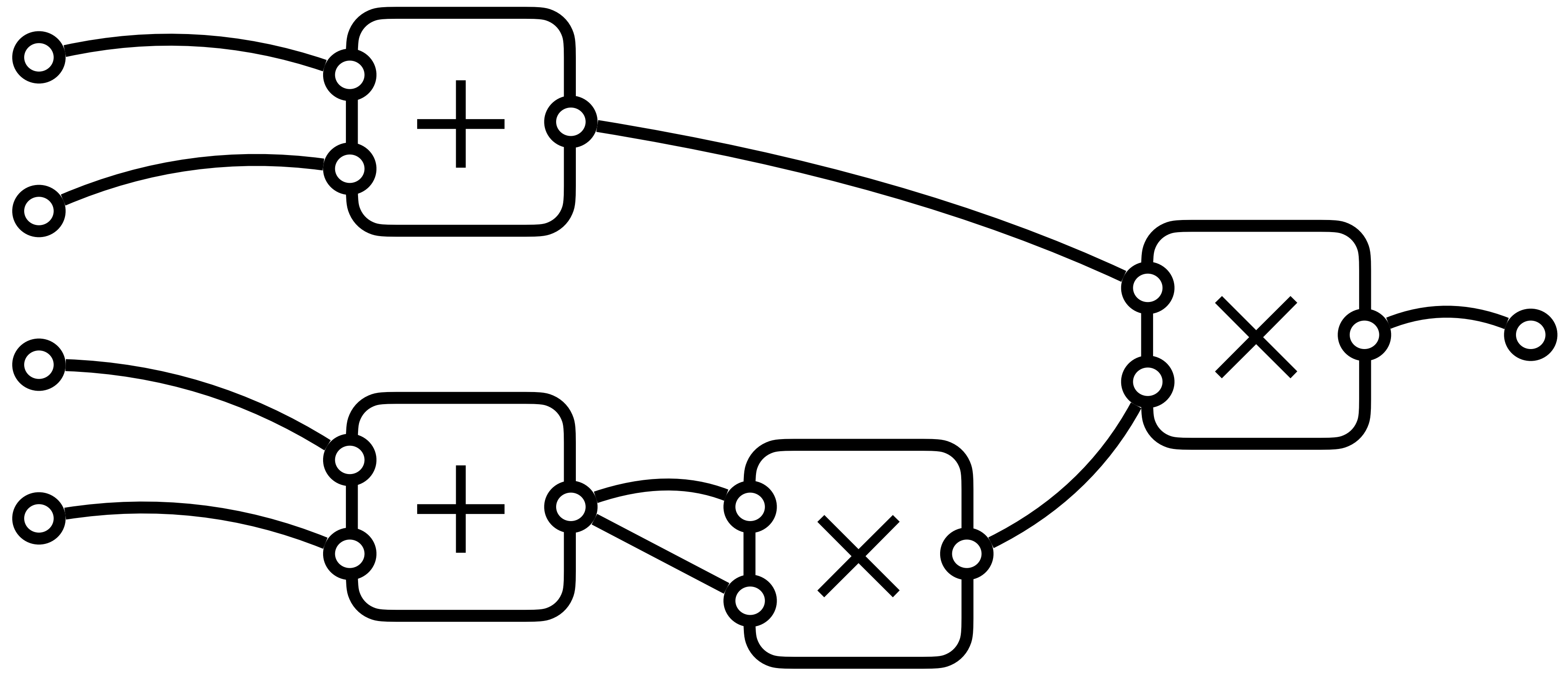
inner_product$(\overrightarrow{tv}, [in_1 \;\; in_2 \;\; in_3 \;\; in_4 \;\; \ell_1 \;\; r_1 \;\; \ell_1 r_1 \;\; \ell_2 \;\; r_2 \;\; \ell_2 r_2]) = [0]$



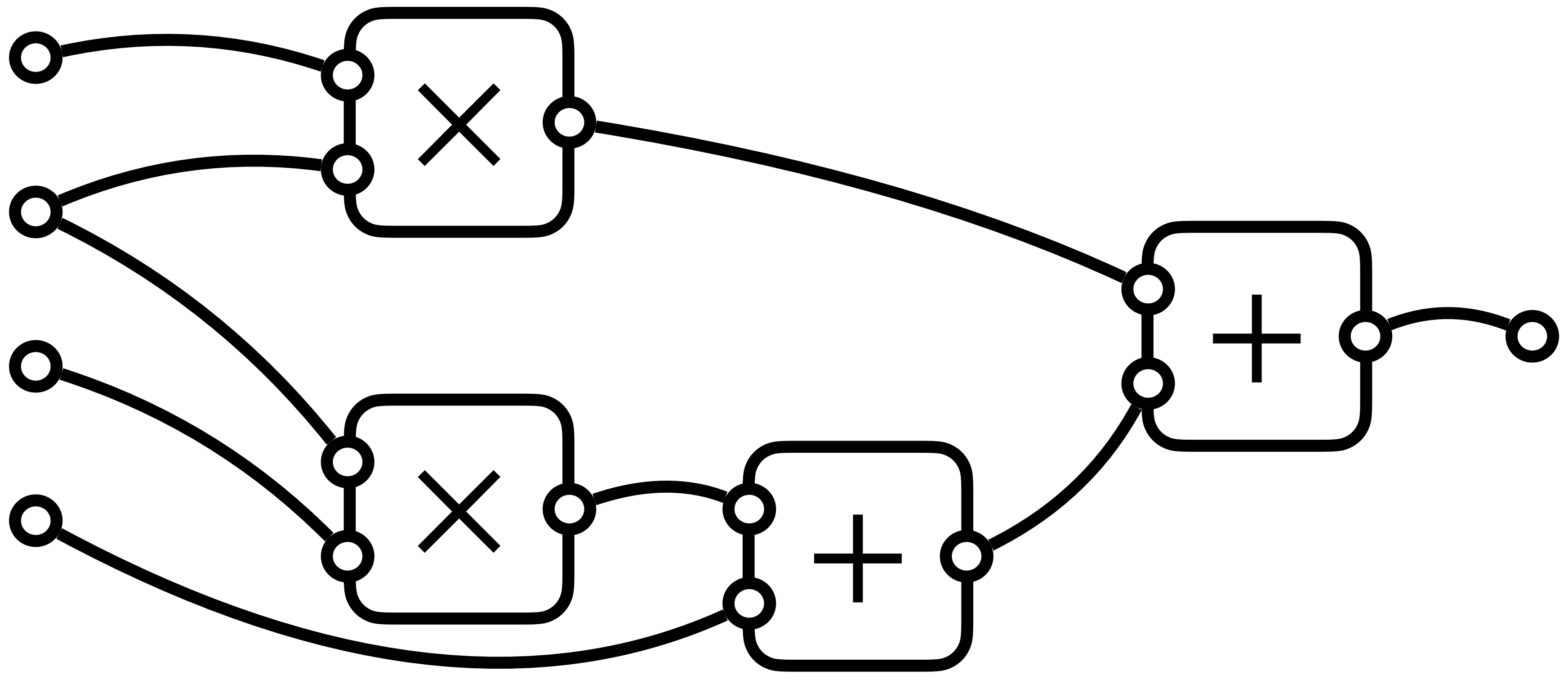
$\overrightarrow{tv_1}$

$\vee \quad \cdots \quad \vee$

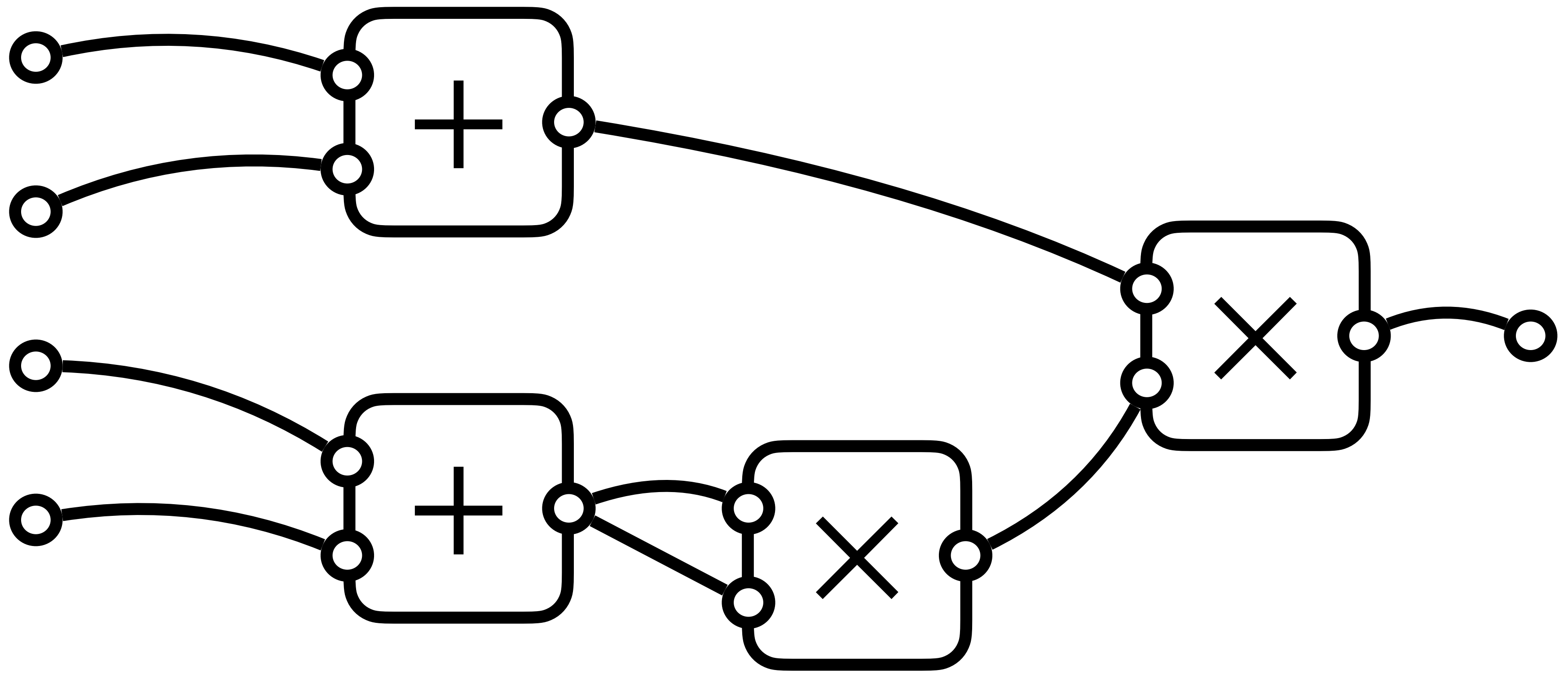
$\cdots$



$\overrightarrow{tv_B}$

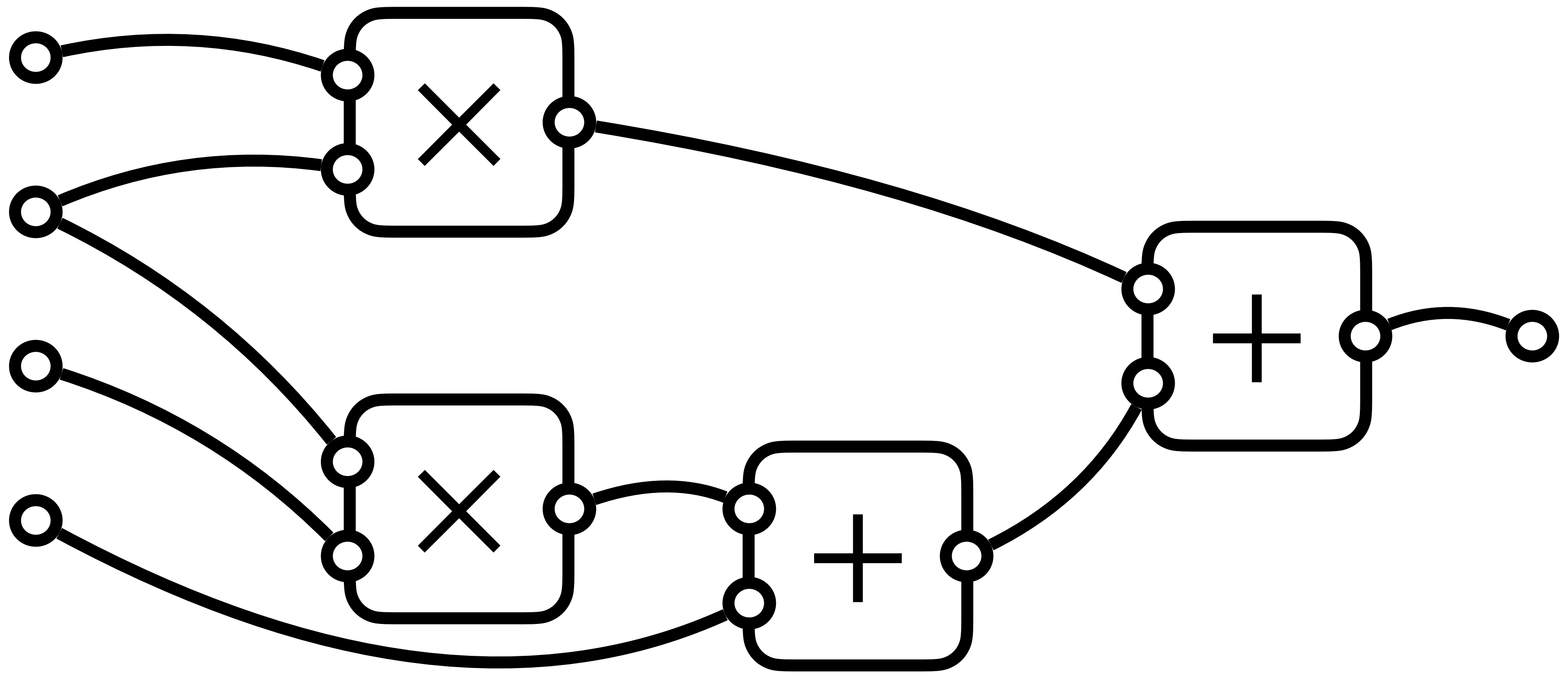


$\vee \quad \cdots \quad \vee$

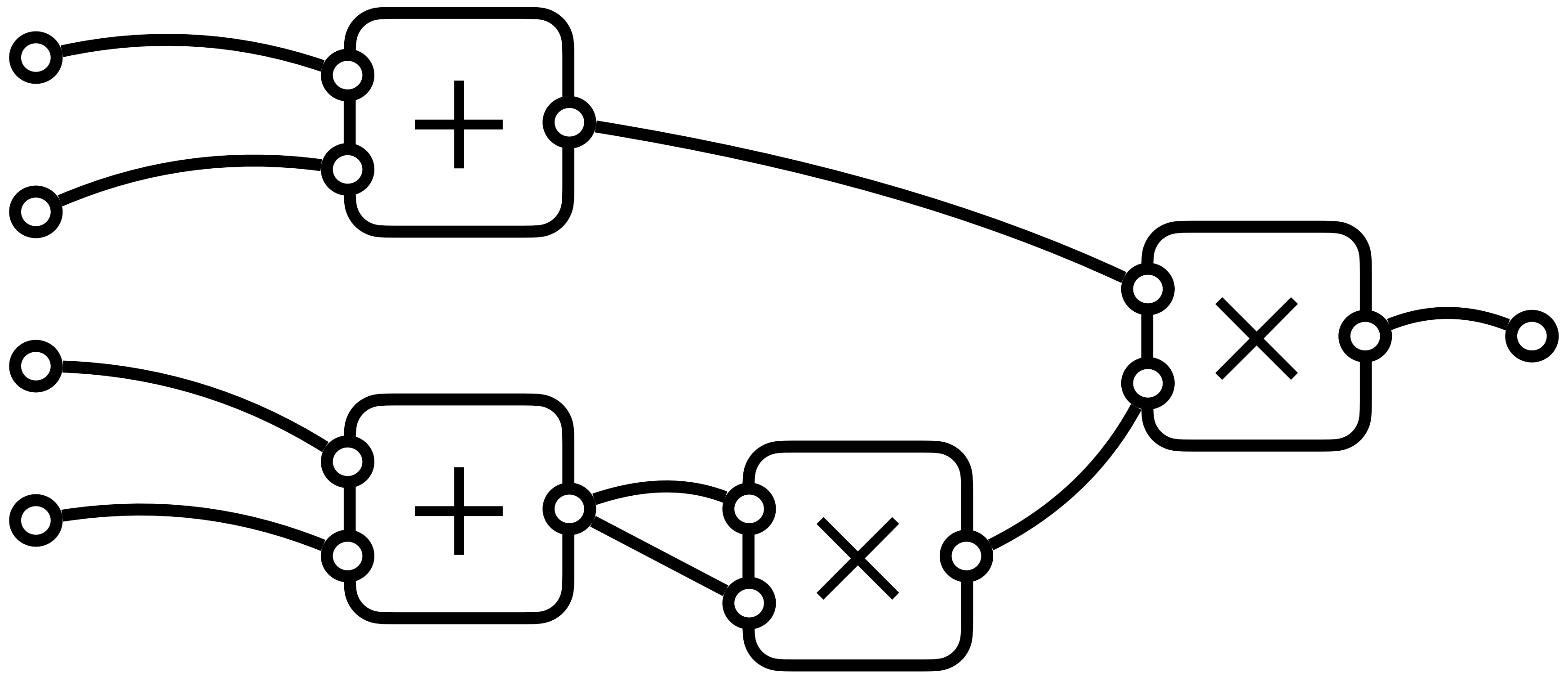


$\overrightarrow{tv_1}$ $\quad \cdots \quad$ $\overrightarrow{tv_B}$

$[in_1 \quad in_2 \quad in_3 \quad in_4 \quad \ell_1 \quad r_1 \quad \ell_1 r_1 \quad \ell_2 \quad r_2 \quad \ell_2 r_2]$

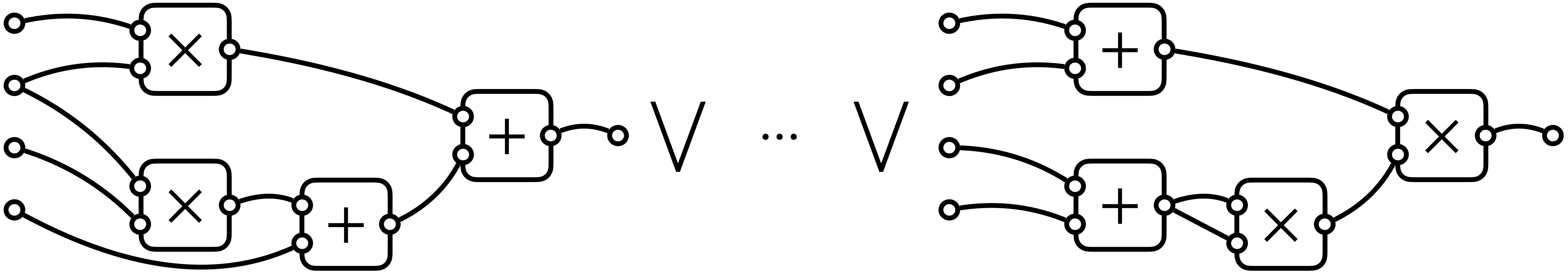


$\bigvee \quad \cdots \quad \bigvee$



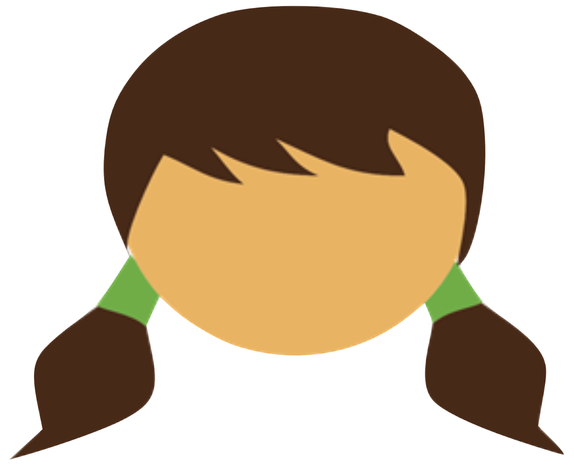
$\overrightarrow{tv_1} \qquad \cdots \qquad \overrightarrow{tv_B}$

$\exists i,$ inner_product($\overrightarrow{tv_i}$,$[in_1 \quad in_2 \quad in_3 \quad in_4 \quad \ell_1 \quad r_1 \quad \ell_1 r_1 \quad \ell_2 \quad r_2 \quad \ell_2 r_2]) = [0]$



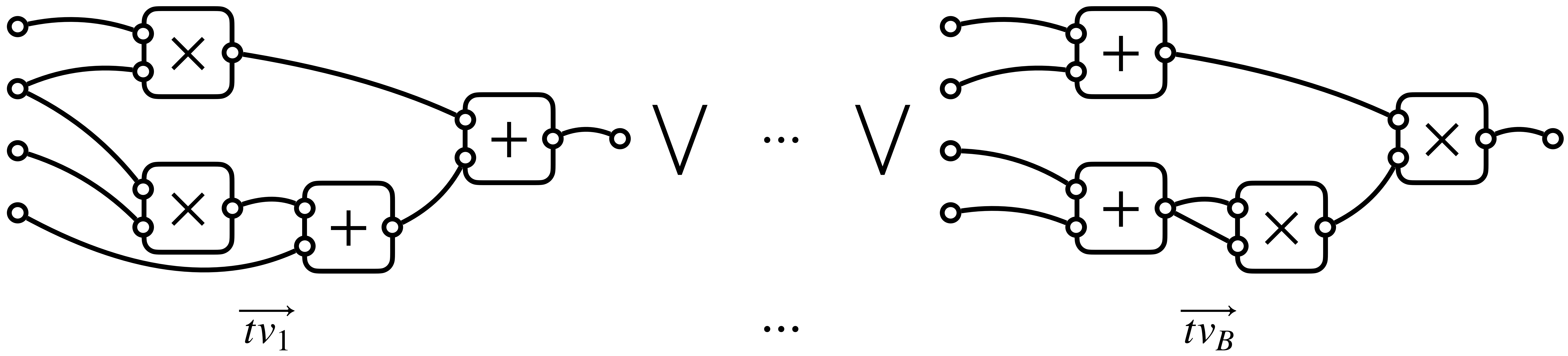
$\overrightarrow{tv_1}$ … $\overrightarrow{tv_B}$

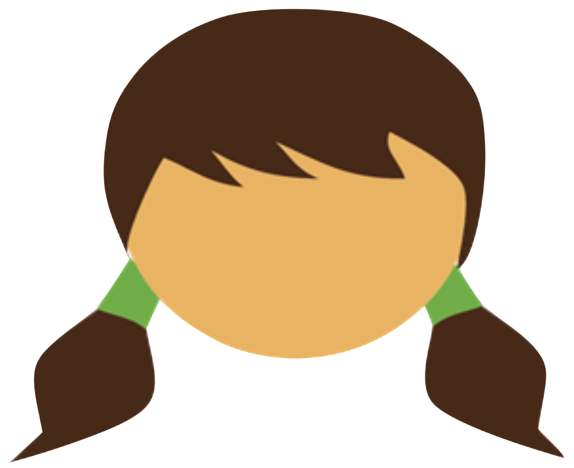
Tech. 2: P knows and helps



■ $[i]$
■

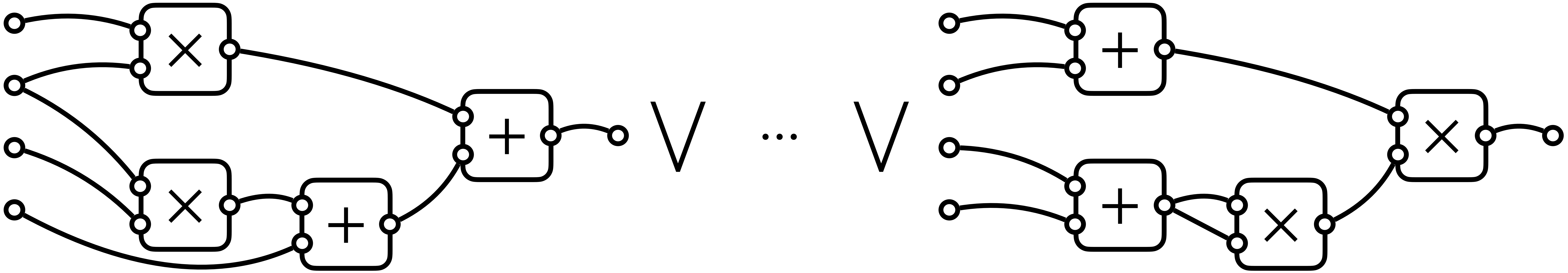
$\exists i,$ inner_product$(\overrightarrow{tv_i},[in_1\ \ in_2\ \ in_3\ \ in_4\ \ \ell_1\ \ r_1\ \ \ell_1 r_1\ \ \ell_2\ \ r_2\ \ \ell_2 r_2]) = [0]$





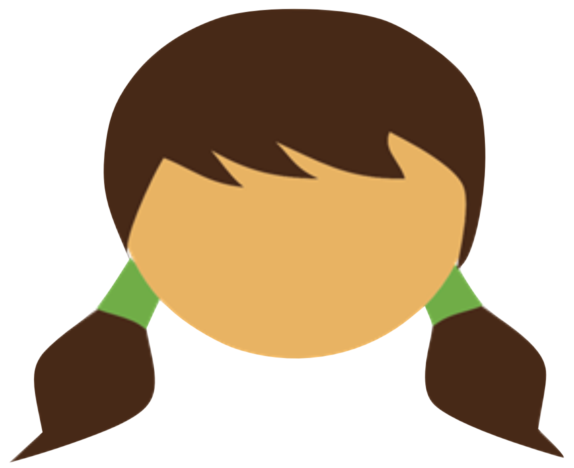
: $[i] \Rightarrow [\overrightarrow{tv_i}]$

inner_product($[\overrightarrow{tv_i}]$,$[in_1\ \ in_2\ \ in_3\ \ in_4\ \ \ell_1\ \ r_1\ \ \ell_1 r_1\ \ \ell_2\ \ r_2\ \ \ell_2 r_2]) = [0]$



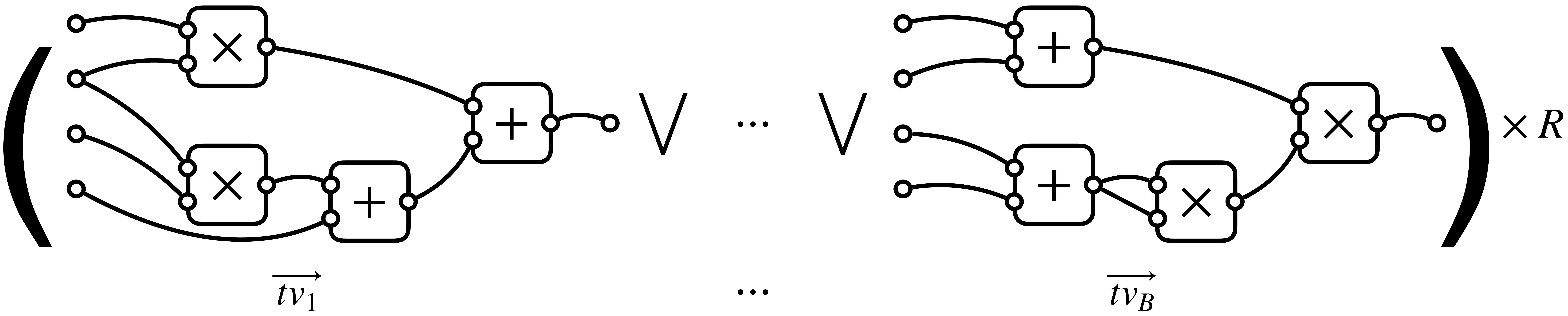
$\overrightarrow{tv_1}$ … $\overrightarrow{tv_B}$

ZK ROM

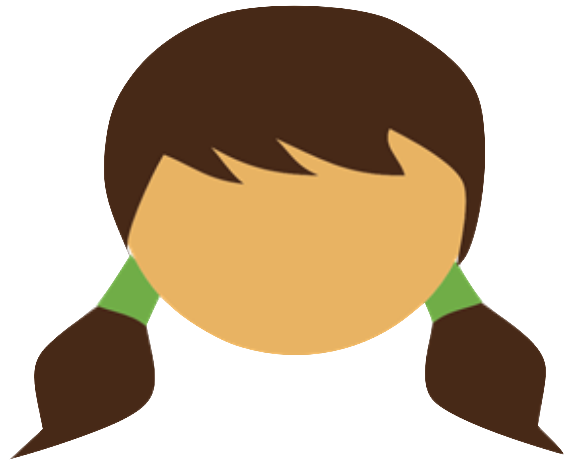


: $[i] \Rightarrow [\overrightarrow{tv_i}]$

inner_product($[\overrightarrow{tv_i}],[in_1 \quad in_2 \quad in_3 \quad in_4 \quad \ell_1 \quad r_1 \quad \ell_1 r_1 \quad \ell_2 \quad r_2 \quad \ell_2 r_2]) = [0]$

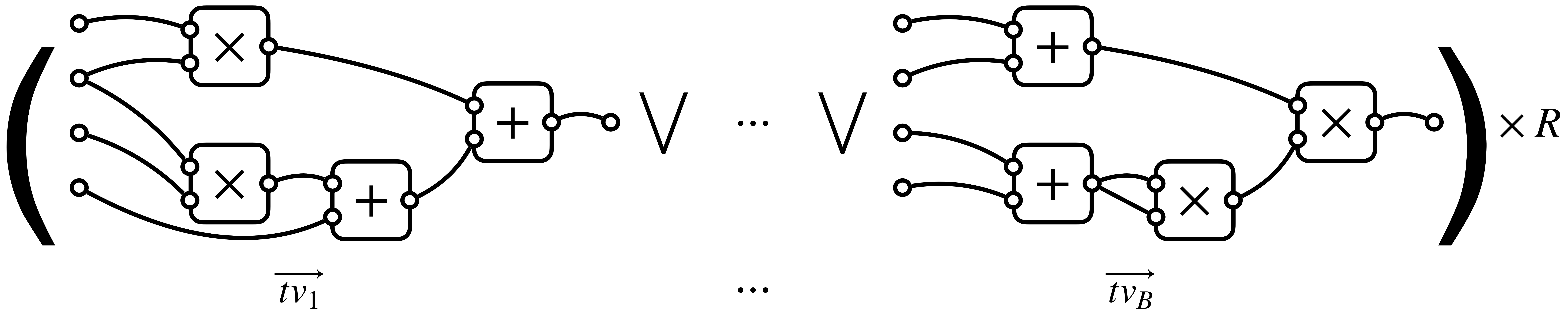


ZK ROM



: $[i]$ ➡ $[\overrightarrow{tv_i}]$

inner_product($[\overrightarrow{tv_i}]$,$[in_1 \;\; in_2 \;\; in_3 \;\; in_4 \;\; \ell_1 \;\; r_1 \;\; \ell_1 r_1 \;\; \ell_2 \;\; r_2 \;\; \ell_2 r_2]) = [0]$



ZK ROM

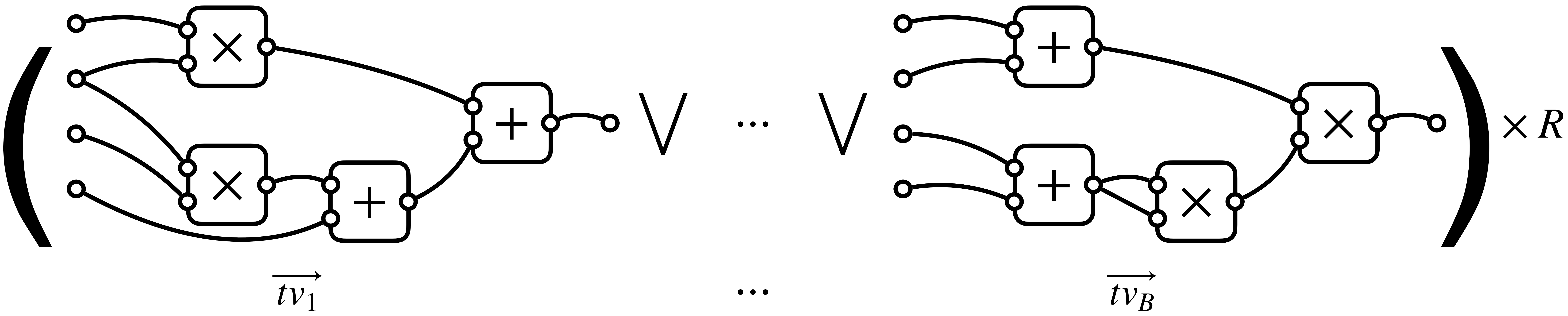
: $[i_1][i_2]\cdots[i_R] \Rightarrow [\overrightarrow{tv_{i_1}}][\overrightarrow{tv_{i_2}}]\cdots[\overrightarrow{tv_{i_R}}]$

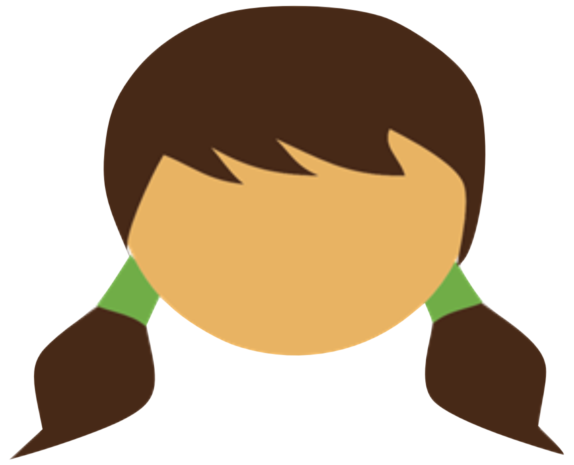
$$\text{inner\_product}([\overrightarrow{tv_{i_1}}],[in_1^{(1)}\ \ in_2^{(1)}\ \ in_3^{(1)}\ \ in_4^{(1)}\ \ \ell_1^{(1)}\ \ r_1^{(1)}\ \ \ell_1^{(1)}r_1^{(1)}\ \ \ell_2^{(1)}\ \ r_2^{(1)}\ \ \ell_2^{(1)}r_2^{(1)}]) = [0]$$

$$\text{inner\_product}([\overrightarrow{tv_{i_2}}],[in_1^{(2)}\ \ in_2^{(2)}\ \ in_3^{(2)}\ \ in_4^{(2)}\ \ \ell_1^{(2)}\ \ r_1^{(2)}\ \ \ell_1^{(2)}r_1^{(2)}\ \ \ell_2^{(2)}\ \ r_2^{(2)}\ \ \ell_2^{(2)}r_2^{(2)}]) = [0]$$

$$\cdots$$

$$\text{inner\_product}([\overrightarrow{tv_{i_R}}],[in_1^{(R)}\ \ in_2^{(R)}\ \ in_3^{(R)}\ \ in_4^{(R)}\ \ \ell_1^{(R)}\ \ r_1^{(R)}\ \ \ell_1^{(R)}r_1^{(R)}\ \ \ell_2^{(R)}\ \ r_2^{(R)}\ \ \ell_2^{(R)}r_2^{(R)}]) = [0]$$





ZK ROM

: $[i_1][i_2]\cdots[i_R]$ ➡ $[\overrightarrow{tv_{i_1}}][\overrightarrow{tv_{i_2}}]\cdots[\overrightarrow{tv_{i_R}}]$

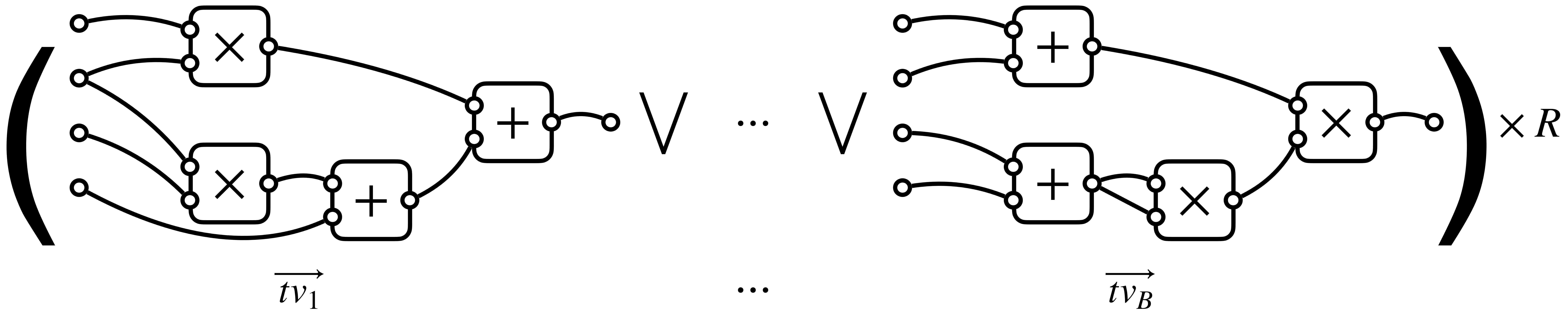
Total cost: $O(Nm + Tm)$ for T accesses and length-m vectors

$$\text{inner\_product}([\overrightarrow{tv_{i_1}}], [in_1^{(1)} \quad in_2^{(1)} \quad in_3^{(1)} \quad in_4^{(1)} \quad \ell_1^{(1)} \quad r_1^{(1)} \quad \ell_1^{(1)} r_1^{(1)} \quad \ell_2^{(1)} \quad r_2^{(1)} \quad \ell_2^{(1)} r_2^{(1)}]) = [0]$$

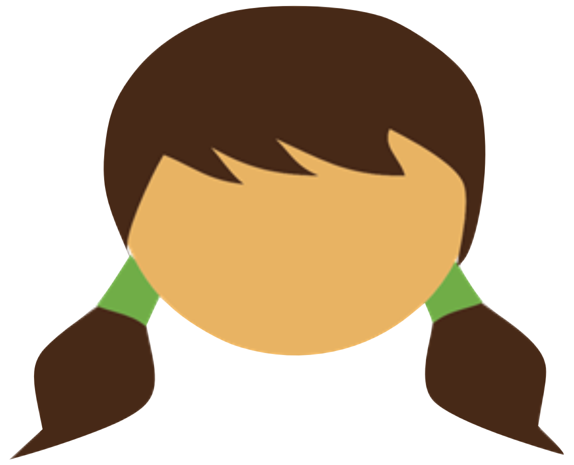
$$\text{inner\_product}([\overrightarrow{tv_{i_2}}], [in_1^{(2)} \quad in_2^{(2)} \quad in_3^{(2)} \quad in_4^{(2)} \quad \ell_1^{(2)} \quad r_1^{(2)} \quad \ell_1^{(2)} r_1^{(2)} \quad \ell_2^{(2)} \quad r_2^{(2)} \quad \ell_2^{(2)} r_2^{(2)}]) = [0]$$

$$\cdots$$

$$\text{inner\_product}([\overrightarrow{tv_{i_R}}], [in_1^{(R)} \quad in_2^{(R)} \quad in_3^{(R)} \quad in_4^{(R)} \quad \ell_1^{(R)} \quad r_1^{(R)} \quad \ell_1^{(R)} r_1^{(R)} \quad \ell_2^{(R)} \quad r_2^{(R)} \quad \ell_2^{(R)} r_2^{(R)}]) = [0]$$



ZK ROM



: $[i_1][i_2]\cdots[i_R]$ ➡ $[\overrightarrow{tv_{i_1}}][\overrightarrow{tv_{i_2}}]\cdots[\overrightarrow{tv_{i_R}}]$

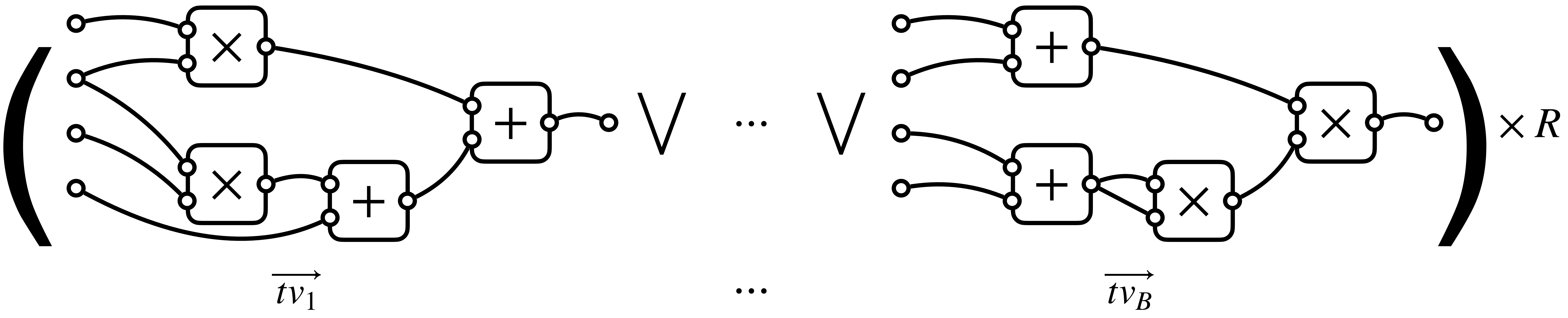
$O(B|C| + R|C|)$

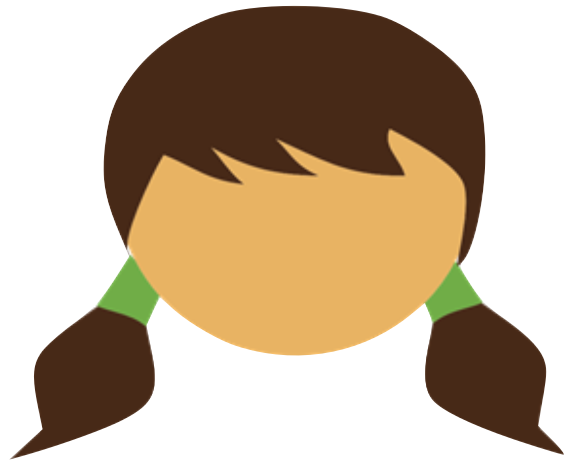
inner_product($[\overrightarrow{tv_{i_1}}], [in_1^{(1)} \quad in_2^{(1)} \quad in_3^{(1)} \quad in_4^{(1)} \quad \ell_1^{(1)} \quad r_1^{(1)} \quad \ell_1^{(1)}r_1^{(1)} \quad \ell_2^{(1)} \quad r_2^{(1)} \quad \ell_2^{(1)}r_2^{(1)}]) = [0]$

inner_product($[\overrightarrow{tv_{i_2}}], [in_1^{(2)} \quad in_2^{(2)} \quad in_3^{(2)} \quad in_4^{(2)} \quad \ell_1^{(2)} \quad r_1^{(2)} \quad \ell_1^{(2)}r_1^{(2)} \quad \ell_2^{(2)} \quad r_2^{(2)} \quad \ell_2^{(2)}r_2^{(2)}]) = [0]$

…

inner_product($[\overrightarrow{tv_{i_R}}], [in_1^{(R)} \quad in_2^{(R)} \quad in_3^{(R)} \quad in_4^{(R)} \quad \ell_1^{(R)} \quad r_1^{(R)} \quad \ell_1^{(R)}r_1^{(R)} \quad \ell_2^{(R)} \quad r_2^{(R)} \quad \ell_2^{(R)}r_2^{(R)}]) = [0]$





$$: \; [i_1][i_2]\cdots[i_R] \xrightarrow{\textbf{ZK ROM}} [\overrightarrow{tv_{i_1}}][\overrightarrow{tv_{i_2}}]\cdots[\overrightarrow{tv_{i_R}}]$$

$O(B|C|+R|C|)$

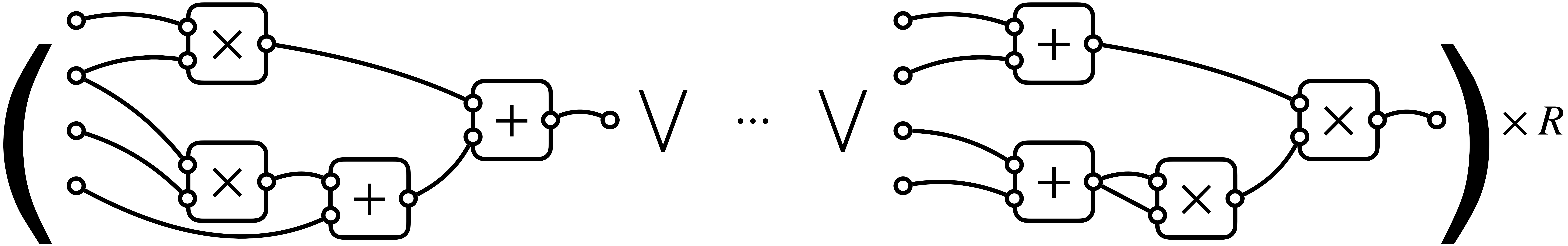
$$\text{inner\_product}([\overrightarrow{tv_{i_1}}], [in_1^{(1)} \;\; in_2^{(1)} \;\; in_3^{(1)} \;\; in_4^{(1)} \;\; \ell_1^{(1)} \;\; r_1^{(1)} \;\; \ell_1^{(1)} r_1^{(1)} \;\; \ell_2^{(1)} \;\; r_2^{(1)} \;\; \ell_2^{(1)} r_2^{(1)}]) = [0]$$

$$\text{inner\_product}([\overrightarrow{tv_{i_2}}], [in_1^{(2)} \;\; in_2^{(2)} \;\; in_3^{(2)} \;\; in_4^{(2)} \;\; \ell_1^{(2)} \;\; r_1^{(2)} \;\; \ell_1^{(2)} r_1^{(2)} \;\; \ell_2^{(2)} \;\; r_2^{(2)} \;\; \ell_2^{(2)} r_2^{(2)}]) = [0]$$

$O(R|C|)$

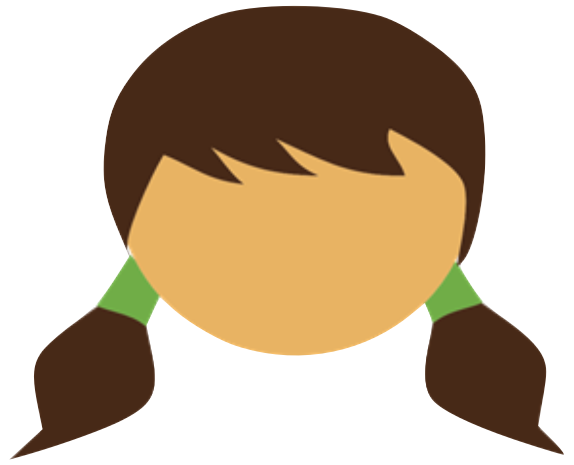
$$\cdots$$

$$\text{inner\_product}([\overrightarrow{tv_{i_R}}], [in_1^{(R)} \;\; in_2^{(R)} \;\; in_3^{(R)} \;\; in_4^{(R)} \;\; \ell_1^{(R)} \;\; r_1^{(R)} \;\; \ell_1^{(R)} r_1^{(R)} \;\; \ell_2^{(R)} \;\; r_2^{(R)} \;\; \ell_2^{(R)} r_2^{(R)}]) = [0]$$



$\overrightarrow{tv_1}$ $\cdots$ $\overrightarrow{tv_B}$ $O(B|C|)$

ZK ROM



: $[i_1][i_2]\cdots[i_R]$ ➡ $[\overrightarrow{tv_{i_1}}][\overrightarrow{tv_{i_2}}]\cdots[\overrightarrow{tv_{i_R}}]$ $O(B|C|+R|C|)$

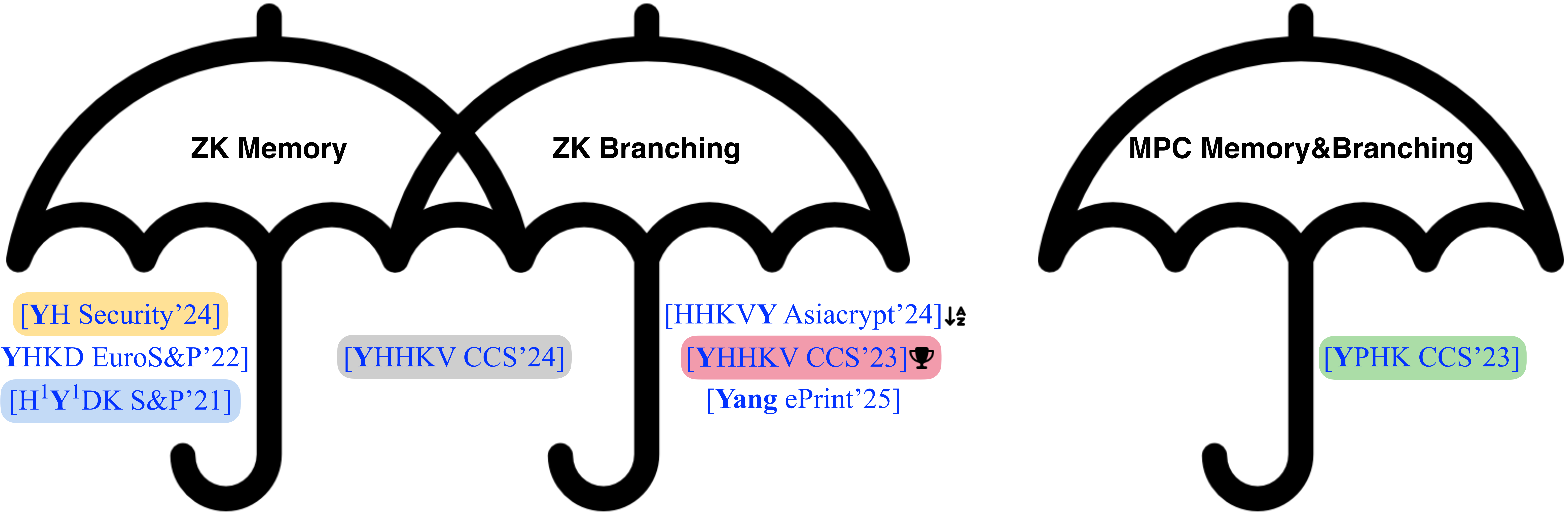
$$\text{inner\_product}([\overrightarrow{tv_{i_1}}],[in_1^{(1)}\ \ in_2^{(1)}\ \ in_3^{(1)}\ \ in_4^{(1)}\ \ \ell_1^{(1)}\ \ r_1^{(1)}\ \ \ell_1^{(1)}r_1^{(1)}\ \ \ell_2^{(1)}\ \ r_2^{(1)}\ \ \ell_2^{(1)}r_2^{(1)}])=[0]$$

$$\text{inner\_product}([\overrightarrow{tv_{i_2}}],[in_1^{(2)}\ \ in_2^{(2)}\ \ in_3^{(2)}\ \ in_4^{(2)}\ \ \ell_1^{(2)}\ \ r_1^{(2)}\ \ \ell_1^{(2)}r_1^{(2)}\ \ \ell_2^{(2)}\ \ r_2^{(2)}\ \ \ell_2^{(2)}r_2^{(2)}])=[0]$$

$O(R|C|)$

$$\cdots$$

$$\text{inner\_product}([\overrightarrow{tv_{i_R}}],[in_1^{(R)}\ \ in_2^{(R)}\ \ in_3^{(R)}\ \ in_4^{(R)}\ \ \ell_1^{(R)}\ \ r_1^{(R)}\ \ \ell_1^{(R)}r_1^{(R)}\ \ \ell_2^{(R)}\ \ r_2^{(R)}\ \ \ell_2^{(R)}r_2^{(R)}])=[0]$$



ZK Memory

[**Y**H Security'24]

[**Y**HKD EuroS&P'22]

[H^1**Y**^{1}DK S&P'21]

[**Y**HHKV CCS'24]

ZK Branching

[HHKV**Y** Asiacrypt'24]↓

[**Y**HHKV CCS'23]🏆

[**Yang** ePrint'25]

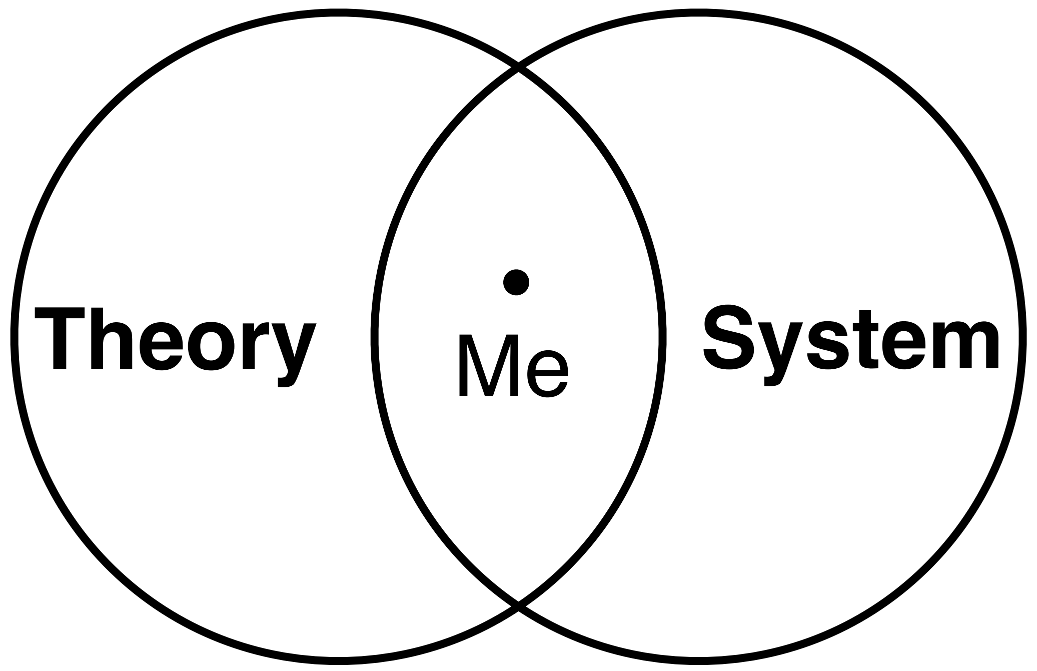
MPC Memory&Branching

[**Y**PHK CCS'23]

1. A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈ 10KHz ($\approx$**1000x**)

2. A zero-knowledge (ZK) read-write memory achieving optimal complexity

3. A zero-knowledge (ZK) branching protocol achieving optimal complexity



↓ Alphabetic Order

🏆 Distinguished Paper Award

✻ Co-first Authorship

Future Work

Crypto VM

KHz

My PhD

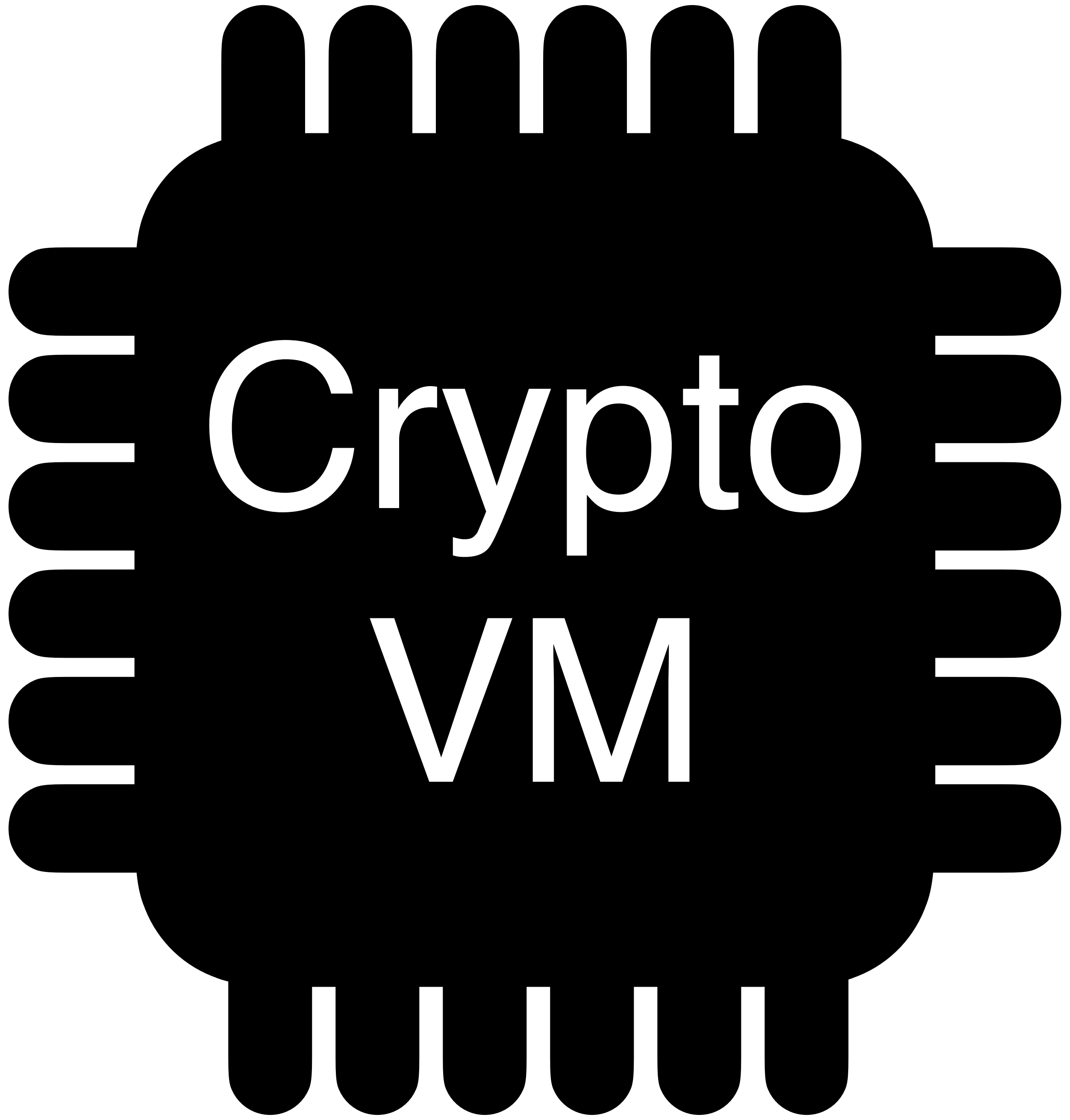
Past

Hz

Future Work

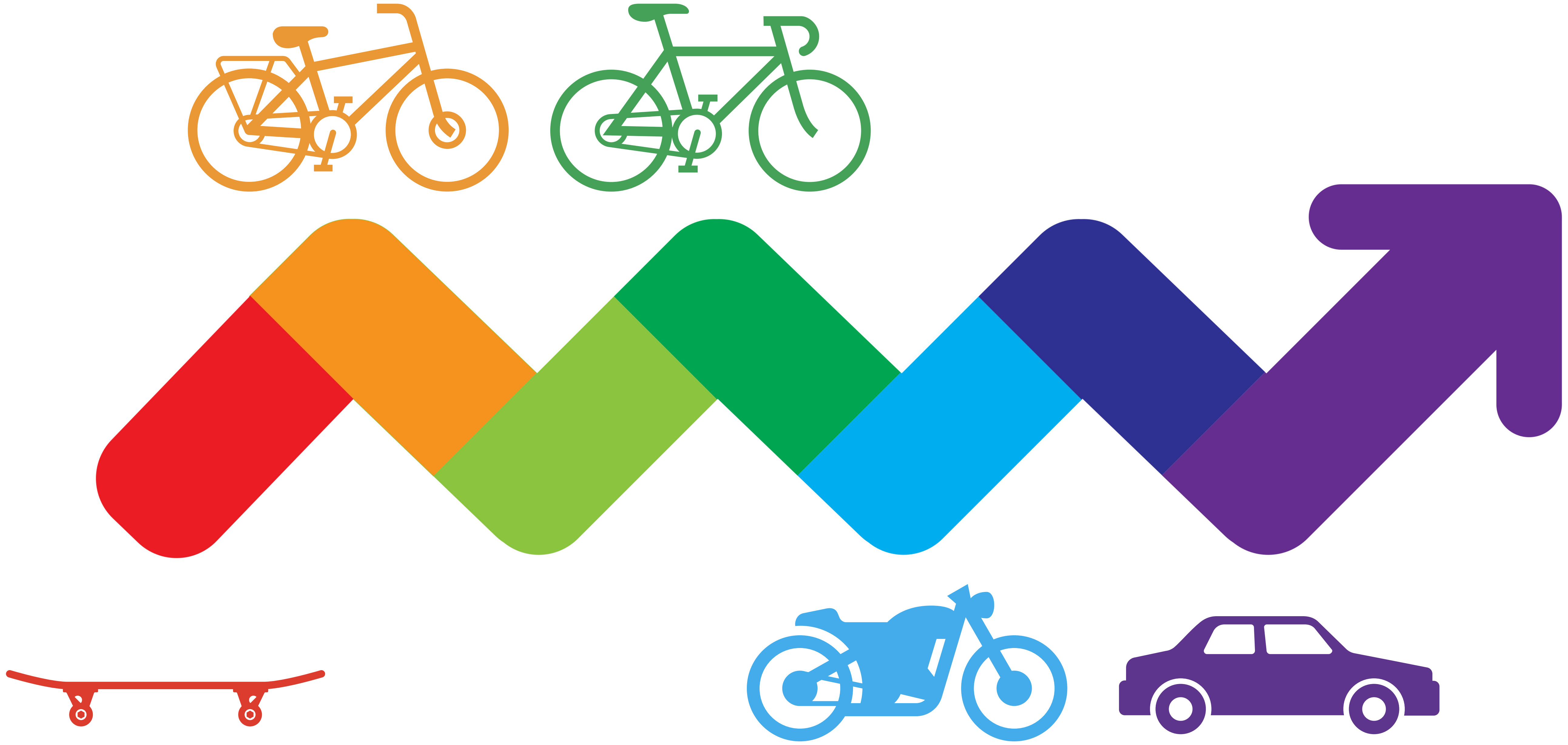
MHz

GHz



KHz

My PhD



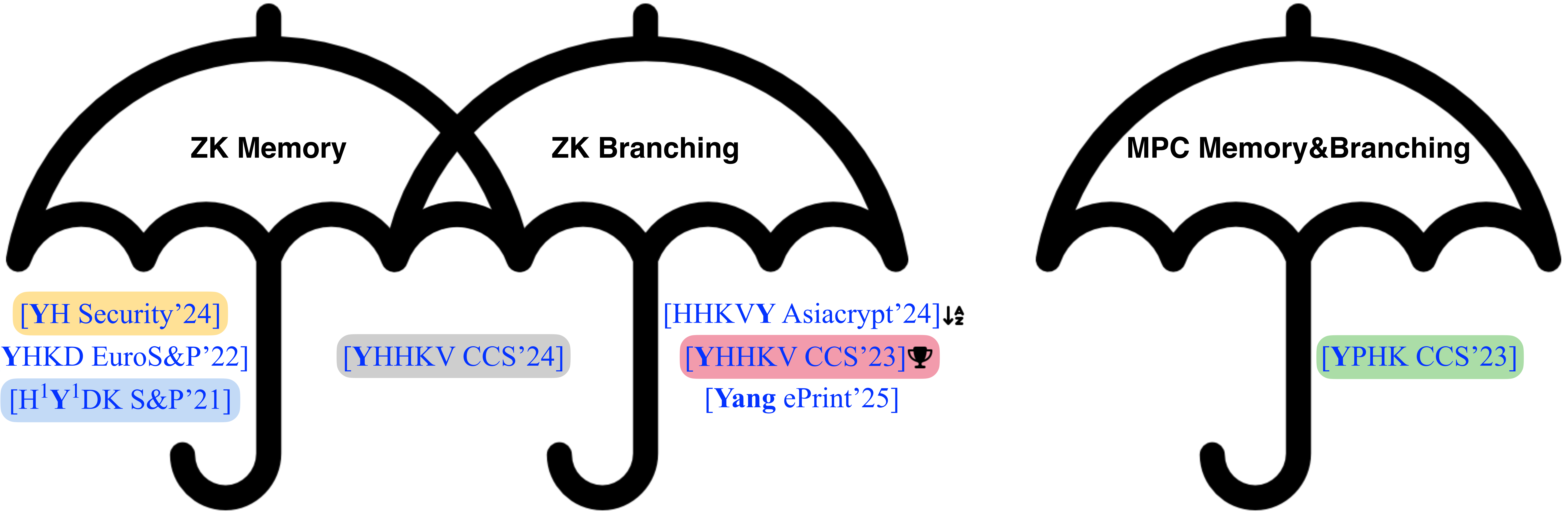
Past

Future Work

Hz

MHz

GHz



ZK Memory

ZK Branching

MPC Memory&Branching

[**Y**H Security'24]

[**Y**HKD EuroS&P'22]

[H^{1}**Y**^{1}DK S&P'21]

[**Y**HHKV CCS'24]

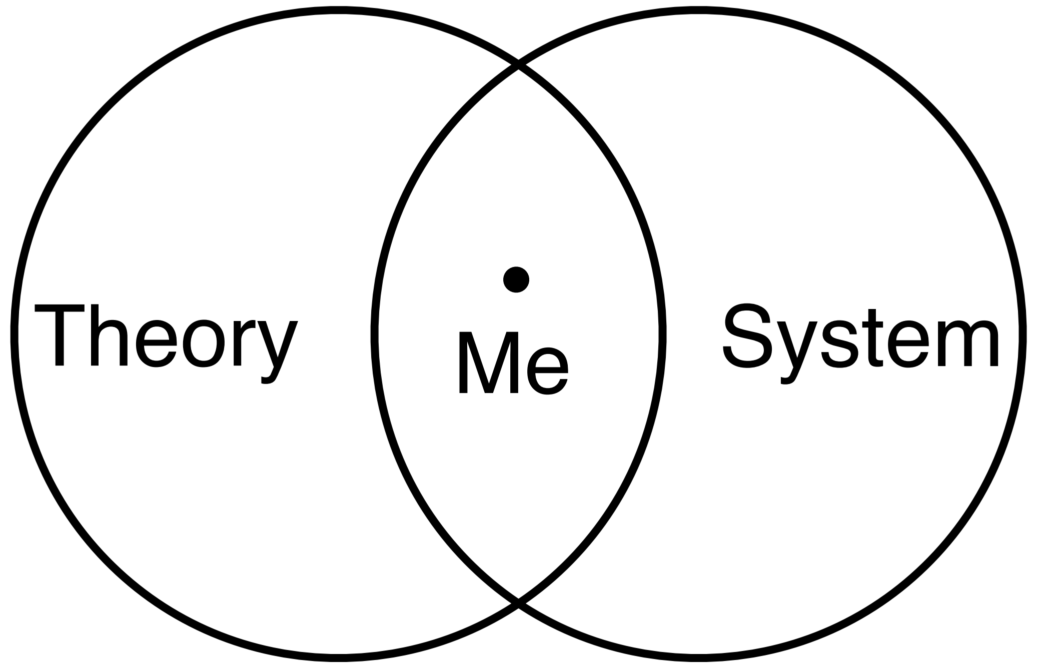
[HHKV**Y** Asiacrypt'24]↓AZ

[**Y**HHKV CCS'23]🏆

[**Yang** ePrint'25]

[**Y**PHK CCS'23]

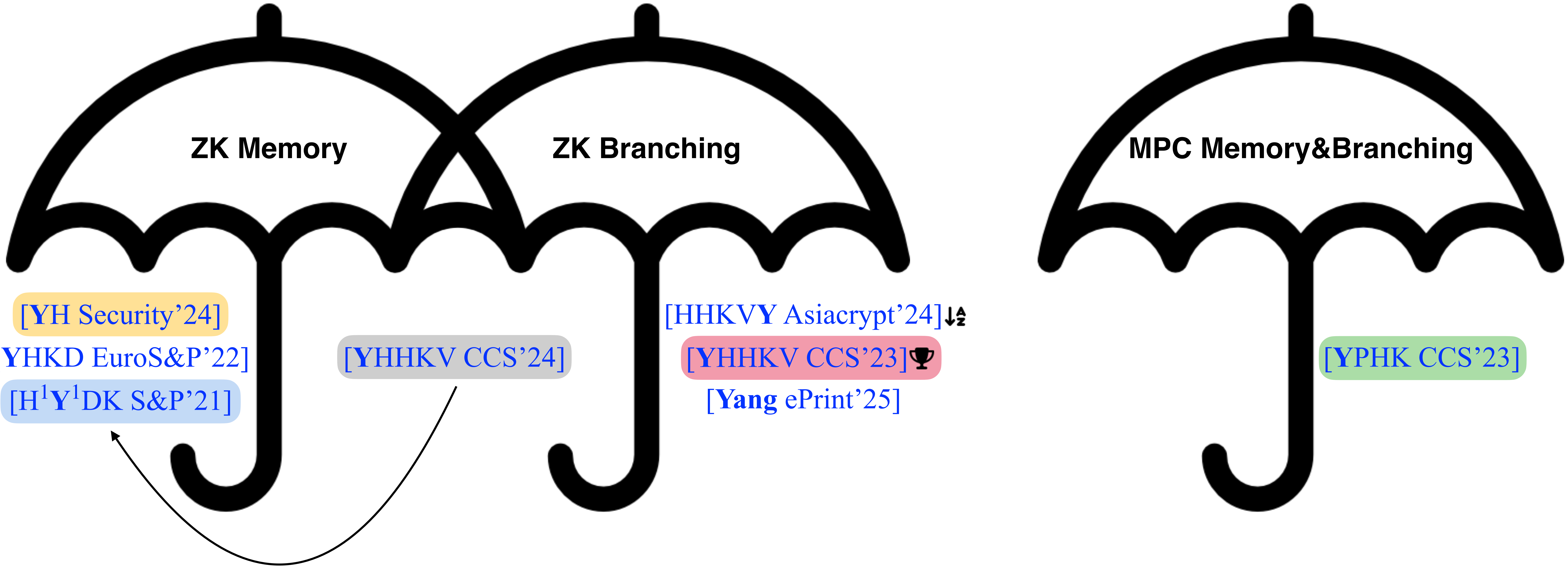
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈1KHz (≈**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity
- A zero-knowledge (ZK) CPU+RAM achieving optimal complexity (≈**100x**)



↓AZ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



[YH Security'24]

[YHKD EuroS&P'22]

[H[1]Y[1]DK S&P'21]

[YHHKV CCS'24]

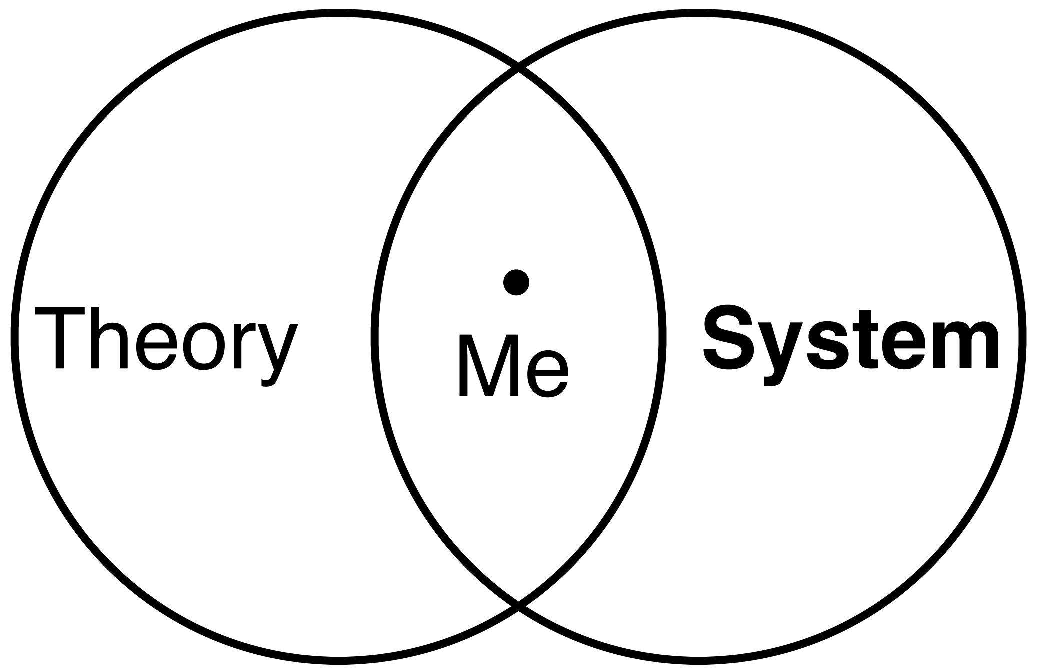
[HHKVY Asiacrypt'24]↓AZ

[YHHKV CCS'23]🏆

[Yang ePrint'25]

[YPHK CCS'23]

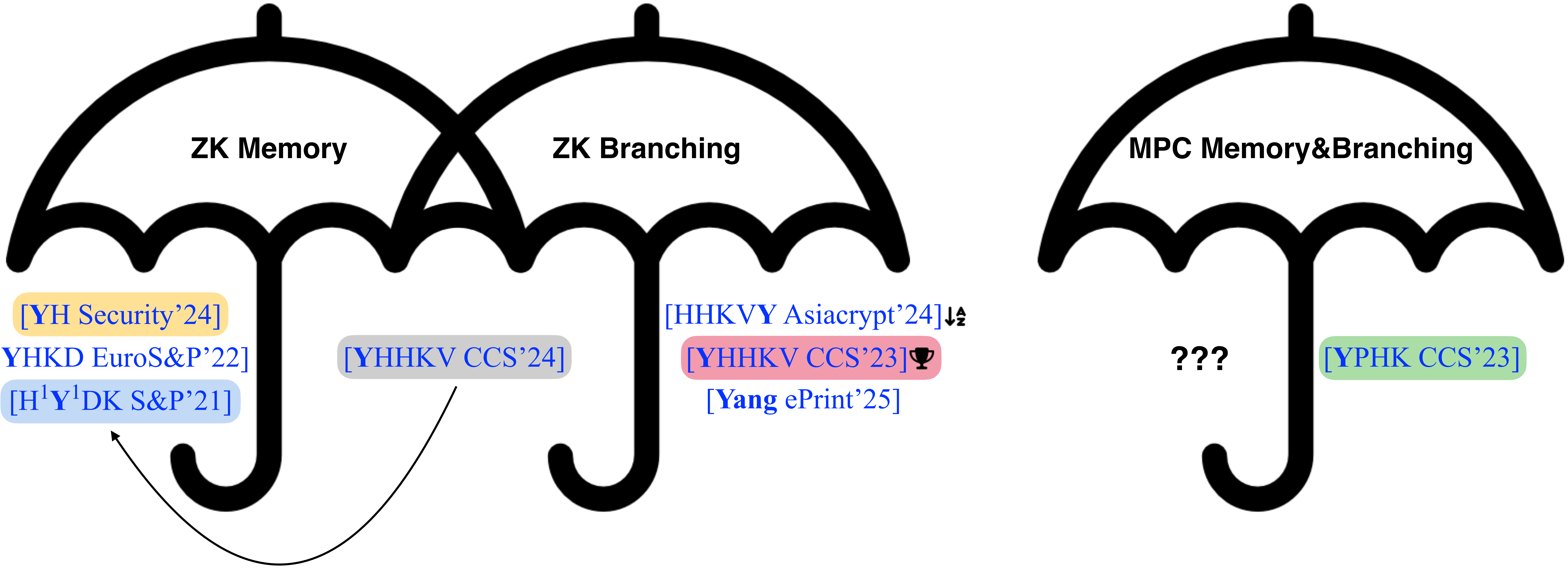
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈ 10KHz ($\approx$**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈ 1KHz ($\approx$**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity
- A zero-knowledge (ZK) CPU+RAM achieving optimal complexity ($\approx$**100x**)



↓AZ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



[YH Security’24]

[YHKD EuroS&P’22]

[H[1]Y[1]DK S&P’21]

[YHHKV CCS’24]

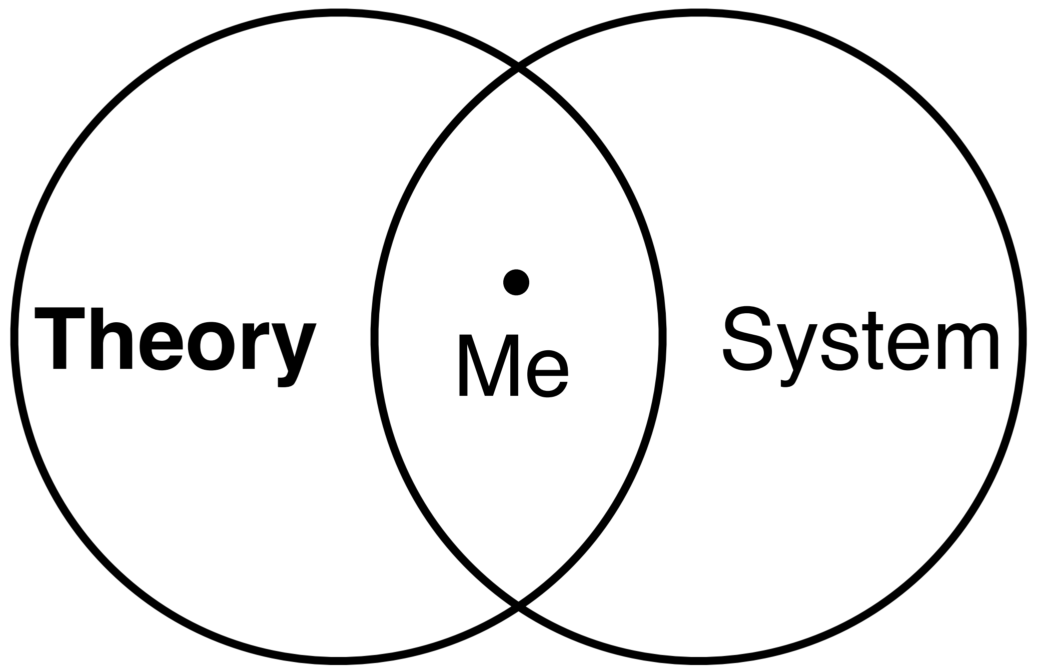
[HHKVY Asiacrypt’24]↓AZ

[YHHKV CCS’23]🏆

[Yang ePrint’25]

[YPHK CCS’23]

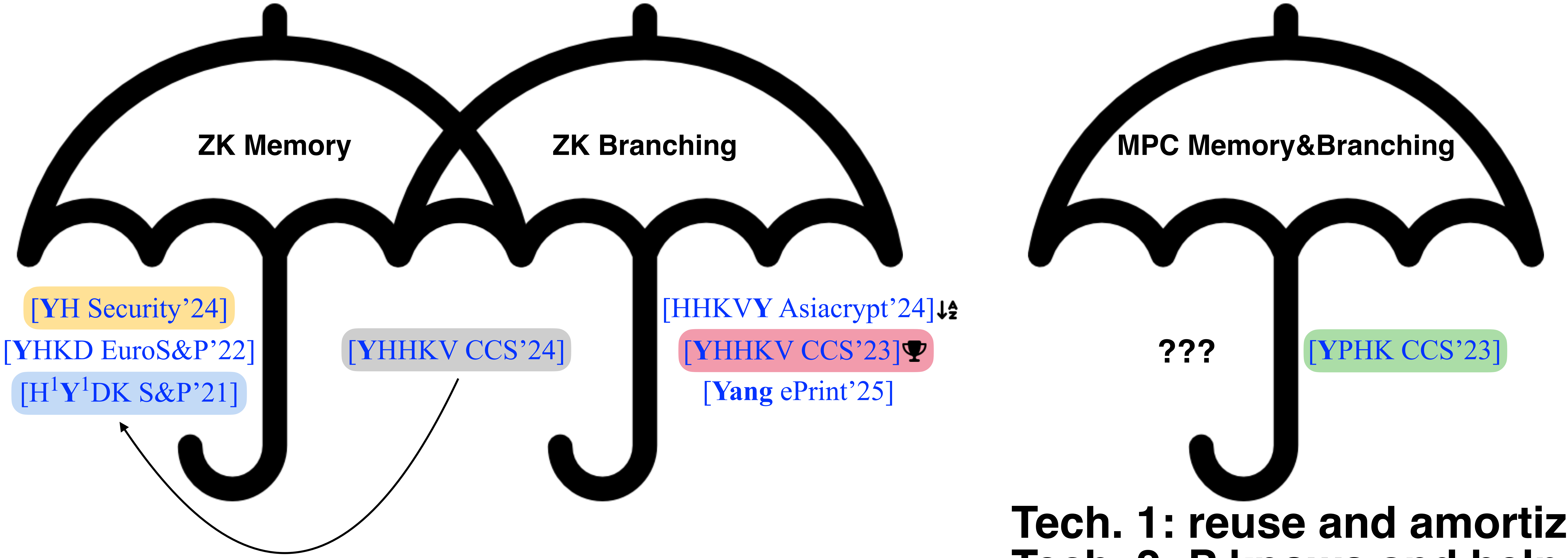
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈1KHz (≈**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity
- A zero-knowledge (ZK) CPU+RAM achieving optimal complexity (≈**100x**)



↓AZ Alphabetic Order

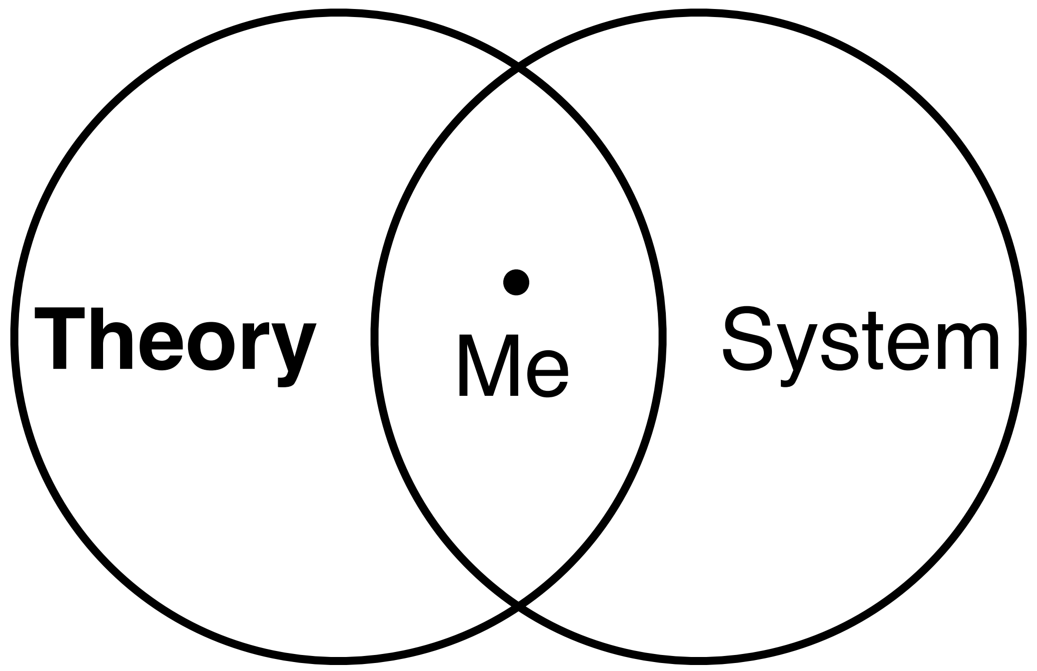
🏆 Distinguished Paper Award

1 Co-first Authorship



Tech. 1: reuse and amortize
Tech. 2: P knows and helps

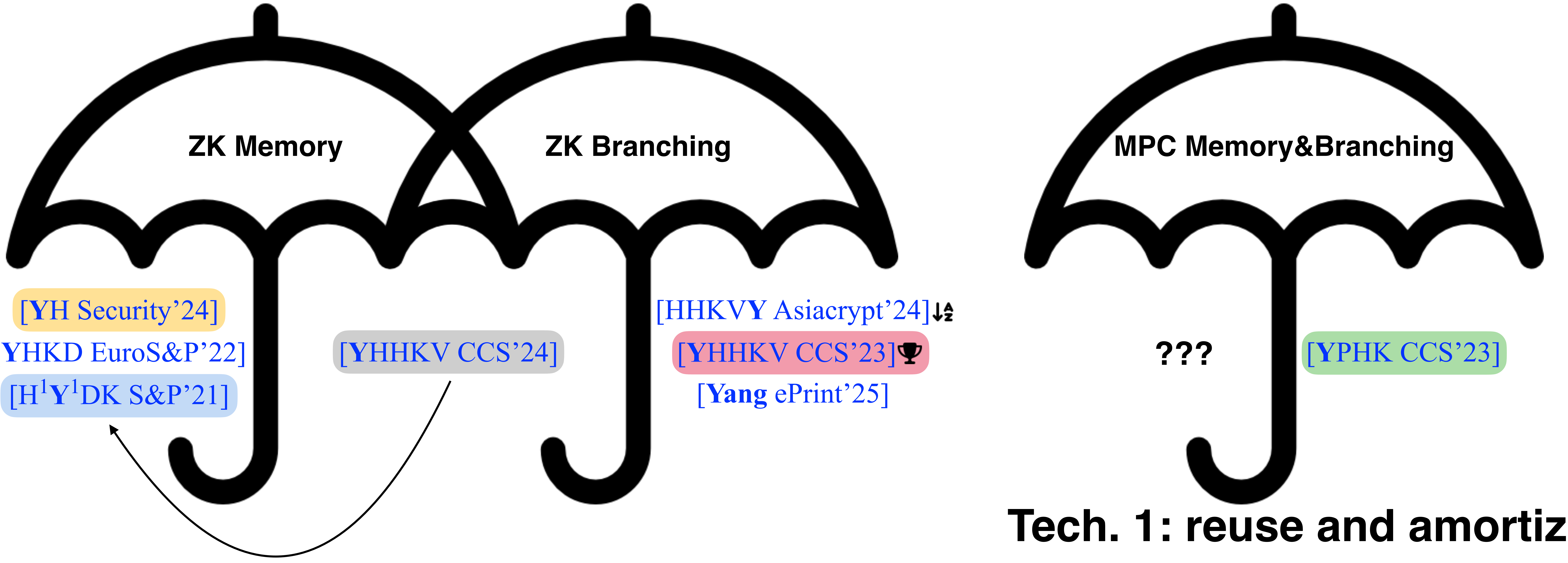
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈10KHz (≈**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈1KHz (≈**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity
- A zero-knowledge (ZK) CPU+RAM achieving optimal complexity (≈**100x**)



⇅ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



[**Y**H Security'24]
[**Y**HKD EuroS&P'22]
[H^{1}**Y**^{1}DK S&P'21]

[**Y**HHKV CCS'24]

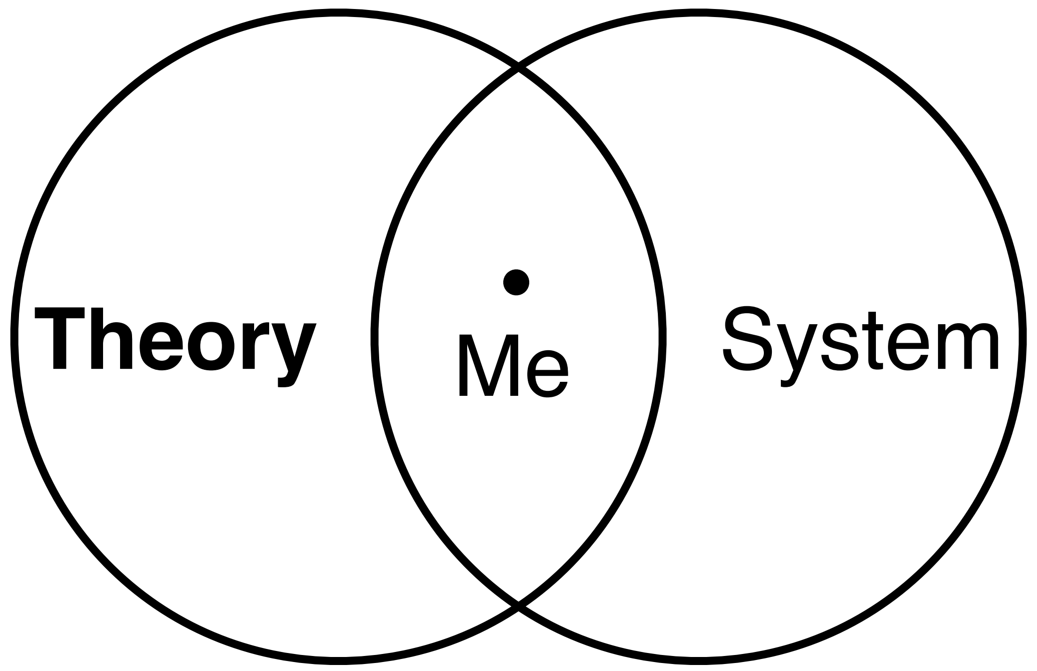
[HHKV**Y** Asiacrypt'24]↓ᴬᶻ
[**Y**HHKV CCS'23]🏆
[**Yang** ePrint'25]

???

[**Y**PHK CCS'23]

Tech. 1: reuse and amortize

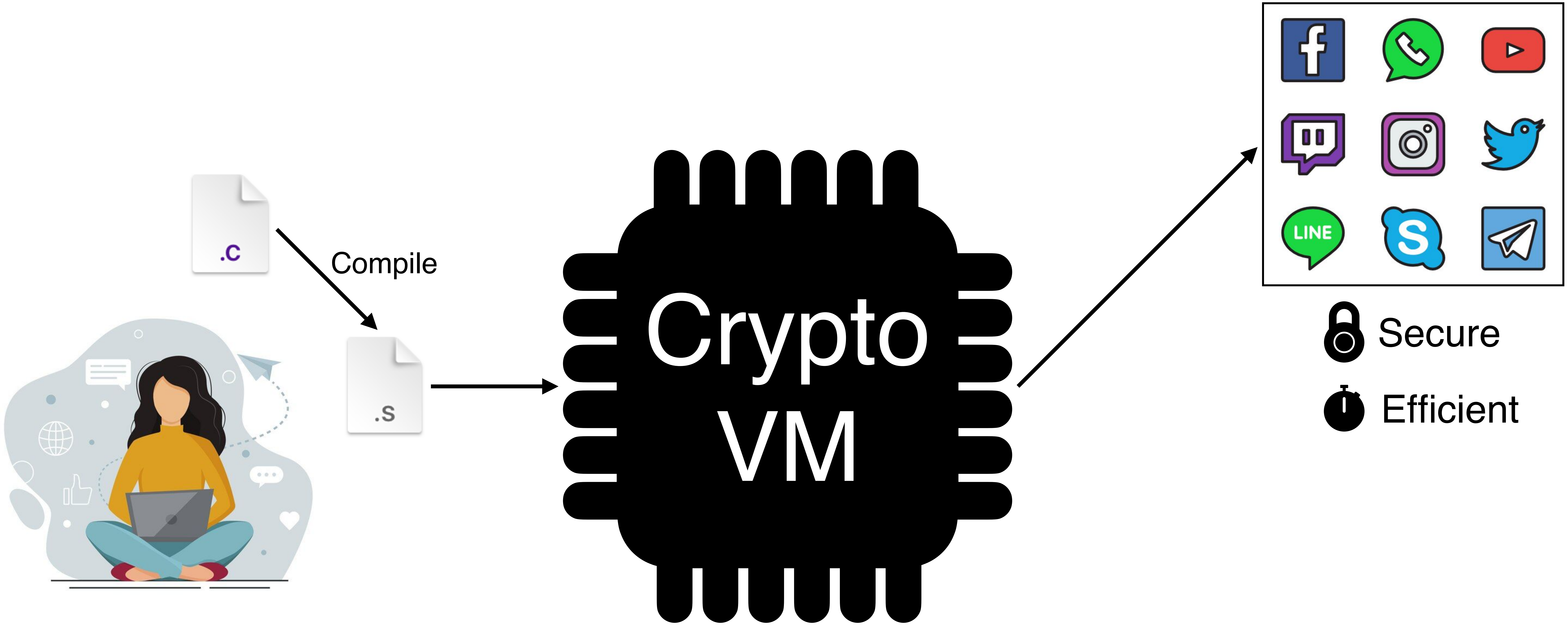
- A zero-knowledge (ZK) full-toolchain system for any ANSI C program at ≈ 10KHz ($\approx$**1000x**)
- A two-party computation (2PC) full-toolchain system for any assembly program at ≈ 1KHz ($\approx$**1000x**)
- A zero-knowledge (ZK) read-write memory achieving optimal complexity
- A zero-knowledge (ZK) branching protocol achieving optimal complexity
- A zero-knowledge (ZK) CPU+RAM achieving optimal complexity ($\approx$**100x**)



↓ᴬᶻ Alphabetic Order
🏆 Distinguished Paper Award
1 Co-first Authorship

Building Better Crypto VM

With the help of compilers, systems, PL, hardware, …



.c

Compile

.s

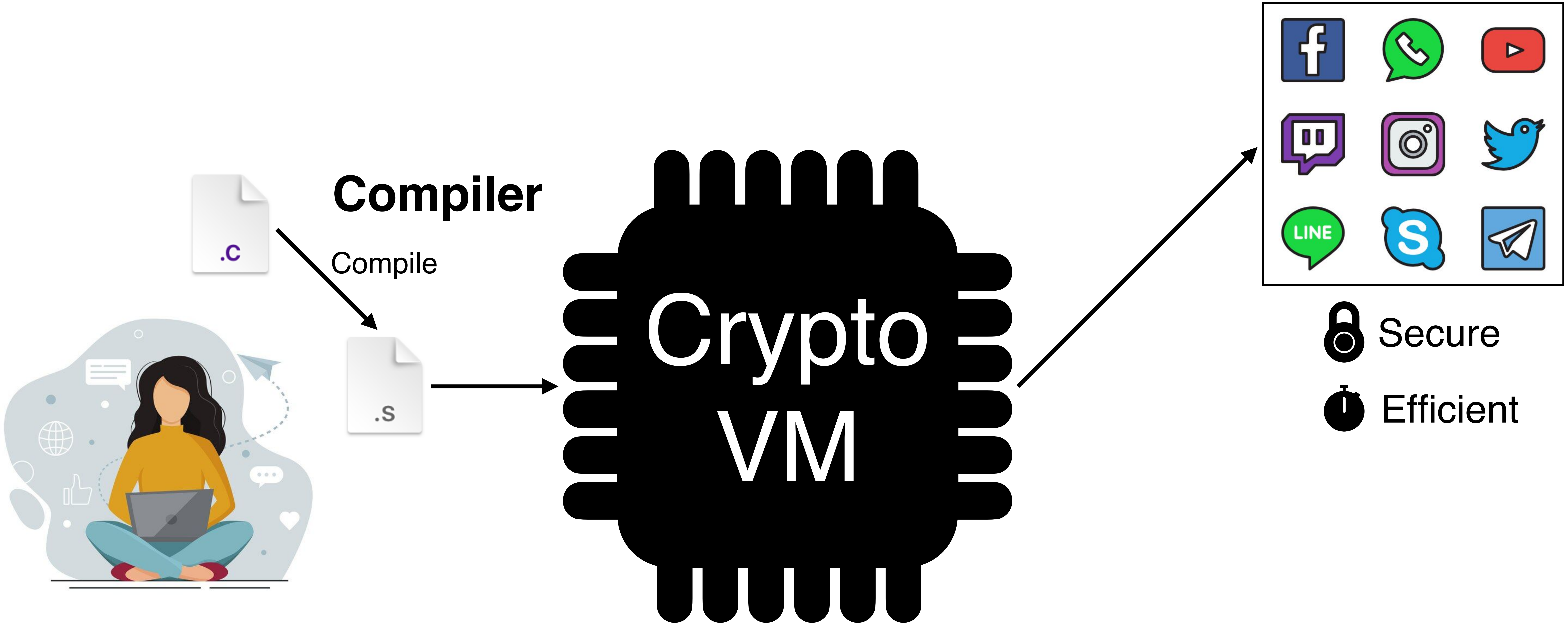
Crypto VM

🔒 Secure

⏱ Efficient

Building Better Crypto VM

With the help of compilers, systems, PL, hardware, …



Compiler

Compile

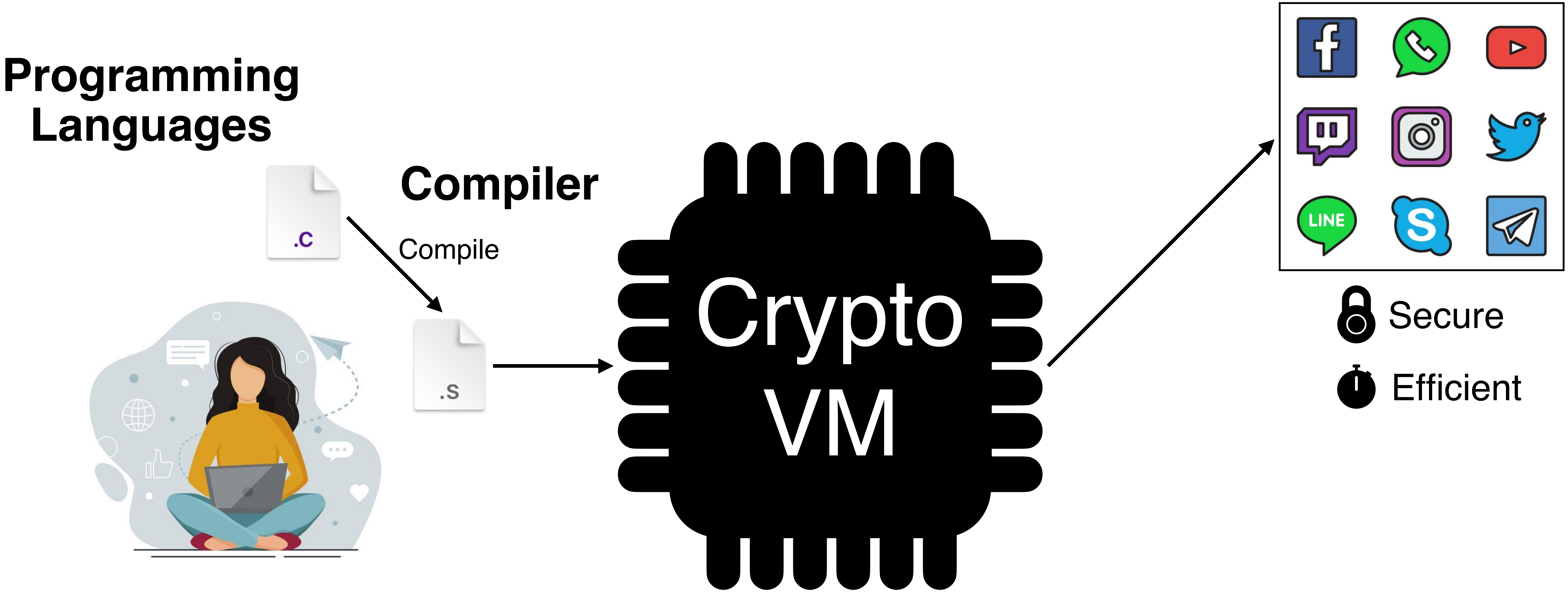
Crypto VM

🔒 Secure

⏱ Efficient

Building Better Crypto VM

With the help of compilers, systems, PL, hardware, …



Programming Languages

Compiler

Compile

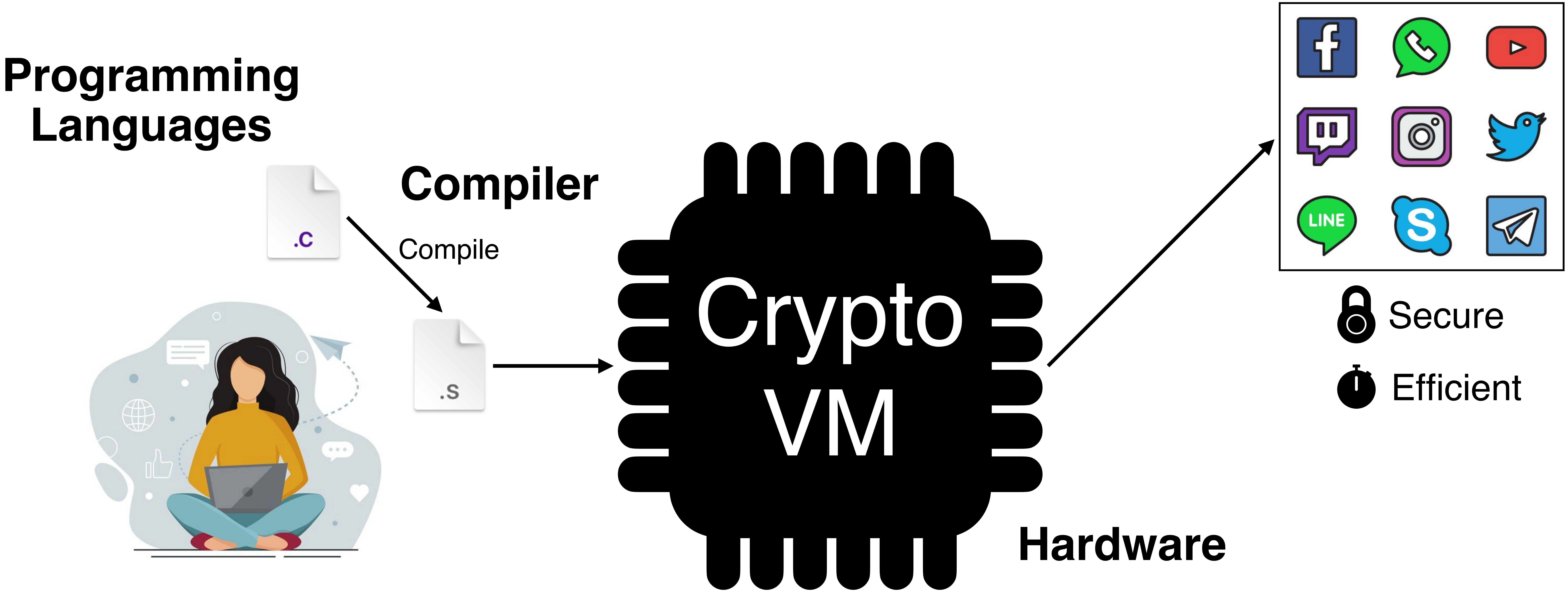
Crypto VM

Secure

Efficient

Building Better Crypto VM

With the help of compilers, systems, PL, hardware, …



Programming Languages

Compiler

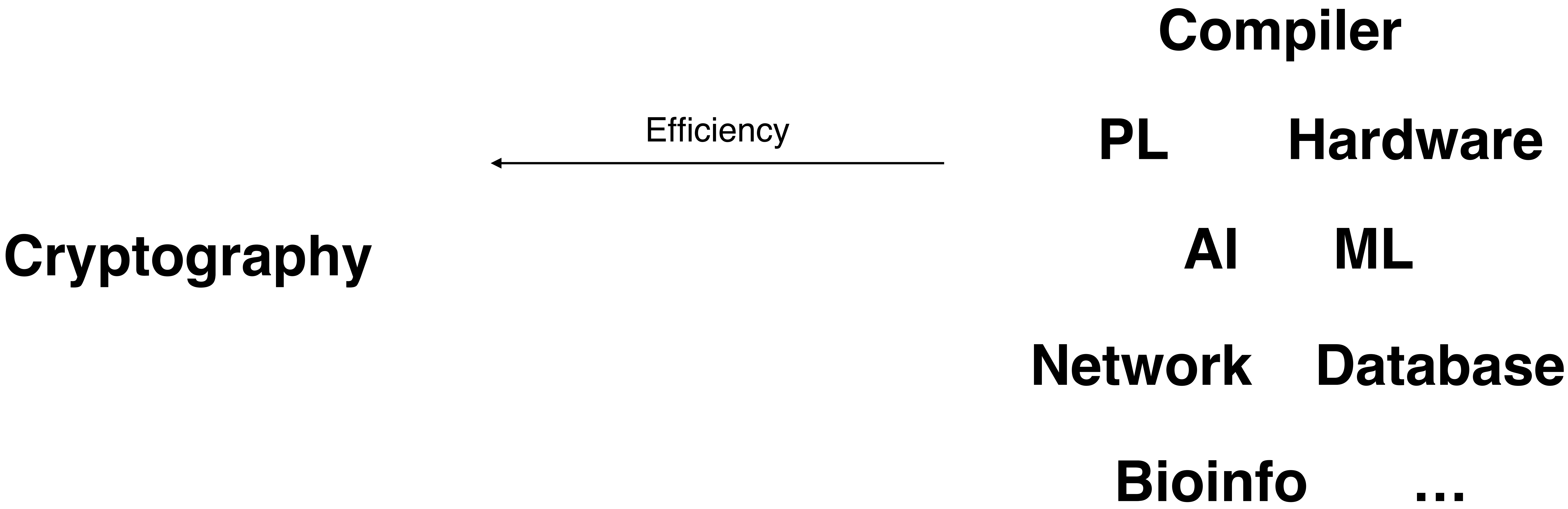
Compile

Crypto VM

Hardware

Secure

Efficient



Compiler

PL Hardware

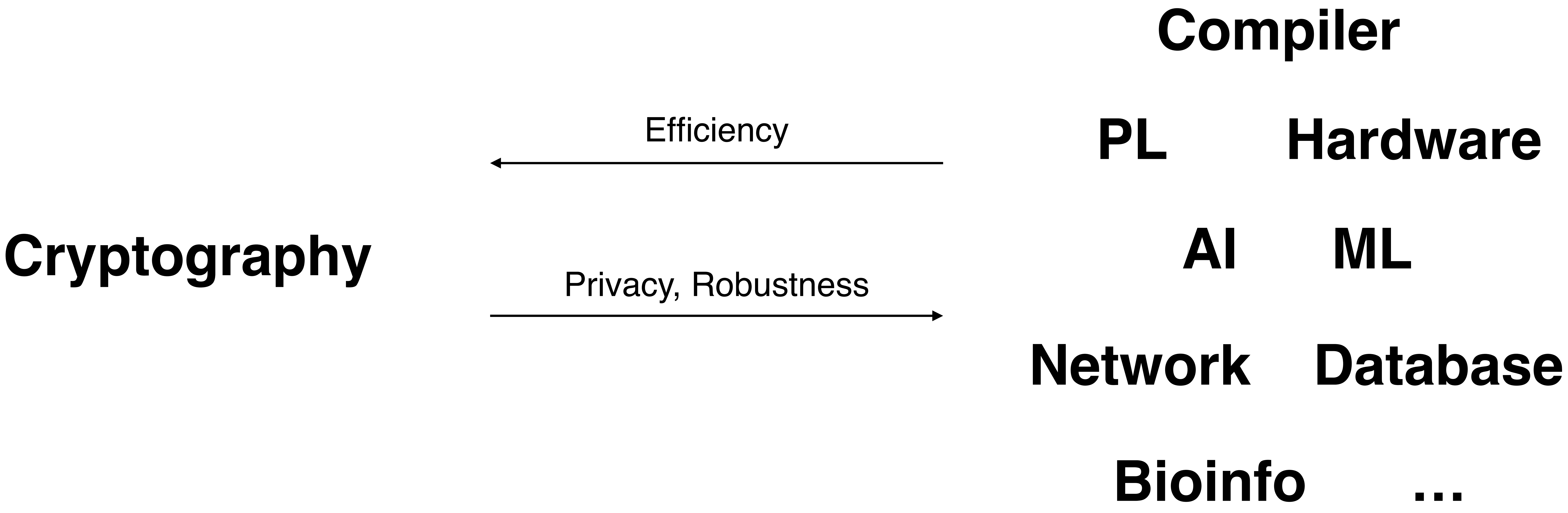
AI ML

Network Database

Bioinfo …

Efficiency

Cryptography



Cryptography

Efficiency

Privacy, Robustness

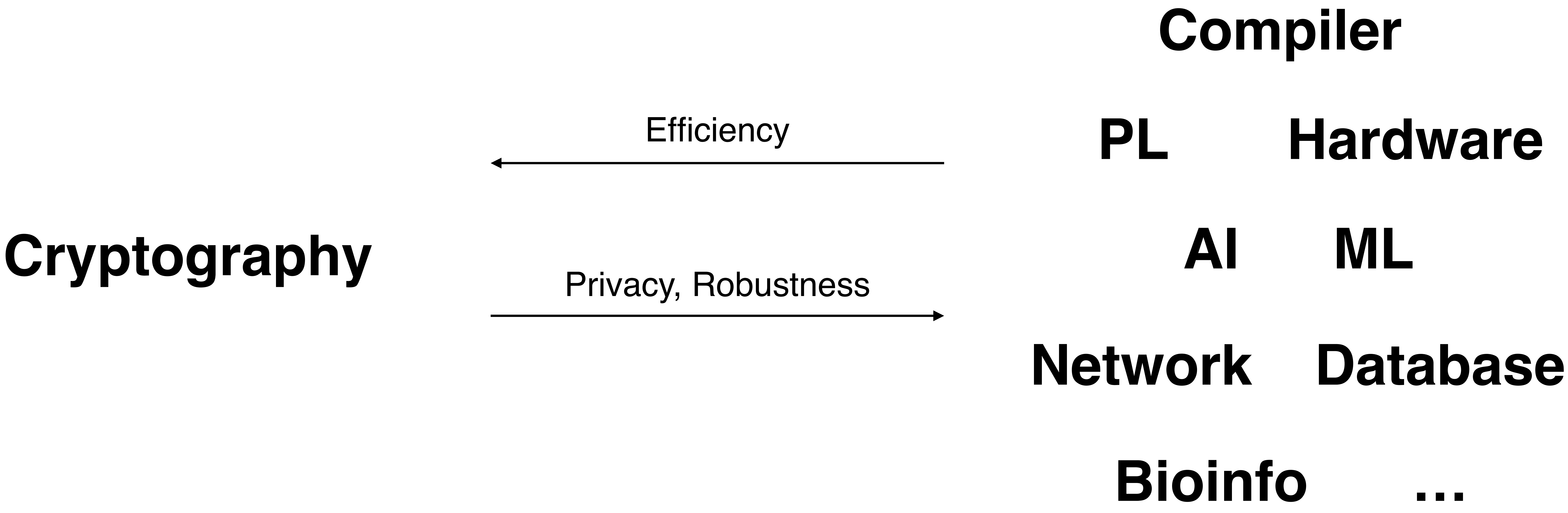
Compiler

PL Hardware

AI ML

Network Database

Bioinfo …



Compiler

Efficiency

PL Hardware

Cryptography

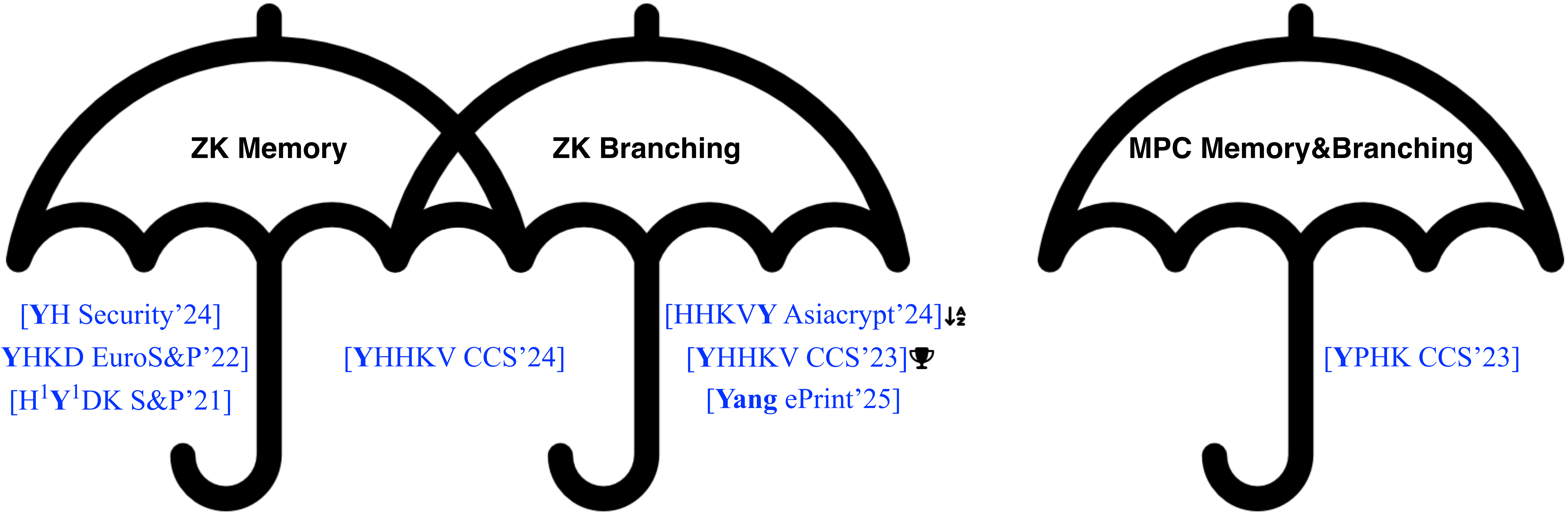
AI ML

Privacy, Robustness

Network Database

Bioinfo …

The state-of-the-art ZKML (Hao et al. USENIX Security'24) uses my *open-sourced* ZK ROM to improve efficiency!



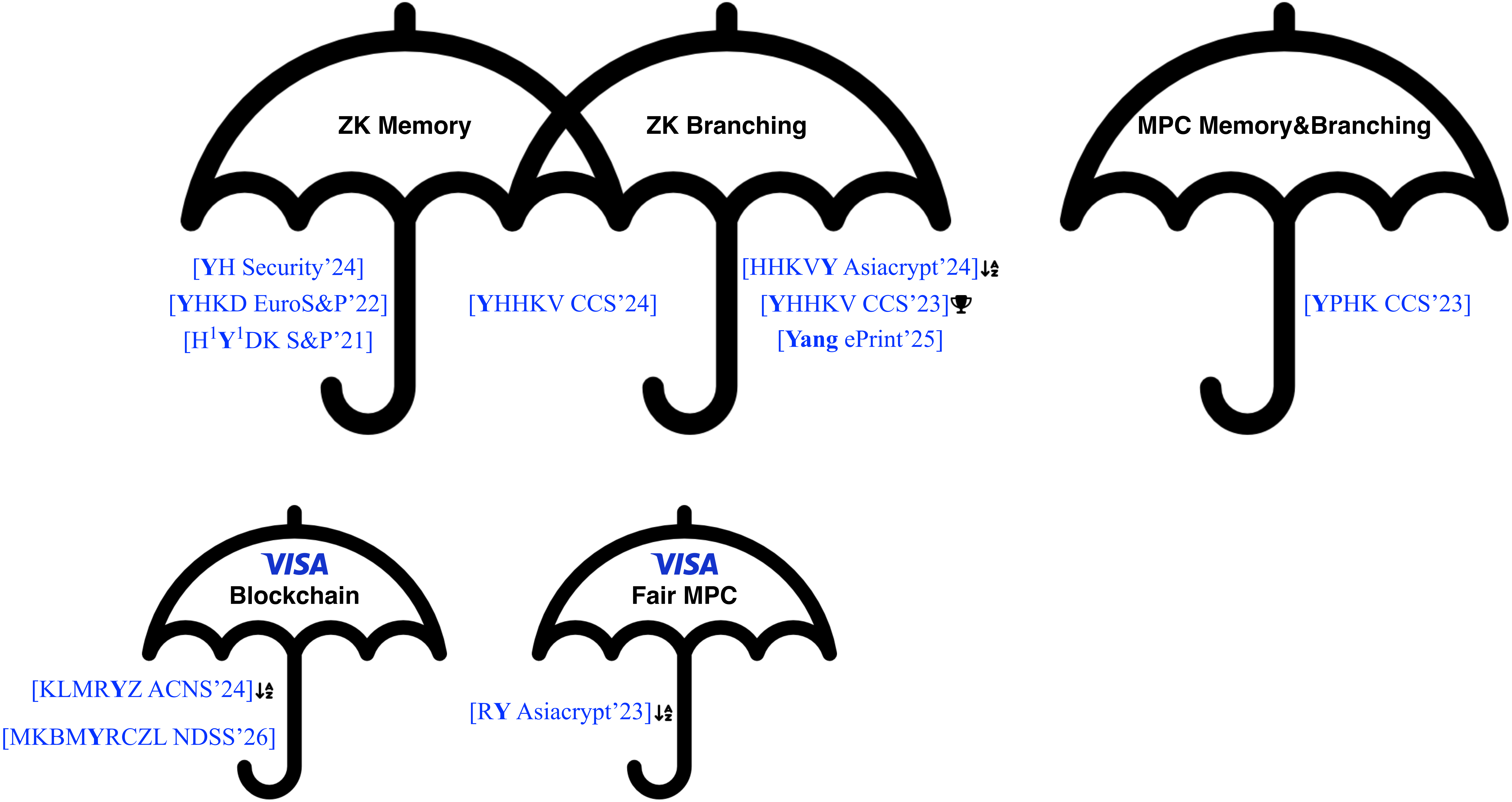
[**Y**H Security'24]
[**Y**HKD EuroS&P'22]
[H^{1}**Y**^{1}DK S&P'21]

[**Y**HHKV CCS'24]

[HHKV**Y** Asiacrypt'24]↓AZ
[**Y**HHKV CCS'23]🏆
[**Yang** ePrint'25]

[**Y**PHK CCS'23]

↓AZ Alphabetic Order
🏆 Distinguished Paper Award
1 Co-first Authorship



ZK Memory

[**Y**H Security’24]

[**Y**HKD EuroS&P’22]

[H^{1}**Y**^{1}DK S&P’21]

[**Y**HHKV CCS’24]

ZK Branching

[HHKV**Y** Asiacrypt’24]↓AZ

[**Y**HHKV CCS’23]🏆

[**Yang** ePrint’25]

MPC Memory&Branching

[**Y**PHK CCS’23]

VISA

Blockchain

[KLMR**Y**Z ACNS’24]↓AZ

[MKBM**Y**RCZL NDSS’26]

VISA

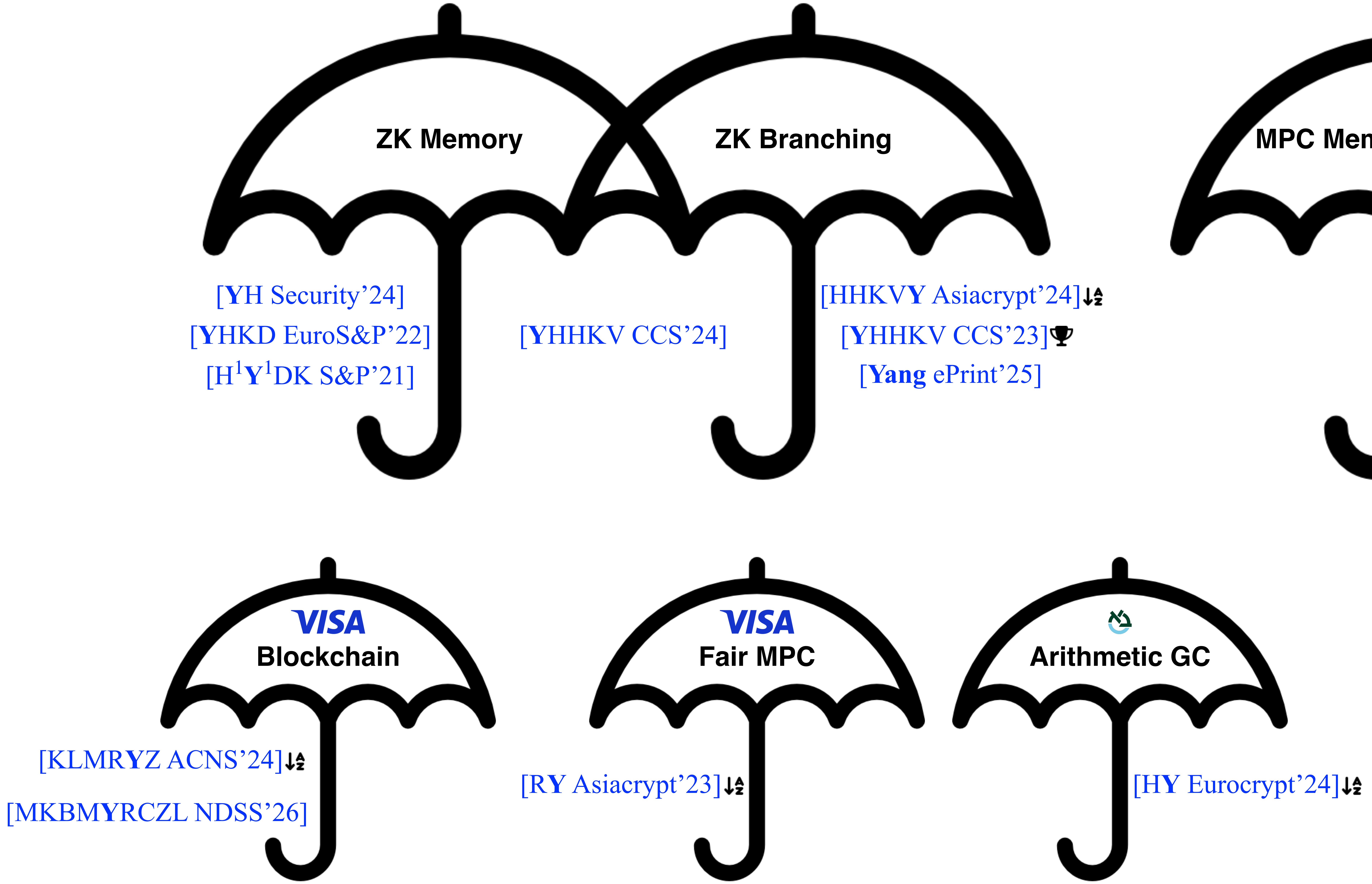
Fair MPC

[R**Y** Asiacrypt’23]↓AZ

↓AZ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



ZK Memory

[**Y**H Security’24]

[**Y**HKD EuroS&P’22]

[H^1**Y**^{1}DK S&P’21]

[**Y**HHKV CCS’24]

ZK Branching

[HHKV**Y** Asiacrypt’24]↓ᴬᴢ

[**Y**HHKV CCS’23]🏆

[**Yang** ePrint’25]

MPC Memory&Branching

[**Y**PHK CCS’23]

VISA

Blockchain

[KLMR**Y**Z ACNS’24]↓ᴬᴢ

[MKBM**Y**RCZL NDSS’26]

VISA

Fair MPC

[R**Y** Asiacrypt’23]↓ᴬᴢ

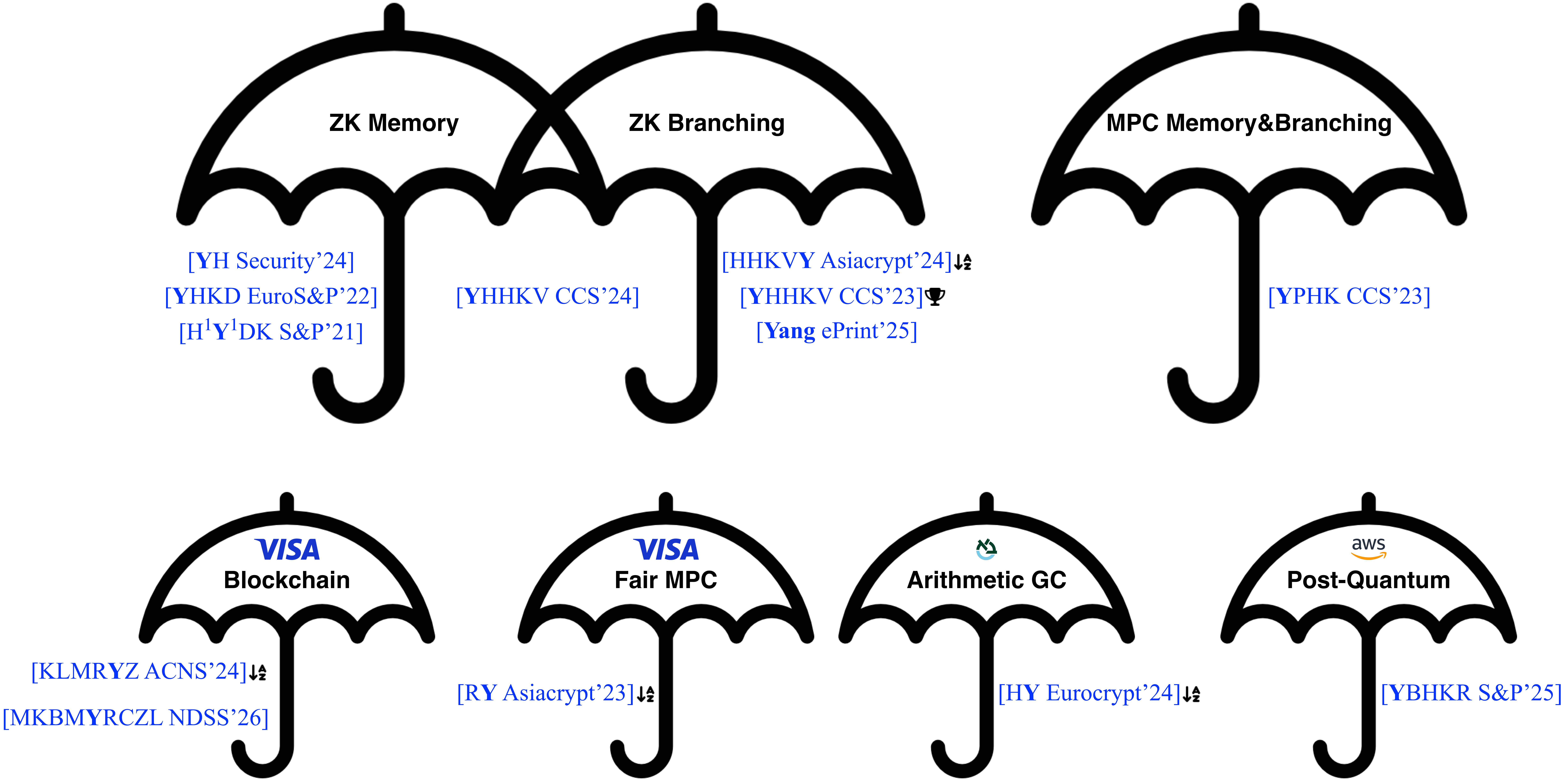
Arithmetic GC

[H**Y** Eurocrypt’24]↓ᴬᴢ

↓ᴬᴢ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship



ZK Memory

[**Y**H Security'24]

[**Y**HKD EuroS&P'22]

[H[1]**Y**[1]DK S&P'21]

[**Y**HHKV CCS'24]

ZK Branching

[HHKV**Y** Asiacrypt'24]↓AZ

[**Y**HHKV CCS'23]🏆

[**Yang** ePrint'25]

MPC Memory&Branching

[**Y**PHK CCS'23]

VISA

Blockchain

[KLMR**Y**Z ACNS'24]↓AZ

[MKBM**Y**RCZL NDSS'26]

VISA

Fair MPC

[R**Y** Asiacrypt'23]↓AZ

Arithmetic GC

[H**Y** Eurocrypt'24]↓AZ

aws

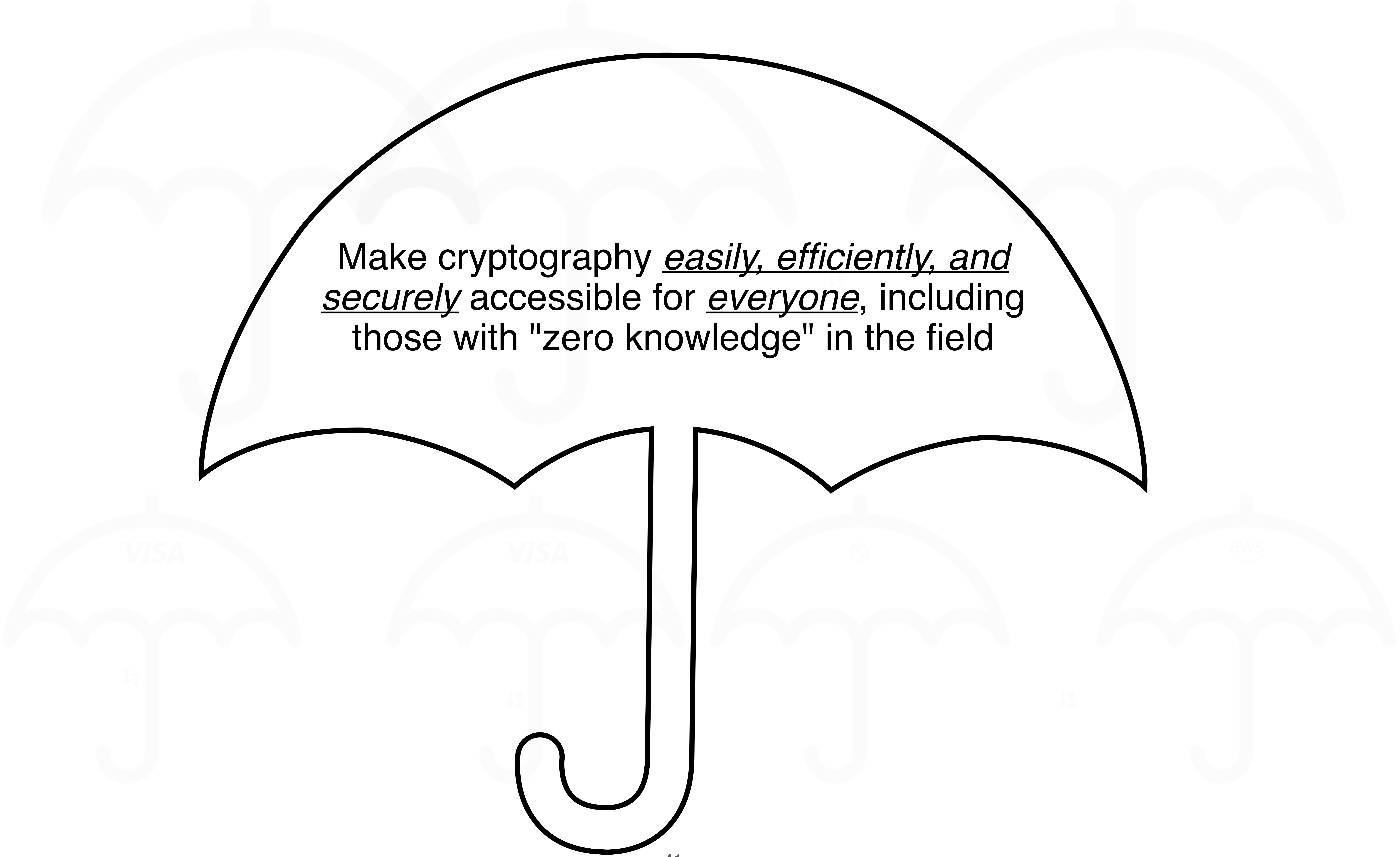
Post-Quantum

[**Y**BHKR S&P'25]

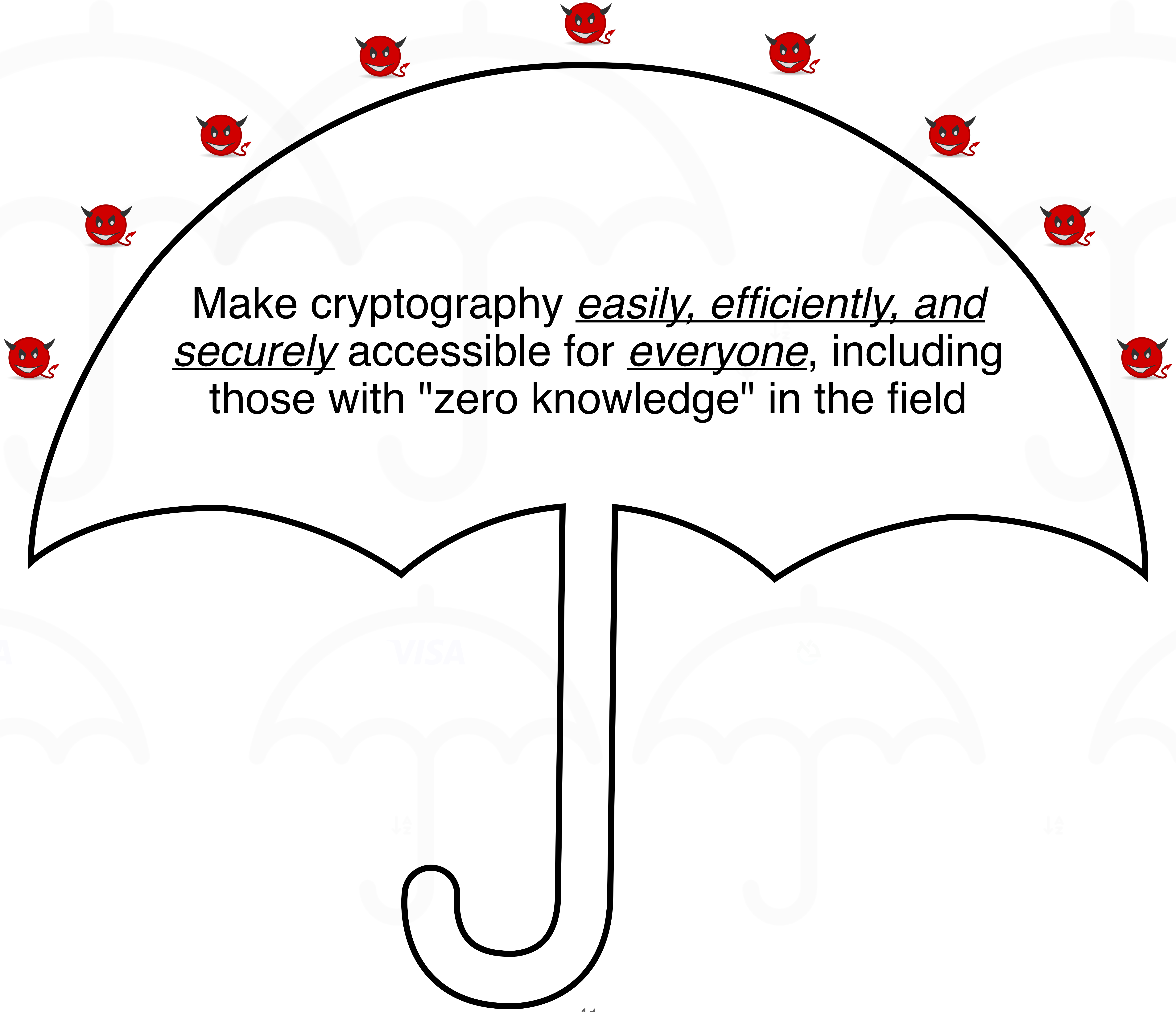
↓AZ Alphabetic Order

🏆 Distinguished Paper Award

1 Co-first Authorship

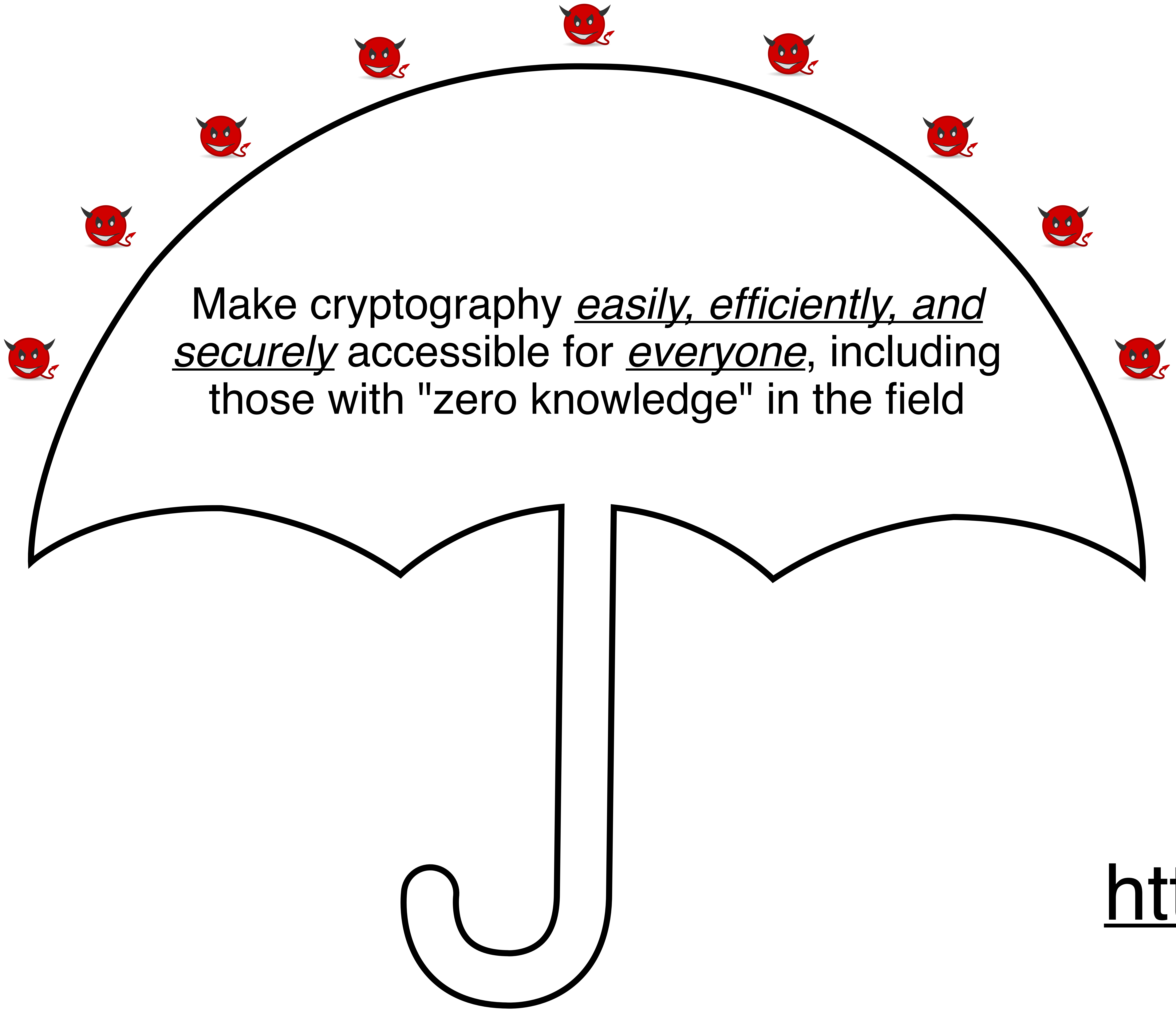


Make cryptography *easily, efficiently, and securely* accessible for *everyone*, including those with "zero knowledge" in the field

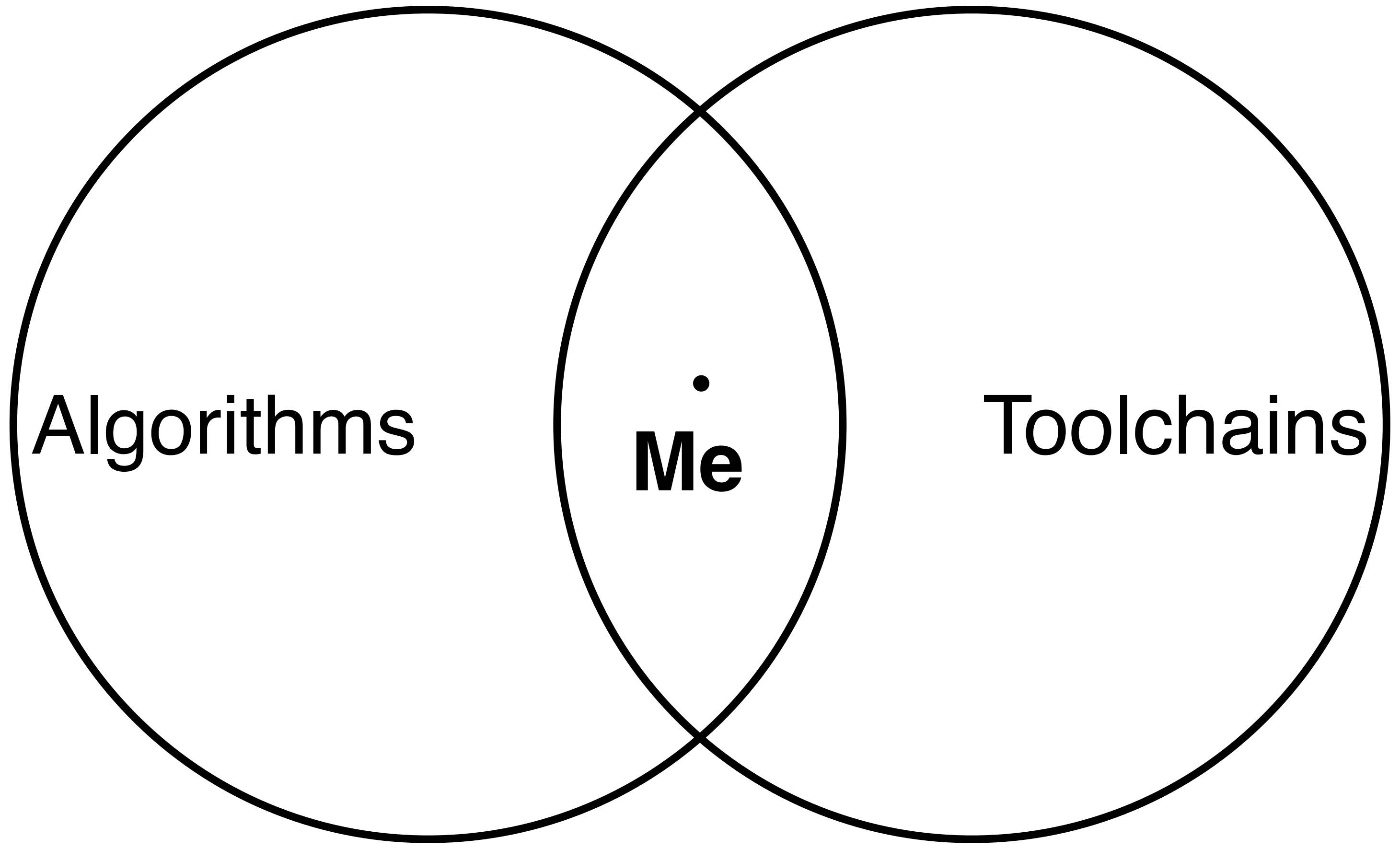


Make cryptography *easily, efficiently, and securely* accessible for *everyone*, including those with "zero knowledge" in the field

Acknowledgements



Make cryptography *easily, efficiently, and securely* accessible for *everyone*, including those with "zero knowledge" in the field



Algorithms

Me

Toolchains

Thank you!

https://yibinyang.info