ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State

Jeongseok Son Griffin Prechter Rishabh Poddar
Raluca Ada Popa Koushik Sen

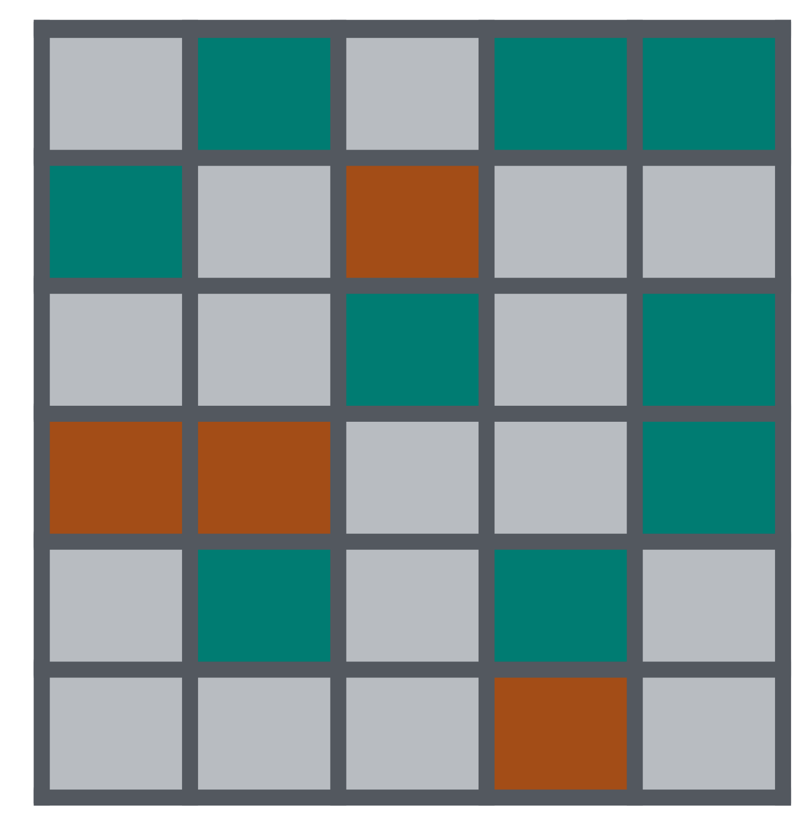
Data Encryption is Widely Adopted

Is encryption enough to hide your confidential data?

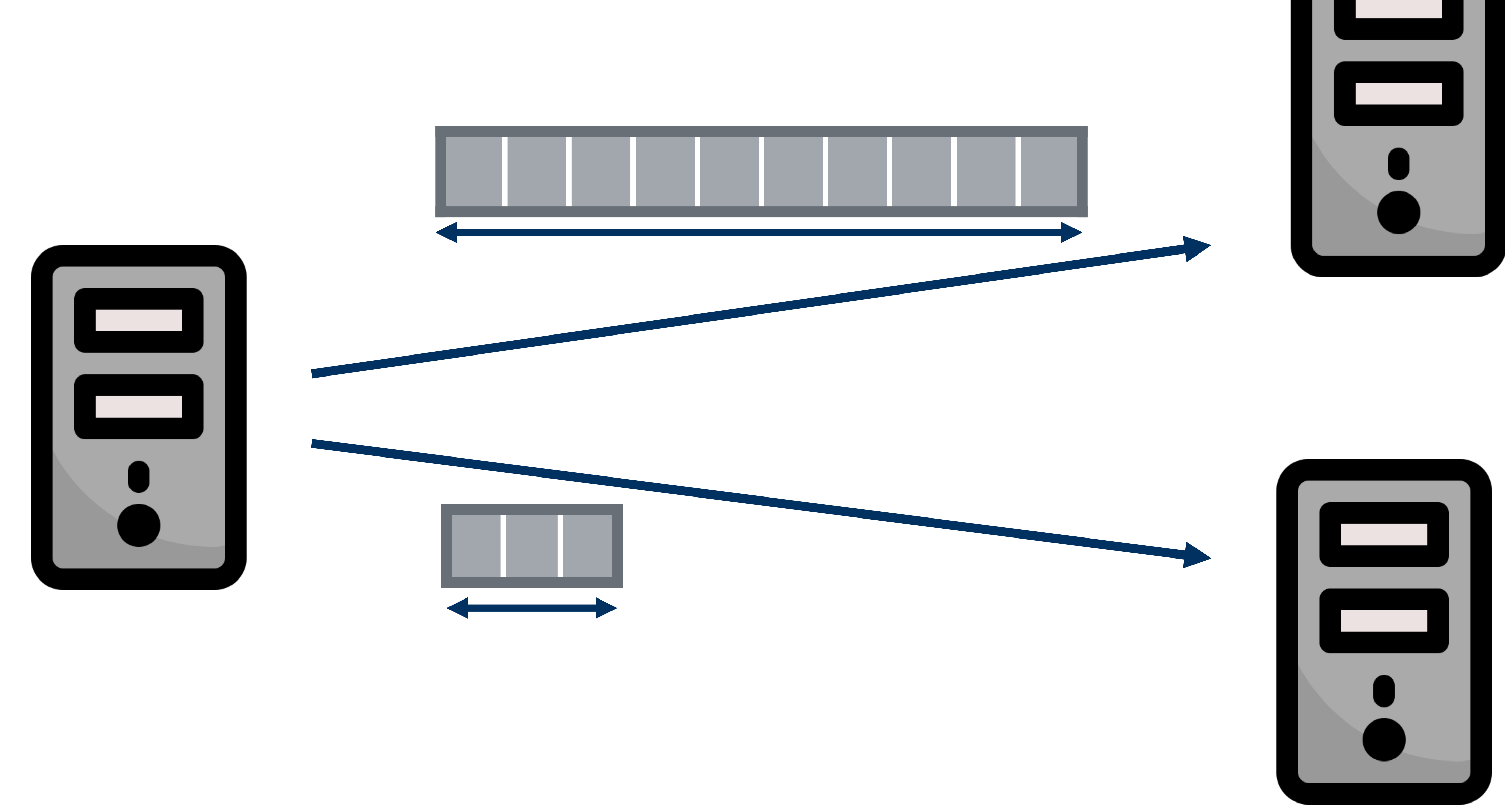
©2017 RISELab

Data Access Patterns Leak Sensitive Information

Data Access Patterns include:



- Accessed Memory Address



- Source / Destination Network Address
- Message Size

©2017 RISELab

Data Access Patterns Leak Sensitive Information

©2017 RISELab

Data Access Patterns Leak Sensitive Information

Alice

SELECT.. disease = Flu

Memory

…	Flu
…	Cancer
…	
…	Flu
…	
…	Flu

©2017 RISELab

Data Access Patterns Leak Sensitive Information

Alice

Bob

SELECT.. disease = Flu

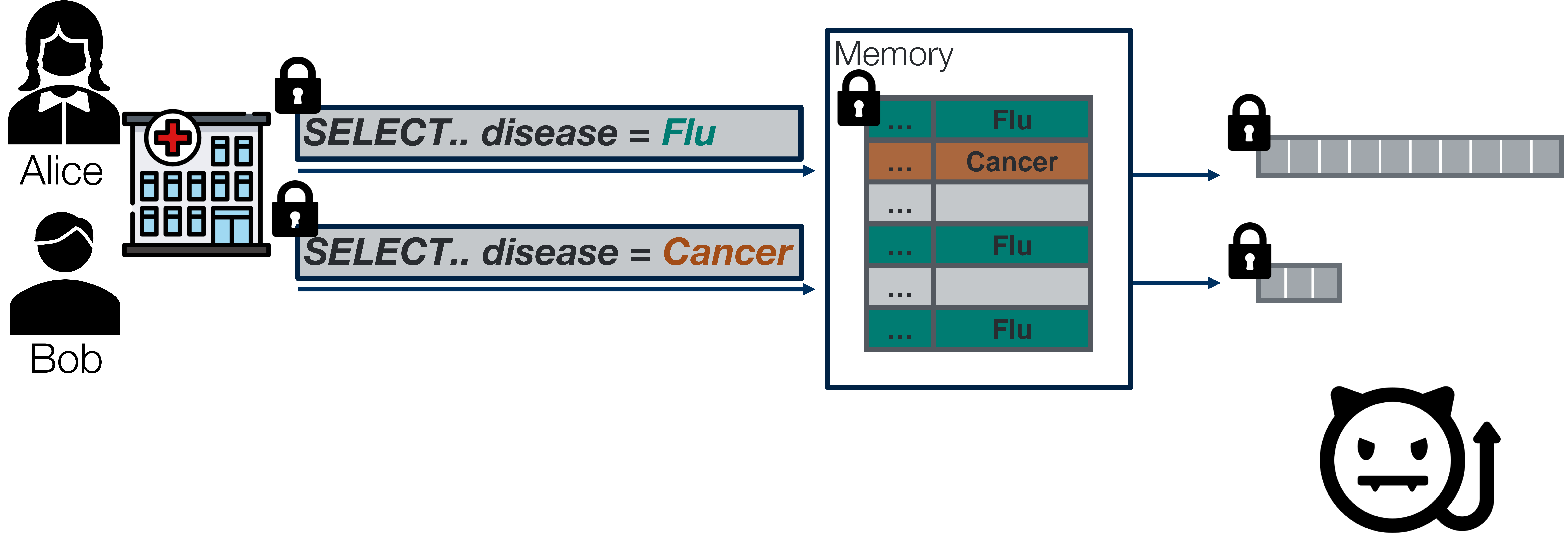
SELECT.. disease = Cancer

Memory

...	Flu
...	Cancer
...	
...	Flu
...	
...	Flu

©2017 RISELab

Data Access Patterns Leak Sensitive Information



©2017 RISELab

Data Access Patterns Leak Sensitive Information

Alice

Bob

SELECT.. disease = *Flu*

SELECT.. disease = *Cancer*

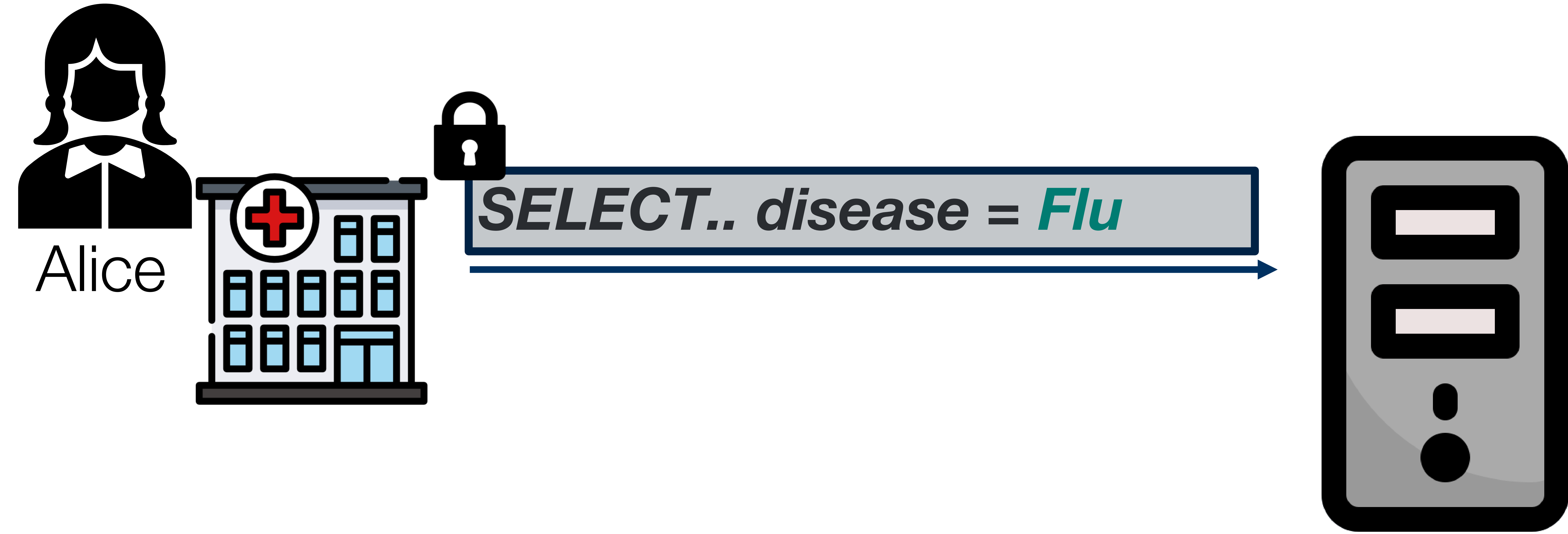
Memory

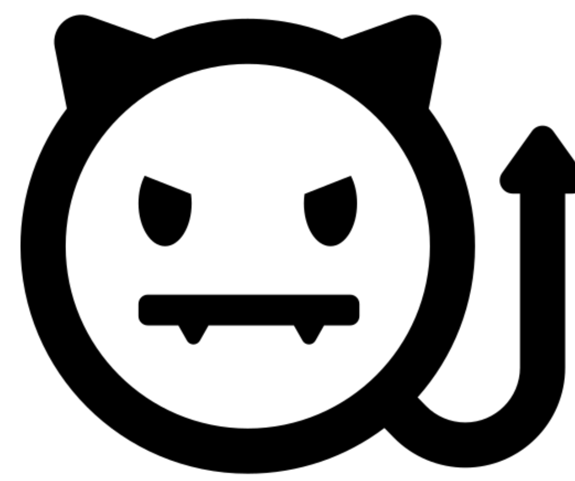
…	Flu
…	Cancer
…	
…	Flu
…	
…	Flu

Bob has cancer!

©2017 RISELab

How To Prevent Access Pattern Leakage





©2017 RISELab

How To Prevent Access Pattern Leakage

Alice

SELECT.. disease = Flu

Memory

...	Flu
...	Cancer
...	
...	Flu
...	
...	Flu

©2017 RISELab

How To Prevent Access Pattern Leakage

Alice

Bob

SELECT.. disease = Flu

SELECT.. disease = Cancer

Memory

...	Flu
...	Cancer
...	
...	Flu
...	
...	Flu

©2017 RISELab

How To Prevent Access Pattern Leakage

Alice

Bob

SELECT.. disease = Flu

SELECT.. disease = Cancer

Memory

...	Flu
...	Cancer
...	
...	Flu
...	
...	Flu

???

©2017 RISELab

How To Prevent Access Pattern Leakage

Alice

Bob

SELECT.. disease = Flu

SELECT.. disease = Cancer

Memory

…	Flu
…	Cancer
…	
…	Flu
…	
…	Flu

???

"Oblivious: Access pattern is independent from private data"

©2017 RISELab

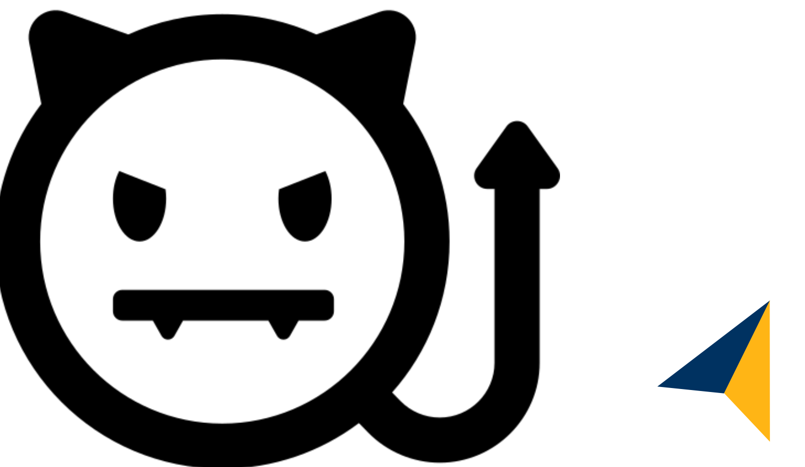
Oblivious Algorithms Are Slow

SELECT..Cancer

Cancer
Flu
Cancer
Flu
Flu
Cancer
Cancer
...

Sorting Network

Cancer
Cancer
Cancer
Cancer
Flu
Flu
Flu
...



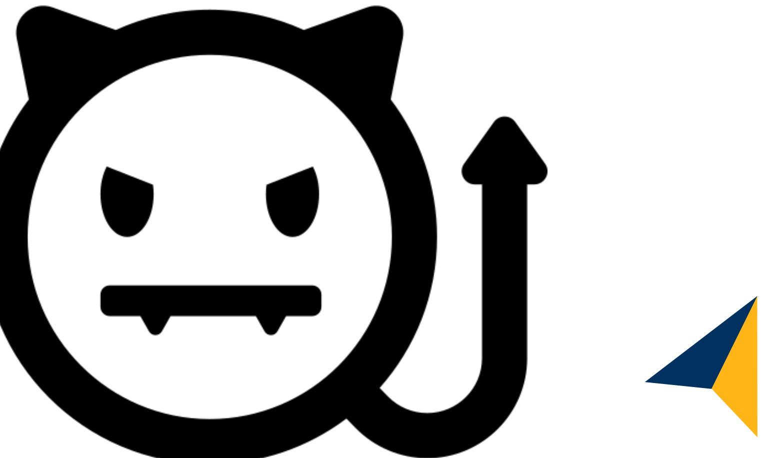
©2017 RISELab

Unobservable Memory Region Speeds Up Oblivious Algorithms

- Oblivious sort operation from Opaque [Zheng et al., USENIX NSDI '17]

SELECT..Cancer

Unobservable Space
(e.g. registers, enclaves)



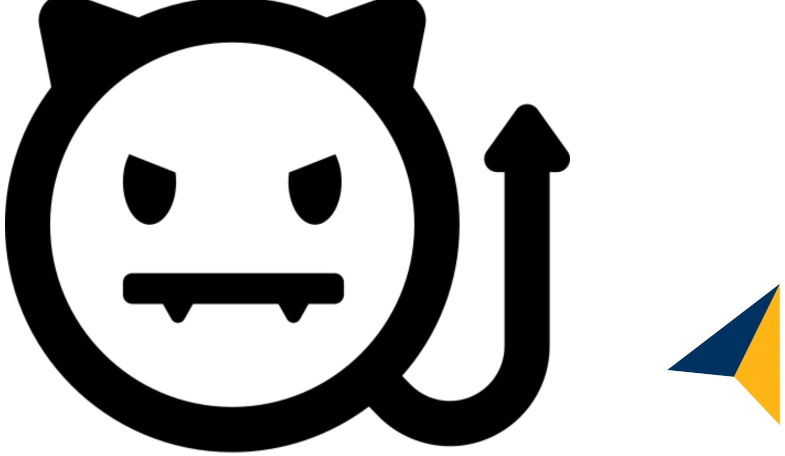
©2017 RISELab

Unobservable Memory Region Speeds Up Oblivious Algorithms

- Oblivious sort operation from Opaque [Zheng et al., USENIX NSDI '17]

SELECT..Cancer

Cancer
Flu
Cancer
Flu
Flu
Flu



©2017 RISELab

Unobservable Memory Region Speeds Up Oblivious Algorithms

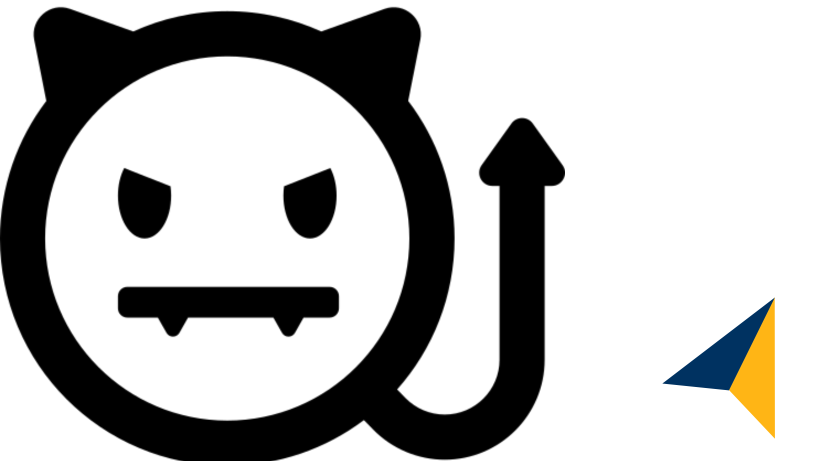
- Oblivious sort operation from Opaque [Zheng et al., USENIX NSDI '17]

SELECT..Cancer

Cancer
Cancer
Flu
Flu
Flu
Flu

Cancer
Cancer
Flu
Flu
Flu
Flu

Sorting Network



©2017 RISELab

Problem: Oblivious Algorithms Are Complicated

Problem

The algorithms are complicated to be efficient and oblivious at the same time
→ Difficult to check one is indeed oblivious

State-of-the-art

Algorithm designers write pen-and-pencil proofs, and the users manually verify it

©2017 RISELab

How can we automatically verify oblivious algorithms in a practical manner?

©2017 RISELab

Existing Solutions Are Insufficient

Code: Oblivious Filter
The program gets a list of private data, compare each element to VALUE and tag either 0 or 1

Threat Model
Attacker can only watch data sent over network, local memory is unobservable

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

Increments the output length by 1 → →

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

Increments the output length by 1 →

Observable output.length is always the same. →
output is encrypted before sent over network.

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```

Increments the output length by 1 →

Observable output.length is always the same. →
output is encrypted before sent over network.

The filter algorithm is **oblivious**

©2017 RISELab

Existing Solutions Are Insufficient

Solution 1. Taint Analysis (Static)
Reject a program with a secret dependent branch

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

Solution 1. Taint Analysis (Static)

Regarded as a problematic branch, →
thus the program gets rejected

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

Solution 1. Taint Analysis (Static)

Shortcoming
Unobservable state propagates taint, which leads to a rejection of an oblivious program

✘ Algorithms leveraging unobservable space are always deemed not oblivious (False-positives)

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

Solution 2. Symbolic Execution

Encounter a branch. Fork the execution →
Path Condition: $e_0 < VALUE$ →

Path Condition: $e_0 \geq VALUE$ →

Path Condition
Constraints possible input values for each execution path
→ Let us reason about all program behaviors

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

e_0 < VALUE

→

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

$e_0 < VALUE$

….append((1, e_0))

….append((0, e_0))

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
→           output.append((1, e));
        } else {
→           output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

2nd Iteration: Fork the execution →

e_0 < VALUE

….append((1, e_0))

….append((0, e_0))

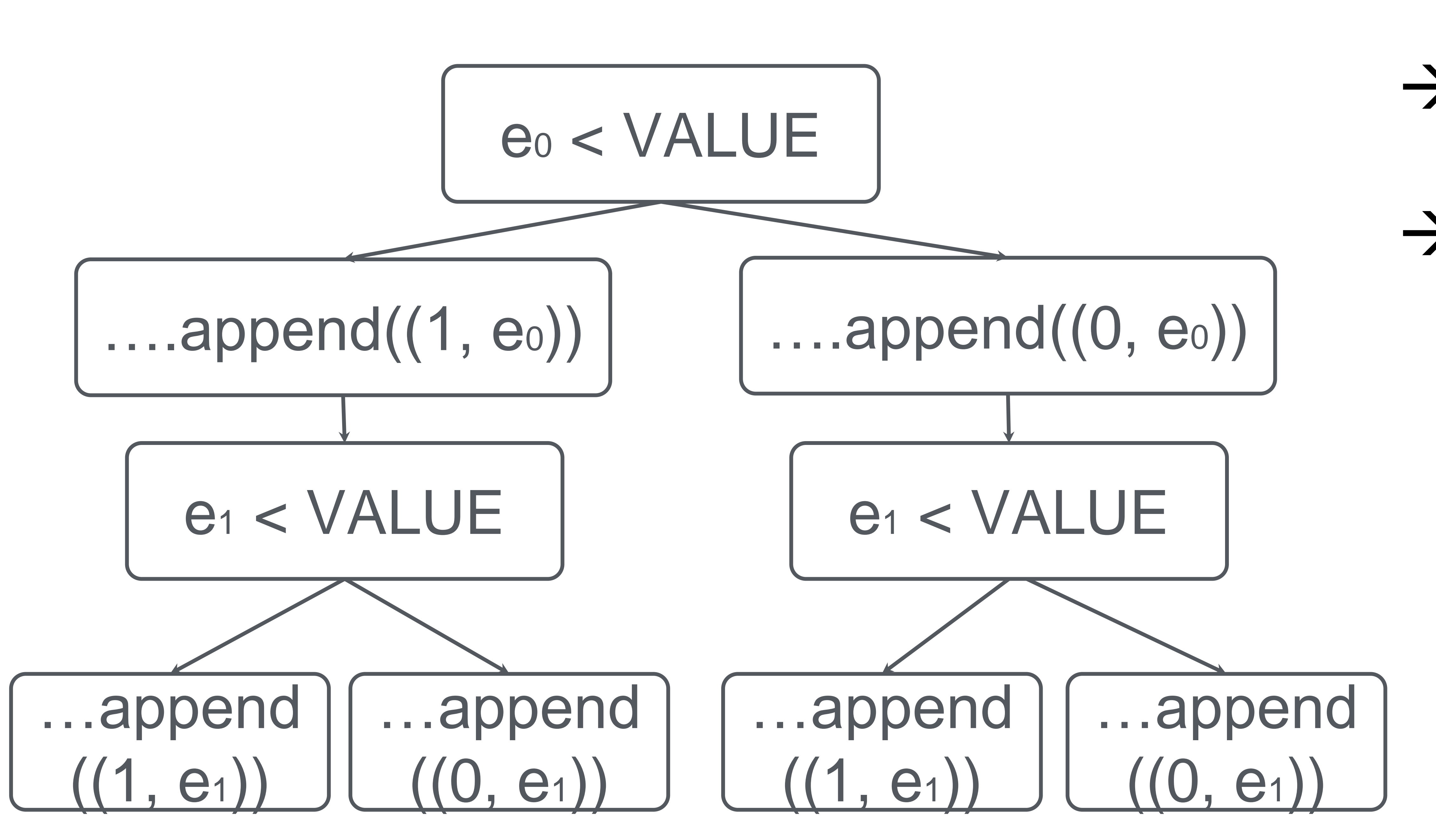
e_1 < VALUE

e_1 < VALUE

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient




```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
→           output.append((1, e));
        } else {
→           output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Existing Solutions Are Insufficient

Solution 2. Symbolic Execution

Shortcoming
Branches on unobservable state makes the number of paths to explore grow exponentially

✘ Analysis time increases exponentially in the length of the input (Path explosion)

```
function Filter(privateData) {
    ...
    for (e in privateData) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

ObliCheck: Symbolic Execution for Verifying Oblivious Algorithms

1. Let users specify observable and unobservable access
→ **ObliCheck API**

2. Leverage domain specific knowledge to enhance symbolic execution performance
→ **Optimistic State Merging**

3. Remove the false-positives incurred by state merging
→ **Iterative State Unmerging**

©2017 RISELab

ObliCheck: Symbolic Execution for Verifying Oblivious Algorithms

1. Let users specify observable and unobservable access
→ **ObliCheck API**

2. Leverage domain specific knowledge to enhance symbolic execution performance
→ **Optimistic State Merging**

3. Remove the false-positives incurred by state merging
→ **Iterative State Unmerging**

©2017 RISELab

API for Accessing Observable Space

Algorithm designers use ObliCheck APIs to express a certain data access is observable

A trace is added to the access sequence under the hood whenever an API is called

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    EncSendTo(NET_ADDR, output);
    ...
}
```


©2017 RISELab

API for Accessing Observable Space

Algorithm designers use ObliCheck APIs to express a certain data access is observable

A trace is added to the access sequence under the hood whenever an API is called

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
→   ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

ObliCheck: Symbolic Execution for Verifying Oblivious Algorithms

1. Let users specify observable and unobservable access
→ **ObliCheck API**

2. Leverage domain specific knowledge to enhance symbolic execution performance
→ **Optimistic State Merging**

3. Remove the false-positives incurred by state merging
→ **Iterative State Unmerging**

©2017 RISELab

Optimistic State merging

Core Idea
Merge state based on domain specific knowledge of which state is unobservable

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

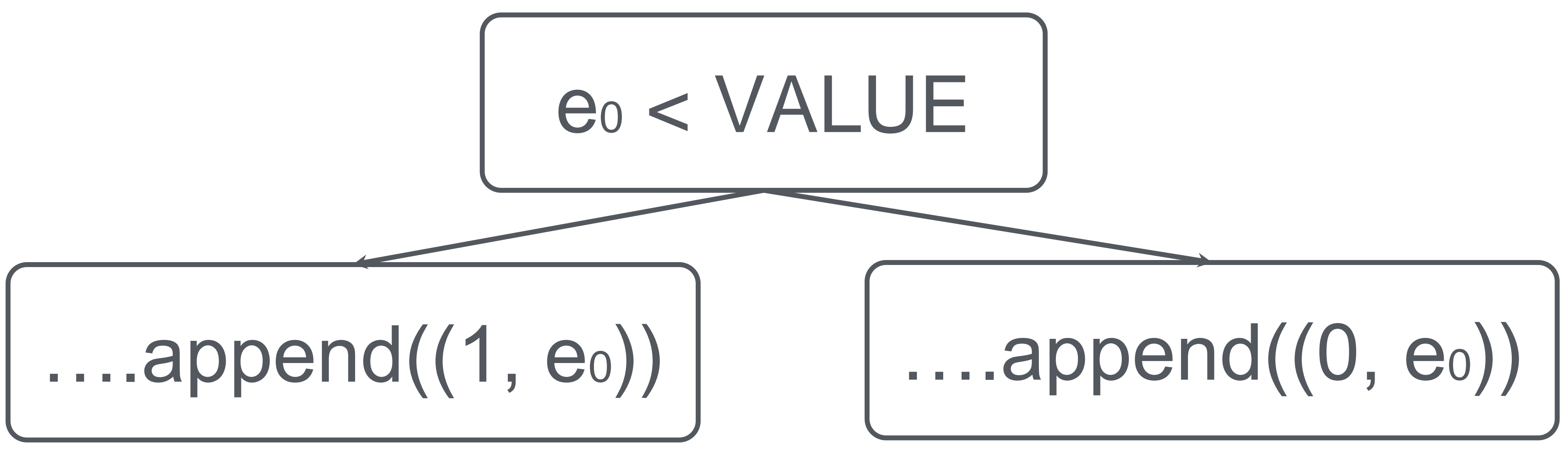
Optimistic State merging

e_0 < VALUE

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
→       if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging




```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
→           output.append((1, e));
        } else {
→           output.append((0, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

output[0] = (1, e_0)
output.length = 1

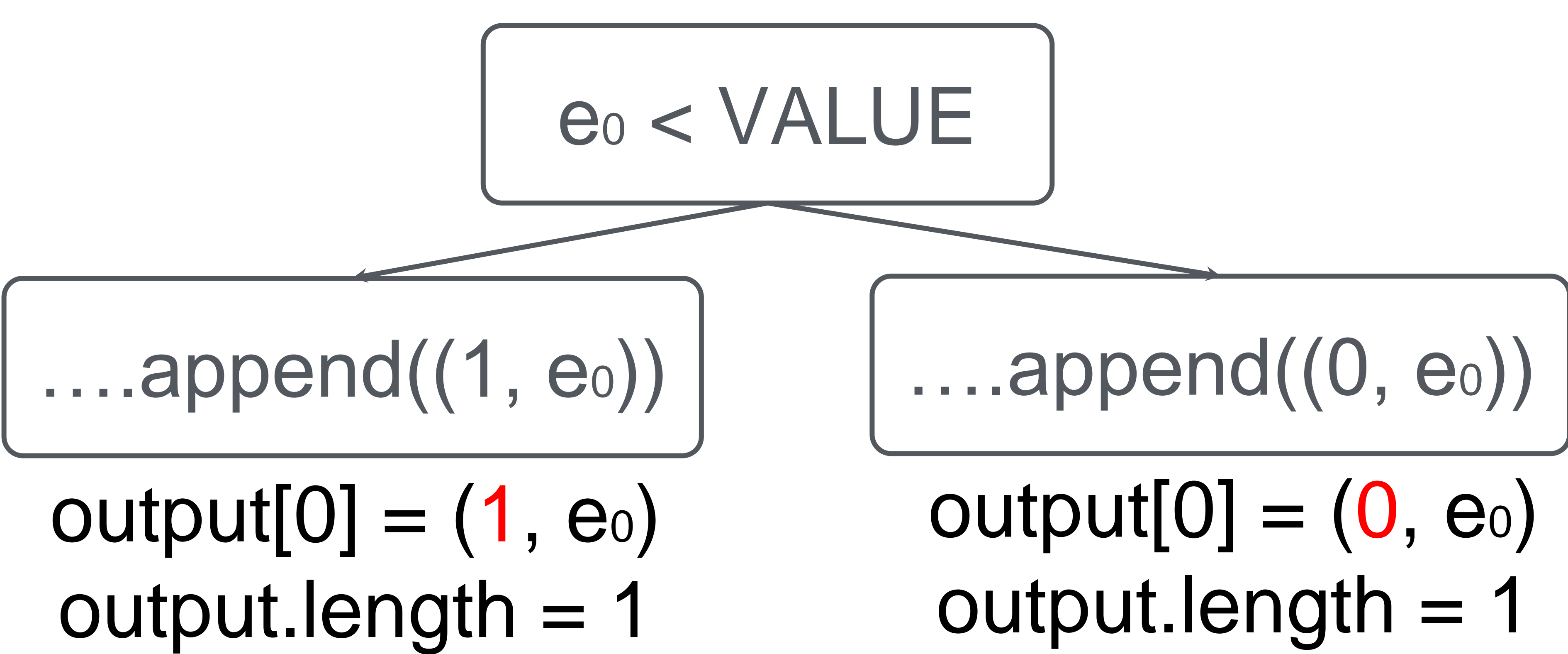
….append((0, e_0))

output[0] = (0, e_0)
output.length = 1

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
→           output.append((1, e));
        } else {
→           output.append((0, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging



output[0] = (1, e_0)
output.length = 1

output[0] = (0, e_0)
output.length = 1

Cannot merge the state because the variable value is different in the two paths

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
→           output.append((1, e));
        } else {
→           output.append((0, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

1 and 0 are unobservable and encrypted when sent →
An attacker cannot distinguish the two values
→

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

1 and 0 are unobservable and encrypted when sent →
An attacker cannot distinguish the two values
→

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

$e_0 < \text{VALUE}$

….append((1, e_0)) | ….append((0, e_0))

output[0] = (1, e_0)
output.length = 1

output[0] = (0, e_0)
output.length = 1

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

output[0] = (★, e_0)
output.length = 1

….append((0, e_0))

output[0] = (★, e_0)
output.length = 1

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((★, e));
        } else {
            output.append((★, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

output[0] = (s_0, e_0)
output.length = 1

….append((0, e_0))

output[0] = (s_0, e_0)
output.length = 1

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

….append((0, e_0))

output[0] = (s_0, e_0)
output.length = 1

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

….append((0, e_0))

output[0] = (s_0, e_0)
output.length = 1

e_1 < VALUE

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((*, e));
        } else {
            output.append((*, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

….append((0, e_0))

output[0] = (s_0, e_0)
output.length = 1

e_1 < VALUE

….append((1, e_1))

….append((0, e_1))

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((*, e));
        } else {
            output.append((*, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

e_0 < VALUE

….append((1, e_0))

….append((0, e_0))

output[0] = (s_0, e_0)
output.length = 1

e_1 < VALUE

….append((1, e_1))

….append((0, e_1))

output[0] = (s_1, e_0)
output.length = 2

...

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((*, e));
        } else {
            output.append((*, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

...

output[n] = s_n
output.length = n

```
ObliCheck.send(NET_ADDR, output);
```

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

...

output[n] = s_n
output.length = n

```
ObliCheck.send(NET_ADDR, output);
```

ObliCheck reports the algorithm is **oblivious**
(output.length is always n in all paths)

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

Optimistic State merging

Benefit
The analysis time grows linearly

Soundness
Over-approximated program includes the cases of the original program

Problem
May falsely reject an oblivious program

```
function Filter(privateRecords) {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((★, e));
        } else {
            output.append((★, e));
        }
    }
    ObliCheck.send(NET_ADDR, output);
    ...
}
```


©2017 RISELab

ObliCheck: Symbolic Execution for Verifying Oblivious Algorithms

1. Let users specify observable and unobservable access
→ **ObliCheck API**

2. Leverage domain specific knowledge to enhance symbolic execution performance
→ **Optimistic State Merging**

3. Remove the false-positives incurred by state merging
→ **Iterative State Unmerging**

©2017 RISELab

Optimistic State Merging Incurs False Positives

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

output[0] = (1, e)→
or
output[0] = (0, e)→

t.first = 0 or 1→

→

result.length always increments by 1→

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```

The algorithm is **oblivious**

©2017 RISELab

...

output[0] = (s_0, e_0)

→

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((*, e));
        } else {
            output.append((*, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

...

output[0] = (s_0, e_0)

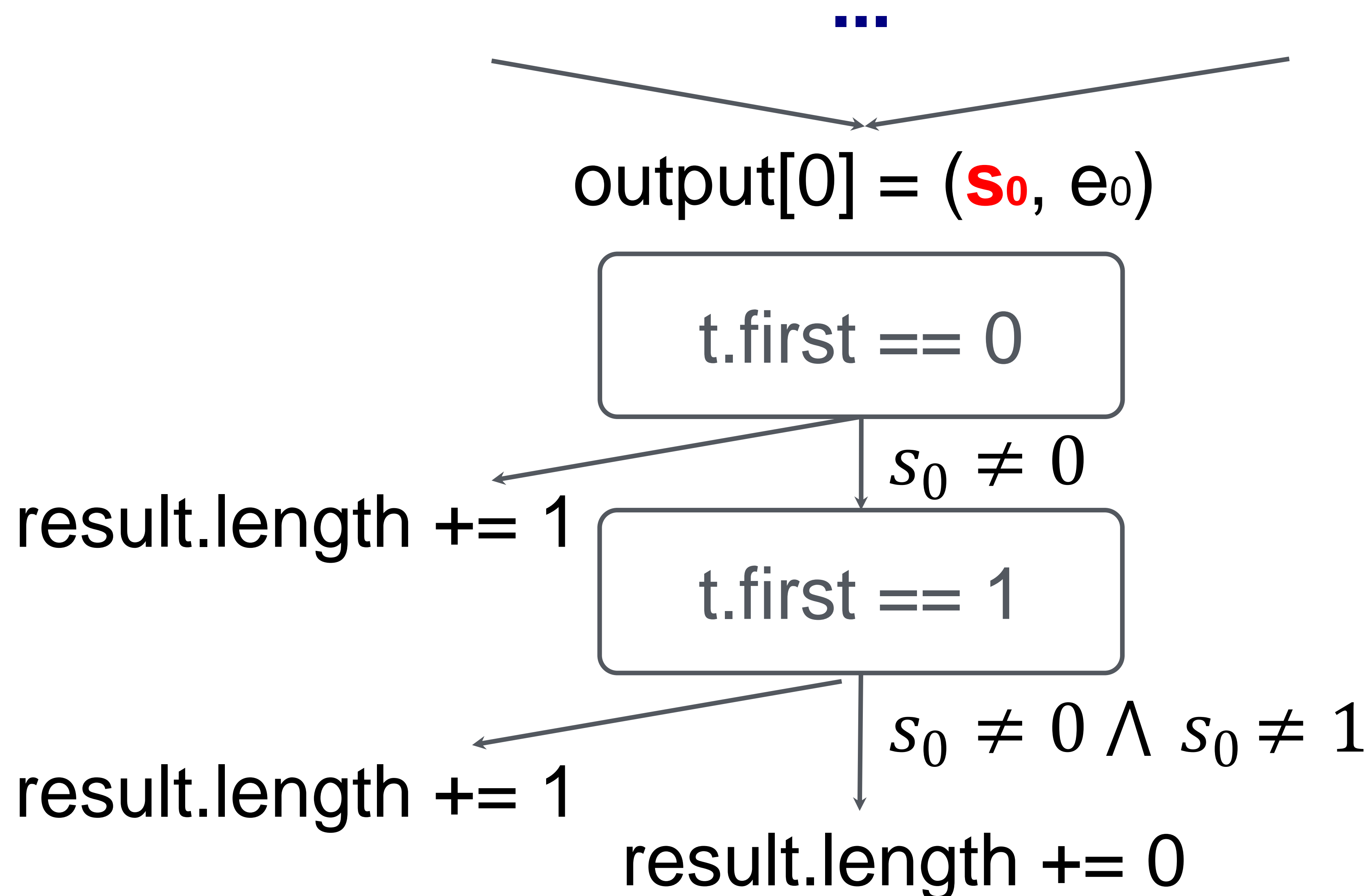
t.first == 0

$s_0 \neq 0$

result.length += 1

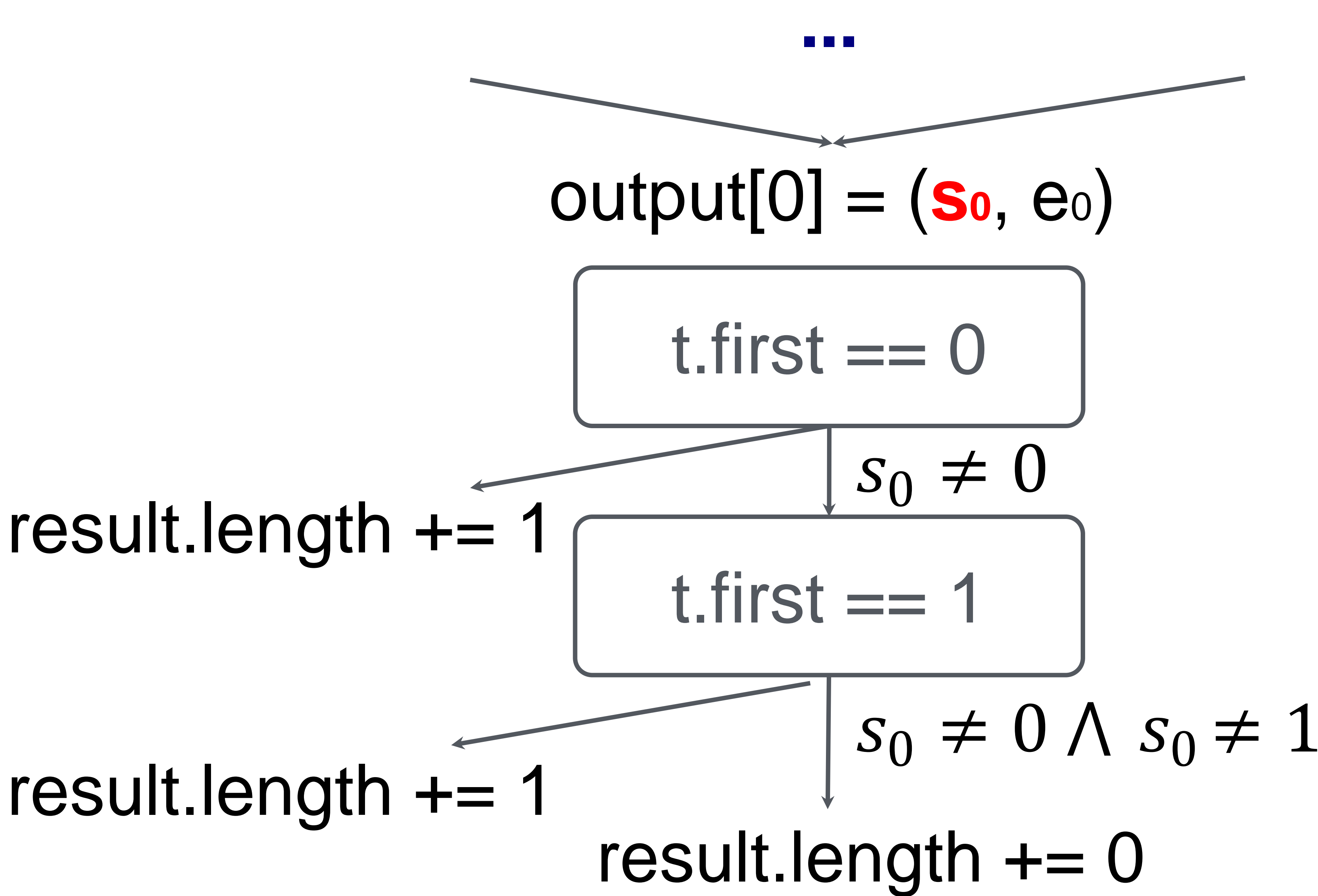
```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((★, e));
        } else {
            output.append((★, e));
        }
    }
    ...
    for (t in output) {
→       if (t.first == 0)
→           result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab




```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((★, e));
        } else {
            output.append((★, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
→       if (t.first == 1)
→           result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab




```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((★, e));
        } else {
            output.append((★, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
→   ObliCheck.Write(MEM_ADDR, result);
}
```

ObliCheck reports the algorithm is **not oblivious**

©2017 RISELab

Optimistic State Merging Incurs False Positives

How should we carefully select variables to merge which do not affect the verification result?

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

Optimistic State Merging Incurs False Positives

How should we carefully select variables to merge which do not affect the verification result?

Solution

Conversely, merge every variable and iteratively un-merge relevant one affecting the verification result

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((★, e));
        } else {
            output.append((★, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

Iteratively Un-Merging State Based on the Verification Result

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

1st iteration:

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

1st iteration:

Save the mapping from symbolic variables (s_0) → to their location of introduction →

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((⋆, e));
        } else {
            output.append((⋆, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

2nd iteration:

Do not merge states at the saved operations ✓
(s_0 was included in the verification condition)
✓

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

...

output[0] = (1, e_0) or (0, e_0)
output.length = 1

→

2nd iteration:

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

...

output[0] = (1, e_0) or (0, e_0)
output.length = 1

t.first == 0

result.length += 1

2nd iteration:

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
→       if (t.first == 0)
→           result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```


©2017 RISELab

2nd iteration:

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
→       if (t.first == 1)
→           result.append(func1(t.second));
    }
    ObliCheck.Write(MEM_ADDR, result);
}
```

...

output[0] = (1, e_0) or (0, e_0)
output.length = 1

t.first == 0

result.length += 1

t.first == 1

result.length += 1

©2017 RISELab

2nd iteration:

```
function Filter() {
    ...
    for (e in privateRecords) {
        if (e < VALUE) {
            output.append((1, e));
        } else {
            output.append((0, e));
        }
    }
    ...
    for (t in output) {
        if (t.first == 0)
            result.append(func0(t.second));
        if (t.first == 1)
            result.append(func1(t.second));
    }
→   ObliCheck.Write(MEM_ADDR, result);
}
```

...

output[0] = (1, e_0) or (0, e_0)
output.length = 1

t.first == 0

result.length += 1

t.first == 1

result.length += 1

ObliCheck reports the algorithm is **oblivious**

©2017 RISELab

Benefit and Cost of Iterative Refinement

Benefit
Achieves better accuracy by only unmerging relevant states
Cost
Extra program runs are added, hence increases the program analysis time

Cost in reality
Turns out the extra cost is tolerable because
(1) Our target programs are intended to be oblivious, extra # of iteration is small
(2) The first iterations are executed quickly, since most paths are merged

©2017 RISELab

Evaluation

Benchmark Programs

Filter: Oblivious SQL operation devised in Opaque [Zheng et al., USENIX NSDI 2017]

Tag & Apply: Tag a bit to data records and apply corresponding function (Our second oblivious program example)

MapReduce: Oblivious MapReduce [Ohrimenko et al., ACM CCS 2015]

DecisionTree: Oblivious inference on a trained decision tree [Ohrimenko et al., USENIX Security 2015]

©2017 RISELab

Does ObliCheck Correctly Judge the Obliviousness?

○: Oblivious ×: Non-oblivious : Correct : Wrong

©2017 RISELab

ObliCheck Correctly Classifies Oblivious Programs

○: Oblivious ×: Non-oblivious [green]: Correct [red]: Wrong

	Oblivious?	Taint Analysis	ObliCheck
Filter	○ (Correct)	× (Wrong)	○ (Correct)
Tag & Apply	○ (Correct)	× (Wrong)	○ (Correct)
MapReduce	× (Correct)	× (Correct)	× (Correct)
Decision Tree	○ (Correct)	× (Wrong)	○ (Correct)

©2017 RISELab

ObliCheck Perform Better Than Symbolic Execution

	Symbolic Execution Time (s)	ObliCheck Time (s)	Speed Up (×)
Filter	14970.5	0.4	35900.4
Tag & Apply	5148.2	0.3	11167.4
MapReduce	8154.9	9.7	212.8
Decision Tree	9305.5	0.2	50300.0

©2017 RISELab

Conclusion

- Oblivious algorithms use unobservable space to optimize the performance, which complicates the verification process
- ObliCheck distinguishes unobservable state to optimize the verification of such algorithms
- ObliCheck merges and iteratively unmerges state based on domain specific knowledge to check oblivious program efficiently and accurately

json@berkeley.edu