SMU Singapore Management University | School of Computing and Information Systems

Boosting Symbolic Execution for Vulnerability Detection

Dissertation Defense

Haoxin Tu

Committee Members: Prof. Lingxiao Jiang, Prof. Xuhua Ding, Prof. David Lo, Prof. Marcel Böhme

May 9th, 2025

Outline

SMU
SINGAPORE MANAGEMENT UNIVERSITY

School of
Computing and
Information Systems

Outline

- **Background**
  - Software vulnerabilities
  - Symbolic execution

Outline

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Background**
  - Software vulnerabilities
  - Symbolic execution
- **Motivation**
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objectives

Outline

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

☐ Background

- ➢ Software vulnerabilities
- ➢ Symbolic execution

☐ Motivation

- ➢ What limitations prevent current symbolic execution for vulnerability detection?
- ➢ Thesis statement and research objectives

☐ Methodology

- ➢ **SymLoc:** A new memory model for symbolic execution
- ➢ **FastKLEE and Vital:** Two new path exploration for symbolic execution
- ➢ **Cottontail:** A new structured input generation for symbolic execution

Outline

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Background**
  - Software vulnerabilities
  - Symbolic execution
- **Motivation**
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objectives
- **Methodology**
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- **Future work**
  - Extension of current solutions
  - Combine program analysis with GenAI

Outline

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

☐ Background

- ➢ Software vulnerabilities
- ➢ Symbolic execution

☐ Motivation

- ➢ What limitations prevent current symbolic execution for vulnerability detection?
- ➢ Thesis statement and research objectives

☐ Methodology

- ➢ **SymLoc:** A new memory model for symbolic execution
- ➢ **FastKLEE and Vital:** Two new path exploration for symbolic execution
- ➢ **Cottontail:** A new structured input generation for symbolic execution

☐ Future work

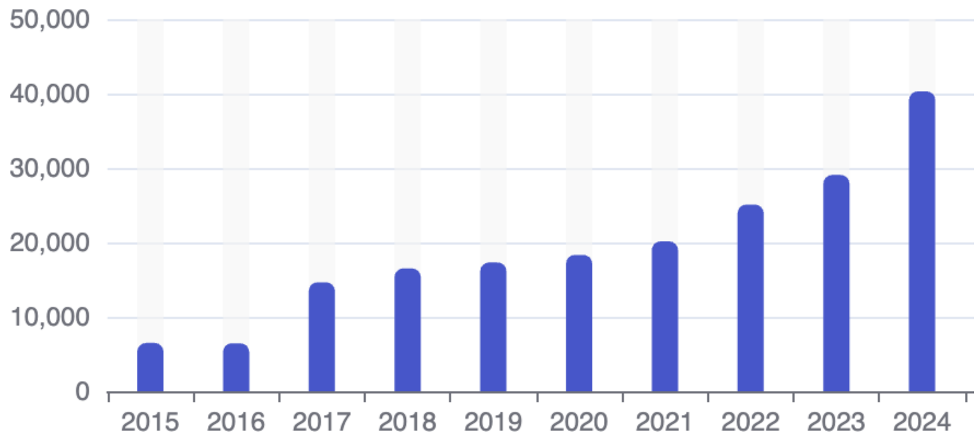
- ➢ Extension of current solutions
- ➢ Combine program analysis with GenAI

☐ Conclusion and Acknowledgment

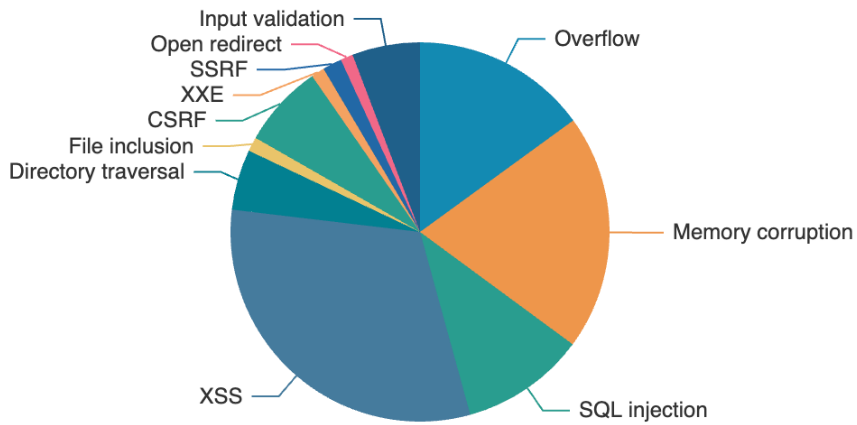
Background: software are everywhere

https://www.coderus.com/software-101-a-complete-guide-to-the-different-types-of-software/

Background: inevitable software defects

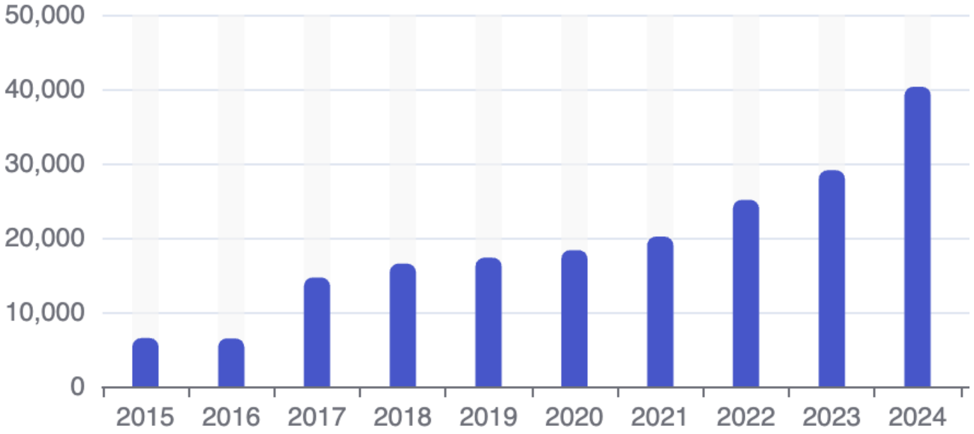


Vulnerability By Yeas (2015-2024)
(https://www.cvedetails.com/vulnerabilities-by-years.php)

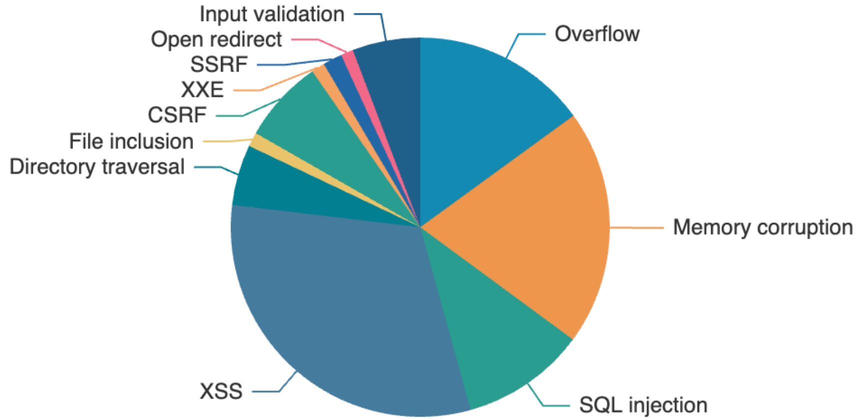


Vulnerability By Type (2015-2024)
(https://www.cvedetails.com/vulnerabilities-by-types.php)

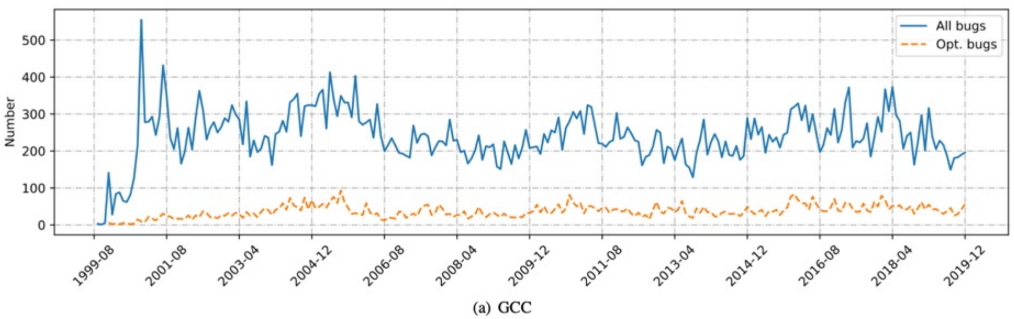
Background: inevitable software defects

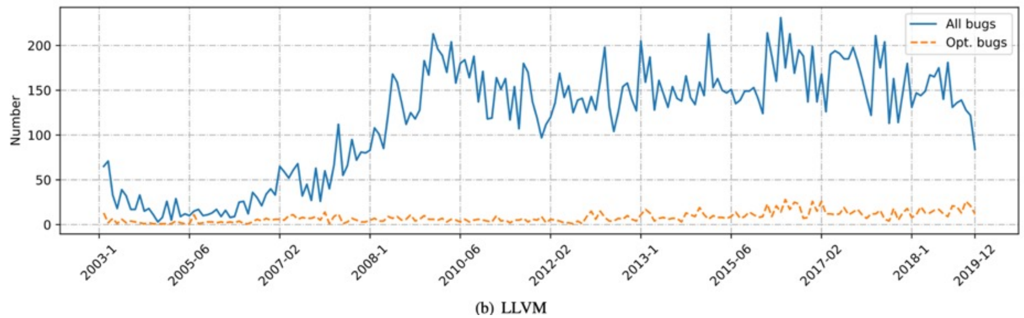


Vulnerability By Yeas (2015-2024)
(https://www.cvedetails.com/vulnerabilities-by-years.php)



Vulnerability By Type (2015-2024)
(https://www.cvedetails.com/vulnerabilities-by-types.php)

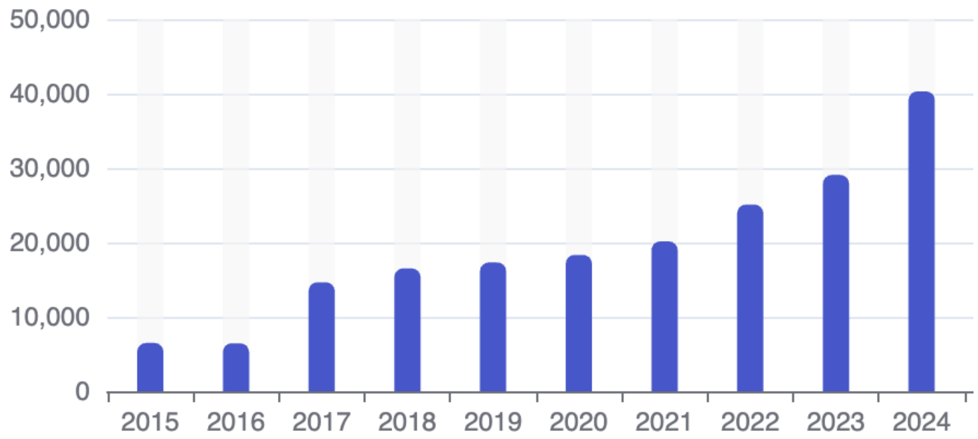




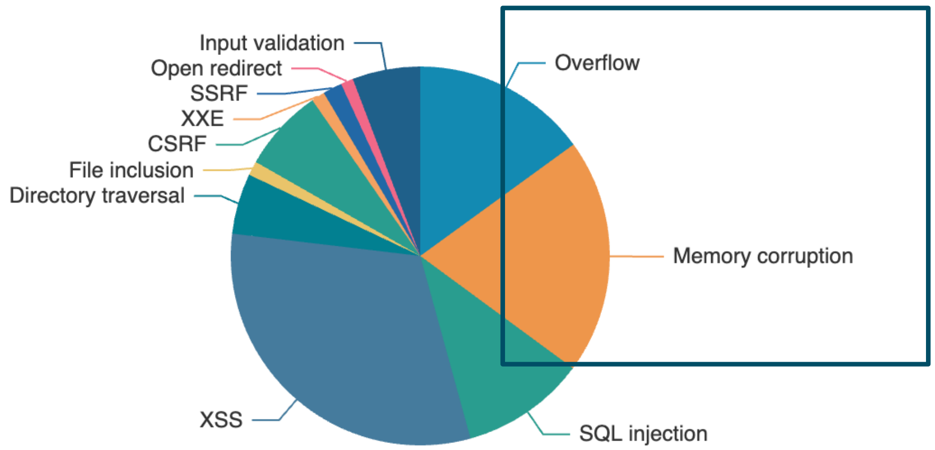
Bug trends on two mainstream compilers (GCC and LLVM)

Background: inevitable software defects

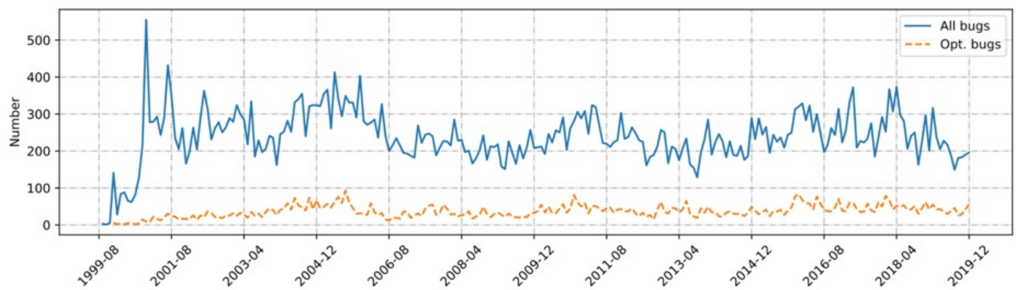
SMU SINGAPORE MANAGEMENT UNIVERSITY
School of Computing and Information Systems



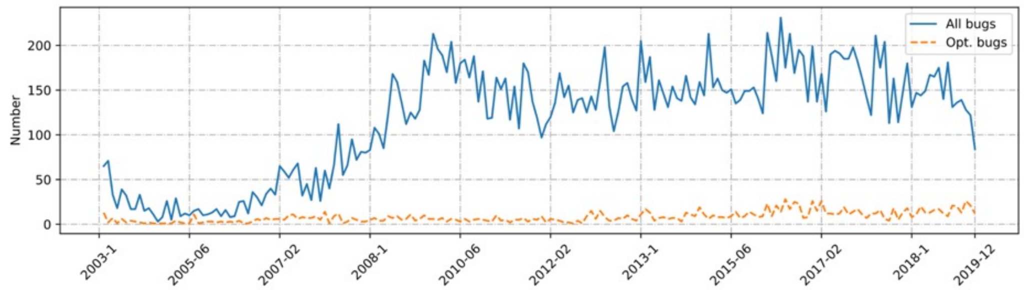
Vulnerability By Yeas (2015-2024)
(https://www.cvedetails.com/vulnerabilities-by-years.php)



Vulnerability By Type (2015-2024)
(https://www.cvedetails.com/vulnerabilities-by-types.php)



(a) GCC



(b) LLVM

Bug trends on two mainstream compilers (GCC and LLVM)

Background: symbolic execution (1/3)

SMU Singapore Management University | School of Computing and Information Systems

❑ What is symbolic execution?

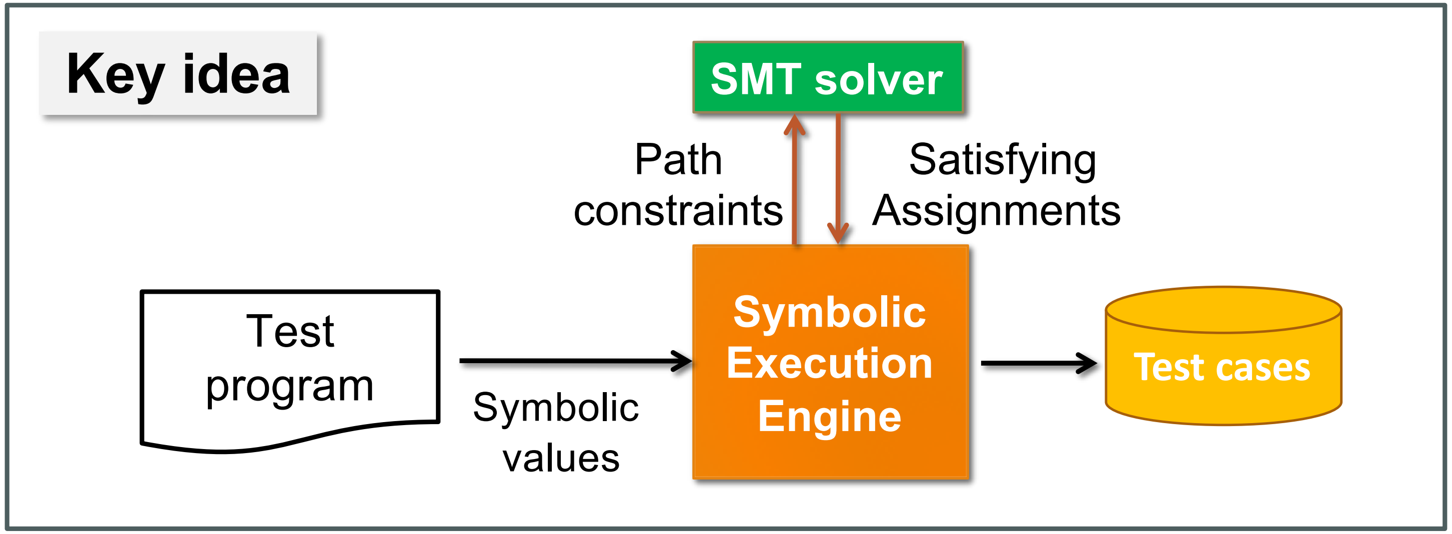
- Proposed in 1976 [1], one of the most popular program analysis techniques, which scales for many **software testing** and **computer security** applications

[1] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.

Background: symbolic execution (1/3)

❑ What is symbolic execution?

➢ Proposed in 1976 [1], one of the most popular program analysis techniques, which scales for many **software testing** and **computer security** applications



[1] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.

Background: symbolic execution (2/3)

❑ A toy example

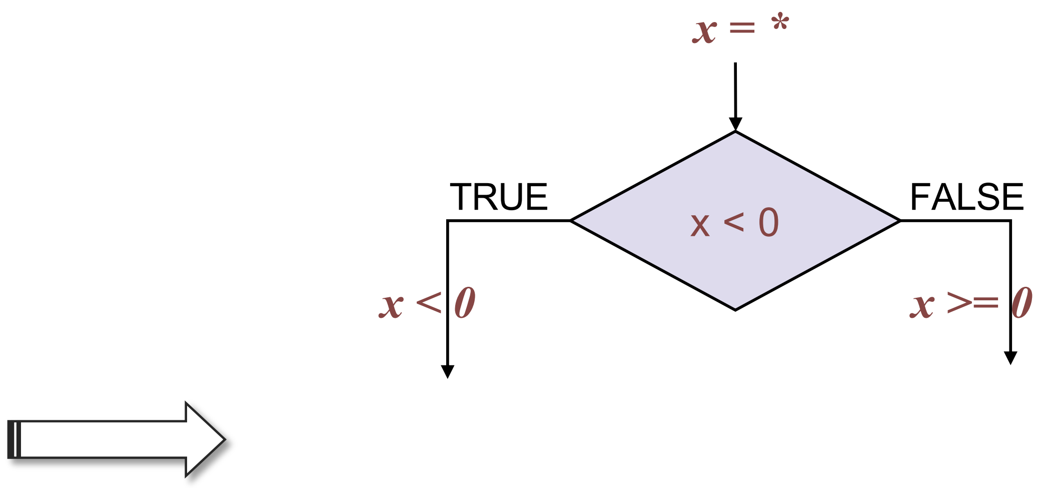
```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```

Background: symbolic execution (2/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- A toy example

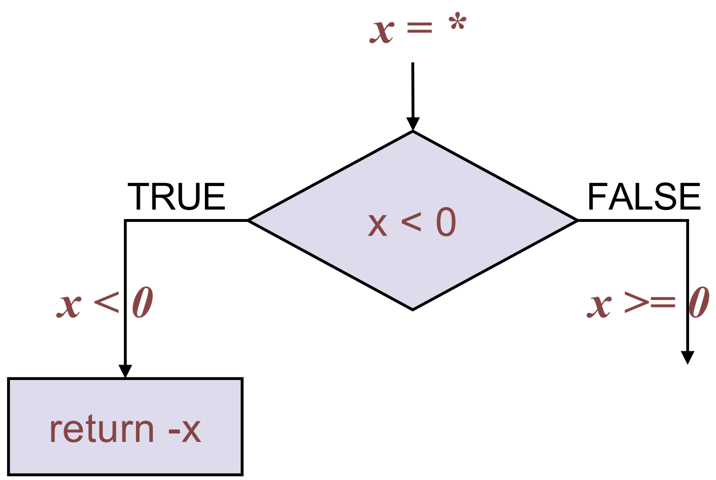
```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```



Background: symbolic execution (2/3)

- A toy example

```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```

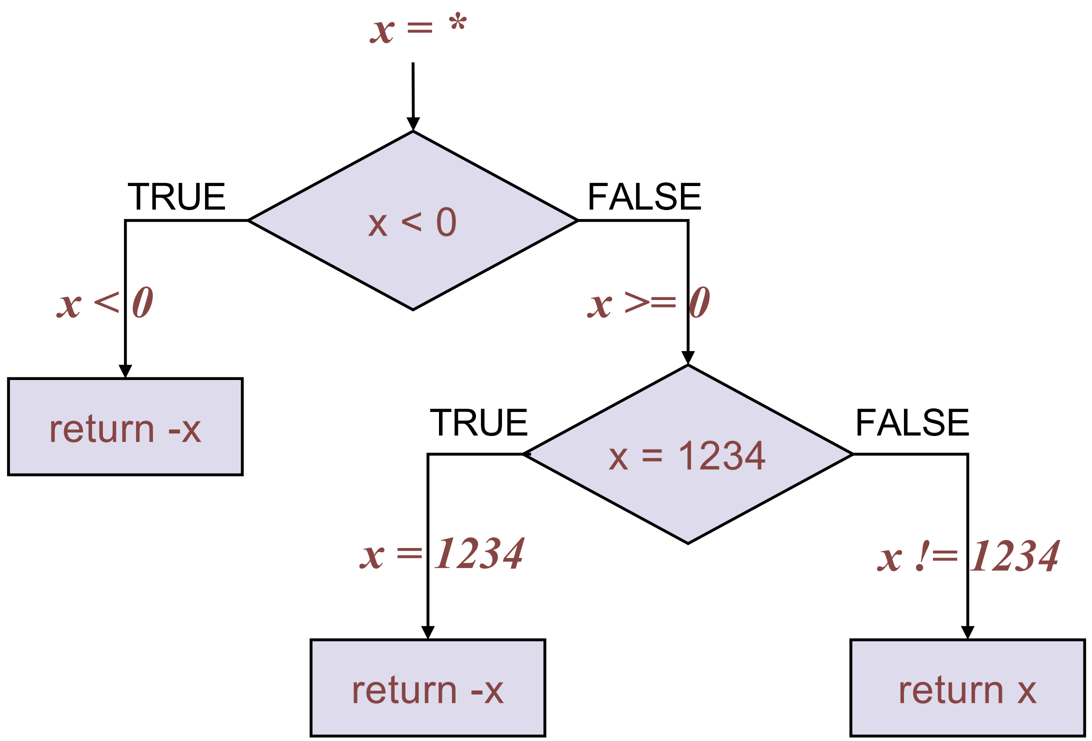


Background: symbolic execution (2/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- A toy example

```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```

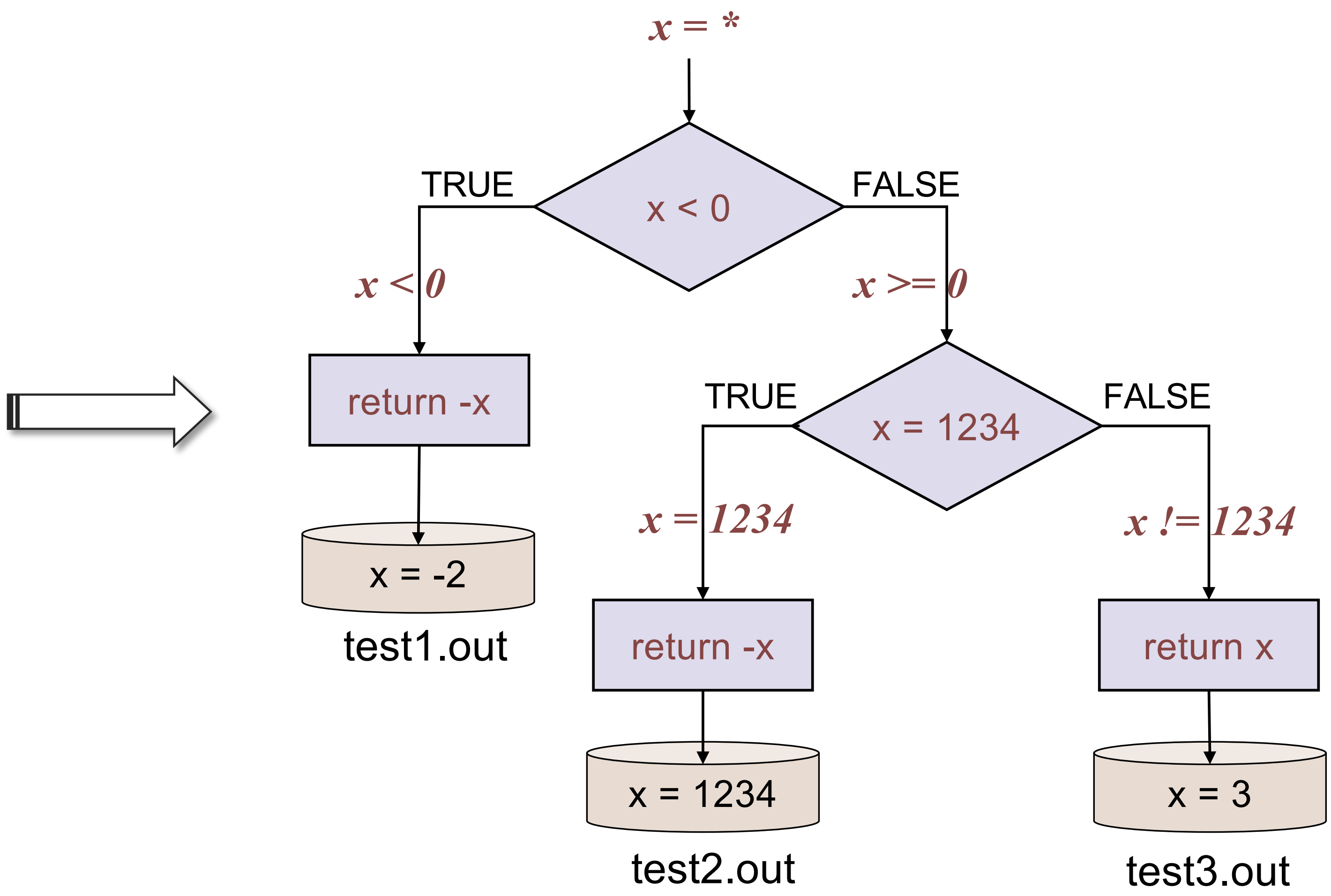


Background: symbolic execution (2/3)

SMU Singapore Management University | School of Computing and Information Systems

- A toy example

```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```

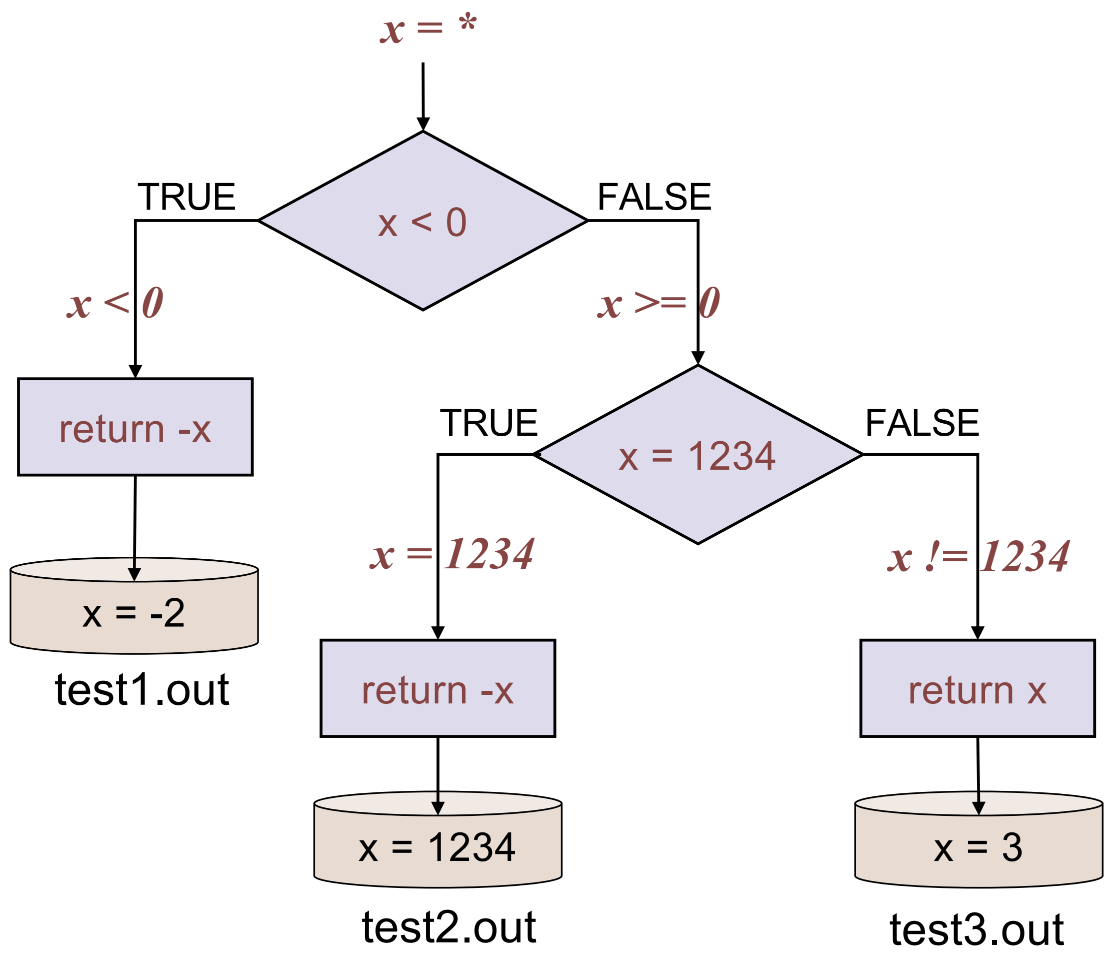


Background: symbolic execution (2/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

❑ A toy example

```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```



x < 0;
x >= 0 && x = 1234;
x >= 0 && x != 1234;
(path constraints)

⇨

test1.out
test2.out
test3.out
(test cases)

Background: symbolic execution (3/3)

Background: symbolic execution (3/3)

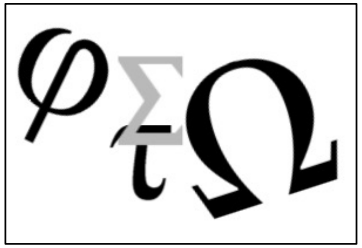
❑ Why symbolic execution could work?

- Execute the program with symbolic inputs
- Represent **equivalent execution paths** with **path constraints**
- **Solve path constraints** to obtain one representative input that exercises the program to go down that specific path

Background: symbolic execution (3/3)

❑ Why symbolic execution could work?

- Execute the program with symbolic inputs
- Represent **equivalent execution paths** with **path constraints**
- **Solve path constraints** to obtain one representative input that exercises the program to go down that specific path



Path constraints

+

Z3 STP
CVC4

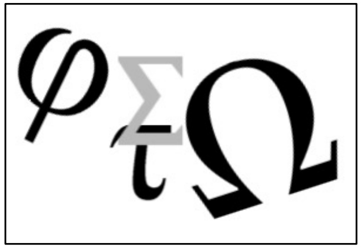
Constraint Solver

⇒ **Worked!**

Background: symbolic execution (3/3)

❑ Why symbolic execution could work?

- Execute the program with symbolic inputs
- Represent **equivalent execution paths** with **path constraints**
- **Solve path constraints** to obtain one representative input that exercises the program to go down that specific path



Path constraints

+

Z3 STP
CVC4

Constraint Solver

⇨ **Worked!**

- Many applications
  - high-coverage test generation, automated debugging, automated program repair, exploit generation, wireless sensor networks, online gaming, …

Background: symbolic execution (3/3)

School of Computing and Information Systems

- Why symbolic execution could work?
  - Execute the program with symbolic inputs

Is current symbolic execution effective enough to detect vulnerabilities in practice?

Path constraints

Constraint Solver

  - Many applications
    - high-coverage test generation, automated debugging, automated program repair, exploit generation, wireless sensor networks, online gaming, …

Background: symbolic execution (3/3)

- Why symbolic execution could work?
  - Execute the program with symbolic inputs

Is current symbolic execution effective enough to detect vulnerabilities in practice?

NO, and why?

Path constraints **Constraint Solver**

  - Many applications
    - high-coverage test generation, automated debugging, automated program repair, exploit generation, wireless sensor networks, online gaming, …

Outline

- Background
  - Software vulnerabilities
  - Symbolic execution
- **Motivation**
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objectives
- Methodology
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- Future work
  - Extension of current solutions
  - Combine program analysis with GenAI
- Conclusion and Acknowledgment

Motivation Example

School of Computing and Information Systems

```
void  buggy () {

   // Vuln1: require complete memory modeling
   void * from = malloc (100);
   void * to = malloc (100);
   if (to > from) { ... }
   if (from > to) {
      vulnerable_func1();
    }

   // Vuln2: require handling path explosion
   ... // deeply nested
      vulnerable_func2();
   ...

   // Vuln3: require structured test inputs
   if (input = "{....}") {
      // application logic
      vulnerable_func3();
   } else {
      earlier_termination();
   }
}
```

Motivation Example

School of Computing and Information Systems

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
     vulnerable_func1();
  }

  // Vuln2: require handling path explosion
  ... // deeply nested
     vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Motivation Example


```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
      vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  ... // deeply nested
      vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
      // application logic
      vulnerable_func3();
  } else {
      earlier_termination();
  }
}
```

Motivation Example

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  ... // deeply nested
     vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Motivation Example

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void  buggy () {
  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { … }
  if (from > to) {
      vulnerable_func1();
  }

  // Vuln2: require handling path explosion
  … // deeply nested
      vulnerable_func2();
  …

  // Vuln3: require structured test inputs
  if (input = "{….}") {
      // application logic
      vulnerable_func3();
  } else {
      earlier_termination();
  }
}
```

Can existing symbolic execution effectively detect the three vulnerabilities?

Motivation Example

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void  buggy () {
    // Vuln1: require complete memory modeling
    void * from = malloc (100);
    void * to = malloc (100);
    if (to > from) { … }
    if (from > to) {
        vulnerable_func1();
    }

    // Vuln2: require handling path explosion
    … // deeply nested
        vulnerable_func2();
    …

    // Vuln3: require structured test inputs
    if (input = "{….}") {
        // application logic
        vulnerable_func3();
    } else {
        earlier_termination();
    }
}
```

Can existing symbolic execution effectively detect the three vulnerabilities?

NO

Limitation 1: Limited memory modeling

- ❑ **How does the engine handle dynamic memory allocations?**

Limitation 1: Limited memory modeling

SMU Singapore Management University | School of Computing and Information Systems

❑ How does the engine handle dynamic memory allocations?

1. int* array = malloc (100 * sizeof (int));
2. array [i] = 10; *// i is symbolic*
3. assert(array[j] != 0); *// j is symbolic*

Limitation 1: Limited memory modeling

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

❑ How does the engine handle dynamic memory allocations?

1. int* array = malloc (100 * sizeof (int));
2. array [i] = 10; *// i is symbolic*
3. assert(array[j] != 0); *// j is symbolic*

➢ **Possible solutions**

1. Fully symbolic (ASE'17)
   - consider **any possible** outcome

   ⇨ **N** states
   accurate but not scale

2. Fully concrete (OSDI'08)
   - consider **one possible** outcome

   ⇨ **1** state
   scale but not accurate

3. Partial symbolic and concrete (S&P'12)
   - concretize writes,
   - Possible ranges to read

   ⇨ **K** states
   scale but (in) accurate

Limitation 1: Limited memory modeling

- ❑ **How does the engine handle dynamic memory allocations?**

1. int* array = malloc (100 * sizeof (int));
2. array [i] = 10; *// i is symbolic*
3. assert(array[j] != 0); *// j is symbolic*

Fixed addresses

Problematic read/write

- **Possible solutions**
  1. Fully symbolic (ASE'17)
     - consider **any possible** outcome
  2. Fully concrete (OSDI'08)
     - consider **one possible** outcome
  3. Partial symbolic and concrete (S&P'12)
     - concretize writes,
     - Possible ranges to read

⇒ **N** states
accurate but not scale

⇒ **1** state
scale but not accurate

⇒ **K** states
scale but (in) accurate

Limitation 1: Limited memory modeling

- **How does the engine handle dynamic memory allocations?**

1. int* array = malloc (100 * sizeof (int));
2. array [i] = 10; *// i is symbolic*
3. assert(array[j] != 0); *// j is symbolic*

Fixed addresses

Problematic read/write

- **Possible solutions**
  1. Fully symbolic (ASE'17)
     - consider **any possible** outcome
  2. Fully concrete (OSDI'08)
     - consider **one possible** outcome
  3. Partial symbolic and concrete (S&P'12)
     - concretize writes,
     - Possible ranges to read

⇒ **N** states accurate but not scale

⇒ **1** state scale but not accurate

⇒ **K** states scale but (in) accurate

We need a more complete and precise memory modeling !

Motivation Example

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { … }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  … // deeply nested
     vulnerable_func2();
  …

  // Vuln3: require structured test inputs
  if (input = "{….}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Motivation Example

School of Computing and Information Systems

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  ... // deeply nested
     vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Can existing symbolic execution effectively detect the three vulnerabilities?

Motivation Example

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void  buggy () {
  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  ... // deeply nested
     vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Can existing symbolic execution effectively detect the three vulnerabilities?

NO, two remaining

Limitation 2: path explosion

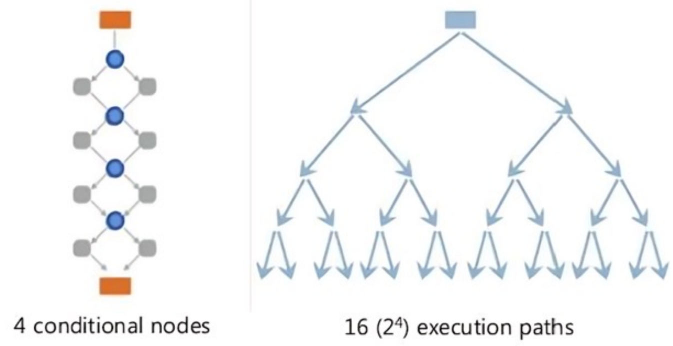
- ❑ **How does symbolic execution deal with path explosion?**

Limitation 2: path explosion

❑ How does symbolic execution deal with path explosion?

```
void process(char input[3]) {
    int counter = 0;
    if (input[0] == 'a') counter++;
    if (input[1] == 'b') counter++;
    if (input[2] == 'c') counter++;
    if (counter >= 3) success();
    error();
}
```

- Exponentially many execution paths

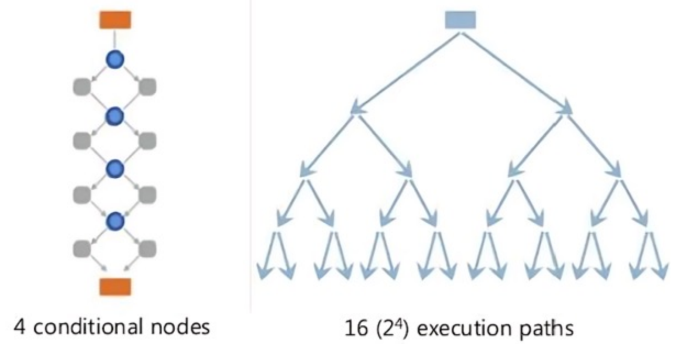


Limitation 2: path explosion

❑ How does symbolic execution deal with path explosion?

```
void process(char input[3]) {
    int counter = 0;
    if (input[0] == 'a') counter++;
    if (input[1] == 'b') counter++;
    if (input[2] == 'c') counter++;
    if (counter >= 3) success();
    error();
}
```

- Exponentially many execution paths

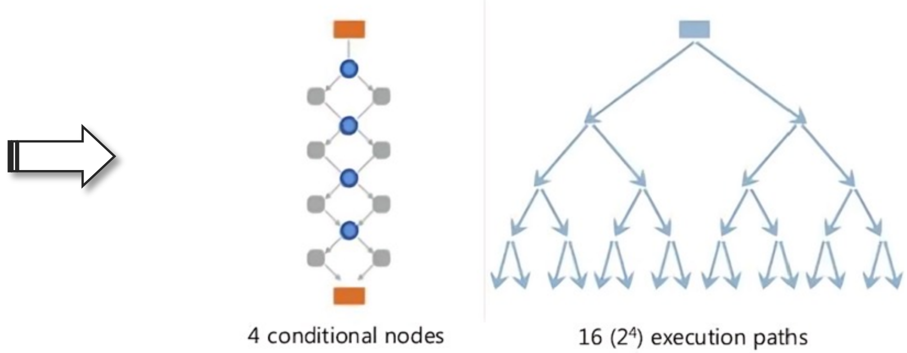


- **Efficient solutions**
  - With specialized optimizations (HotOS’13)
  - With code transformation (ECOOP’18)

Limitation 2: path explosion

❑ How does symbolic execution deal with path explosion?

```
void process(char input[3]) {
    int counter = 0;
    if (input[0] == 'a') counter++;
    if (input[1] == 'b') counter++;
    if (input[2] == 'c') counter++;
    if (counter >= 3) success();
    error();
}
```



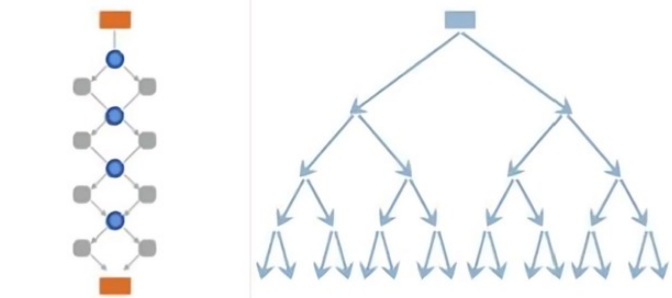
- **Efficient solutions**
  - With specialized optimizations (HotOS'13)
  - With code transformation (ECOOP'18)
- **Effective solutions**
  - Random search (DFS and BFS)
  - Heuristic guided search (OSDI'08)
    - coverage, instruction, etc.

Limitation 2: path explosion

❑ How does symbolic execution deal with path explosion?

```
void process(char input[3]) {
    int counter = 0;
    if (input[0] == 'a') counter++;
    if (input[1] == 'b') counter++;
    if (input[2] == 'c') counter++;
    if (counter >= 3) success();
    error();
}
```

• Exponentially many execution paths



4 conditional nodes

16 (2^4) execution paths

- **Efficient solutions**
  - With specialized optimizations (HotOS'13)
  - With code transformation (ECOOP'18)

- **Effective solutions**
  - Random search (DFS and BFS)
  - Heuristic guided search (OSDI'08)
    - coverage, instruction, etc.

Inefficient path search

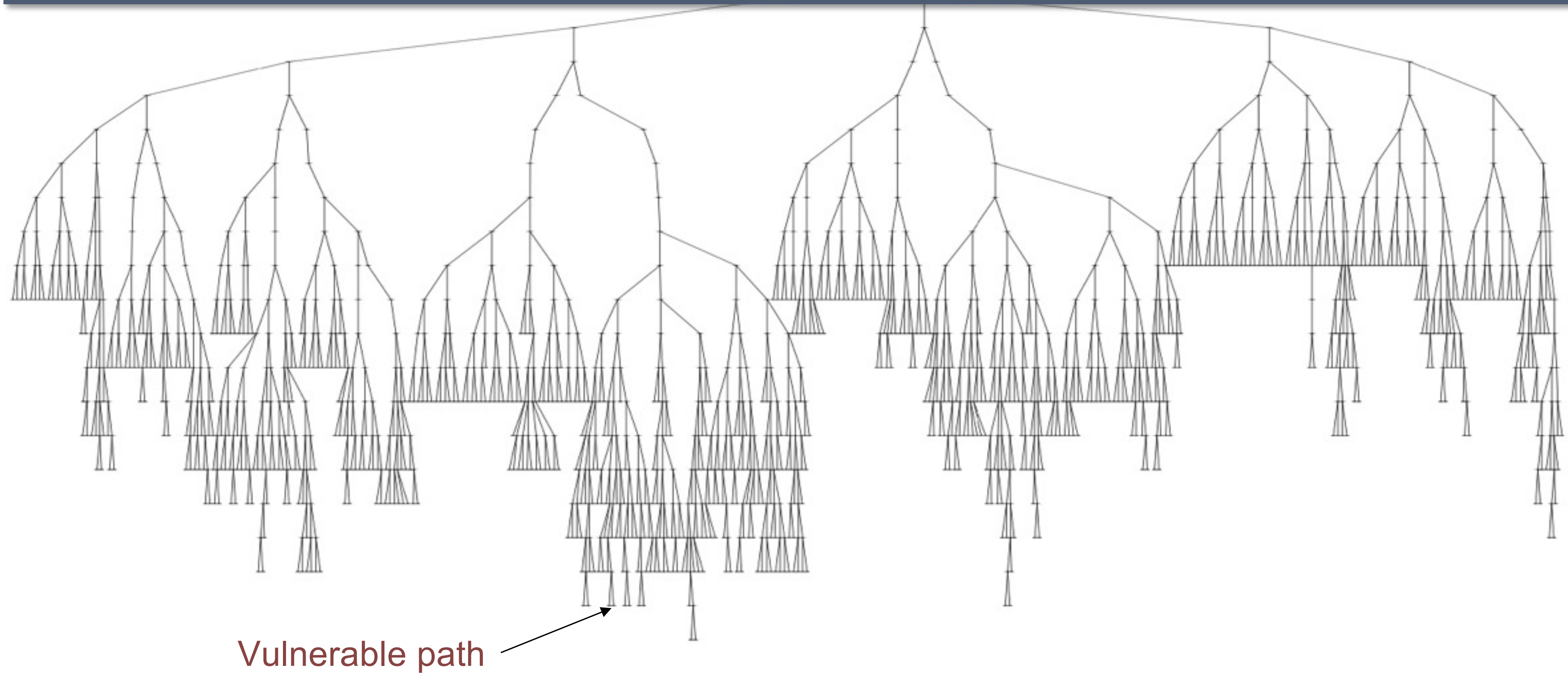
No unified targets

Ineffective path search

Path explosion visualization

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

How to search vulnerable paths in rather complex software systems?



Path explosion visualization

How to search vulnerable paths in rather complex software systems?

We need a more efficient and effective path exploration !

Motivation Example

SMU Singapore Management University | School of Computing and Information Systems

```
void  buggy () {
  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { … }
  if (from > to) {
      vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  … // deeply nested
      vulnerable_func2();
  …

  // Vuln3: require structured test inputs
  if (input = "{….}") {
      // application logic
      vulnerable_func3();
  } else {
      earlier_termination();
  }
}
```

Motivation Example

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void  buggy () {
    // Vuln1: require complete memory modeling
    void * from = malloc (100);
    void * to = malloc (100);
    if (to > from) { ... }
    if (from > to) {
        vulnerable_func1();
    }

    // Vuln2: require handling path explosion
    ... // deeply nested
        vulnerable_func2();
    ...

    // Vuln3: require structured test inputs
    if (input = "{....}") {
        // application logic
        vulnerable_func3();
    } else {
        earlier_termination();
    }
}
```

Can existing symbolic execution effectively detect the three vulnerabilities?

Motivation Example

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

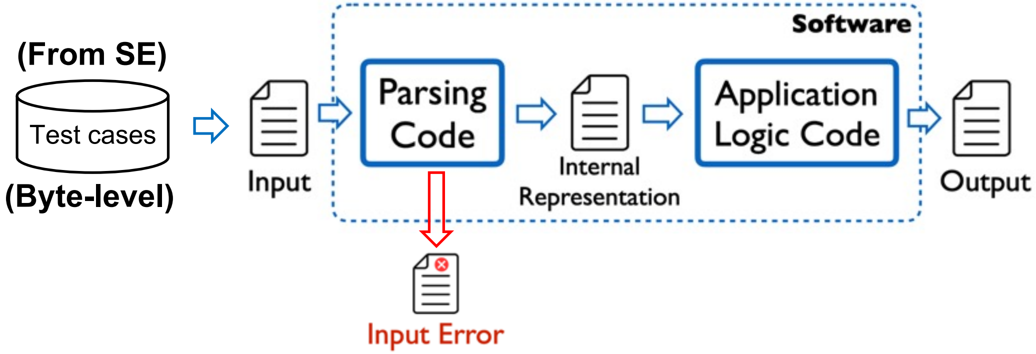
```
void  buggy () {
  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
      vulnerable_func1();
  }

  // Vuln2: require handling path explosion
  ... // deeply nested
      vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
      // application logic
      vulnerable_func3();
  } else {
      earlier_termination();
  }
}
```

Can existing symbolic execution effectively detect the three vulnerabilities?

NO, one remaining still

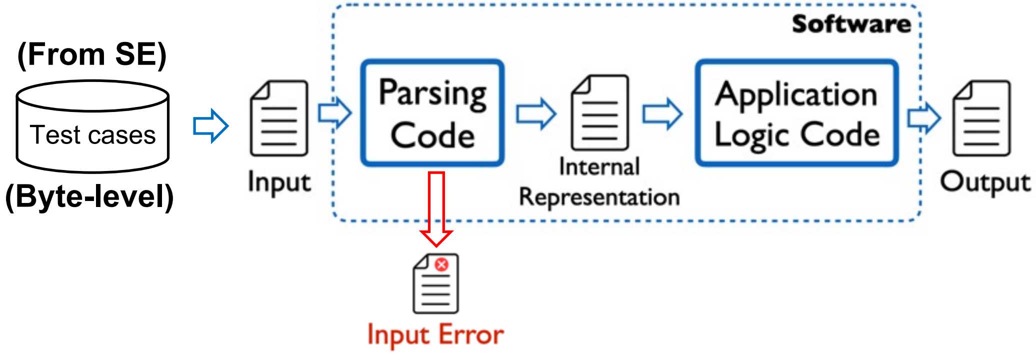
Limitation 3: structured input generation

❑ **How does symbolic execution generate structured test cases?**



Limitation 3: structured input generation

❑ **How does symbolic execution generate structured test cases?**

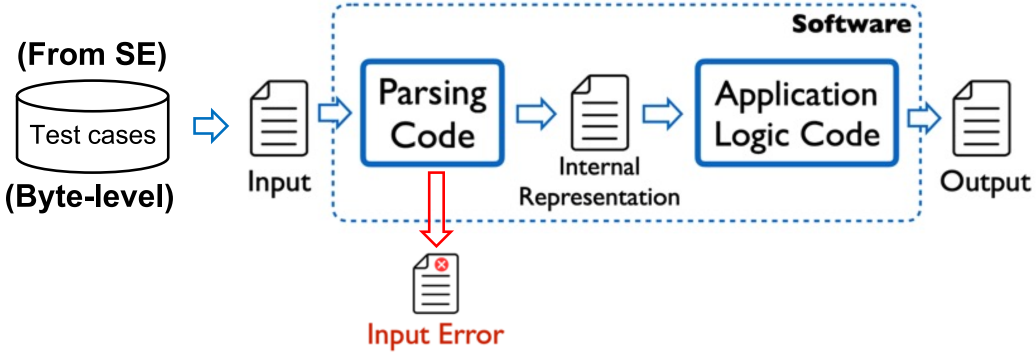


- **Possible solutions**
  - Grammar-based generation (PLDI'08)
    - Use grammar specifications to guide generation
  - Token-level symbolization (ISSTA'21)
    - Treating tokens rather than bytes as symbolic data

Limitation 3: structured input generation

School of Computing and Information Systems

❑ How does symbolic execution generate structured test cases?



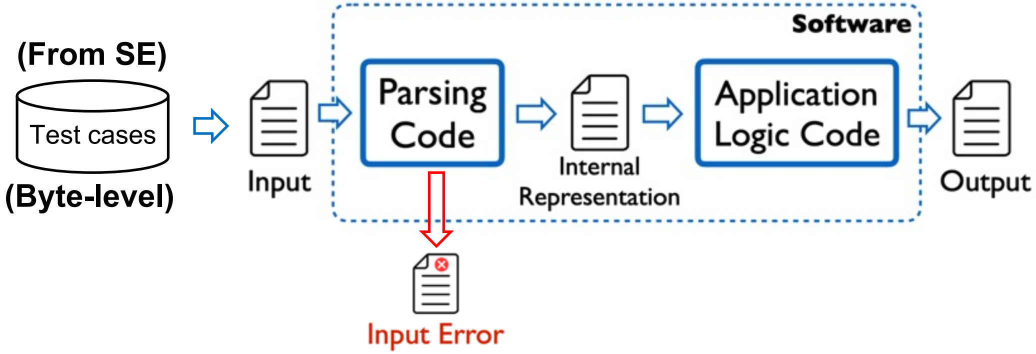
- **Possible solutions**
  - Grammar-based generation (PLDI’08)
    - Use grammar specifications to guide generation
  - Token-level symbolization (ISSTA’21)
    - Treating tokens rather than bytes as symbolic data

Structure-agnostic path selection

Ineffective constraint solving

Limitation 3: structured input generation

❑ How does symbolic execution generate structured test cases?



- **Possible solutions**
  - Grammar-based generation (PLDI'08)
    - Use grammar specifications to guide generation
  - Token-level symbolization (ISSTA'21)
    - Treating tokens rather than bytes as symbolic data

Structure-agnostic path selection

Ineffective constraint solving

We need a new structured test case generation strategy !

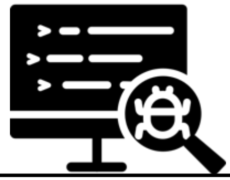
Thesis statement and research objectives

- ***Statement*: This thesis aims to boost symbolic execution** by designing new solutions to alleviate three key limitations in **memory modeling/path exploration/test input generation, for efficient and effective automatic vulnerability detection.**

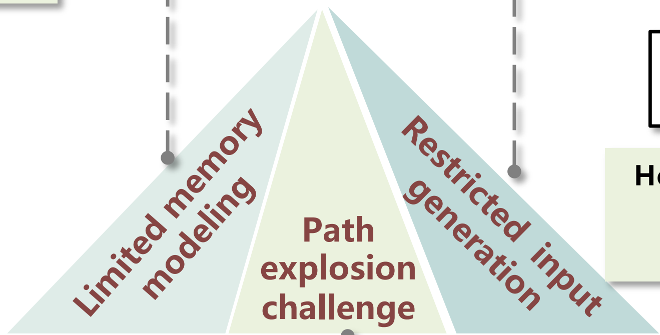
Summary of Key Research Problems

How can we support a more complete memory model for dynamic memory allocations ?

Fixed addresses



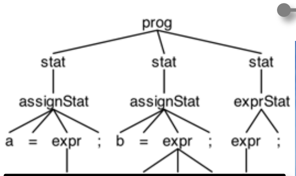
Problematic W/R



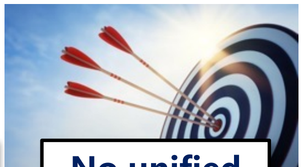
Structure-agnostic path selection

Ineffective constraint solving

How can we perform concolic execution to generate highly structured test inputs for systematically testing parsing programs?



Inefficient path search



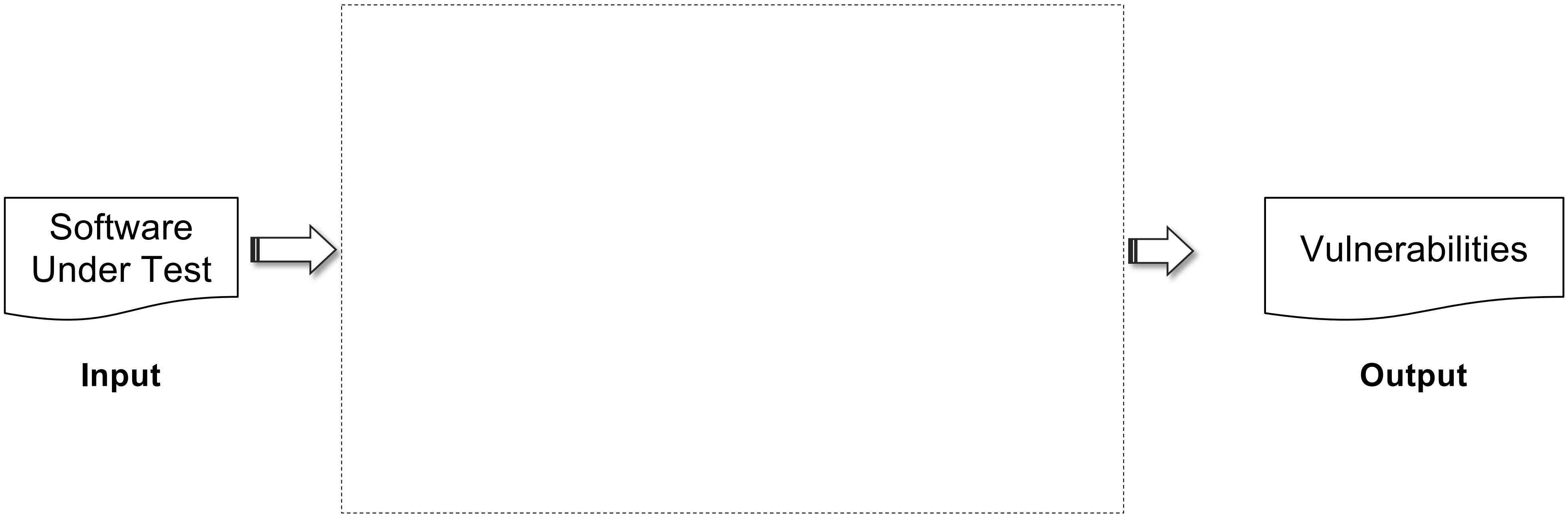
No unified targets

Ineffective path search

How can we efficiently and effectively search vulnerable paths in rather complex software systems?

Thesis statement and research objectives

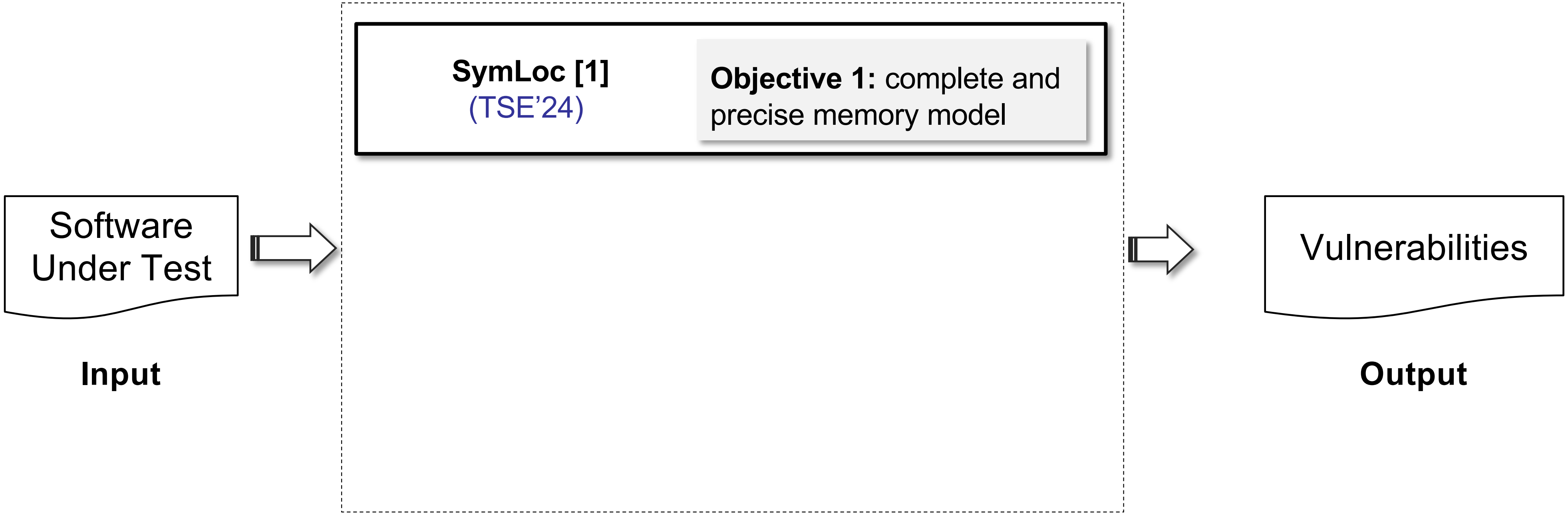
- ***Statement*: This thesis aims to boost symbolic execution** by designing new solutions to alleviate three key limitations in **memory modeling/path exploration/test input generation, for efficient and effective automatic vulnerability detection.**



[1] **Haoxin Tu**, Lingxiao Jiang, and et.al, "*Concretely Mapped Symbolic Memory Locations for Memory Error Detection*" (IEEE TSE).
[2] **Haoxin Tu**, Lingxiao Jiang, and et.al,, "*FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers*", in Tool Demonstrations Track of FSE 2022.
[3] **Haoxin Tu**, Lingxiao Jiang and et.al,. "*Vital: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search.*" arXiv:2408.08772 (2024).
[4] **Haoxin Tu**, Seongmin Lee, and et.al. "*Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation.*" arXiv:2504.17542 (2025).

Thesis statement and research objectives

- ***Statement*: This thesis aims to boost symbolic execution** by designing new solutions to alleviate three key limitations in **memory modeling/path exploration/test input generation, for efficient and effective automatic vulnerability detection.**



[1] **Haoxin Tu**, Lingxiao Jiang, and et.al, "*Concretely Mapped Symbolic Memory Locations for Memory Error Detection*" (IEEE TSE).
[2] **Haoxin Tu**, Lingxiao Jiang, and et.al,, "*FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers*", in Tool Demonstrations Track of FSE 2022.
[3] **Haoxin Tu**, Lingxiao Jiang and et.al,. "*Vital: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search.*" arXiv:2408.08772 (2024).
[4] **Haoxin Tu**, Seongmin Lee, and et.al. "*Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation.*" arXiv:2504.17542 (2025).

Thesis statement and research objectives

SMU Singapore Management University | School of Computing and Information Systems

- ***Statement*: This thesis aims to boost symbolic execution** by designing new solutions to alleviate three key limitations in **memory modeling/path exploration/test input generation, for efficient and effective automatic vulnerability detection.**

Software Under Test

Input

SymLoc [1]
(TSE'24)

Objective 1: complete and precise memory model

FastKLEE [2]
(FSE'22 Demonstration)
Vital [3]
(under review)

Objective 2: efficient and effective path exploration

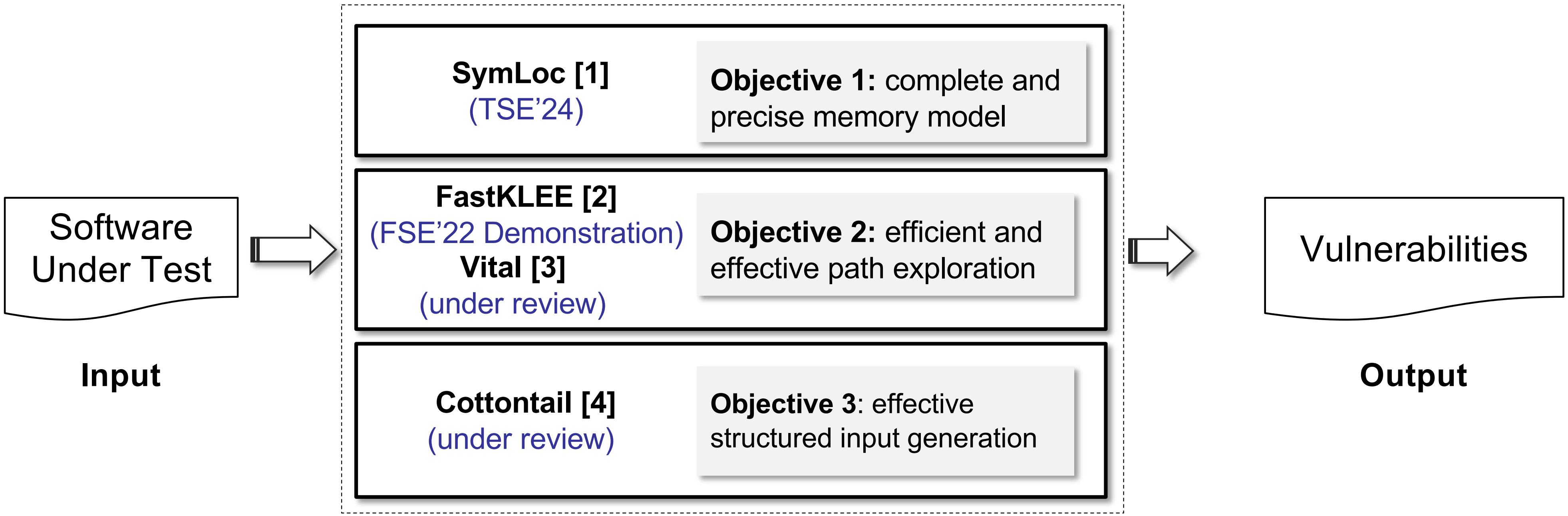
Vulnerabilities

Output

[1] **Haoxin Tu**, Lingxiao Jiang, and et.al, "*Concretely Mapped Symbolic Memory Locations for Memory Error Detection*" (IEEE TSE).
[2] **Haoxin Tu**, Lingxiao Jiang, and et.al,, "*FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers*", in Tool Demonstrations Track of FSE 2022.
[3] **Haoxin Tu**, Lingxiao Jiang and et.al,. "*Vital: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search.*" arXiv:2408.08772 (2024).
[4] **Haoxin Tu**, Seongmin Lee, and et.al. "*Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation.*" arXiv:2504.17542 (2025).

Thesis statement and research objectives

- ***Statement*: This thesis aims to boost symbolic execution** by designing new solutions to alleviate three key limitations in **memory modeling/path exploration/test input generation, for efficient and effective automatic vulnerability detection.**



[1] **Haoxin Tu**, Lingxiao Jiang, and et.al, “*Concretely Mapped Symbolic Memory Locations for Memory Error Detection*” (IEEE TSE).
[2] **Haoxin Tu**, Lingxiao Jiang, and et.al,, “*FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers*”, in Tool Demonstrations Track of FSE 2022.
[3] **Haoxin Tu**, Lingxiao Jiang and et.al,. "*Vital: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search.*" arXiv:2408.08772 (2024).
[4] **Haoxin Tu**, Seongmin Lee, and et.al. "*Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation.*" arXiv:2504.17542 (2025).

Motivation Example (with our solutions)

SMU Singapore Management University | School of Computing and Information Systems

```
void  buggy () {
  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { … }
  if (from > to) {
     vulnerable_func1();
  }

  // Vuln2: require handling path explosion
  … // deeply nested
     vulnerable_func2();
  …

  // Vuln3: require structured test inputs
  if (input = "{….}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Motivation Example (with our solutions)

SMU Singapore Management University | School of Computing and Information Systems

```
void  buggy () {
    // Vuln1: require complete memory modeling
    void * from = malloc (100);
    void * to = malloc (100);
    if (to > from) { … }
    if (from > to) {
        vulnerable_func1();
    }

    // Vuln2: require handling path explosion
    … // deeply nested
        vulnerable_func2();
    …

    // Vuln3: require structured test inputs
    if (input = "{….}") {
        // application logic
        vulnerable_func3();
    } else {
        earlier_termination();
    }
}
```

Can our boosted symbolic execution effectively detect the three vulnerabilities?

YES

Outline

- Background
  - Software vulnerabilities
  - Symbolic execution
- Motivation
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objects
- **Methodology**
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- Future work
  - Extension of current solutions
  - Combine Program Analysis with GenAI
- Conclusion and Acknowledgment

A New Memory Model: insights

A New Memory Model: insights

- Three fundamental designs are required

A New Memory Model: insights

- **Three fundamental designs are required**
  A. Symbolization of addresses and modeling them into path constraints

A New Memory Model: insights

- **Three fundamental designs are required**
  A. Symbolization of addresses and modeling them into path constraints
  B. Practical read/write operation from/to symbolic addresses

A New Memory Model: insights

➢ **Three fundamental designs are required**

A. Symbolization of addresses and modeling them into path constraints

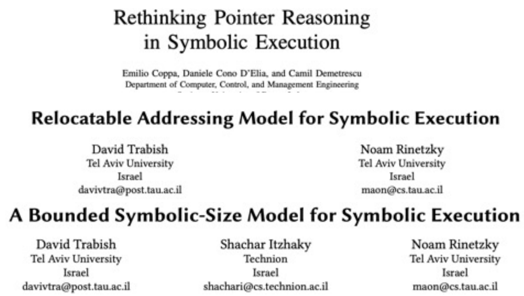
B. Practical read/write operation from/to symbolic addresses

C. Effectively tracking the uses of symbolic addresses

A New Memory Model: insights

- **Three fundamental designs are required**
  A. Symbolization of addresses and modeling them into path constraints
  B. Practical read/write operation from/to symbolic addresses
  C. Effectively tracking the uses of symbolic addresses

- **Existing approaches are difficult to satisfy all above requirements**
  - KLEE and Symsize (FSE’21): none of the requirements can be satisfied
  - RAM (ICSE’18): satisfies requirements #B and partially #A but not #C
  - Memsight (ASE’17): satisfies requirements #A and #B but not #C


Rethinking Pointer Reasoning in Symbolic Execution

Emilio Coppa, Daniele Cono D’Elia, and Camil Demetrescu
Department of Computer, Control, and Management Engineering

Relocatable Addressing Model for Symbolic Execution

David Trabish, Tel Aviv University, Israel, davivtra@post.tau.ac.il
Noam Rinetzky, Tel Aviv University, Israel, maon@cs.tau.ac.il

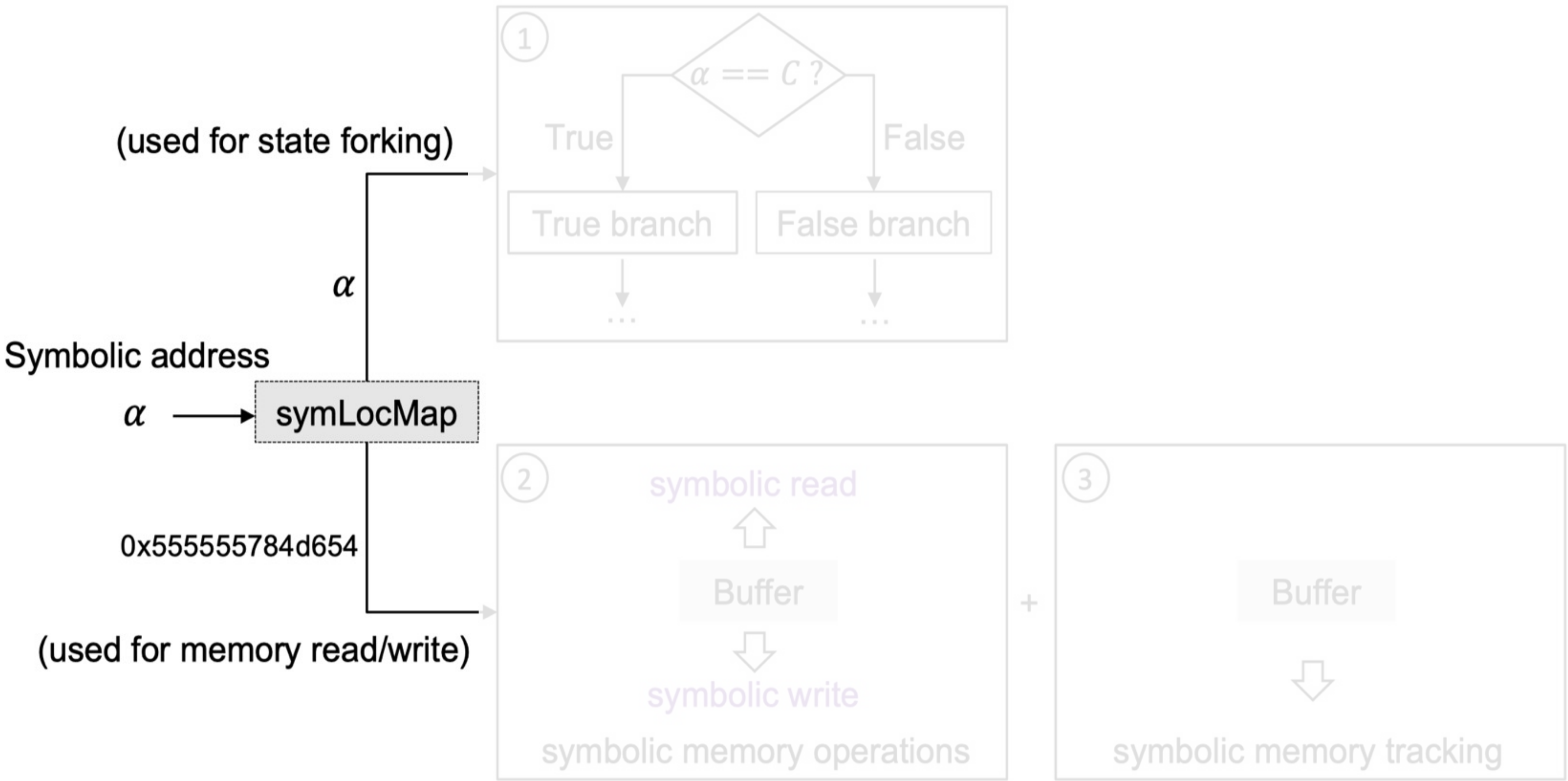
A Bounded Symbolic-Size Model for Symbolic Execution

David Trabish, Tel Aviv University, Israel, davivtra@post.tau.ac.il
Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il
Noam Rinetzky, Tel Aviv University, Israel, maon@cs.tau.ac.il

Solution: SymLoc (1/3)

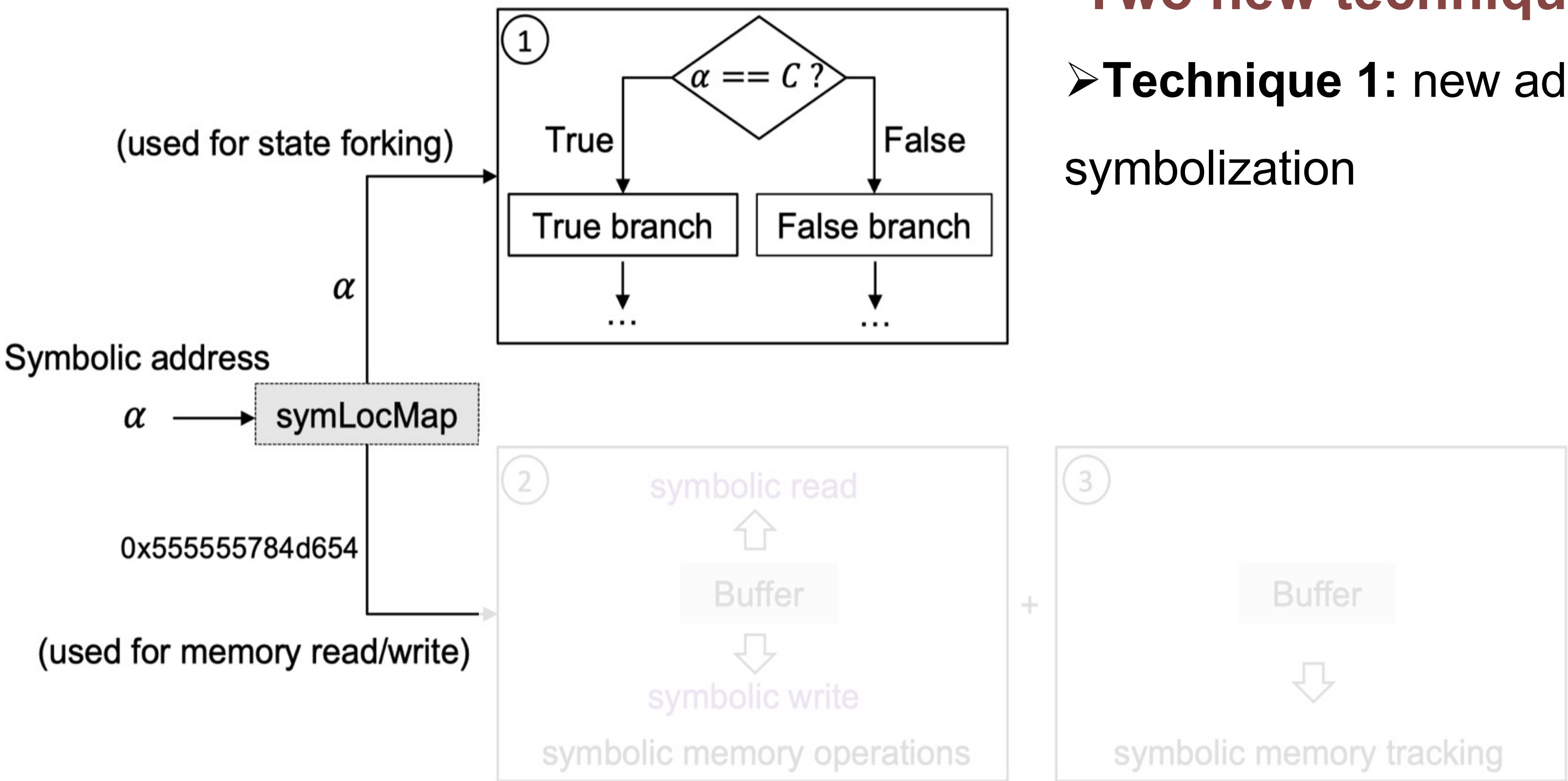
Solution: SymLoc (1/3)

High-level Idea



Solution: SymLoc (1/3)

High-level Idea



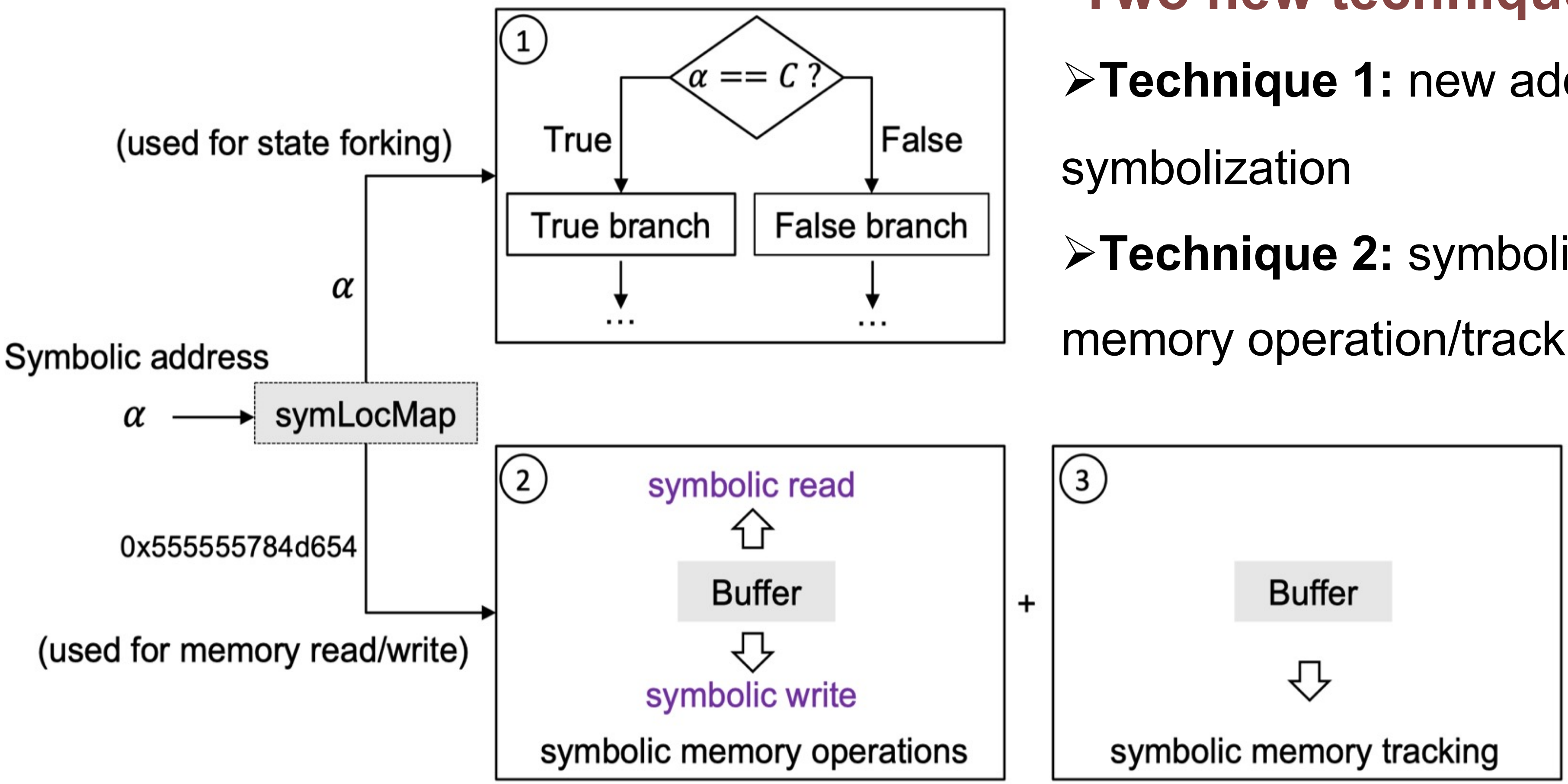
Two new techniques

- **Technique 1:** new address symbolization

Solution: SymLoc (1/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

High-level Idea

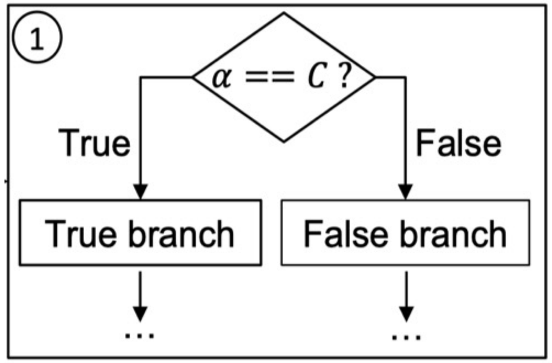


Two new techniques

- **Technique 1:** new address symbolization
- **Technique 2:** symbolic memory operation/tracking

Solution: SymLoc (2/3)

Solution: SymLoc (2/3)



Solution: SymLoc (2/3)

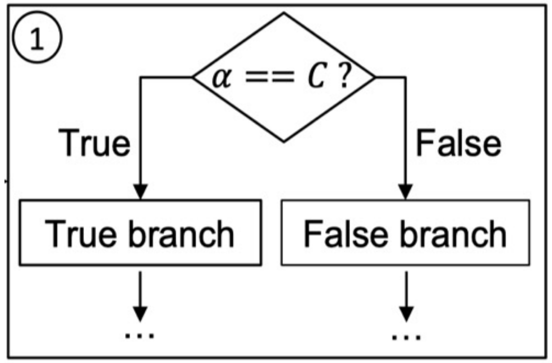
☐ New address symbolization

- **Symbolic addressing model**
  - Encoding the symbolic address into path constraints

Existing: $(addr, size, arry) \in N^{+} \times N^{+} \times A$

$\downarrow$

Ours: $(symAddr, size, arry) \in N^{+} \times N^{+} \times A$



Solution: SymLoc (2/3)

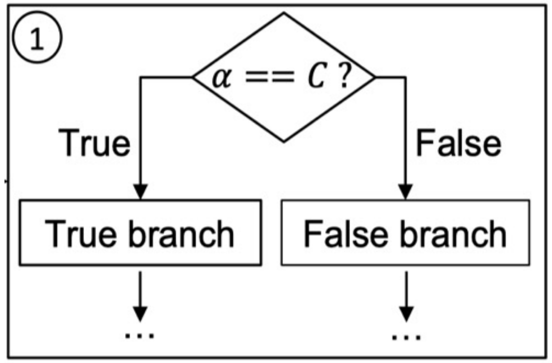
☐ New address symbolization

- **Symbolic addressing model**
  - Encoding the symbolic address into path constraints

Existing: $(addr, size, arry) \in N^+ \times N^+ \times A$

$\downarrow$

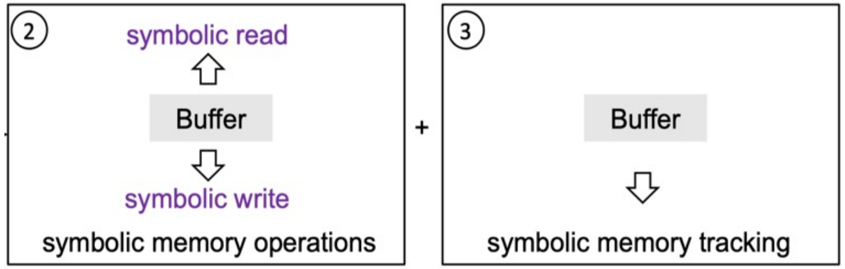
Ours: $(symAddr, size, arry) \in N^+ \times N^+ \times A$



- **Input**
  - A set of variables to return from malloc function
- **Output**
  - A symbolic-concrete memory map (symLocMap)
  - Will be used in the latter phase

Solution: SymLoc (3/3)

Solution: SymLoc (3/3)



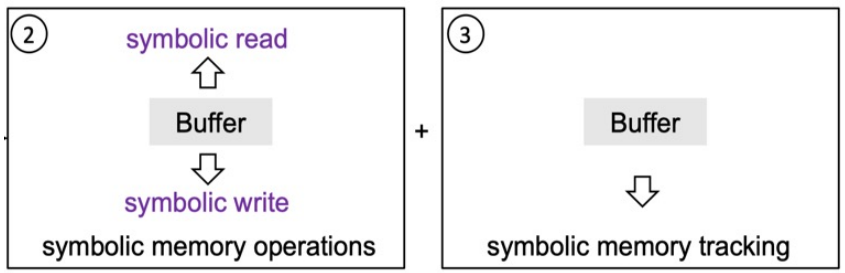
Solution: SymLoc (3/3)

Symbolic memory operation and tracking

Algorithm 1: Symbolic memory operations and tracking

Input: the map `symLocMap`, a symbolic expression `symExpr`, and a function `func` being executed
Output: a concrete or symbolic expression, or an error

```
conExpr ⟵ ∅ // initialize a concrete expression
FreeList ⟵ ∅ // initialize a list to store freed objects
Function SymAddrRes (symLocMap, symExpr, func):
  ...
```



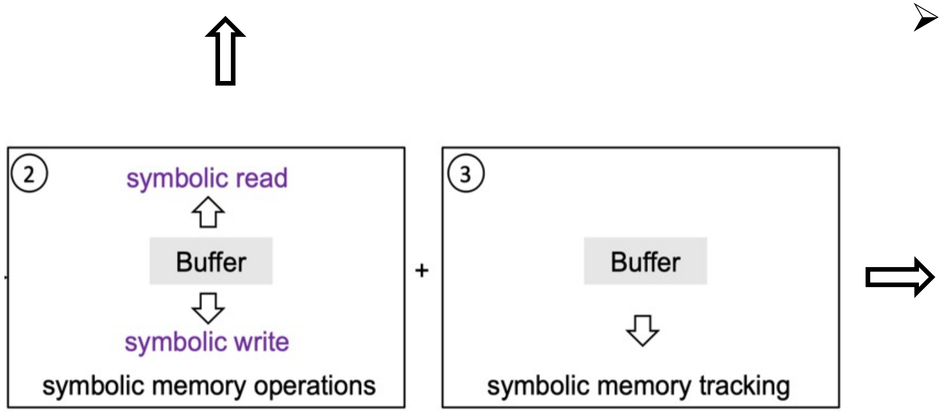
- **Input**
  - symLocMap, a symbolic expression, and a function
- **Output**
  - A concrete address or normal symbolic variable or a bug

Solution: SymLoc (3/3)

☐ Symbolic memory operation and tracking

Algorithm 1: Symbolic memory operations and tracking

```
Input: the map symLocMap, a symbolic expression symExpr,
       and a function func being executed
Output: a concrete or symbolic expression, or an error
conExpr ⟵ ∅ // initialize a concrete expression
FreeList ⟵ ∅ // initialize a list to store freed objects
Function SymAddrRes (symLocMap, symExpr, func):
  ...
```



- **Input**
  - symLocMap, a symbolic expression, and a function
- **Output**
  - A concrete address or normal symbolic variable or a bug
- **Tracking example**
  - A memory address is symbolized as “sym_a”
  - If the freed object is “**sym_a**” or “**sym_a + 100**”
    - Indicating UAF bugs

SymLoc: Evaluation Results & Takeaways

SymLoc: Evaluation Results & Takeaways

RQ1: How does SymLoc perform in detecting spatial memory errors?

– SymLoc could cover **15%** and **48%** more unique lines of code on average than the two baseline approaches.

– SymLoc could detect **169%** and **218%** more spatial memory errors than the two baseline approaches.

– **Two** new vulnerabilities have been detected

SymLoc: Evaluation Results & Takeaways

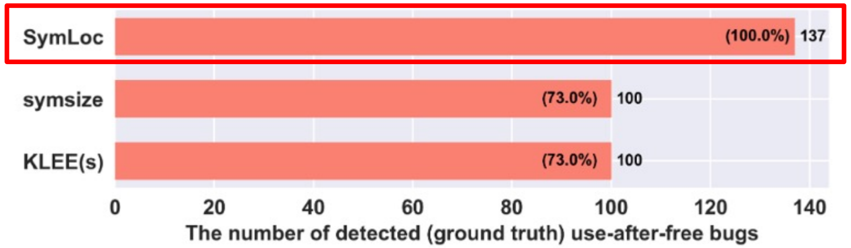
SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

RQ1: How does SymLoc perform in detecting spatial memory errors?

- SymLoc could cover **15%** and **48%** more unique lines of code on average than the two baseline approaches.
- SymLoc could detect **169%** and **218%** more spatial memory errors than the two baseline approaches.
- **Two** new vulnerabilities have been detected

RQ2: How does SymLoc perform in detecting temporal memory errors?

- SymLoc has an **overall better** temporal memory error detection capability for detecting UAF and DoF errors than static, dynamic, and symbolic execution-based approaches.



SymLoc: Evaluation Results & Takeaways

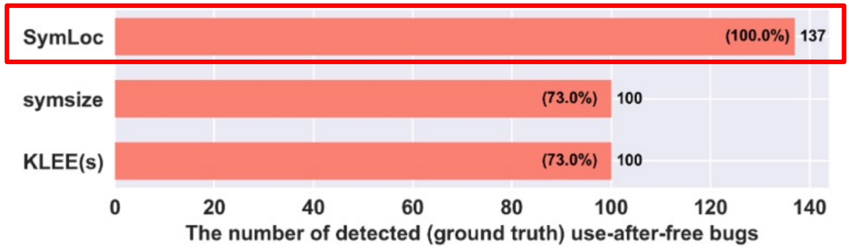
SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

RQ1: How does SymLoc perform in detecting spatial memory errors?

- SymLoc could cover **15%** and **48%** more unique lines of code on average than the two baseline approaches.
- SymLoc could detect **169%** and **218%** more spatial memory errors than the two baseline approaches.
- **Two** new vulnerabilities have been detected

RQ2: How does SymLoc perform in detecting temporal memory errors?

- SymLoc has an **overall better** temporal memory error detection capability for detecting UAF and DoF errors than static, dynamic, and symbolic execution-based approaches.



- **Takeaways**
  - **A more complete memory model could help detect tricky vulnerabilities**

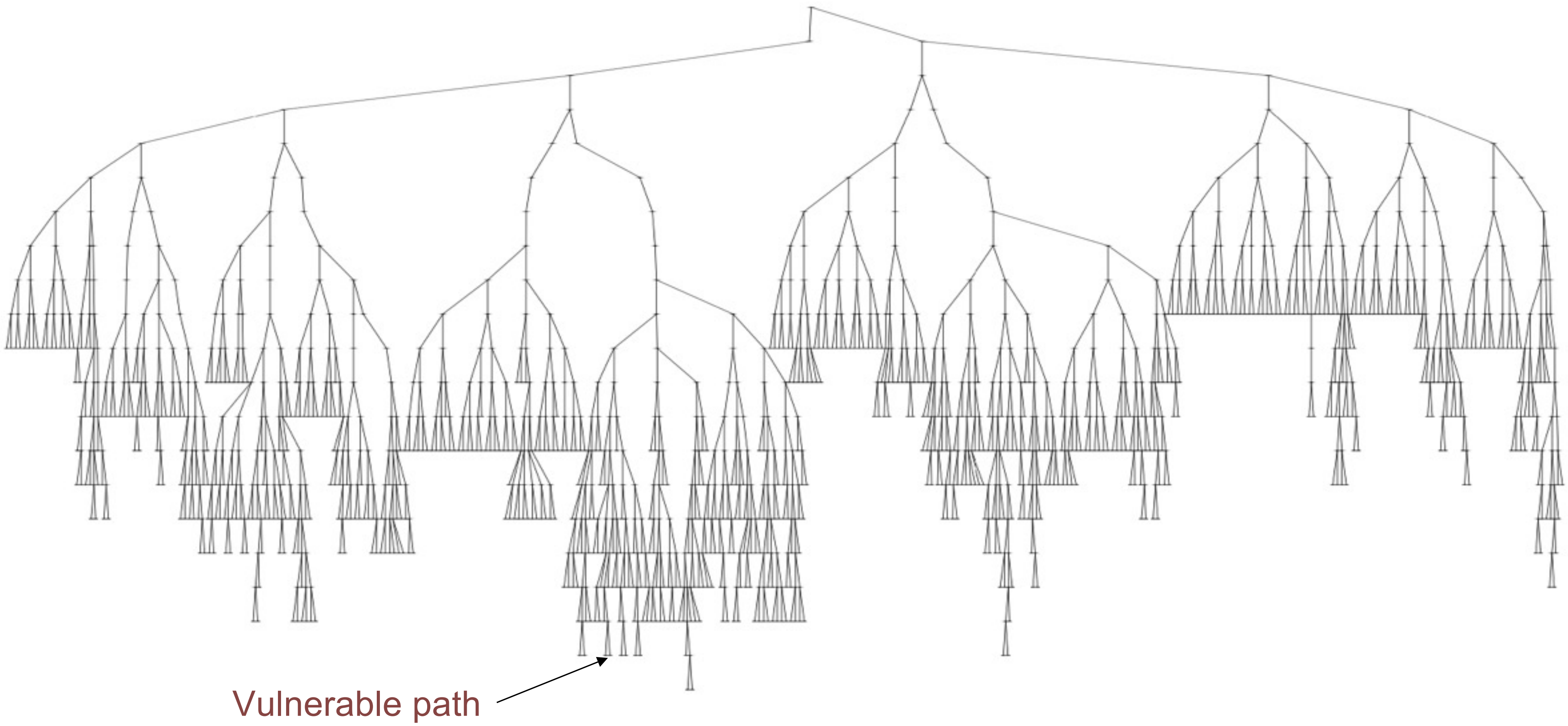
Outline

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Background**
  - Software vulnerabilities
  - Symbolic execution
- **Motivation**
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objects
- **Methodology**
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- **Future work**
  - Extension of current solutions
  - Combine Program Analysis with GenAI
- **Conclusion and Acknowledgment**

Two New Path Explorations (1/2)

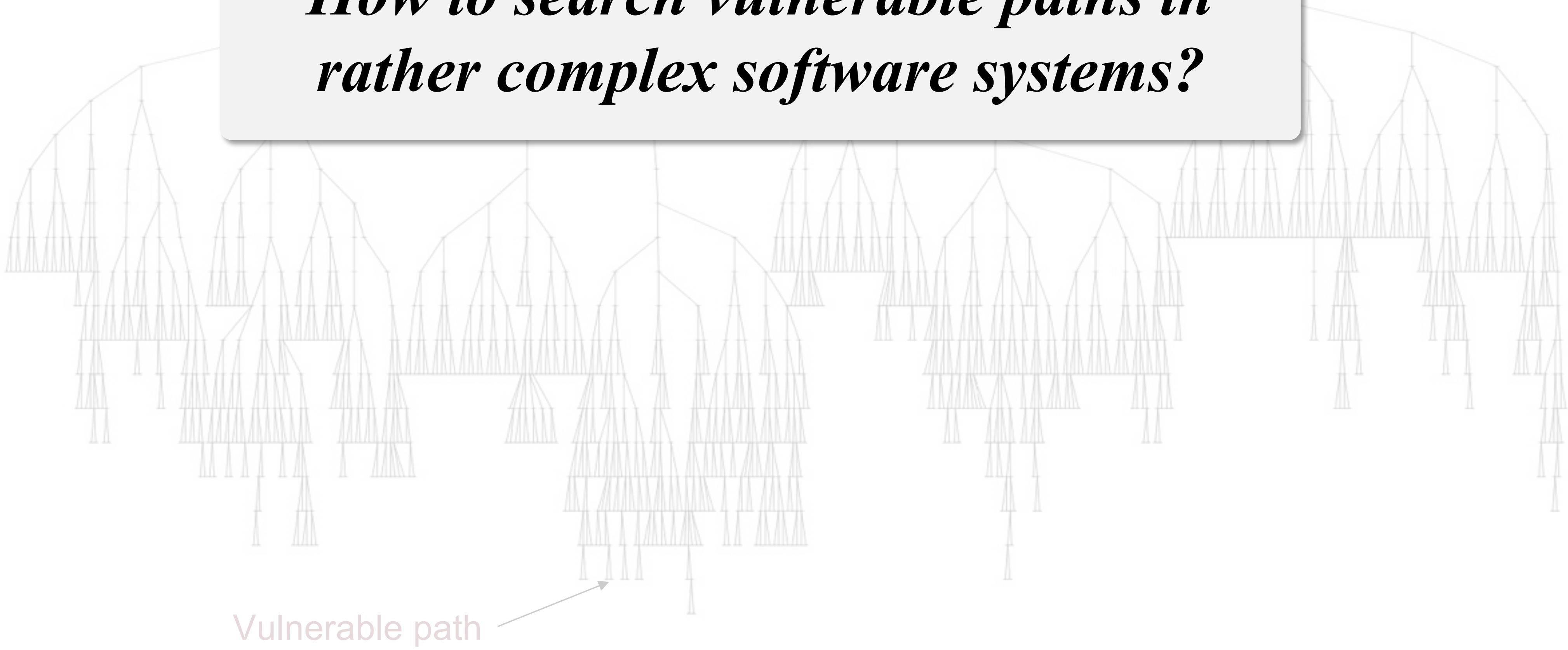
School of Computing and Information Systems



Two New Path Explorations (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

How to search vulnerable paths in rather complex software systems?

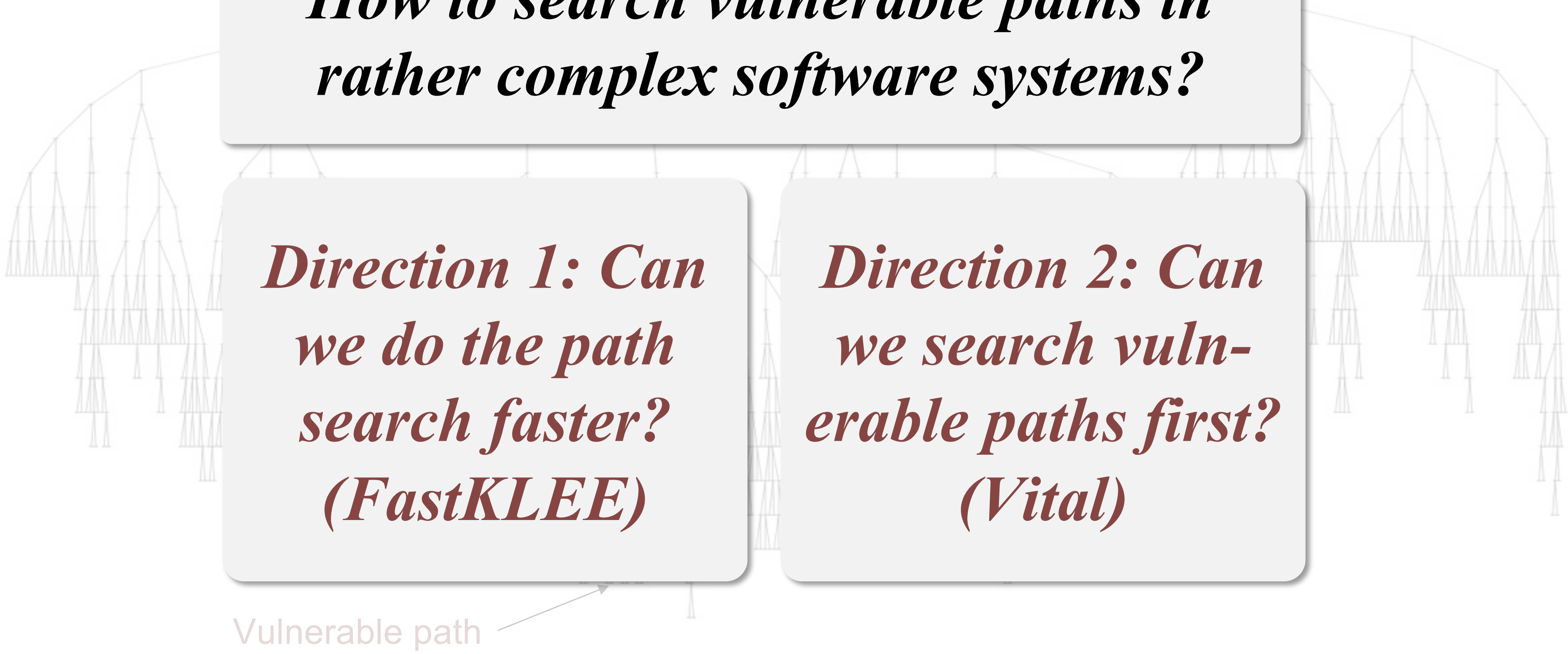


Two New Path Explorations (1/2)

How to search vulnerable paths in rather complex software systems?

Direction 1: Can we do the path search faster? (FastKLEE)

Direction 2: Can we search vulnerable paths first? (Vital)

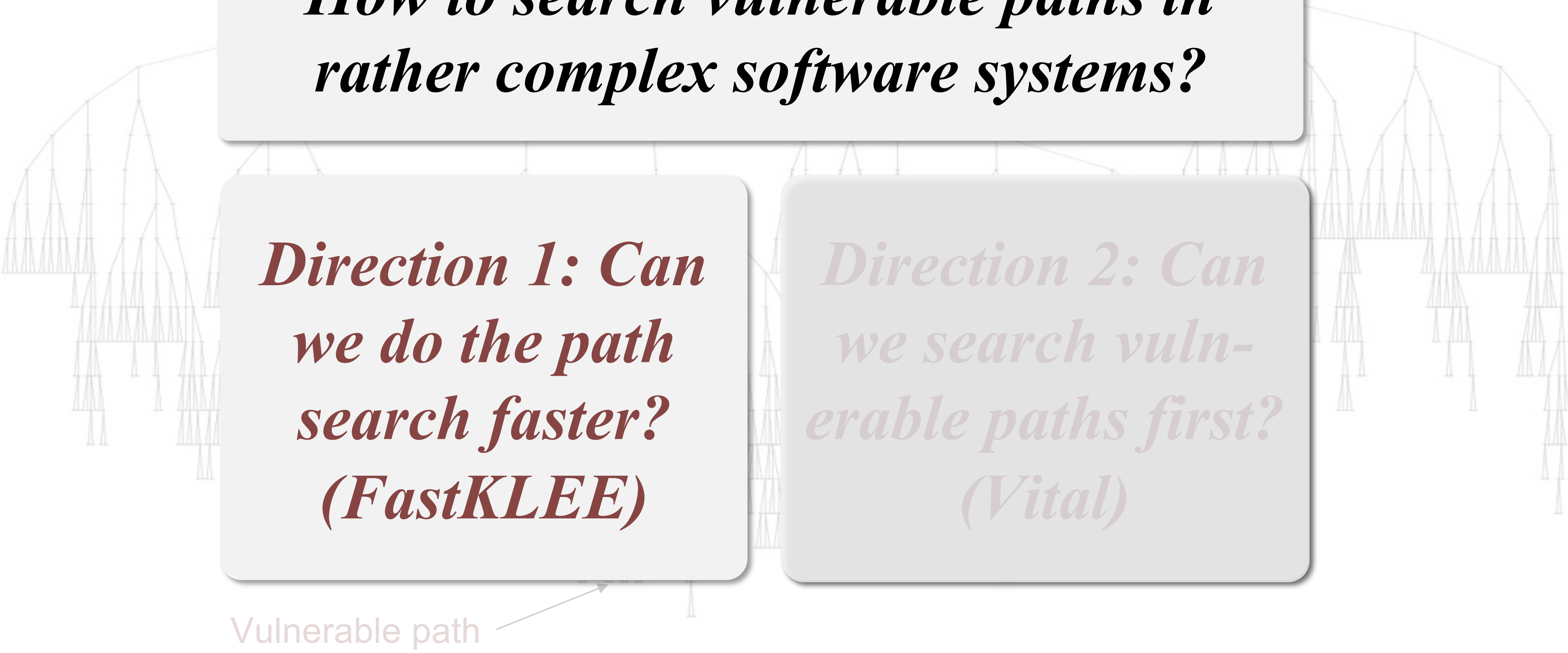


Two New Path Explorations (1/2)

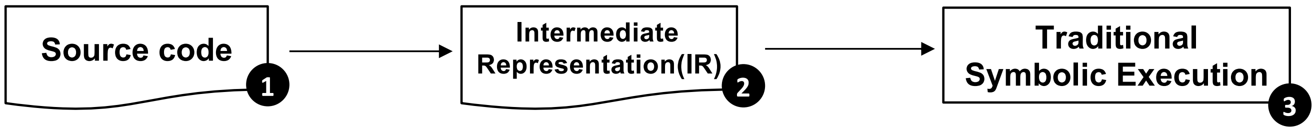
How to search vulnerable paths in rather complex software systems?

Direction 1: Can we do the path search faster? (FastKLEE)

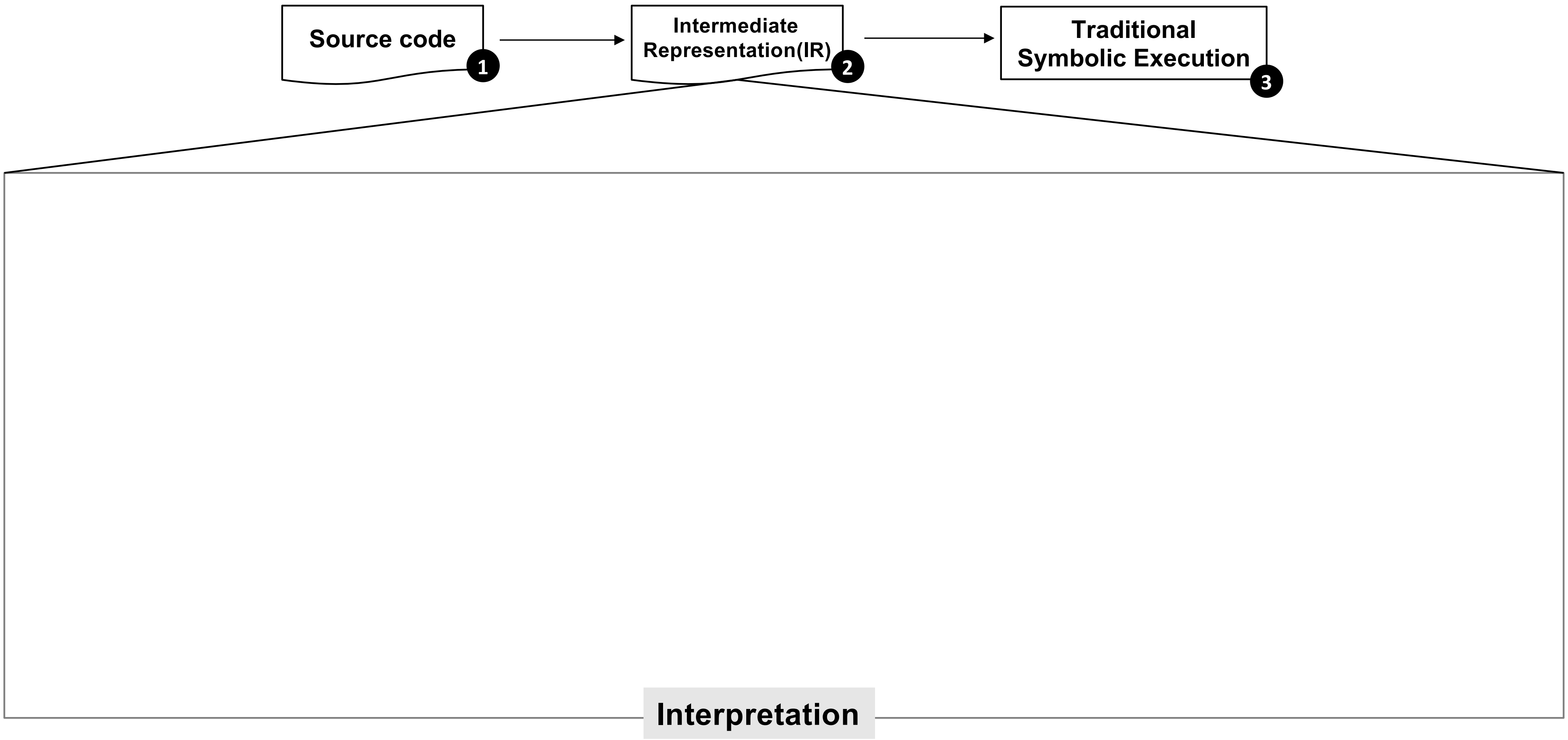
Direction 2: Can we search vulnerable paths first? (Vital)



Efficient Path Exploration: Rethinking



Efficient Path Exploration: Rethinking



Efficient Path Exploration: Rethinking

Source code ❶ → Intermediate Representation(IR) ❷ → Traditional Symbolic Execution ❸

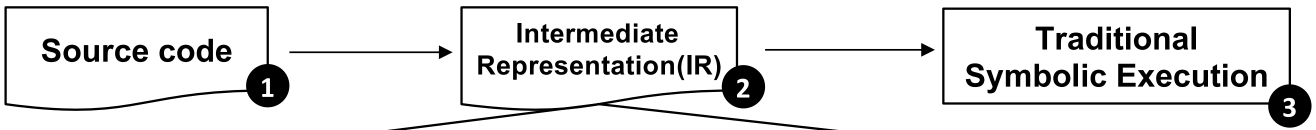
(1) Observation

– The number of interpreted instructions tends to be **huge** (several billion only in one hour run)

```
Elapsed: 01:00:04
KLEE: done: explored paths = 125017
KLEE: done: avg. constructs per query = 74
KLEE: done: total queries = 8859
KLEE: done: valid queries = 6226
KLEE: done: invalid queries = 2633
KLEE: done: query cex = 8859

KLEE: done: total instructions = 605113213
KLEE: done: completed paths = 125017
KLEE: done: generated tests = 65
```

Interpretation

Efficient Path Exploration: Rethinking



(1) Observation

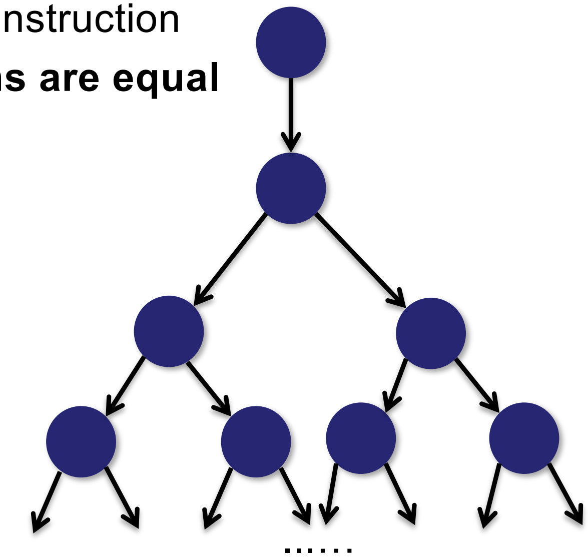
– The number of interpreted instructions tends to be **huge** (several billion only in one hour run)

```
Elapsed: 01:00:04
KLEE: done: explored paths = 125017
KLEE: done: avg. constructs per query = 74
KLEE: done: total queries = 8859
KLEE: done: valid queries = 6226
KLEE: done: invalid queries = 2633
KLEE: done: query cex = 8859

KLEE: done: total instructions = 605113213
KLEE: done: completed paths = 125017
KLEE: done: generated tests = 65
```

(2) Overheads in current symbolic execution

– The color depth represents the overheads of an interpreted instruction

– **All instructions are equal**



Interpretation

Efficient Path Exploration: Rethinking

Source code (1) → Intermediate Representation(IR) (2) → Traditional Symbolic Execution (3)

(1) Observation

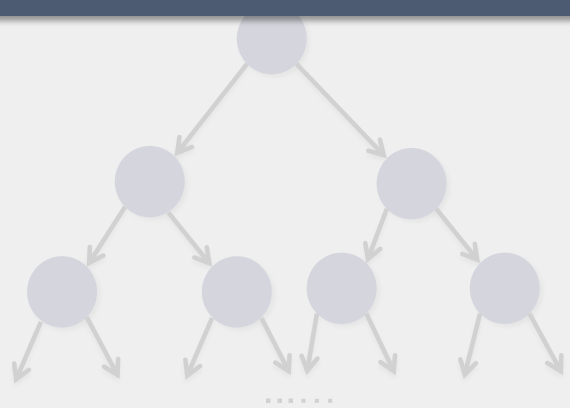
– The number of interpreted instructions tends to

(2) Overheads in current symbolic execution

– The color depth represents the overheads of

Can we reduce the overhead of interpreted instructions for faster symbolic execution?

```
KLEE: done: avg. constructs per query = 74
KLEE: done: total queries = 8859
KLEE: done: valid queries = 6226
KLEE: done: invalid queries = 2633
KLEE: done: query cex = 8859

KLEE: done: total instructions = 605113213
KLEE: done: completed paths = 125017
KLEE: done: generated tests = 65
```



Interpretation

Solution: FastKLEE (1/2)

- **Key insights**

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: FastKLEE (1/2)

- **Key insights**
  - Only a small portion of memory-related instructions need bound checking

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: FastKLEE (1/2)

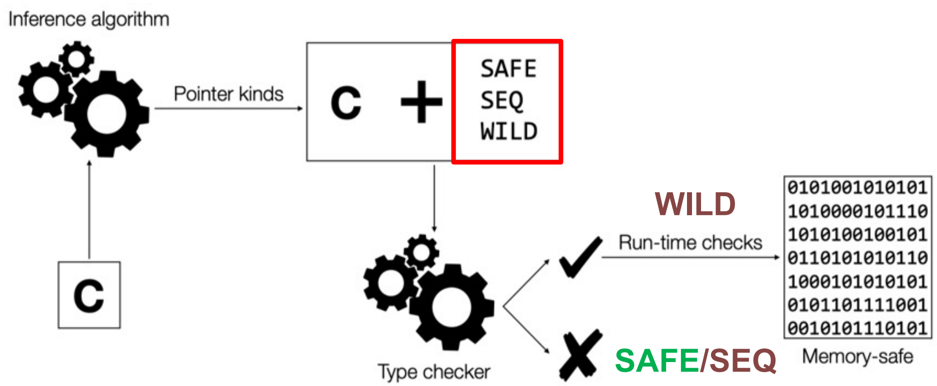
SMU Singapore Management University | School of Computing and Information Systems

- **Key insights**
  - Only a small portion of memory-related instructions need bound checking
  - Reduce the interpreting overhead of the most frequently interpreted ones (i.e., load/store instructions)

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: FastKLEE (1/2)

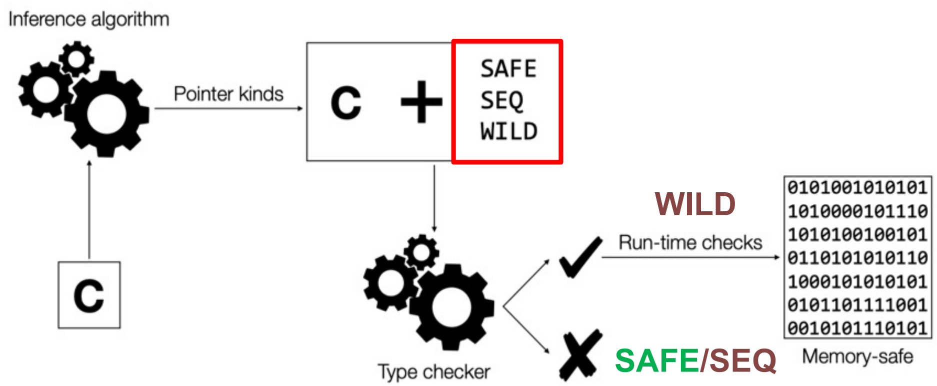
- **Key insights**
  - Only a small portion of memory-related instructions need bound checking
  - Reduce the interpreting overhead of the most frequently interpreted ones (i.e., load/store instructions)
  - Inspired by ***Type Inference*** system [1]



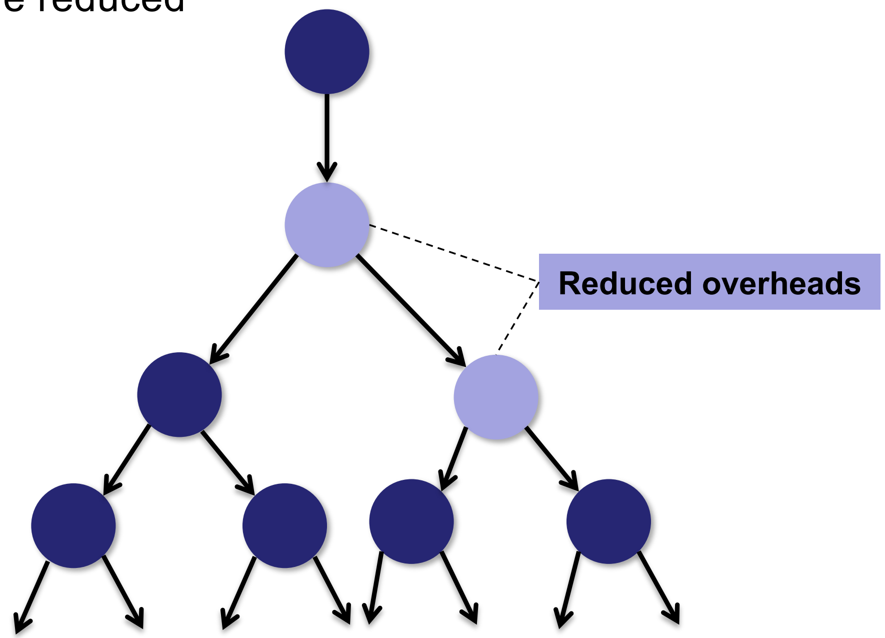
[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: FastKLEE (1/2)

- **Key insights**
  - Only a small portion of memory-related instructions need bound checking
  - Reduce the interpreting overhead of the most frequently interpreted ones (i.e., load/store instructions)
  - Inspired by ***Type Inference*** system [1]



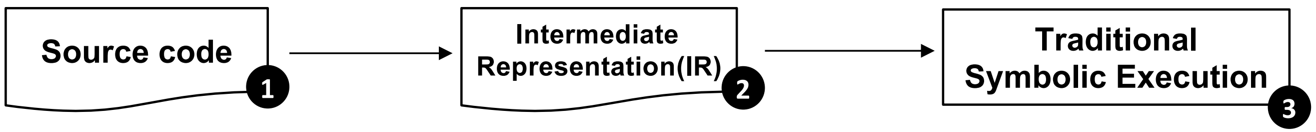
- **Advantage: overheads in FastKLEE**
  - Interpretation overheads for some instructions are reduced



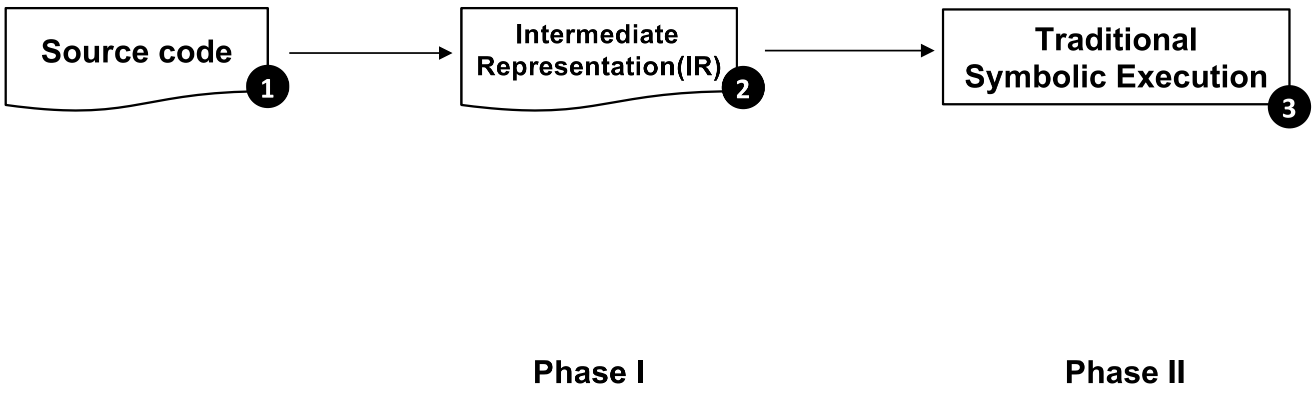
[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: FastKLEE (2/2)

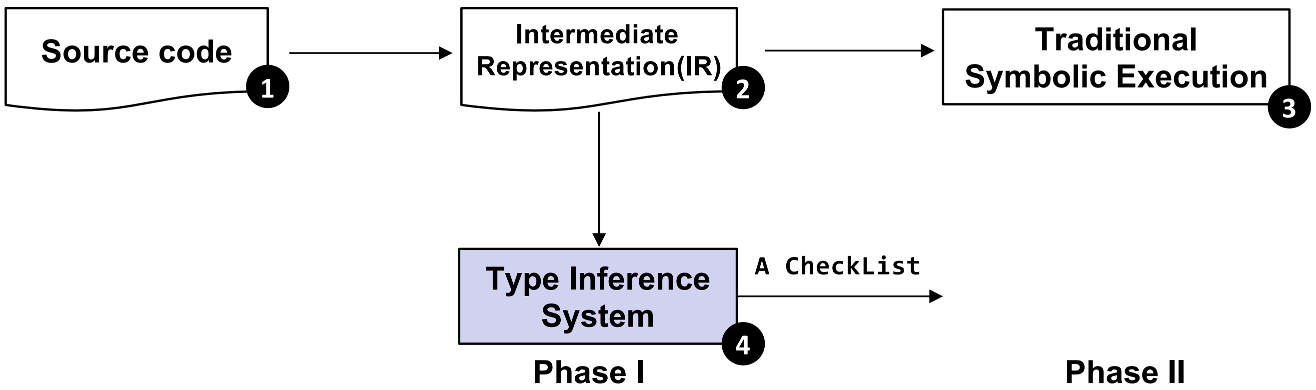
Solution: FastKLEE (2/2)



Solution: FastKLEE (2/2)

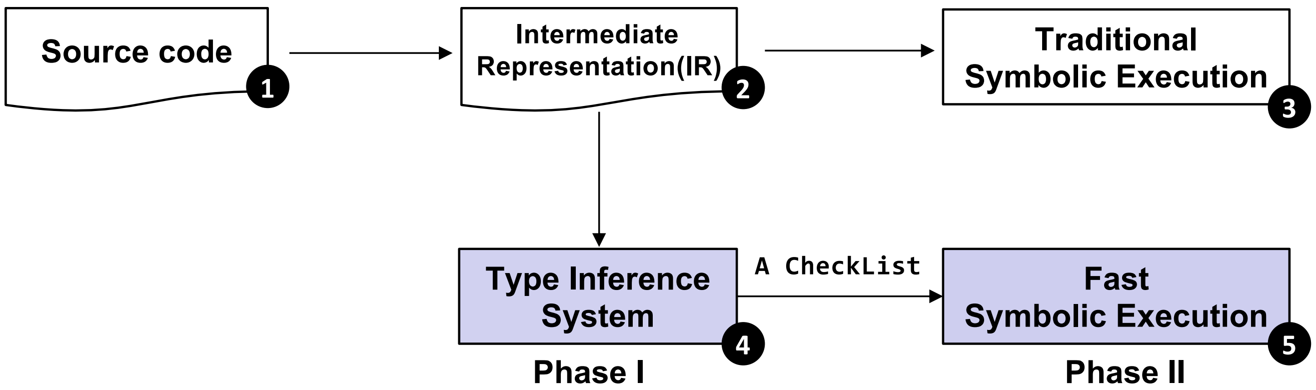


Solution: FastKLEE (2/2)



4 • **Phase I**: Introduce a **Type Inference System** to classify memory-related instruction types
  – ***Unsafe*** memory instructions will be stored in **CheckList**

Solution: FastKLEE (2/2)



4 • **Phase I**: Introduce a **Type Inference System** to classify memory-related instruction types
  – ***Unsafe*** memory instructions will be stored in **CheckList**

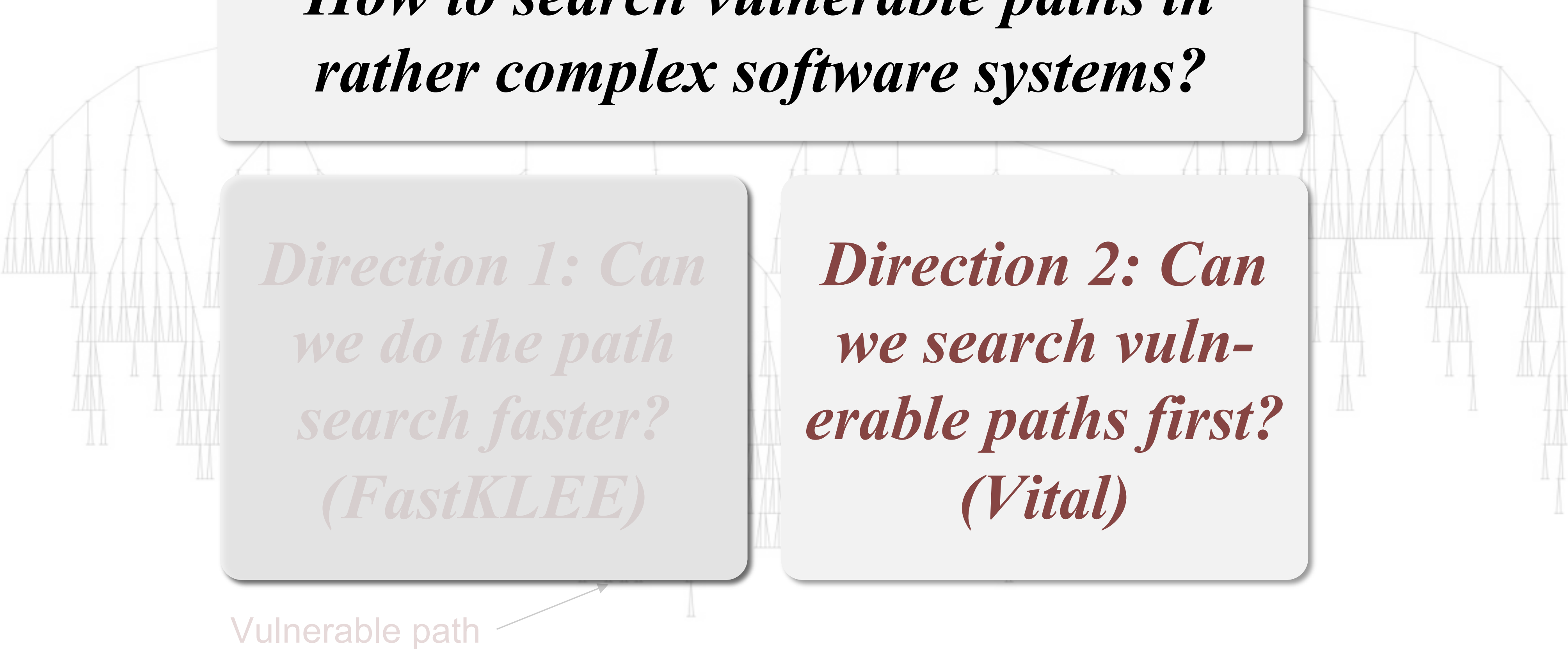
5 • **Phase II**: Conduct **Customized Memory Operation** in Fast symbolic execution
  – Only perform checking for ***Unsafe*** memory instructions during interpretation

Two New Path Explorations (2/2)

How to search vulnerable paths in rather complex software systems?

Direction 1: Can we do the path search faster? (FastKLEE)

Direction 2: Can we search vuln-erable paths first? (Vital)



Insight 1: approximate a vulnerable path

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

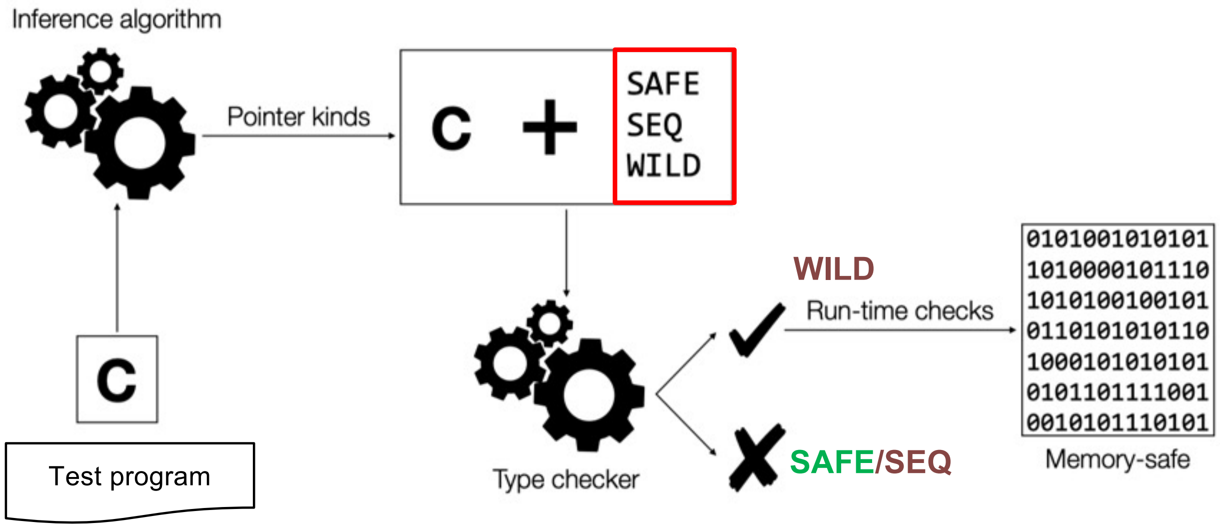
Insight 1: approximate a vulnerable path

- **Revisit type inference (Ccurd [1])**

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Insight 1: approximate a vulnerable path

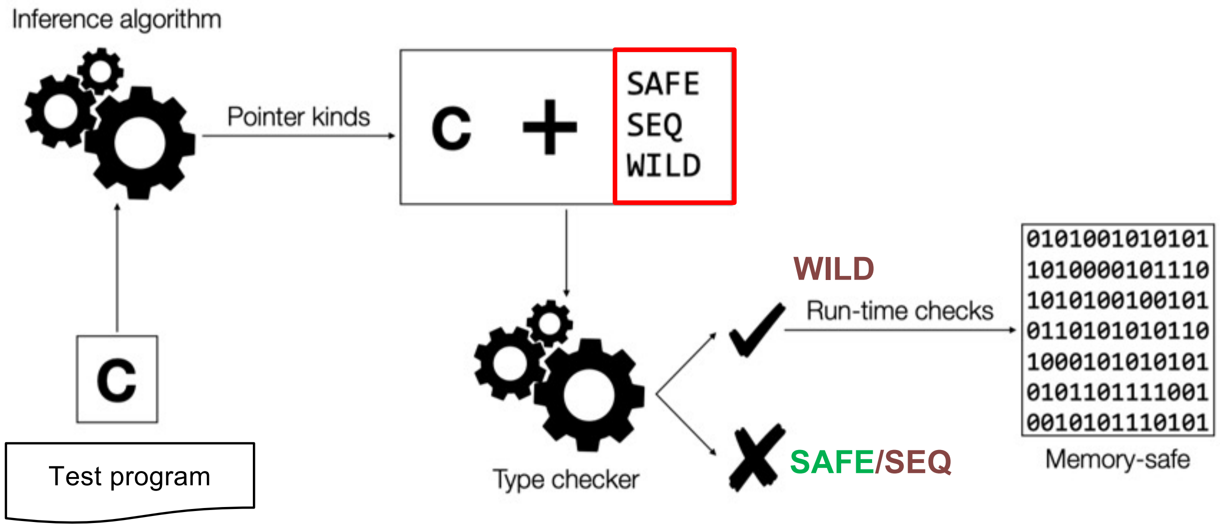
- **Revisit type inference (Ccurd [1])**



[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Insight 1: approximate a vulnerable path

- **Revisit type inference (Ccurd [1])**



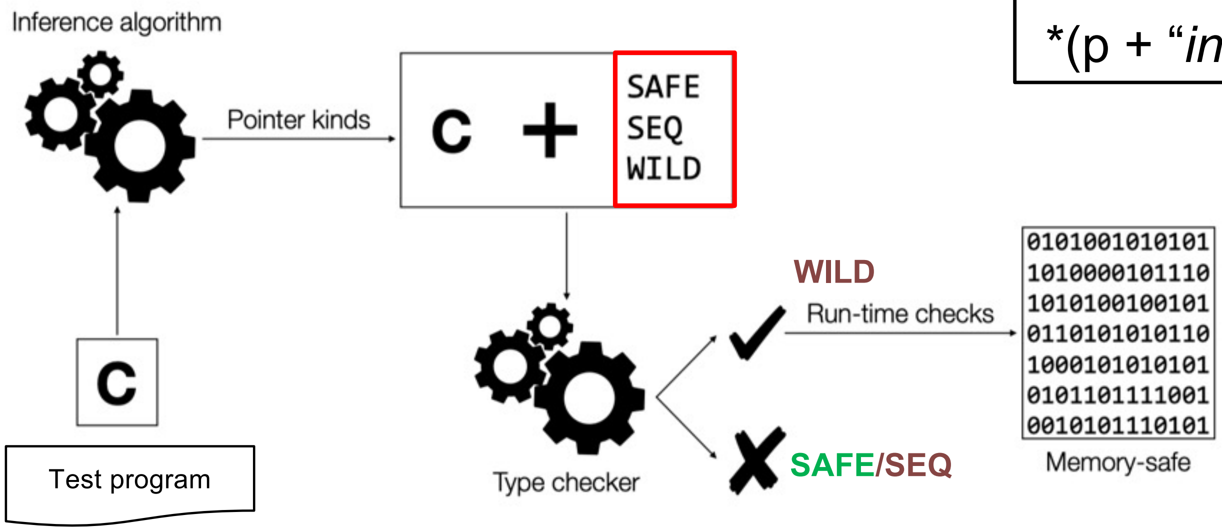
Observation: Memory errors can only happen in ***unsafe*** pointer operations (**SEQ/WILD**)!

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Insight 1: approximate a vulnerable path

- **Revisit type inference (Ccurd [1])**

```
int * p = malloc (100);
*(p + "input") = 1;  // unsafe pointer operation
```



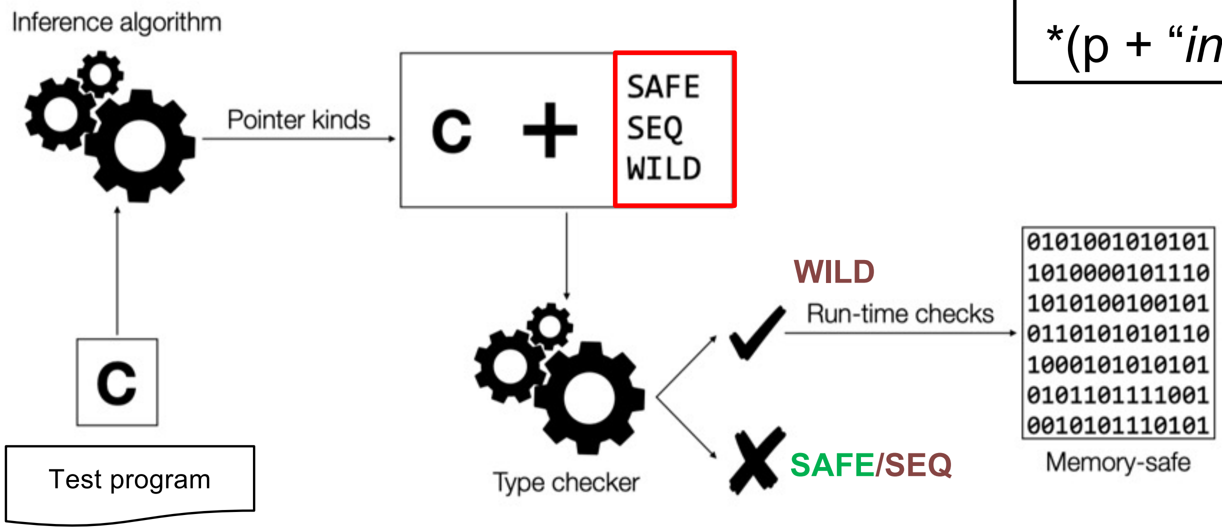
Observation: Memory errors can only happen in ***unsafe*** pointer operations (**SEQ/WILD**)!

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Insight 1: approximate a vulnerable path

- **Revisit type inference (Ccurd [1])**

```
int * p = malloc (100);
*(p + "input") = 1;  // unsafe pointer operation
```



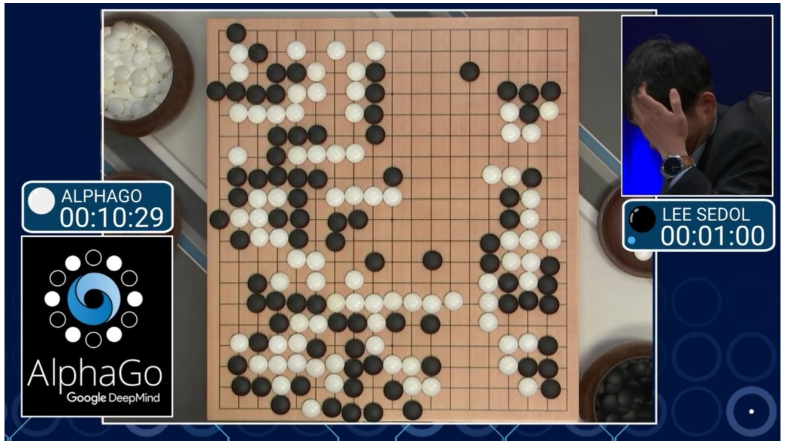
Observation: Memory errors can only happen in ***unsafe*** pointer operations (**SEQ/WILD**)!

We can exploit such information from type inference to guide the path search!

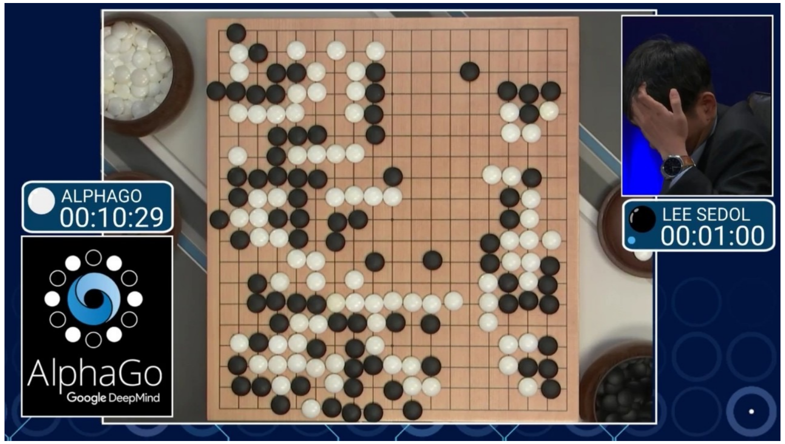
[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

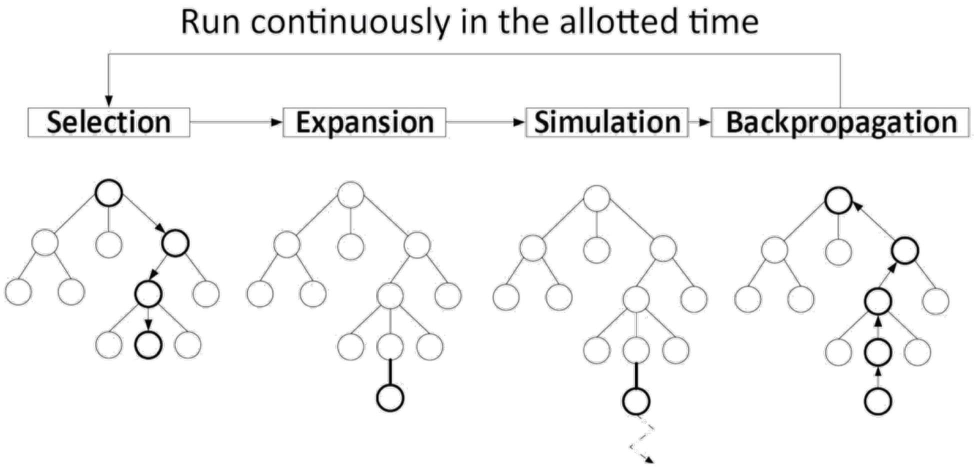
Insight 2: search the path smartly

Insight 2: search the path smartly



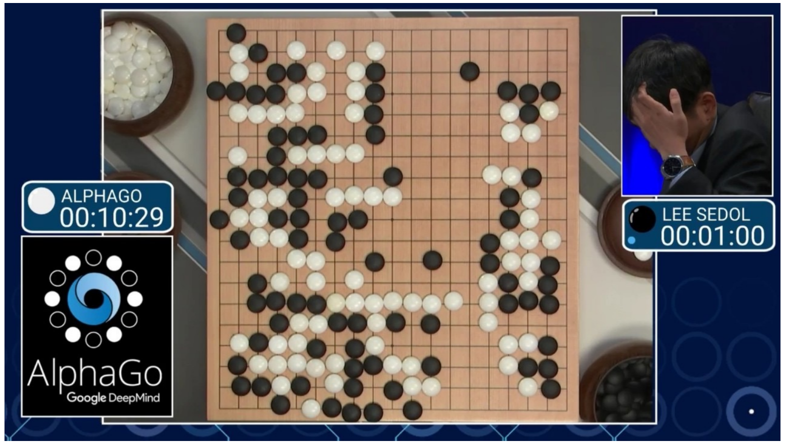
Insight 2: search the path smartly

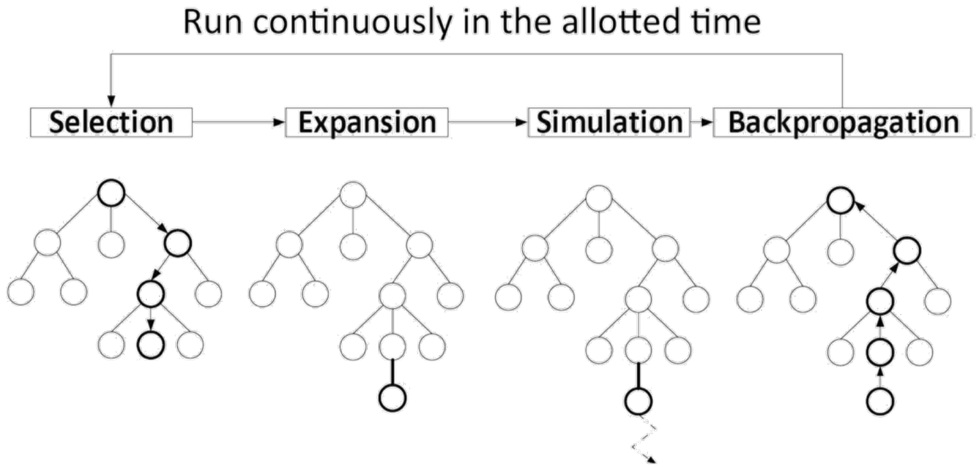




Monte Carlo Tree Search (MCTS) in AlphaGo

Insight 2: search the path smartly



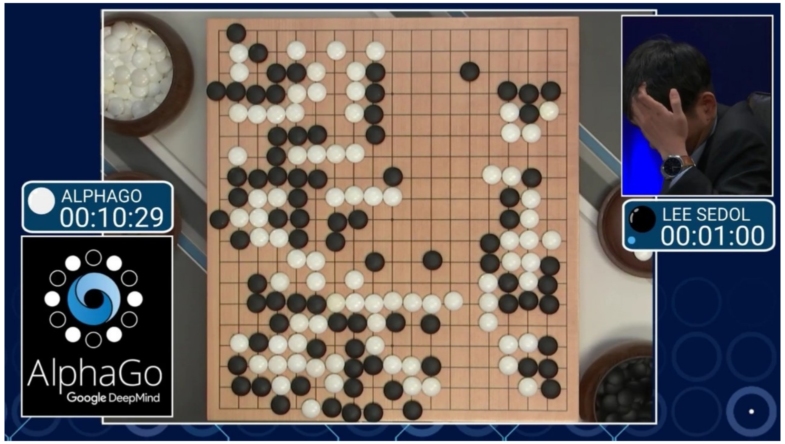


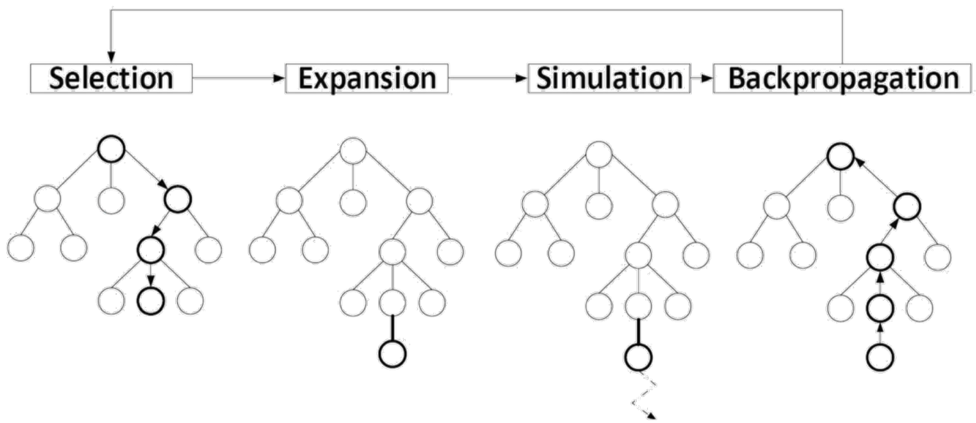
Monte Carlo Tree Search (MCTS) in AlphaGo

- **Analogy: Game tree (win) VS Execution tree (a vulnerable path)**

We can use MCTS to guide the path search towards the vulnerable paths!

Insight 2: search the path smartly





Monte Carlo Tree Search (MCTS) in AlphaGo

- **Analogy: Game tree (win) VS Execution tree (a vulnerable path)**

We can use MCTS to guide the path search towards the vulnerable paths!

Key question 1: *How to select/expand tree nodes?*

Key question 2: *How to evaluate rewards?*

Solution: Vital (vulnerability-oriented MCTS)

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: Vital (vulnerability-oriented MCTS)

How to select/expand tree nodes?

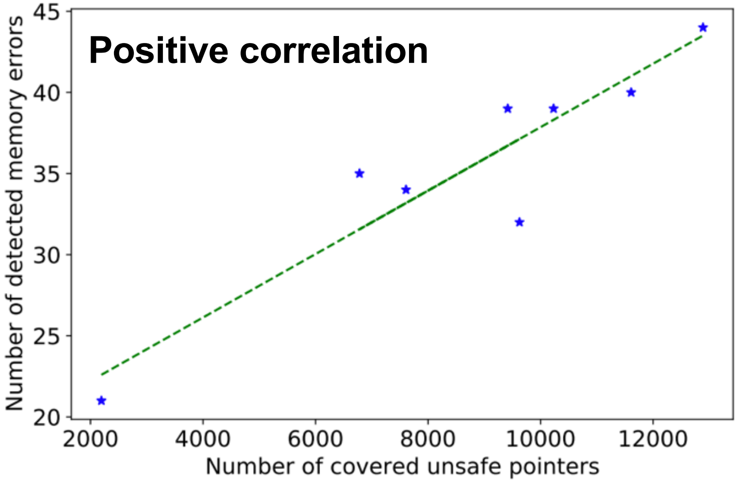
- **Use the number of unsafe pointers**
- **Fact:** vulnerabilities always happen on *type-unsafe* pointers

[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: Vital (vulnerability-oriented MCTS)

How to select/expand tree nodes?

- **Use the number of unsafe pointers**
- **Fact:** vulnerabilities always happen on *type-unsafe* pointers



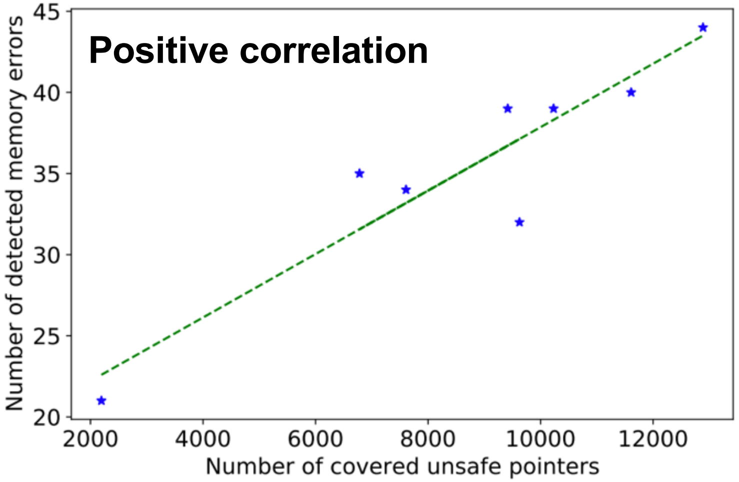
[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: Vital (vulnerability-oriented MCTS)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

How to select/expand tree nodes?

- **Use the number of unsafe pointers**
- **Fact:** vulnerabilities always happen on *type-unsafe* pointers



How to evaluate rewards?

- **Use state simulation (+backpropagation)**
- **Reward:** number of unsafe pointers

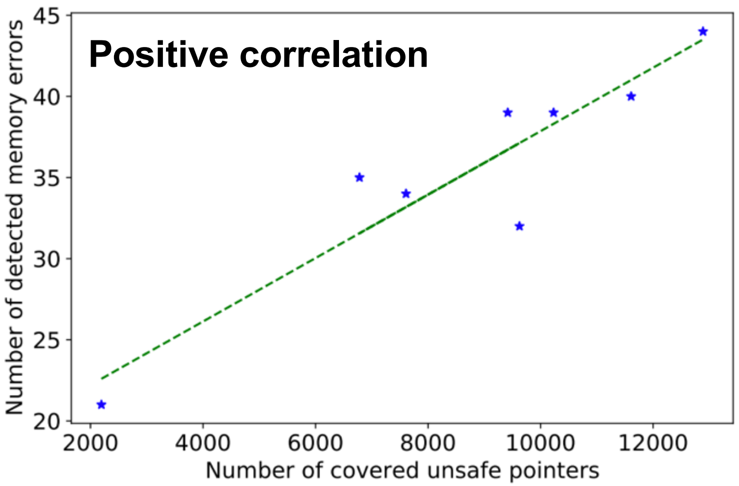
[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Solution: Vital (vulnerability-oriented MCTS)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

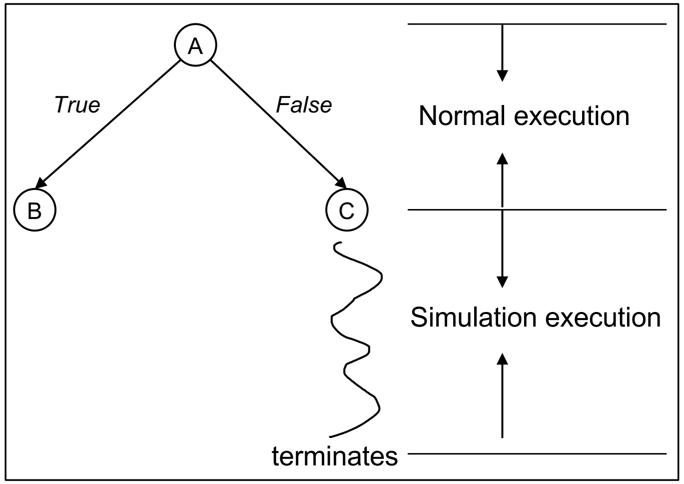
How to select/expand tree nodes?

- **Use the number of unsafe pointers**
- **Fact:** vulnerabilities always happen on *type-unsafe* pointers



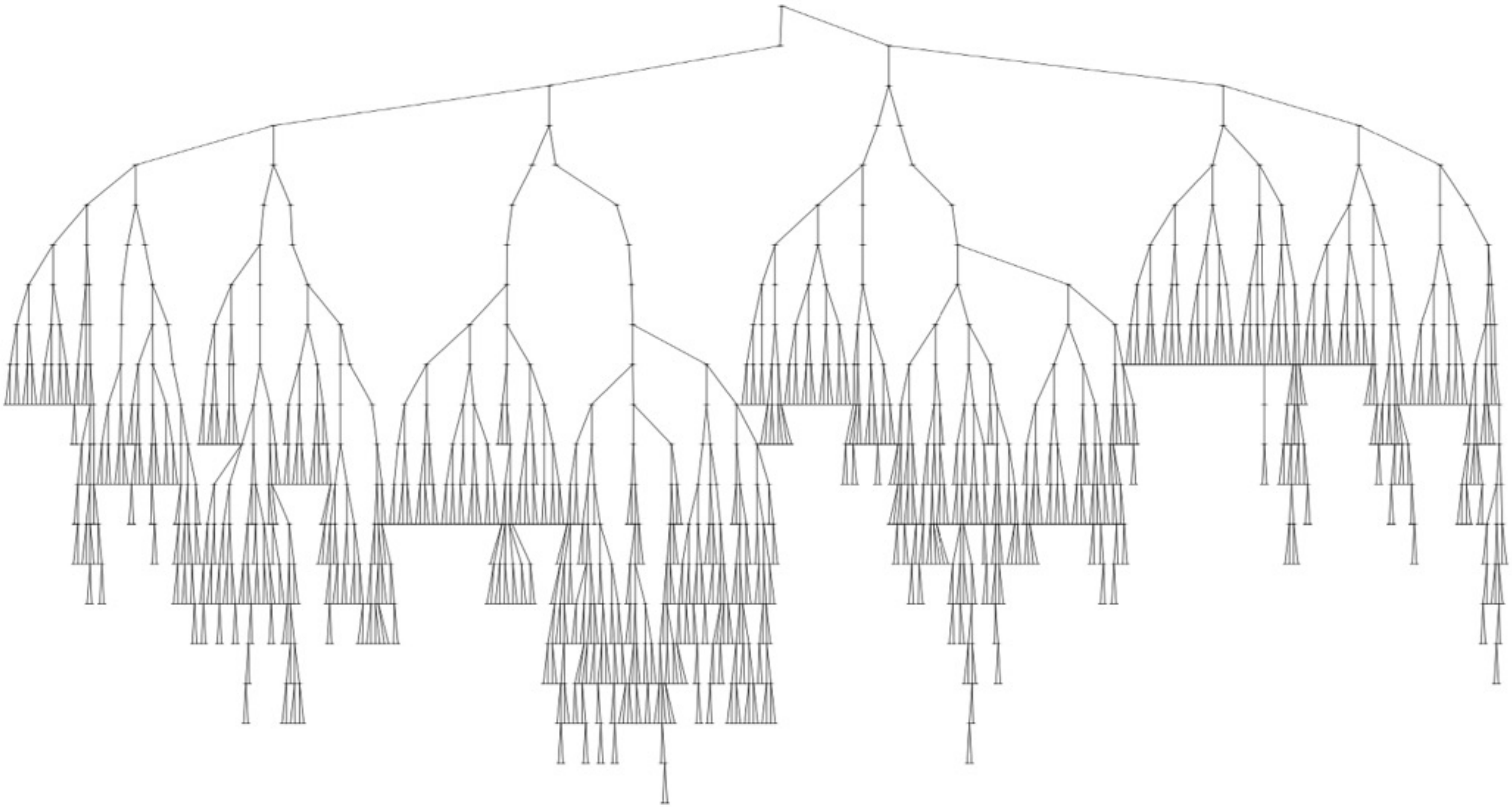
How to evaluate rewards?

- **Use state simulation (+backpropagation)**
- **Reward:** number of unsafe pointers



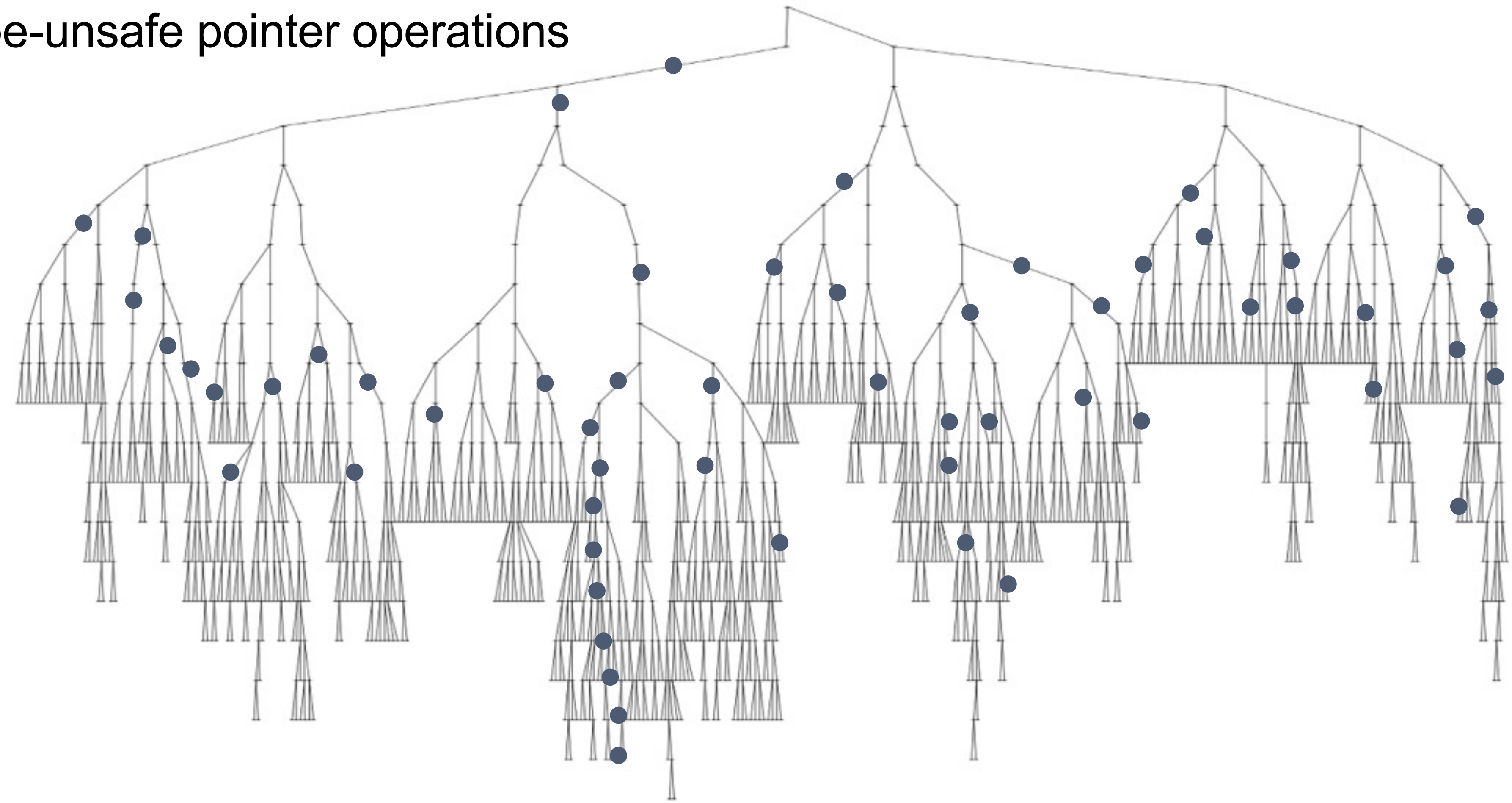
[1] Ccured: George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. **CCured: type-safe retrofitting of legacy software**. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526.

Effective Path Exploration: outcome



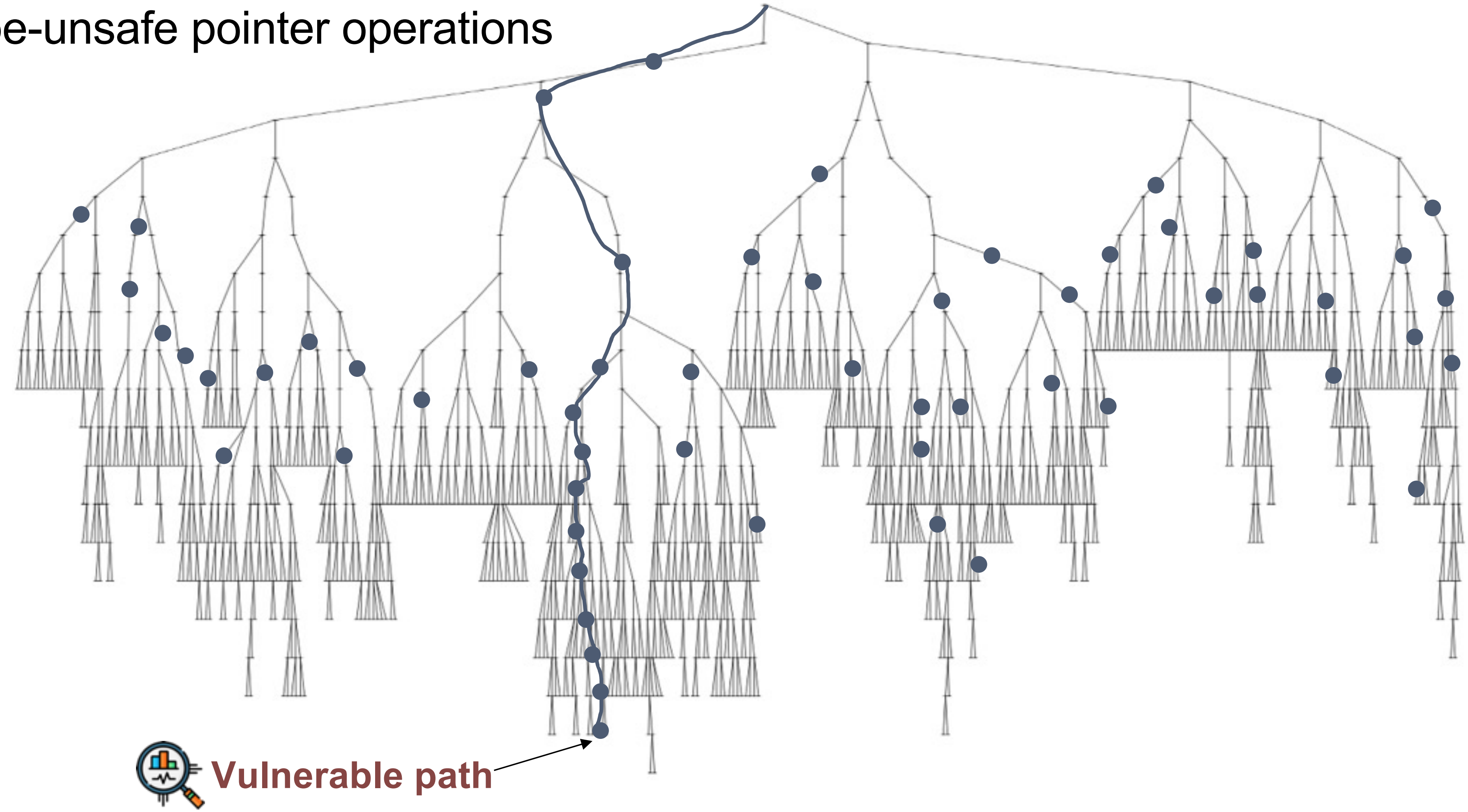
Effective Path Exploration: outcome

- Type-unsafe pointer operations



Effective Path Exploration: outcome

- Type-unsafe pointer operations



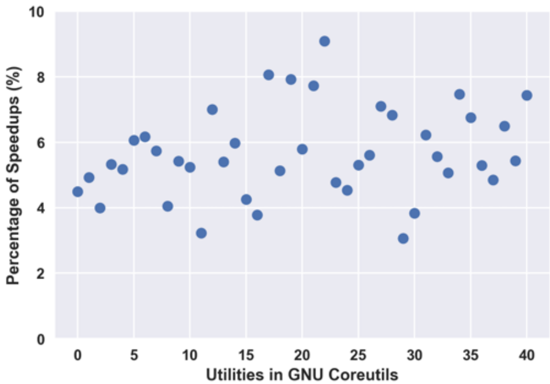
Evaluation Results & Takeaways

Evaluation Results & Takeaways

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Results for Direction 1: Can we do the path search faster?

– **FastKLEE** can reduce by up to **9.1%** time compared with the state-of-the-art approach (i.e., KLEE)

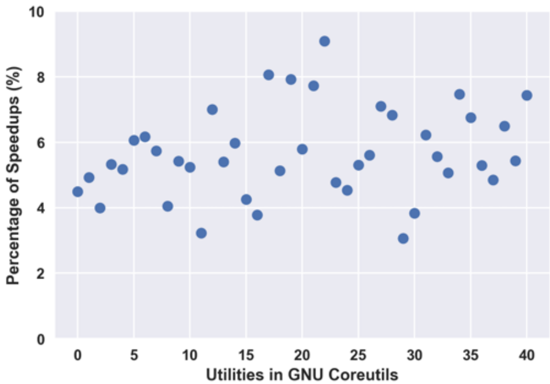


Evaluation Results & Takeaways

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Results for Direction 1: Can we do the path search faster?

- **FastKLEE** can reduce by up to **9.1%** time compared with the state-of-the-art approach (i.e., KLEE)



Results for Direction 2: Can we search vulnerable paths first?

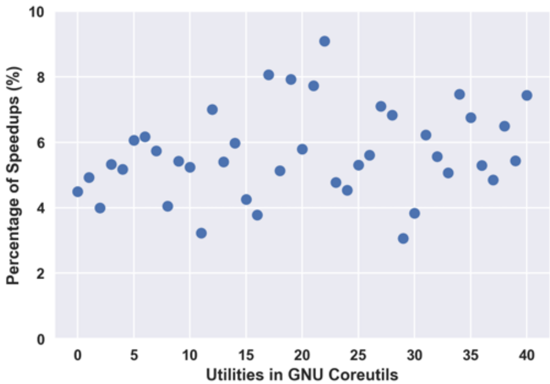
- **Vital** outperforms existing search strategies by covering up to **90.03%** unsafe pointers and detecting up to **57.14%** more unique memory errors.
- **Vital** outperforms existing solutions by achieving a speedup of up to **30x** execution time and a reduction of up to **20x** memory consumption.
- Detected an unknown vulnerability (a new CVE)

Evaluation Results & Takeaways

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Results for Direction 1: Can we do the path search faster?

- **FastKLEE** can reduce by up to **9.1%** time compared with the state-of-the-art approach (i.e., KLEE)



Results for Direction 2: Can we search vulnerable paths first?

- **Vital** outperforms existing search strategies by covering up to **90.03%** unsafe pointers and detecting up to **57.14%** more unique memory errors.
- **Vital** outperforms existing solutions by achieving a speedup of up to **30x** execution time and a reduction of up to **20x** memory consumption.
- Detected an unknown vulnerability (a new CVE)

- **Takeaways**
  - **Combing type inference with symbolic execution could help do a better path exploration**

Outline

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- Background
  - Software vulnerabilities
  - Symbolic execution
- Motivation
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objects
- **Methodology**
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- Future work
  - Extension of current solutions
  - Combine Program Analysis with GenAI
- Conclusion and Acknowledgment

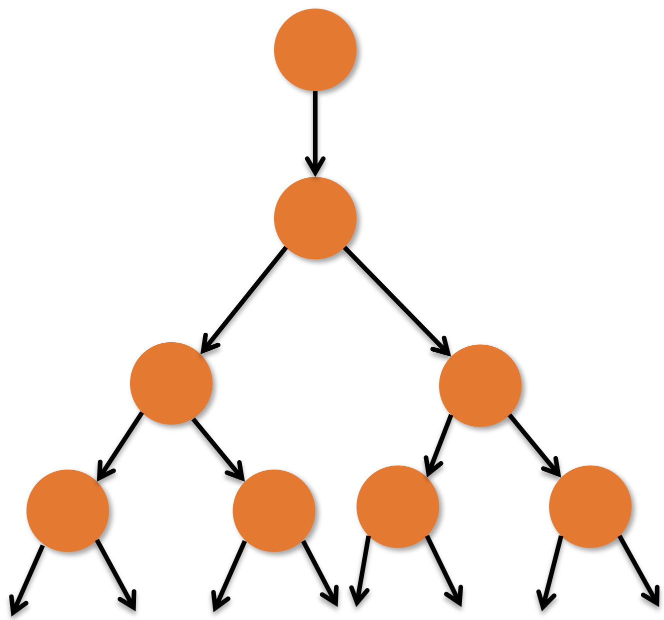
Preliminary: online and offline

- ***Online*** and ***offline*** symbolic execution

Preliminary: online and offline

- ***Online*** and ***offline*** symbolic execution

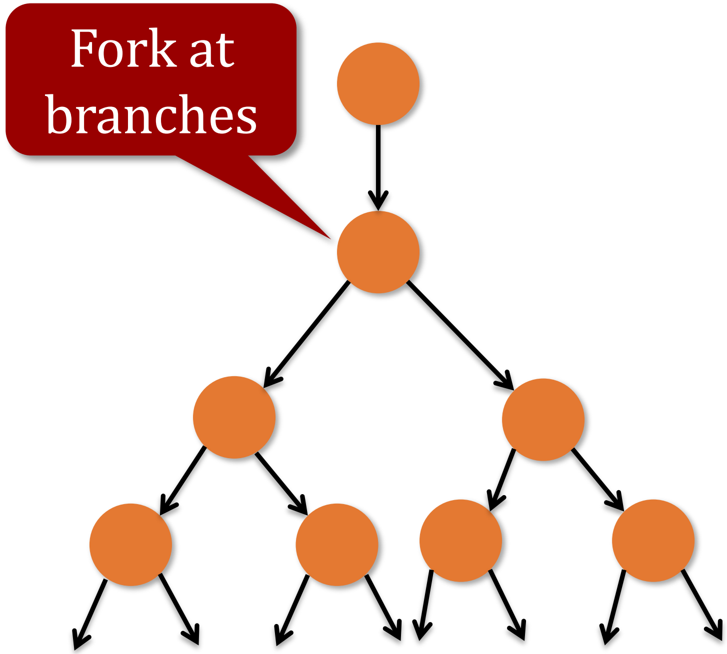
Online (without initial seeds)



Preliminary: online and offline

- ***Online*** and ***offline*** symbolic execution

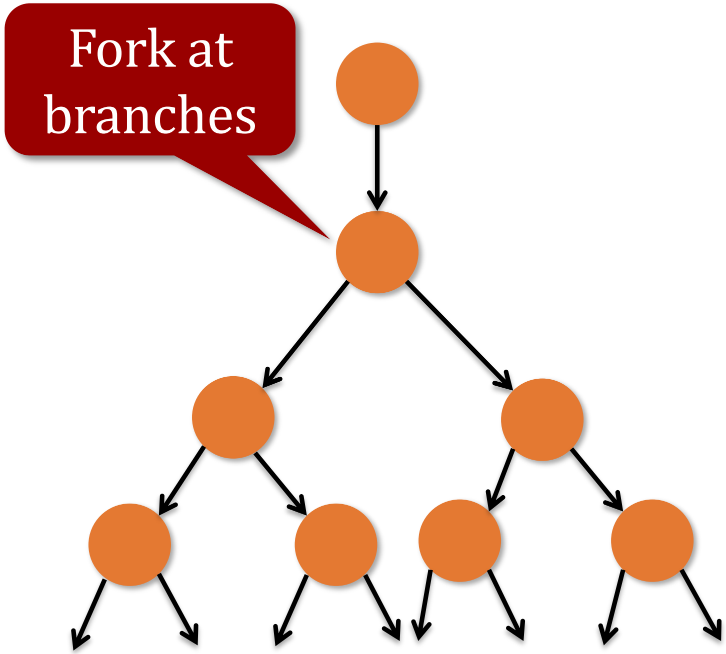
<u>Online (without initial seeds)</u>



Preliminary: online and offline

- ❑ ***Online*** and ***offline*** symbolic execution

<u>Online (without initial seeds)</u>



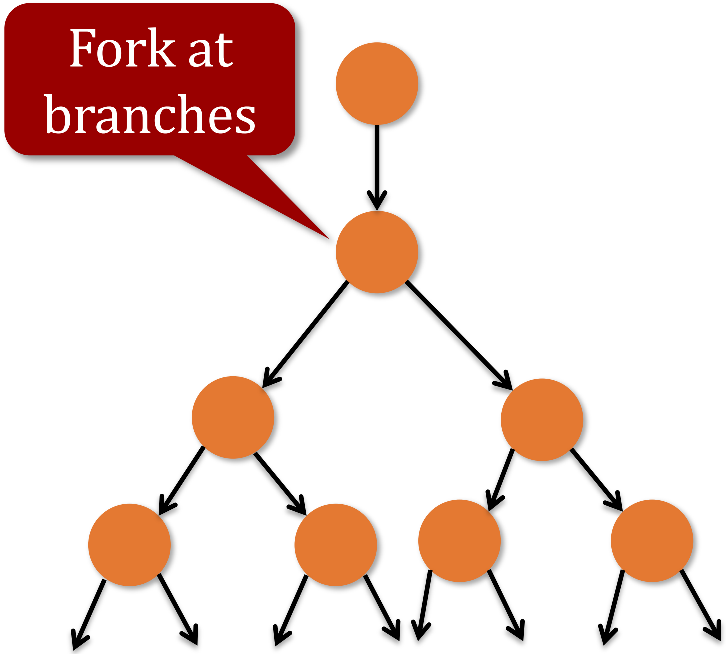
- Example: **KLEE**

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In OSDI.

Preliminary: online and offline

- ***Online*** and ***offline*** symbolic execution

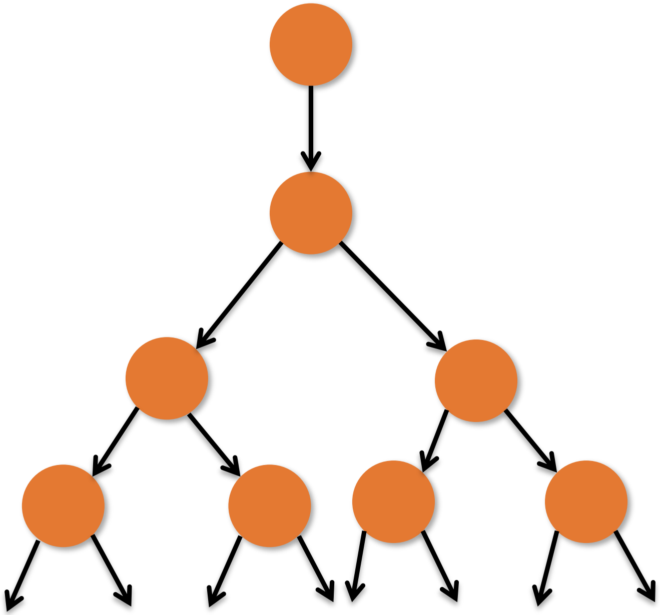
Online (without initial seeds)



- Example: **KLEE**

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In OSDI.

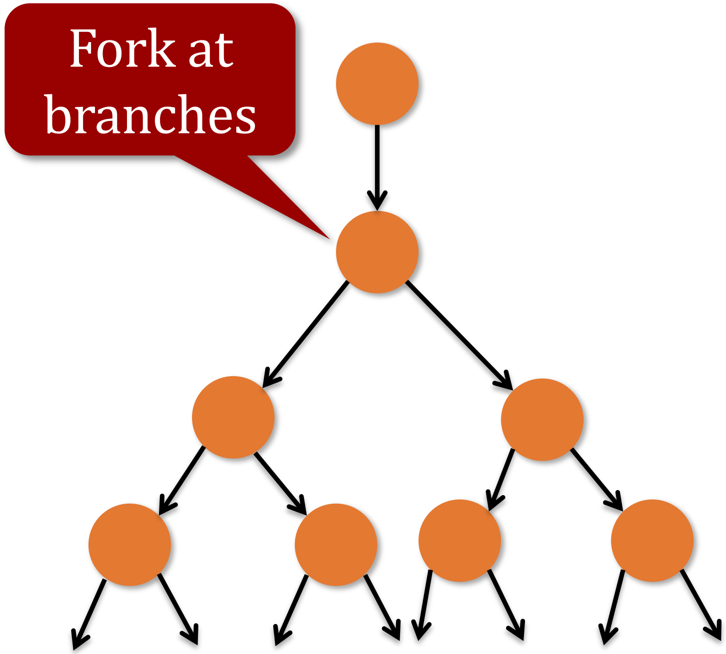
Offline (require initial seeds to setup)



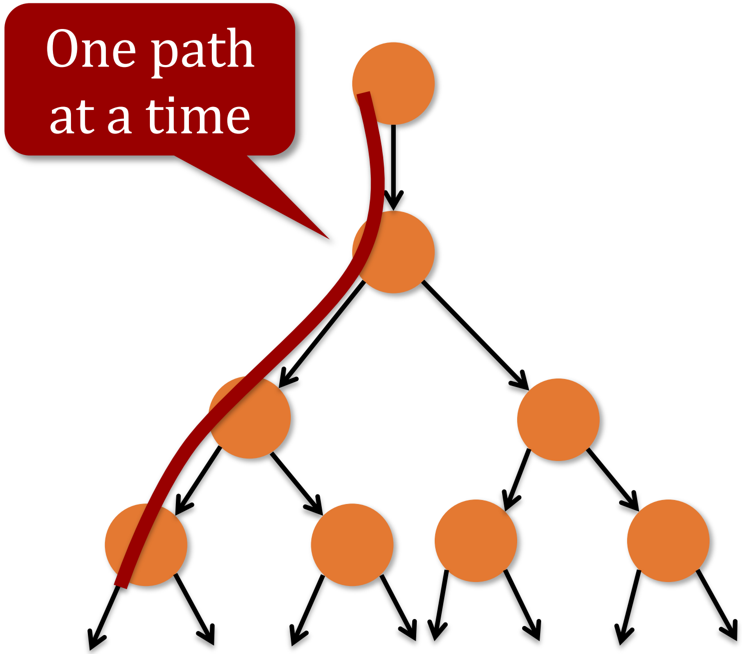
Preliminary: online and offline

- ***Online*** and ***offline*** symbolic execution

<u>Online (without initial seeds)</u>



<u>Offline (require initial seeds to setup)</u>



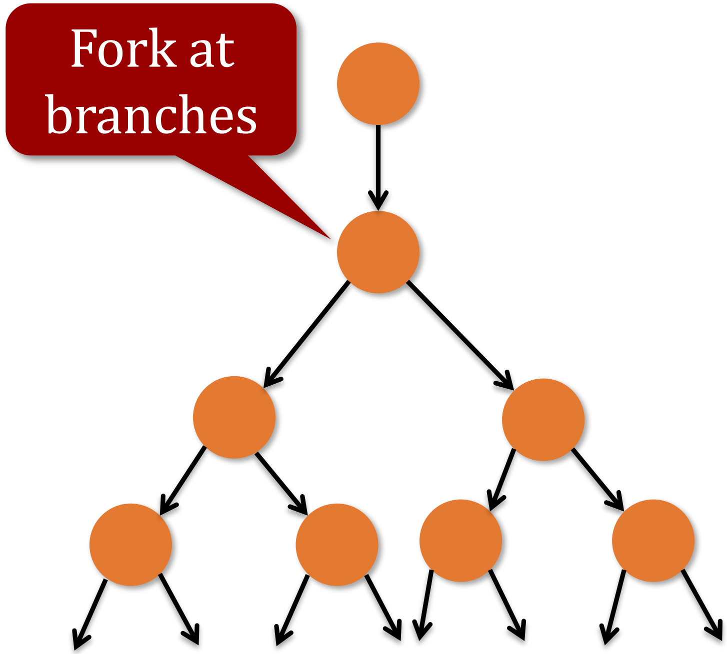
- Example: **KLEE**

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In OSDI.

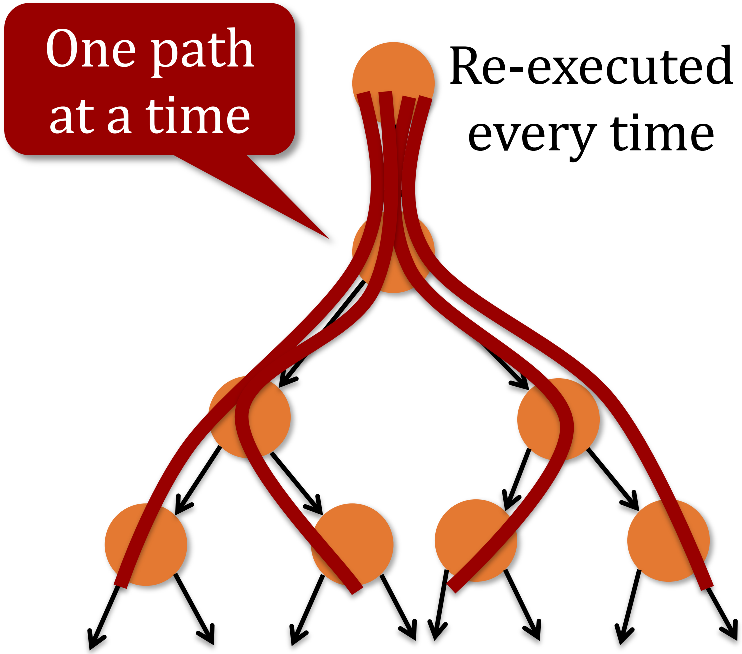
Preliminary: online and offline

- ❑ ***Online*** and ***offline*** symbolic execution

Online (without initial seeds)



Offline (require initial seeds to setup)



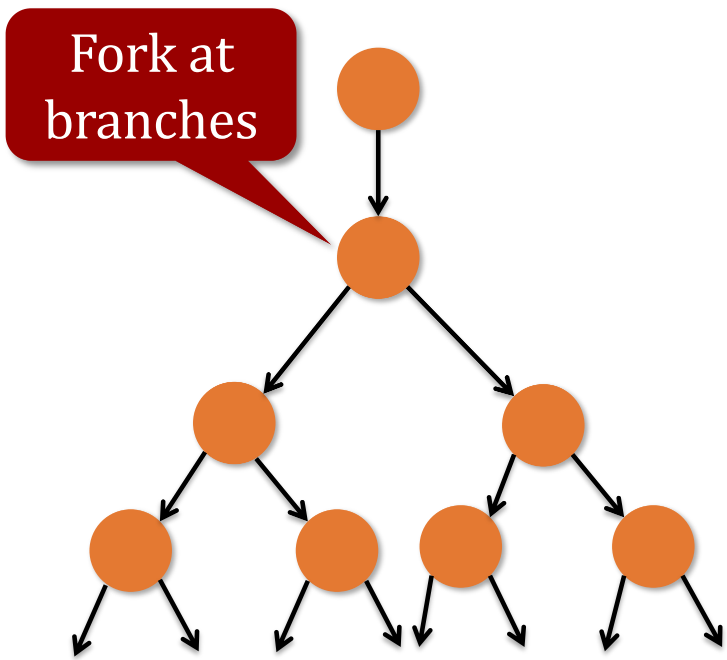
- Example: **KLEE**

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In OSDI.

Preliminary: online and offline

- *Online* and *offline* symbolic execution

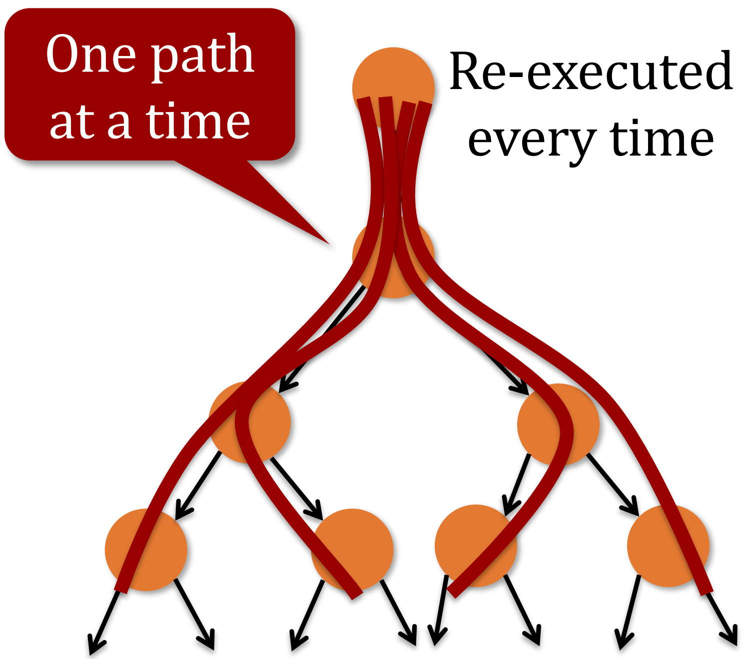
Online (without initial seeds)



- Example: **KLEE**

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In OSDI.

Offline (require initial seeds to setup)



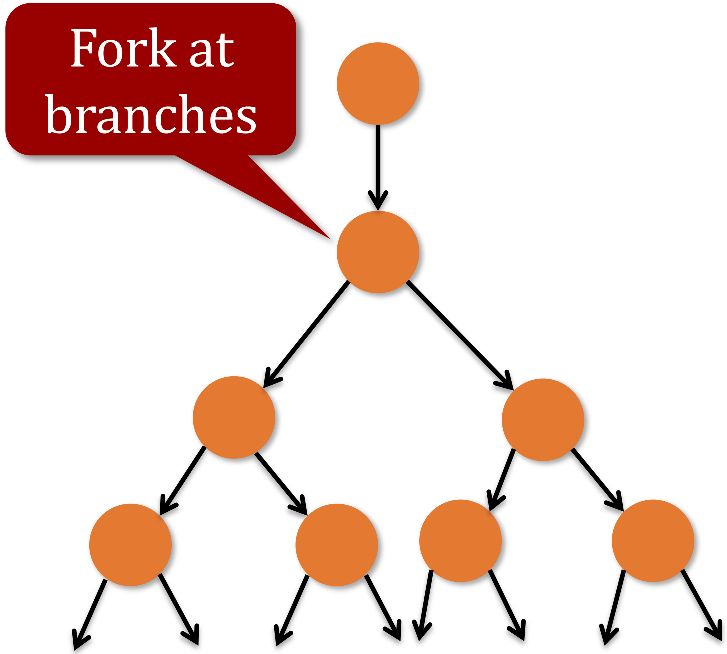
- Example: **SAGE**

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. Commun. ACM.

Preliminary: online and offline

- *Online* and *offline* symbolic execution

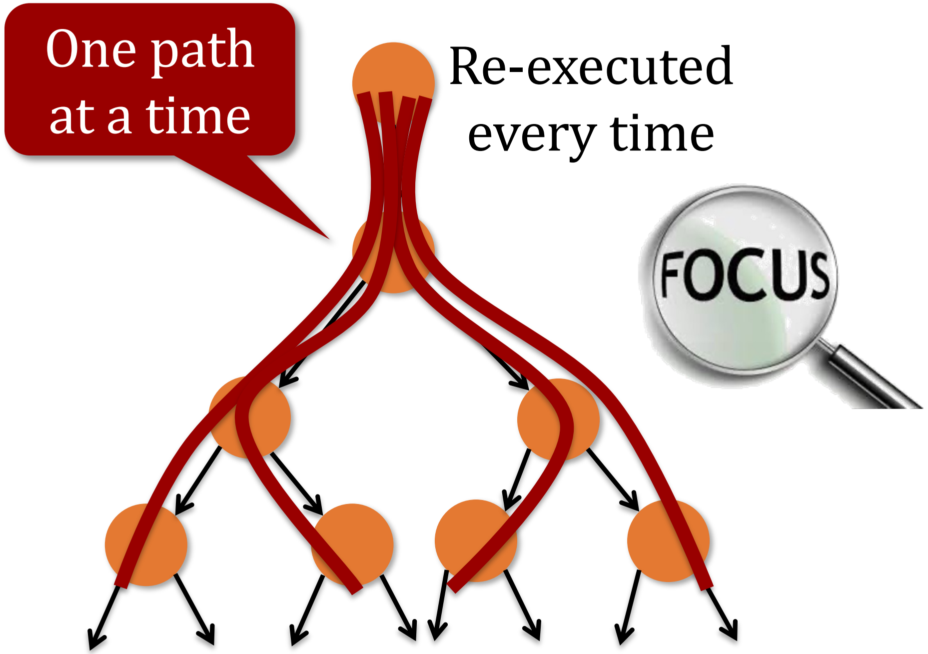
Online (without initial seeds)



- Example: **KLEE**

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In OSDI.

Offline (require initial seeds to setup)



- Example: **SAGE**

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. Commun. ACM.

Preliminary: compilation-based

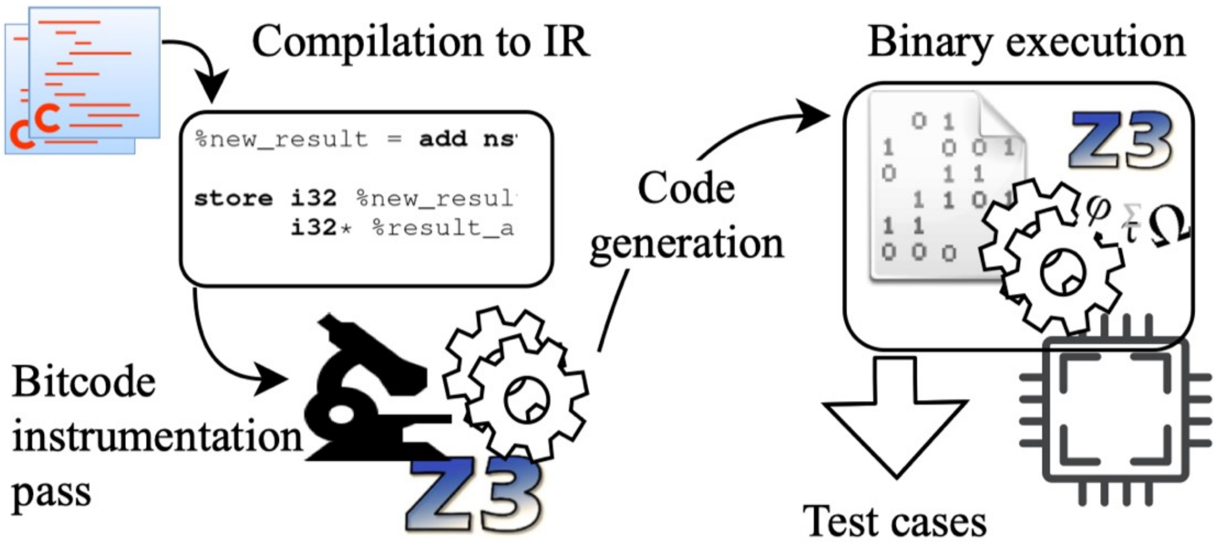
Preliminary: compilation-based

- **Prevalent concolic execution style**
  - Compilation-based concolic/symbolic execution: SymCC (Usenix Sec'20)
  - Potentially (ideally) can be used to test any scale software systems

Preliminary: compilation-based

❑ Prevalent concolic execution style

- Compilation-based concolic/symbolic execution: SymCC (Usenix Sec'20)
- Potentially (ideally) can be used to test any scale software systems



Preliminary: compilation-based

- Prevalent concolic execution style
  - Compilation-based concolic/symbolic execution: SymCC (Usenix Sec'20)
  - Potentially (ideally) can be used to test any scale software systems

How can we perform concolic execution to generate highly structured test inputs for systematically testing parsing programs?

Challenges

Challenges

- **#C1: what to solve?**
  - Not every path is worth solving (reduce resources/costs)
  - Existing path constraint selection strategy is not structure-aware

Challenges

SMU SINGAPORE MANAGEMENT UNIVERSITY
School of Computing and Information Systems

- **#C1: what to solve?**
  - Not every path is worth solving (reduce resources/costs)
  - Existing path constraint selection strategy is not structure-aware
- **#C2: how to solve?**
  - Need to make sure resulting test cases are both satisfactory to constraints and valid to syntax
  - Existing constraint solving only solve for satisfactory, which yields a large number of invalid inputs

Challenges

- **#C1: what to solve?**
  - Not every path is worth solving (reduce resources/costs)
  - Existing path constraint selection strategy is not structure-aware
- **#C2: how to solve?**
  - Need to make sure resulting test cases are both satisfactory to constraints and valid to syntax
  - Existing constraint solving only solve for satisfactory, which yields a large number of invalid inputs
- **#C3: How to acquire new seeds after saturation?**
  - For a better continuous testing
  - Existing measurements (e.g., using gcov) could be time-consuming to get during runtime

Key insights

Key insights

- **Insight 1: *unique* implementation characteristic**
  - Can help distinguish structural program paths
    - Select only interesting path constraints

Key insights

❑ Insight 1: *unique* implementation characteristic

- ➢ Can help distinguish structural program paths
  - ➢ Select only interesting path constraints

Seed

```
// seed input
c = 30000; a = [];
for (i = 0; i < 2 * c; i += 1) { a.push(i%c); }
a.sort(function (x, y) { return x - y; }); print(a[2 * c - 2]);
```

Input string

MuJS implementation

```
// Parsing logic (/* adapted from jslex.c */)
static int jsY_isidentifierpart(int c) {
    return isdigit(c) //"(bvsge #x00000039 (concat #x000000 k!95))"
    || isalpha(c) || c == '$' || c == '_' || isalpharune(c);
}
static int jsY_lexx(js_State *J){
    while (1) {
        //...
        switch (J->lexchar) {
            case '(': jsY_next(J); return '(';
            case ')': jsY_next(J); return ')';
            case ',': jsY_next(J); return ',';
            //...
        }
    //...
    }
}
```

Key insights

- ❑ **Insight 1: *unique* implementation characteristic**
  - ➢ Can help distinguish structural program paths
    - ➢ Select only interesting path constraints
- ❑ **Insight 2: strong completion capabilities of LLMs**
  - ➢ Can help solve constraints smartly (more details later)
    - ➢ Solve – Complete paradigm (our key contribution)

Seed

```
// seed input
c = 30000; a = [];
for (i = 0; i < 2 * c; i += 1) { a.push(i%c); }
a.sort(function (x, y) { return x - y; }); print(a[2 * c - 2]);
```

Input string

MuJS implementation

```
// Parsing logic (/* adapted from jslex.c */)
static int jsY_isidentifierpart(int c) {
    return isdigit(c) //"(bvsge #x00000039 (concat #x000000 k!95))"
    || isalpha(c) || c == '$' || c == '_' || isalpharune(c);
}
static int jsY_lexx(js_State *J){
    while (1) {
        //...
        switch (J->lexchar) {
           case '(': jsY_next(J); return '(';
           case ')': jsY_next(J); return ')';
           case ',': jsY_next(J); return ',';
            //...
        }
    //...
    }
}
```

Key insights

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

❑ Insight 1: *unique* implementation characteristic

- ➢ Can help distinguish structural program paths
  - ➢ Select only interesting path constraints

❑ Insight 2: strong completion capabilities of LLMs

- ➢ Can help solve constraints smartly (more details later)
  - ➢ Solve – Complete paradigm (our key contribution)

Seed

```
// seed input
c = 30000; a = [];
for (i = 0; i < 2 * c; i += 1) { a.push(i%c); }
a.sort(function (x, y) { return x - y; }); print(a[2 * c - 2]);
```

Input string

MuJS implementation

```
// Parsing logic (/* adapted from jslex.c */)
static int jsY_isidentifierpart(int c) {
    return isdigit(c) //"(bvsge #x00000039 (concat #x000000 k!95))"
    || isalpha(c) || c == '$' || c == '_' || isalpharune(c);
}
static int jsY_lexx(js_State *J){
    while (1) {
        //...
        switch (J->lexchar) {
            case '(': jsY_next(J); return '(';
            case ')': jsY_next(J); return ')';
            case ',': jsY_next(J); return ',';
            //...
        }
    //...
    }
}
```

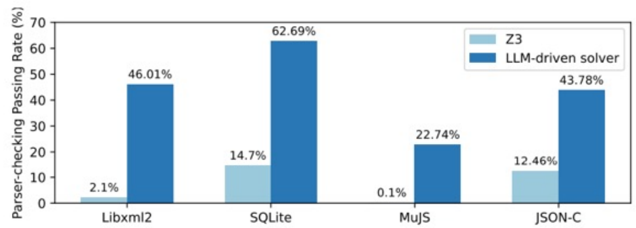


Figure 1: Parser checking passing rate comparison between traditional constraint solver (i.e., Z3) and LLM-driven solver (designed in COTTONTAIL).

Key insights

- **Insight 1: *unique* implementation characteristic**
  - Can help distinguish structural program paths
    - Select only interesting path constraints
- **Insight 2: strong completion capabilities of LLMs**
  - Can help solve constraints smartly (more details later)
    - Solve – Complete paradigm (our key contribution)
- **Insight 3: knowledgeable resources for seeds**
  - Can help generate fresh seeds when saturated

Seed

```
// seed input
c = 30000; a = [];
for (i = 0; i < 2 * c; i += 1) { a.push(i%c); }
a.sort(function (x, y) { return x - y; }); print(a[2 * c - 2]);
```

Input string

```
// Parsing logic (/* adapted from jslex.c */)
static int jsY_isidentifierpart(int c) {
    return isdigit(c) //"(bvsge #x00000039 (concat #x000000 k!95))"
    || isalpha(c) || c == '$' || c == '_' || isalpharune(c);
}
static int jsY_lexx(js_State *J){
    while (1) {
        //...
        switch (J->lexchar) {
            case '(': jsY_next(J); return '(';
            case ')': jsY_next(J); return ')';
            case ',': jsY_next(J); return ',';
            //...
        }
    //...
    }
}
```

MuJS implementation

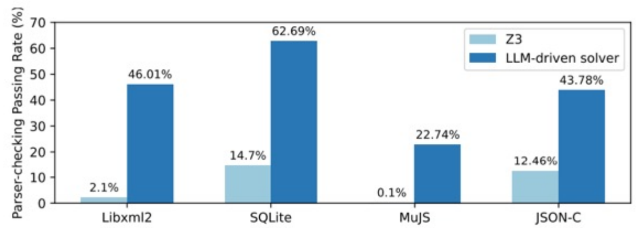
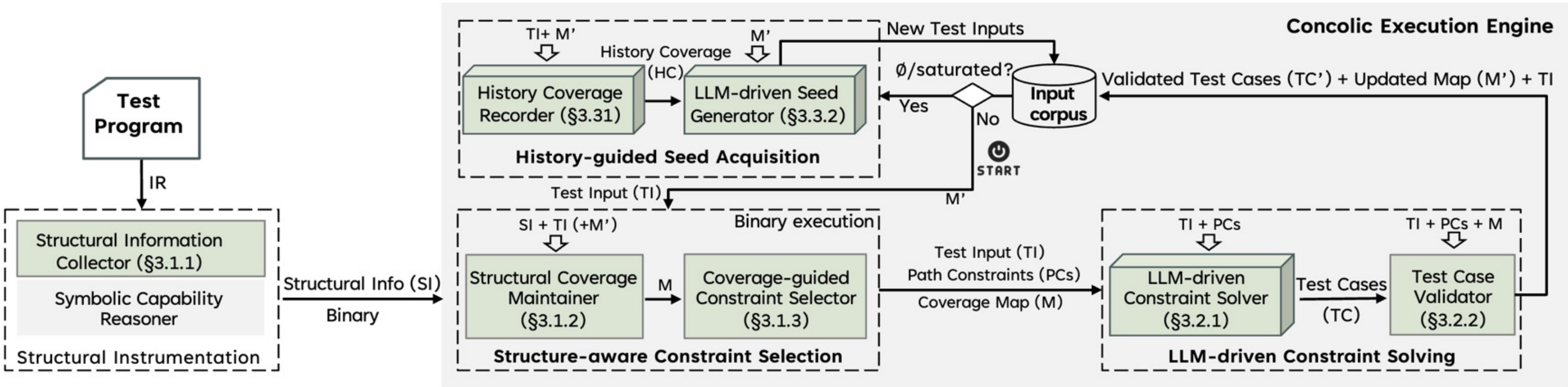
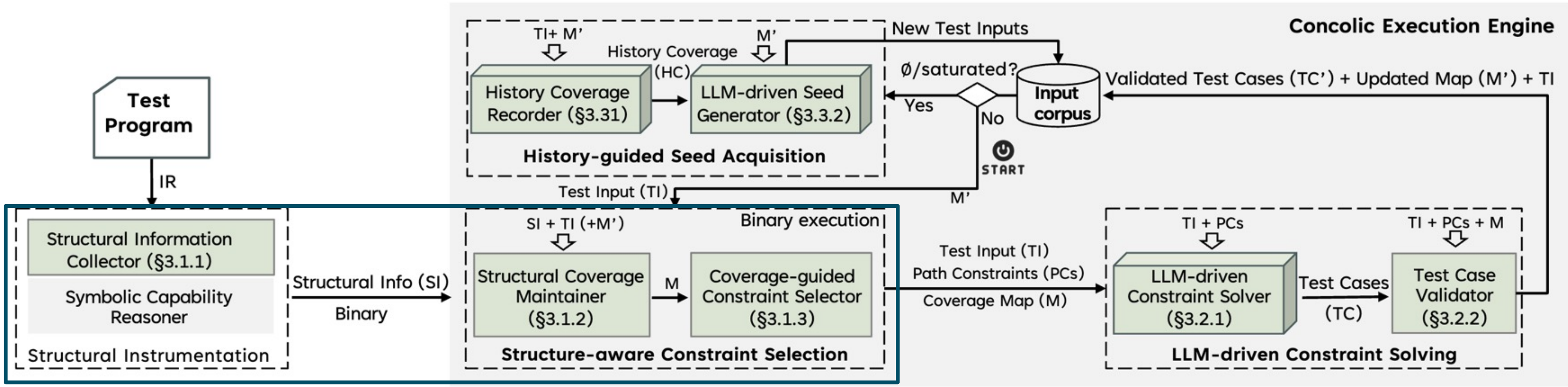


Figure 1: Parser checking passing rate comparison between traditional constraint solver (i.e., Z3) and LLM-driven solver (designed in CottonTail).

Our solution: Cottontail

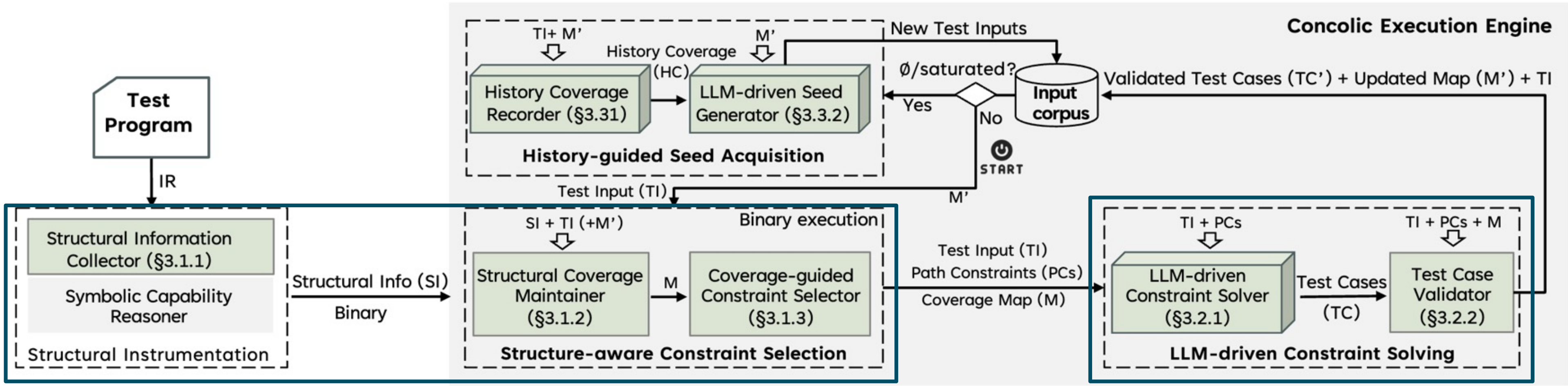


Our solution: Cottontail



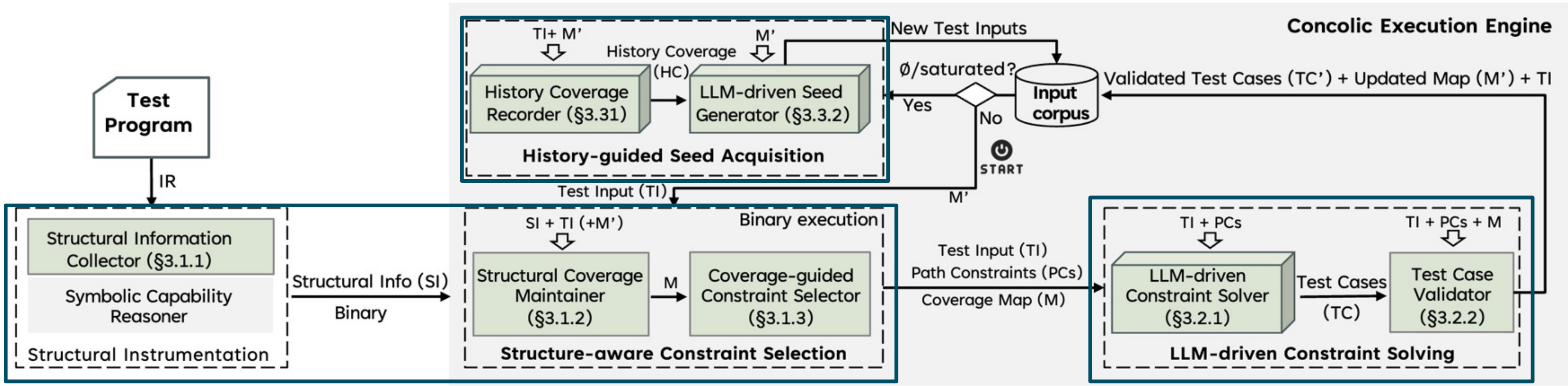
- **Three new techniques**
  - **Structure-aware constraint selection**: select only interesting path constraints (#C1)

Our solution: Cottontail



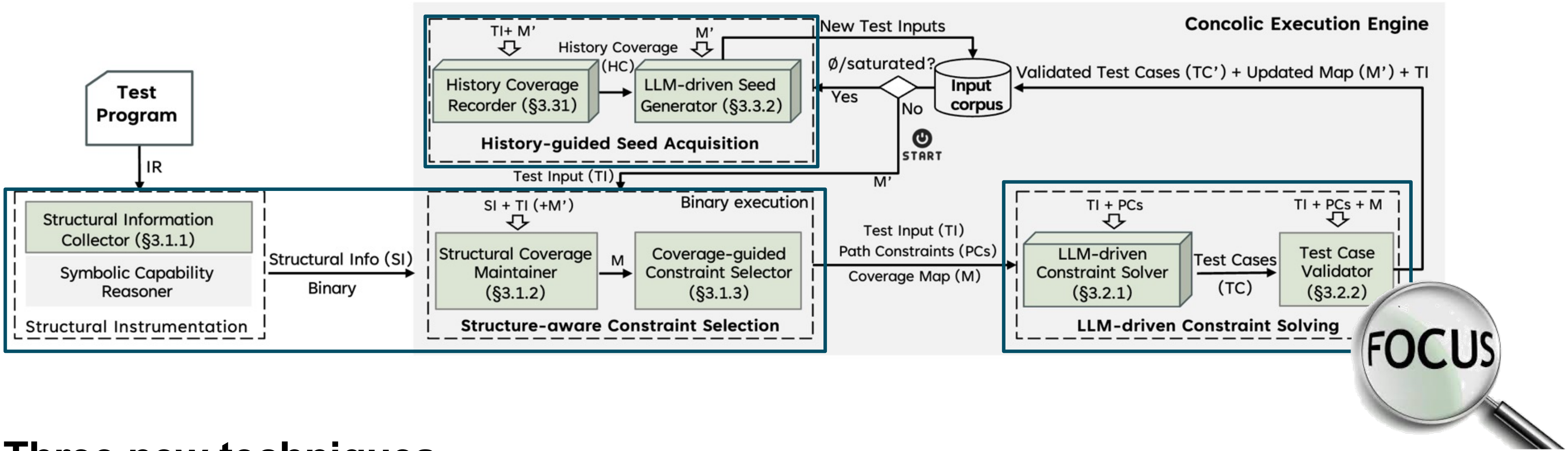
- **Three new techniques**
  - **Structure-aware constraint selection**: select only interesting path constraints (#C1)
  - **LLM-driven constraint solving:** smart LLM-solving + test case validator (#C2)

Our solution: Cottontail



- **Three new techniques**
  - **Structure-aware constraint selection**: select only interesting path constraints (#C1)
  - **LLM-driven constraint solving:** smart LLM-solving + test case validator (#C2)
  - **History-guided seed acquisition**: capable seed generation (#C3)

Our solution: Cottontail



- **Three new techniques**
  - **Structure-aware constraint selection**: select only interesting path constraints (#C1)
  - **LLM-driven constraint solving:** smart LLM-solving + test case validator (#C2)
  - **History-guided seed acquisition**: capable seed generation (#C3)

Novelty: Solve-Complete paradigm

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

</> System Prompt (*Solve-Complete* Paradigm with Chain of Thought)
You are a smart Constraint Solver.
Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

</> User Prompt

Given the path constraint ```bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> Response

Let's solve the constraint step by step:

1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```
```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);```
```
```

Novelty: Solve-Complete paradigm

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

</> **System Prompt (*Solve-Complete* Paradigm with Chain of Thought)**
You are a smart Constraint Solver.
Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

</> **User Prompt**

Given the path constraint ```bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> **Response**

Let's solve the constraint step by step:

1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```
```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;}
print(a[100]);```
```

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r**?**
```

Novelty: Solve-Complete paradigm

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

</> System Prompt (*Solve-Complete* Paradigm with Chain of Thought)

You are a smart Constraint Solver.

Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

</> User Prompt

Given the path constraint ```bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> Response

Let's solve the constraint step by step:

1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```
```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);```
```

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r?

Step 1: syntax-aware *solve* process

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {re
```

Novelty: Solve-Complete paradigm

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

</> **System Prompt (*Solve-Complete* Paradigm with Chain of Thought)**
You are a smart Constraint Solver.
Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

Tradition solver (Z3): r9

</> **User Prompt**
Given the path constraint ``bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> **Response**
Let's solve the constraint step by step:

1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```
```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);```
```

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r?

Step 1: syntax-aware *solve* process

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {re
```

Novelty: Solve-Complete paradigm

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

</> **System Prompt (*Solve-Complete* Paradigm with Chain of Thought)**
You are a smart Constraint Solver.
Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

</> **User Prompt**

Tradition solver (Z3): r9

Given the path constraint ```bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> **Response**

Let's solve the constraint step by step:

1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```
c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);
```

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r?

Step 1: syntax-aware *solve* process

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {re

Step 2: syntax-aware *complete* process

c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);```

Novelty: Solve-Complete paradigm

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

</> **System Prompt (*Solve-Complete* Paradigm with Chain of Thought)**
You are a smart Constraint Solver.
Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

Tradition solver (Z3): r9

</> **User Prompt**
Given the path constraint ``bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> **Response**
Let's solve the constraint step by step:
1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```
c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);
```

```
c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r?
```

Step 1: syntax-aware *solve* process

```
c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {re
```

Step 2: syntax-aware *complete* process

```
c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);```
```

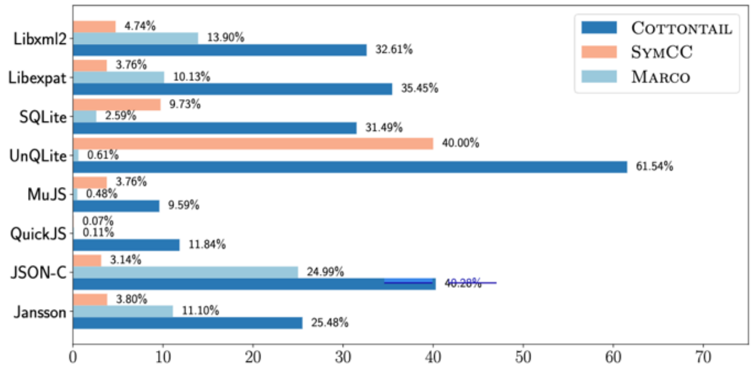
Solve **and** ***Complete*** **for not only constraint satisfiability but also syntax validity**

Evaluation Results & Takeaways

Evaluation Results & Takeaways

RQ1: How does Cottontail perform compared with baseline approaches?

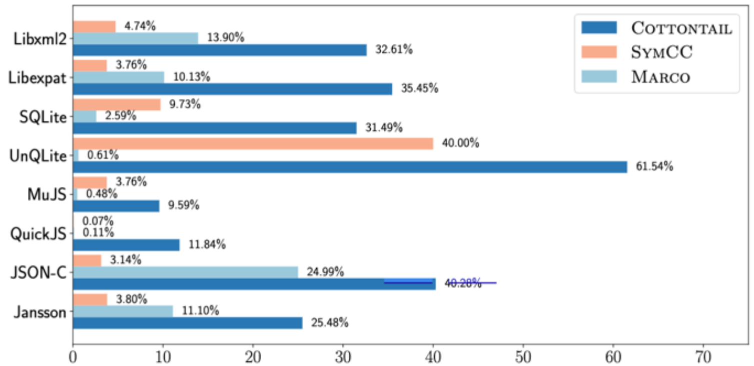
– **Cottontail** outperforms state-of-the-art approaches (SymCC and Marco) by **14.15%** and **14.31%** in terms of line coverage



Evaluation Results & Takeaways

RQ1: How does Cottontail perform compared with baseline approaches?

– **Cottontail** outperforms state-of-the-art approaches (SymCC and Marco) by **14.15%** and **14.31%** in terms of line coverage



RQ2: Can Cottontail find new vulnerabilities in practice?

– **Cottontail** detected **6** new vulnerabilities, showing practical vulnerability detection capability

– Marco can only detect one and SymCC can detect four

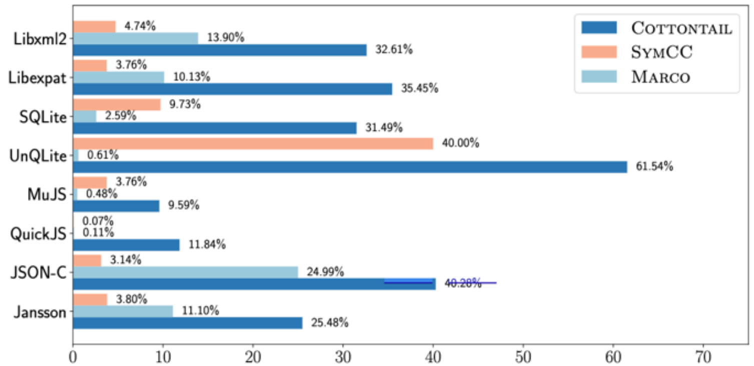
ID	Subject	Description	Status	CVE-Assigned
#1	MuJS	Memory leak	Fixed	CVE-2024-55061
#2	MuJS	Heap overflow	Fixed	CVE-2025-26082
#3	QuickJS	Stack overflow	Fixed	CVE-2024-13903
#4	QuickJS	Stack overflow	Fixed	CVE-2025-26081
#5	UnQLite	Global overflow	Reported	CVE-2025-26083
#6	UnQLite	Heap overflow	Reported	CVE-2025-3791

Evaluation Results & Takeaways

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

RQ1: How does Cottontail perform compared with baseline approaches?

- **Cottontail** outperforms state-of-the-art approaches (SymCC and Marco) by **14.15%** and **14.31%** in terms of line coverage



RQ2: Can Cottontail find new vulnerabilities in practice?

- **Cottontail** detected **6** new vulnerabilities, showing practical vulnerability detection capability
- Marco can only detect one and SymCC can detect four

ID	Subject	Description	Status	CVE-Assigned
#1	MuJS	Memory leak	Fixed	CVE-2024-55061
#2	MuJS	Heap overflow	Fixed	CVE-2025-26082
#3	QuickJS	Stack overflow	Fixed	CVE-2024-13903
#4	QuickJS	Stack overflow	Fixed	CVE-2025-26081
#5	UnQLite	Global overflow	Reported	CVE-2025-26083
#6	UnQLite	Heap overflow	Reported	CVE-2025-3791

- **Takeaways**
  - **Smart LLM-driven constraint solving could help generate more complex test cases**

Retrospection of Motivation Example

```
void  buggy () {
  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { … }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  … // deeply nested
     vulnerable_func2();
  …

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

Retrospection of Motivation Example

```
void  buggy () {

    // Vuln1: require complete memory modeling
    void * from = malloc (100);
    void * to = malloc (100);
    if (to > from) { ... }
    if (from > to) {
        vulnerable_func1();
     }

    // Vuln2: require handling path explosion
    ... // deeply nested
        vulnerable_func2();
    ...

    // Vuln3: require structured test inputs
    if (input = "{....}") {
        // application logic
        vulnerable_func3();
    } else {
        earlier_termination();
    }
}
```

- ☐ **SymLoc: new memory modeling**

Retrospection of Motivation Example

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { … }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  … // deeply nested
     vulnerable_func2();
  …

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

- ☐ **SymLoc: new memory modeling**
- ☐ **FastKLEE: efficient path exploration**
- ☐ **Vital: effective path exploration**

Retrospection of Motivation Example

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
     vulnerable_func1();
  }

  // Vuln2: require handling path explosion
  ... // deeply nested
     vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

- ☐ SymLoc: new memory modeling
- ☐ FastKLEE: efficient path exploration
- ☐ Vital: effective path exploration
- ☐ Cottontail: smart test input generation

Retrospection of Motivation Example

```
void  buggy () {

  // Vuln1: require complete memory modeling
  void * from = malloc (100);
  void * to = malloc (100);
  if (to > from) { ... }
  if (from > to) {
     vulnerable_func1();
   }

  // Vuln2: require handling path explosion
  ... // deeply nested
     vulnerable_func2();
  ...

  // Vuln3: require structured test inputs
  if (input = "{....}") {
     // application logic
     vulnerable_func3();
  } else {
     earlier_termination();
  }
}
```

- ☐ **SymLoc: new memory modeling**
- ☐ **FastKLEE: efficient path exploration**
- ☐ **Vital: effective path exploration**
- ☐ **Cottontail: smart test input generation**

But not all vulnerabilities can be caught

Outline

- Background
  - Software vulnerabilities
  - Symbolic execution
- Motivation
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objects
- Methodology
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- **Future work**
  - Extension of current solutions
  - Combine program analysis with GenAI
- Conclusion and Acknowledgment

Future work

Future work

☐ Extension of Current Solutions

- Extend SymLoc with more realistic memory modeling
- Extend Vital into greybox/hybrid as targets to guide directed fuzzing
- Extend Cottontail to support complex input formats (e.g., binaries)

Future work

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

☐ Extension of Current Solutions

- Extend SymLoc with more realistic memory modeling
- Extend Vital into greybox/hybrid as targets to guide directed fuzzing
- Extend Cottontail to support complex input formats (e.g., binaries)

☐ Program analysis + GenAI

Future work

☐ Extension of Current Solutions

- Extend SymLoc with more realistic memory modeling
- Extend Vital into greybox/hybrid as targets to guide directed fuzzing
- Extend Cottontail to support complex input formats (e.g., binaries)

☐ Program analysis + GenAI

– New testing engine that can be more general (e.g., across different langauges)

Future work

☐ Extension of Current Solutions

- ➢ Extend SymLoc with more realistic memory modeling
- ➢ Extend Vital into greybox/hybrid as targets to guide directed fuzzing
- ➢ Extend Cottontail to support complex input formats (e.g., binaries)

☐ Program analysis + GenAI

- – New testing engine that can be more general (e.g., across different langauges)
- – Combine LLM with formal verification techniques to guarantee the robustness

Future work

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

☐ Extension of Current Solutions

- Extend SymLoc with more realistic memory modeling
- Extend Vital into greybox/hybrid as targets to guide directed fuzzing
- Extend Cottontail to support complex input formats (e.g., binaries)

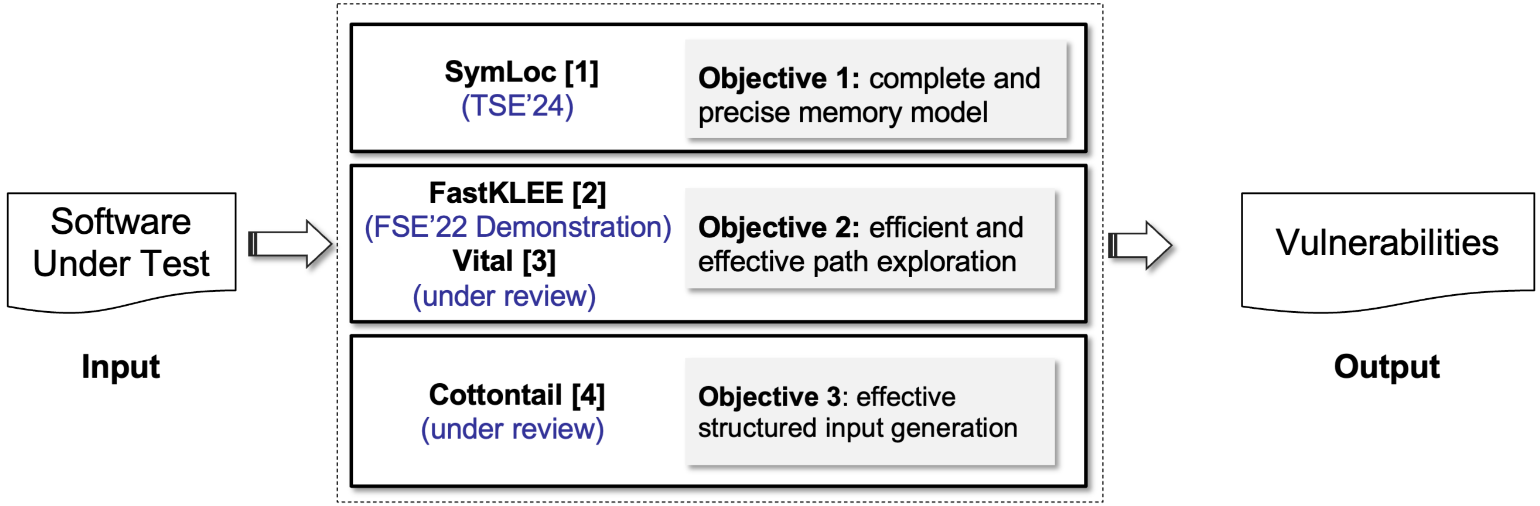
☐ Program analysis + GenAI

- New testing engine that can be more general (e.g., across different langauges)
- Combine LLM with formal verification techniques to guarantee the robustness
- Practical hybrid selective symbolic execution for mixed programs
  - New software ecosystem (human written code + AI generated code)

Outline

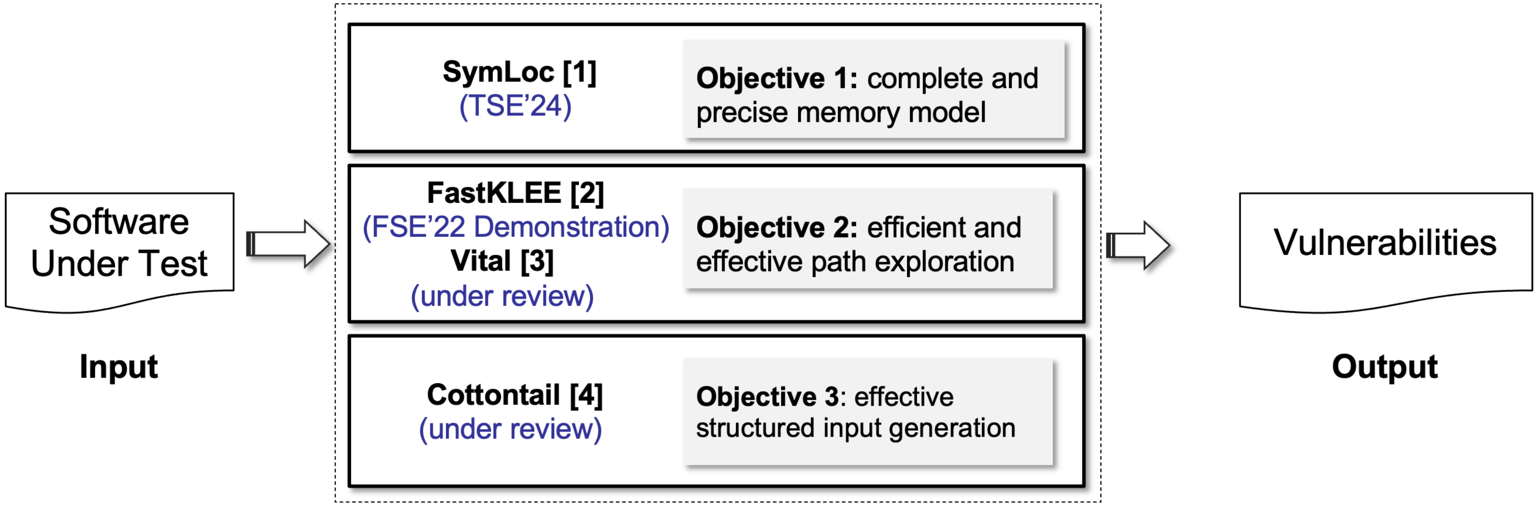
- Background
  - Software vulnerabilities
  - Symbolic execution
- Motivation
  - What limitations prevent current symbolic execution for vulnerability detection?
  - Thesis statement and research objects
- Methodology
  - **SymLoc:** A new memory model for symbolic execution
  - **FastKLEE and Vital:** Two new path exploration for symbolic execution
  - **Cottontail:** A new structured input generation for symbolic execution
- Future work
  - Extend key ideas to hybrid fuzzing
  - Combine Program Analysis with GenAI
- **Conclusion and Acknowledgment**

Conclusion



Software Under Test
Input
SymLoc [1]
(TSE'24)
Objective 1: complete and precise memory model
FastKLEE [2]
(FSE'22 Demonstration)
Vital [3]
(under review)
Objective 2: efficient and effective path exploration
Cottontail [4]
(under review)
Objective 3: effective structured input generation
Vulnerabilities
Output

Conclusion



- **Research impact**
  - **Four** open-sourced tools to foster further symbolic execution research
  - **10+** new memory related vulnerabilities are detected (with 7 new CVE assigned)
    - All of them have been fixed by developers so far

[1] **Haoxin Tu**, Lingxiao Jiang, and et.al, "*Concretely Mapped Symbolic Memory Locations for Memory Error Detection*" (IEEE TSE).
[2] **Haoxin Tu**, Lingxiao Jiang, and et.al,, "*FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers*", in Tool Demonstrations Track of FSE 2022.
[3] **Haoxin Tu**, Lingxiao Jiang and et.al,. "*Vital: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search*." arXiv:2408.08772 (2024).
[4] **Haoxin Tu**, Seongmin Lee, and et.al. "*Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation*." arXiv:2504.17542 (2025).

Acknowledgement

Thank you & Questions?

Boosting Symbolic Execution for Vulnerability Detection

Software Under Test

Input

SymLoc [1]
(TSE'24)

Objective 1: complete and precise memory model

FastKLEE [2]
(FSE'22 Demonstration)
Vital [3]
(under review)

Objective 2: efficient and effective path exploration

Cottontail [4]
(under review)

Objective 3: effective structured input generation

Vulnerabilities

Output

Dissertation Defense by Haoxin Tu (May 9th, 2024)

References

[1] Automated Test Generation: **“A Journey from Symbolic Execution to Smart Fuzzing and Beyond”** (Keynote by Koushik Sen)

[2] Zhide Zhou, Zhilei Ren, Guojun Gao, He Jiang. “**An empirical study of optimization bugs in GCC and LLVM”**. JSS, 2021.

[3] James C. King. 1976. **Symbolic execution and program testing**. Commun. ACM 19, 7 (July 1976), 385–394.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. 2018. **“A Survey of Symbolic Execution Techniques”.** ACM Computer Survey. 51, 3, Article 50 (July 2018), 39 pages.

[5] Seo, Hyunmin, and Sunghun Kim. "**How we get there: a context-guided search strategy in concolic testing**." Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008.” **KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”.** In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, USA, 209–224.

[7] C. Cadar and K. Sen, “**Symbolic execution for software testing: three decades later**,” Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, “**S2E: a platform for in-vivo multi-path analysis of software systems**,” in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, New York, NY, USA, Mar. 2011, pp. 265–278.

[9] S. Poeplau and A. Francillon, “**SymQEMU: Compilation-based symbolic execution for binaries**,” presented at the in Proceedings of the 2021 Network and Distributed System Security Symposium, 2021.

[10] Y. Shoshitaishvili et al., “**SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis**,” in 2016 IEEE Symposium on Security and Privacy (SP), May 2016, pp. 138–157.

[11] S. Poeplau and A. Francillon, “**Symbolic execution with SymCC: Don’t interpret, compile!,”** in 29th USENIX Security Symposium, 2020, pp. 181–198.

[12] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. “**Past-sensitive pointer analysis for symbolic execution**”. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). New York, NY, USA, 197–208.