The Long Road towards Testing Multi-Threaded Programs with KLEE

Daniel Schemmel, Julian Büning, David Laprell, Klaus Wehrle
RWTH Aachen University

https://comsys.rwth-aachen.de

Zoomspace, 2021-06-11

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent
- We believe that Symbolic Execution can help test those programs [1]

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs.
D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent
- We believe that Symbolic Execution can help test those programs [1]

- Adding support for multi-threaded programs to KLEE initially sounds simple:

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs.
D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent
- We believe that Symbolic Execution can help test those programs [1]

- Adding support for multi-threaded programs to KLEE initially sounds simple:
  1. Extend KLEE states to manage multiple thread stacks

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs.
D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent
- We believe that Symbolic Execution can help test those programs [1]

- Adding support for multi-threaded programs to KLEE initially sounds simple:
  1. Extend KLEE states to manage multiple thread stacks
  2. Add a concurrency-aware searcher

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs.
D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent
- We believe that Symbolic Execution can help test those programs [1]

- Adding support for multi-threaded programs to KLEE initially sounds simple:
  1. Extend KLEE states to manage multiple thread stacks
  2. Add a concurrency-aware searcher
  3. Model the `pthread` API

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs.
D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Introduction

- As core counts rise, multi-threaded programs become more and more prevalent
- We believe that Symbolic Execution can help test those programs [1]

- Adding support for multi-threaded programs to KLEE initially sounds simple:
  1. Extend KLEE states to manage multiple thread stacks
  2. Add a concurrency-aware searcher
  3. Model the `pthread` API

- **Where is the catch?**

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs.
D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Adding Basic Thread-Support to KLEE

- KLEE states are easily extended to track multiple stacks and instruction pointers

Adding Basic Thread-Support to KLEE

- KLEE states are easily extended to track multiple stacks and instruction pointers
- Executing every instruction interleaving causes **immediate state explosion**!

Adding Basic Thread-Support to KLEE

- KLEE states are easily extended to track multiple stacks and instruction pointers
- Executing every instruction interleaving causes **immediate state explosion**!
  - # of possible paths $=$ #threads$^{\text{\#instructions per thread}}$

Adding Basic Thread-Support to KLEE

- KLEE states are easily extended to track multiple stacks and instruction pointers
- Executing every instruction interleaving causes **immediate state explosion**!
  - # of possible paths $=$ #threads$^{\text{#instructions per thread}}$
  - Tiny example: $8^{20} = 2^{60}$

Adding Basic Thread-Support to KLEE

- KLEE states are easily extended to track multiple stacks and instruction pointers
- Executing every instruction interleaving causes **immediate state explosion**!
    - # of possible paths $=$ #threads$^{\text{\#instructions per thread}}$
    - Tiny example: $8^{20} = 2^{60}$
    - Executing any one instruction turns one state intp #threads states

Adding Basic Thread-Support to KLEE

- KLEE states are easily extended to track multiple stacks and instruction pointers
- Executing every instruction interleaving causes **immediate state explosion**!
  - # of possible paths $=$ #threads$^{\text{#instructions per thread}}$
  - Tiny example: $8^{20} = 2^{60}$
  - Executing any one instruction turns one state intp #threads states

- **A posteriori approaches (searchers, state pruning) alone are insufficient!**

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4
```

Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4
```

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4

; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

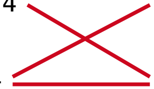
Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4

; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4

; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```



Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4

; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4

; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4

; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

Data races are undefined behavior!
[C11 §5.1.2.4/25] [C18 §5.1.2.4/35]

Data Race Detection

- Our KLEE fork detects data races as a new bug category

Data Race Detection

- Our KLEE fork detects data races as a new bug category
- For each byte, we record which thread read/wrote it last

Data Race Detection

- Our KLEE fork detects data races as a new bug category
- For each byte, we record which thread read/wrote it last
- Efficient if a memory object is used by one thread exclusively

Data Race Detection

- Our KLEE fork detects data races as a new bug category
- For each byte, we record which thread read/wrote it last
- Efficient if a memory object is used by one thread exclusively
  - Most memory objects are not shared between threads

Data Race Detection

- Our KLEE fork detects data races as a new bug category
- For each byte, we record which thread read/wrote it last
- Efficient if a memory object is used by one thread exclusively
  - Most memory objects are not shared between threads
  - Symbolic accesses to shared memory objects may require SMT solving

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4


; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4


; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
```

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4

call void @lock()
; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
call void @unlock()
```

Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4

call void @lock()
; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
call void @unlock()
```

Coarse Thread-Scheduling

Thread 1

```
; a += 1
%1 = load i32, i32* @a, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @a, align 4

call void @lock()
; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
call void @unlock()
```

Thread 2

```
; b += 1
%1 = load i32, i32* @b, align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @b, align 4

call void @lock()
; x += 1
%3 = load i32, i32* @x, align 4
%4 = add nsw i32 %3, 1
store i32 %4, i32* @x, align 4
call void @unlock()
```

Coarse Thread-Scheduling

Thread 1

```
  ; a += 1
  %1 = load i32, i32* @a, align 4
  %2 = add nsw i32 %1, 1
  store i32 %2, i32* @a, align 4

● call void @lock()
  ; x += 1
  %3 = load i32, i32* @x, align 4
  %4 = add nsw i32 %3, 1
  store i32 %4, i32* @x, align 4
● call void @unlock()
```

Thread 2

```
  ; b += 1
  %1 = load i32, i32* @b, align 4
  %2 = add nsw i32 %1, 1
  store i32 %2, i32* @b, align 4

● call void @lock()
  ; x += 1
  %3 = load i32, i32* @x, align 4
  %4 = add nsw i32 %3, 1
  store i32 %4, i32* @x, align 4
● call void @unlock()
```

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further
  - Happens all the time in parallel programs

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further
  - Happens all the time in parallel programs
  - E.g., consider two threads atomically incrementing a counter

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further
  - Happens all the time in parallel programs
  - E.g., consider two threads atomically incrementing a counter
- We detect such **cutoff events** by hashing all memory

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further
  - Happens all the time in parallel programs
  - E.g., consider two threads atomically incrementing a counter
- We detect such **cutoff events** by hashing all memory
- Also, our partial-order-reduction algorithm **re-executes some states**

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further
  - Happens all the time in parallel programs
  - E.g., consider two threads atomically incrementing a counter
- We detect such **cutoff events** by hashing all memory
- Also, our partial-order-reduction algorithm **re-executes some states**
  - These re-executions must yield the exact same state as the original execution

Cutoff Events and Re-executions

- If two paths yield the exact same state, only one needs to be pursued further
  - Happens all the time in parallel programs
  - E.g., consider two threads atomically incrementing a counter
- We detect such **cutoff events** by hashing all memory
- Also, our partial-order-reduction algorithm **re-executes some states**
  - These re-executions must yield the exact same state as the original execution

A deterministic engine is key for our approach!

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables
- Relies on recognizing data races as bugs (and does so)

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables
- Relies on recognizing data races as bugs (and does so)
- Requires deterministic re-execution

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables
- Relies on recognizing data races as bugs (and does so)
- Requires deterministic re-execution

Symbolic Execution + **Partial-Order Reduction**

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables
- Relies on recognizing data races as bugs (and does so)
- Requires deterministic re-execution

Symbolic Execution
data non-determinism

+

Partial-Order Reduction
thread non-determinism

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables
- Relies on recognizing data races as bugs (and does so)
- Requires deterministic re-execution

Symbolic Execution
data non-determinism

+

Partial-Order Reduction
thread non-determinism

Check out our tool: **https://github.com/por-se/por-se**

Conclusion

- Our KLEE fork can symbolically execute multi-threaded programs
- Support for most POSIX threading primitives, including condition variables
- Relies on recognizing data races as bugs (and does so)
- Requires deterministic re-execution

Symbolic Execution
data non-determinism

+

Partial-Order Reduction
thread non-determinism

Check out our tool: **https://github.com/por-se/por-se**
Read the extended version of our CAV'20 paper [1]: **https://arxiv.org/abs/2005.06688**

[1] Symbolic Partial-Order Execution for Testing Multi-Threaded Programs. D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, K. Wehrle. CAV 2020.

Acknowledgements

This research is supported by the European Research Council (ERC) under the European Union’s Horizon 2020 Research and Innovation Programme (grant agreement №. 647295 (SYMBIOSYS)).

European Research Council
Established by the European Commission