SIFT: A Multithreading Extension to KLEE

Tuba Yavuz

Associate Professor

University of Florida

KLEE Workshop 2022

Imperial College, London

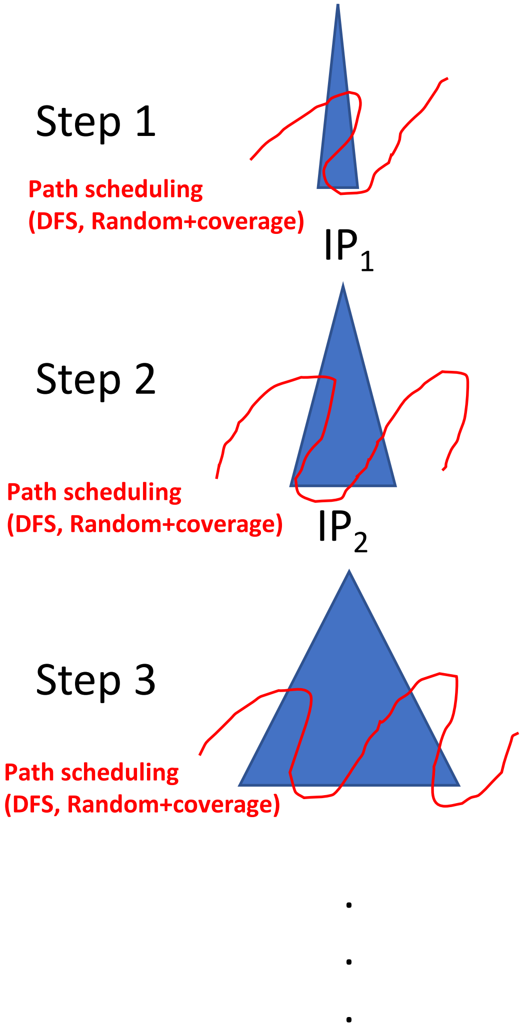
Motivation

- Detecting memory vulnerabilities in multithreaded code is challenging
- Existing work use various heuristics such as context bounding or schedule variation w.r.t. some interference points
- Existing property directed scheduling approaches handle assertions only and rely on an offline static analysis
- Symbolic execution is effective in memory vulnerabilities
- The path explosion in symbolic execution gets exacerbated for multithreaded code
- ***There is a need for property directed symbolic execution of multithreaded code.***

Approach

- Compute data-flow facts for property relevant code locations based on explored symbolic execution paths
  - Memory deallocations, memory accesses based on pointer arithmetic, assertion checks
- Identify instructions or *Interleaving Points* (context-switch points)
  - Impact property relevant code locations
  - Interference points of multiple threads
- As new paths get explored, update the interleaving points
- Until the property violation is detected or a timeout is reached

SIFT’s Exploration Steps



Step 1

Path scheduling
(DFS, Random+coverage)

IP_1

Conservative thread scheduling,
i.e., when a thread blocks schedule another

Step 2

Path scheduling
(DFS, Random+coverage)

IP_2

New thread scheduling scenarios due to interleaving points in IP_1

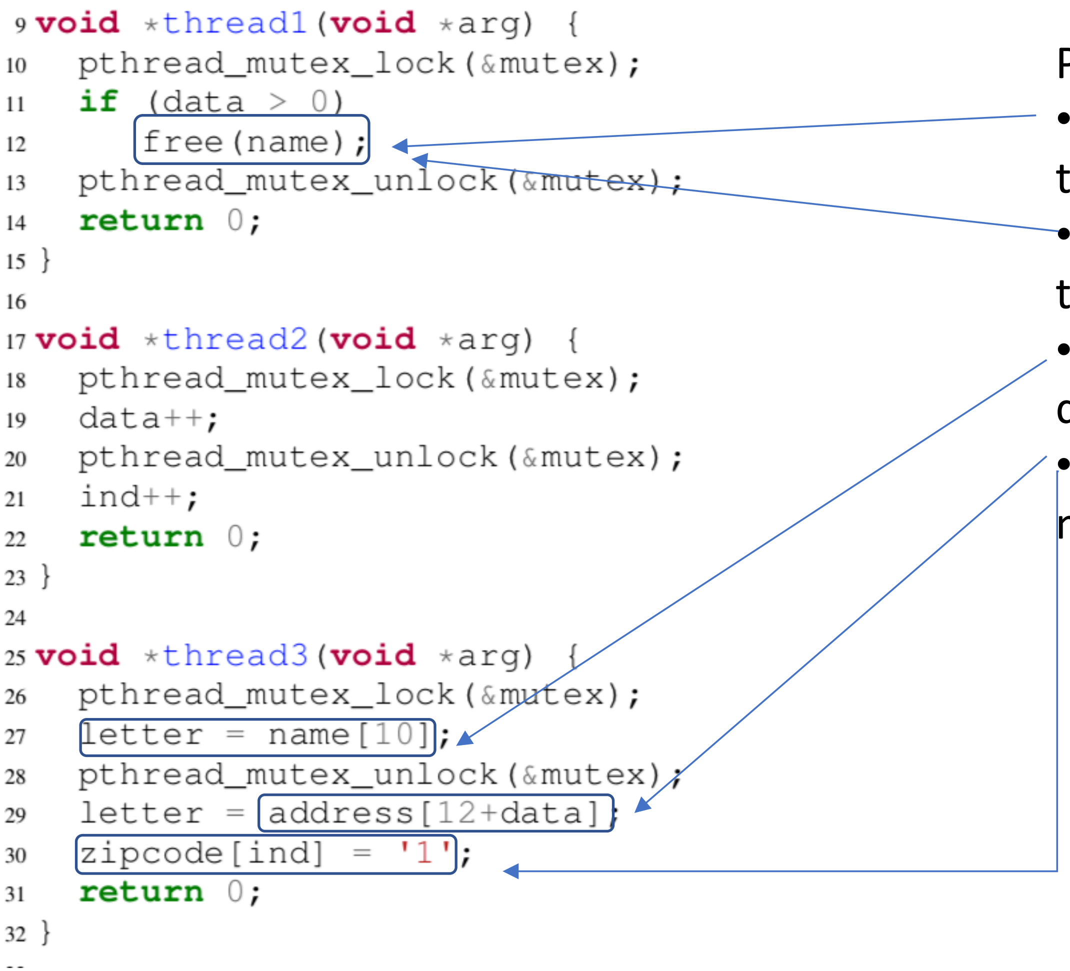
Step 3

Path scheduling
(DFS, Random+coverage)

New thread scheduling scenarios due to interleaving points in $IP_2 \setminus IP_1$

.
.
.

One Type of Property: Memory Safety



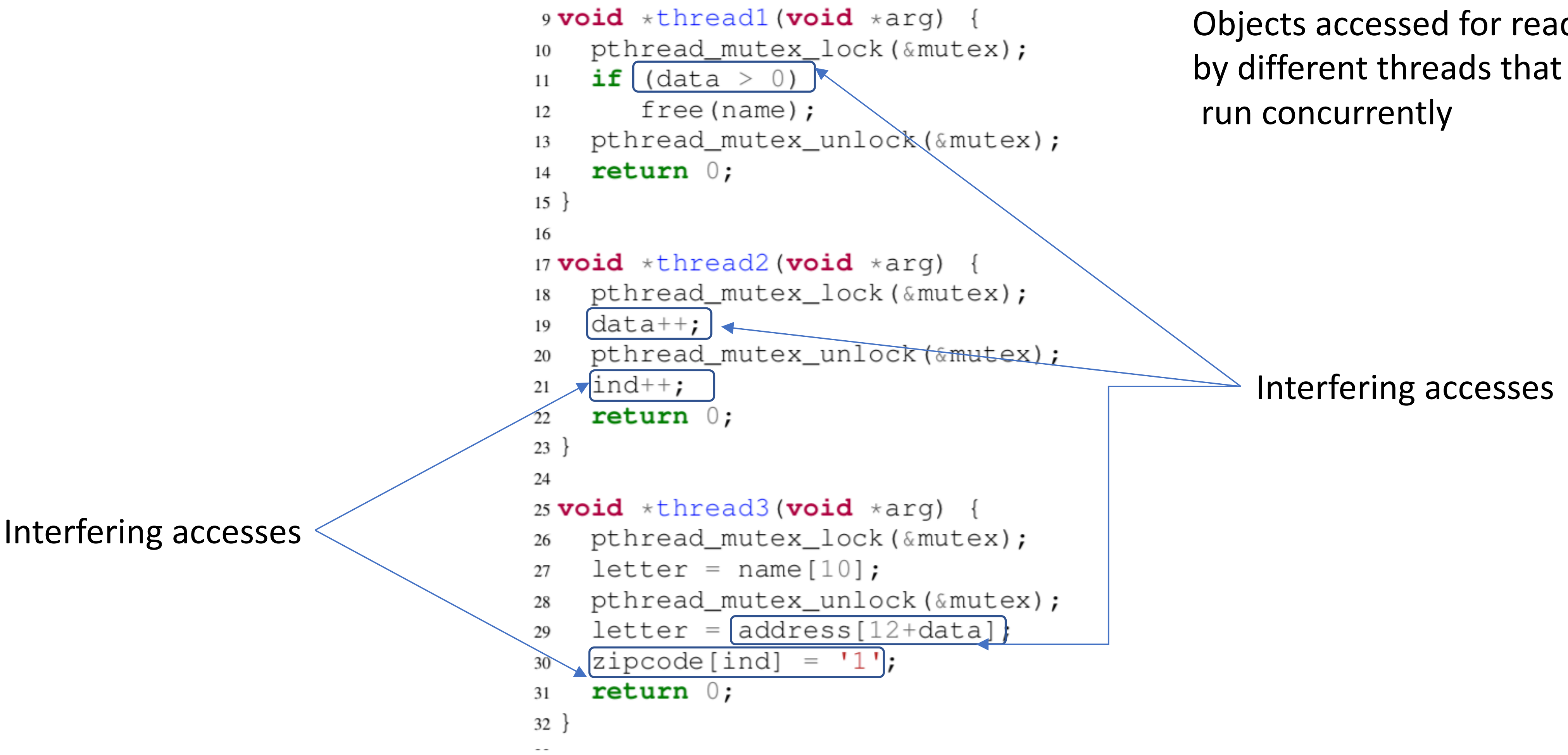
```
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
     free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}
```

Property: Memory Safety

- Are there any accesses to deallocated memory?
- Are there any memory that get deallocated twice?
- Are there any NULL pointer dereferences?
- Are there any out of bounds memory accesses?

Identifying Property Relevant Memory Objects & Instructions from Explored Paths

Objects accessed for read/write and write by different threads that may run concurrently



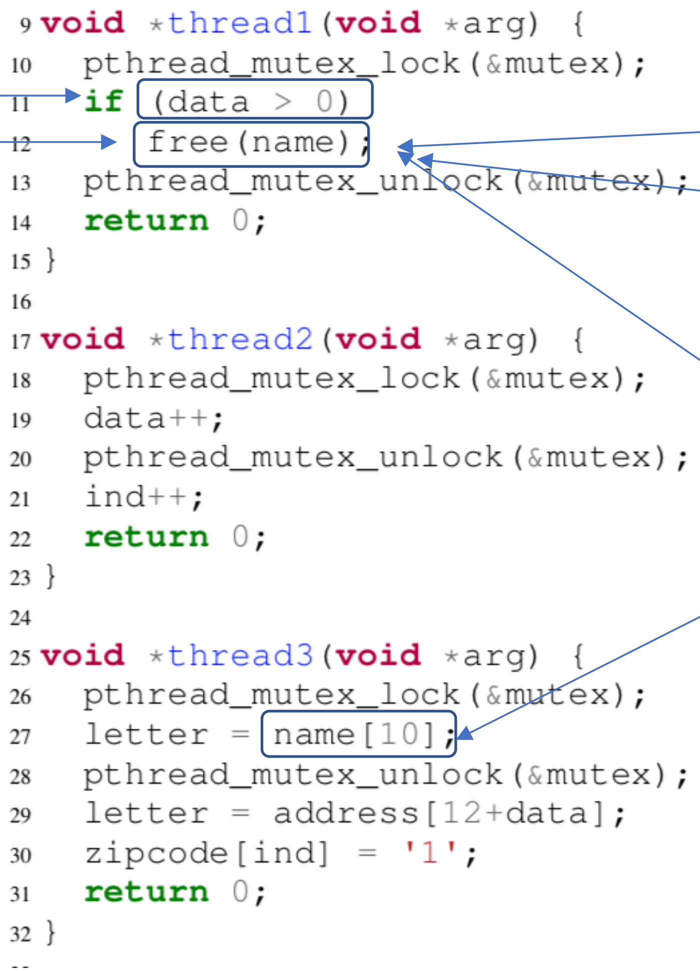
```
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
     free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}
```

Interfering accesses

Interfering accesses

Identifying Property Relevant Memory Objects & Instructions from Explored Paths

- ***Static analysis for identifying target function callsites reachable from untaken branches***
  - makes the branch instruction as property relevant even if data was not global



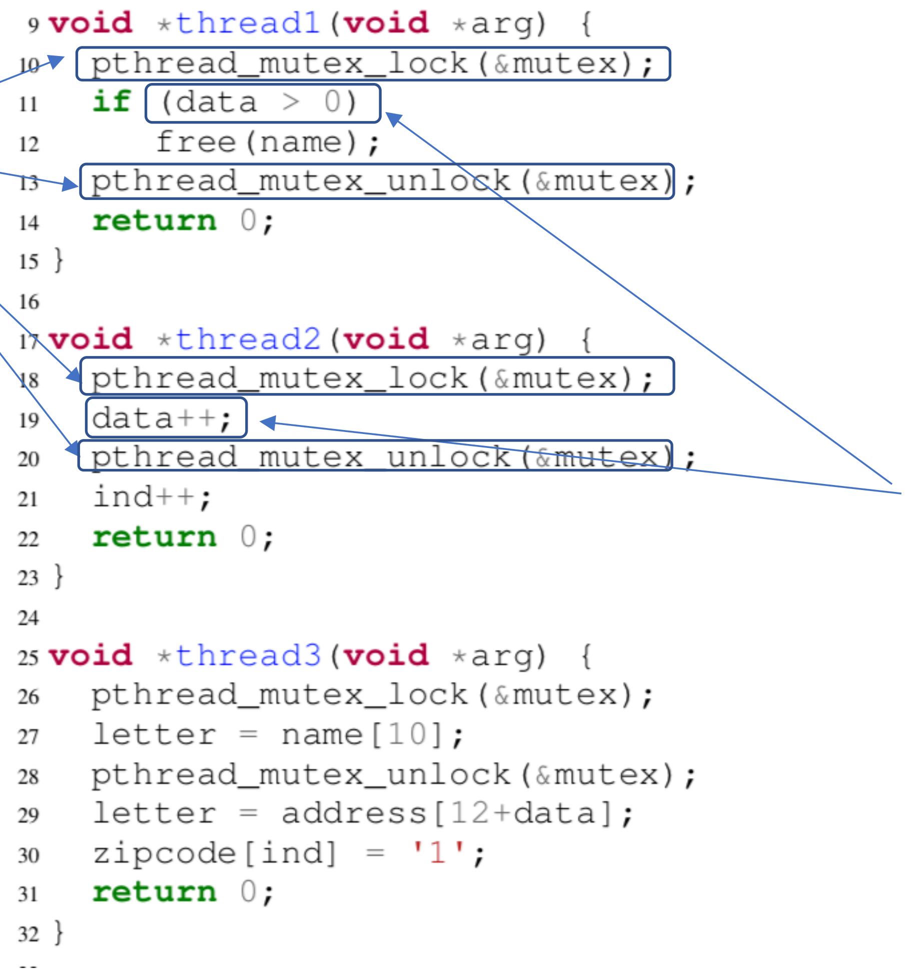
```
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
    free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}
```

Used as an argument (name) of a target function (free) ***(User Input)***

Hidden dependency to be explored later

Identifying Property Relevant Memory Objects & Instructions from Explored Paths

- **Lock acquire & release** instructions that enclose interfering instructions using a common lock

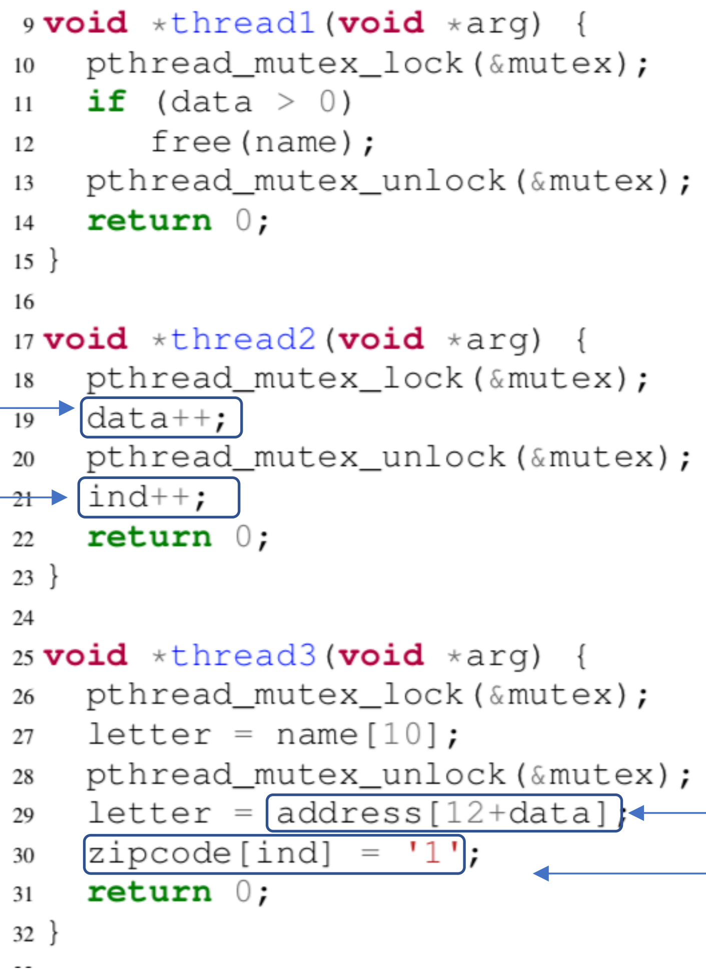


```
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
      free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}
```

Interfering accesses

Identifying Property Relevant Memory Objects & Instructions from Explored Paths

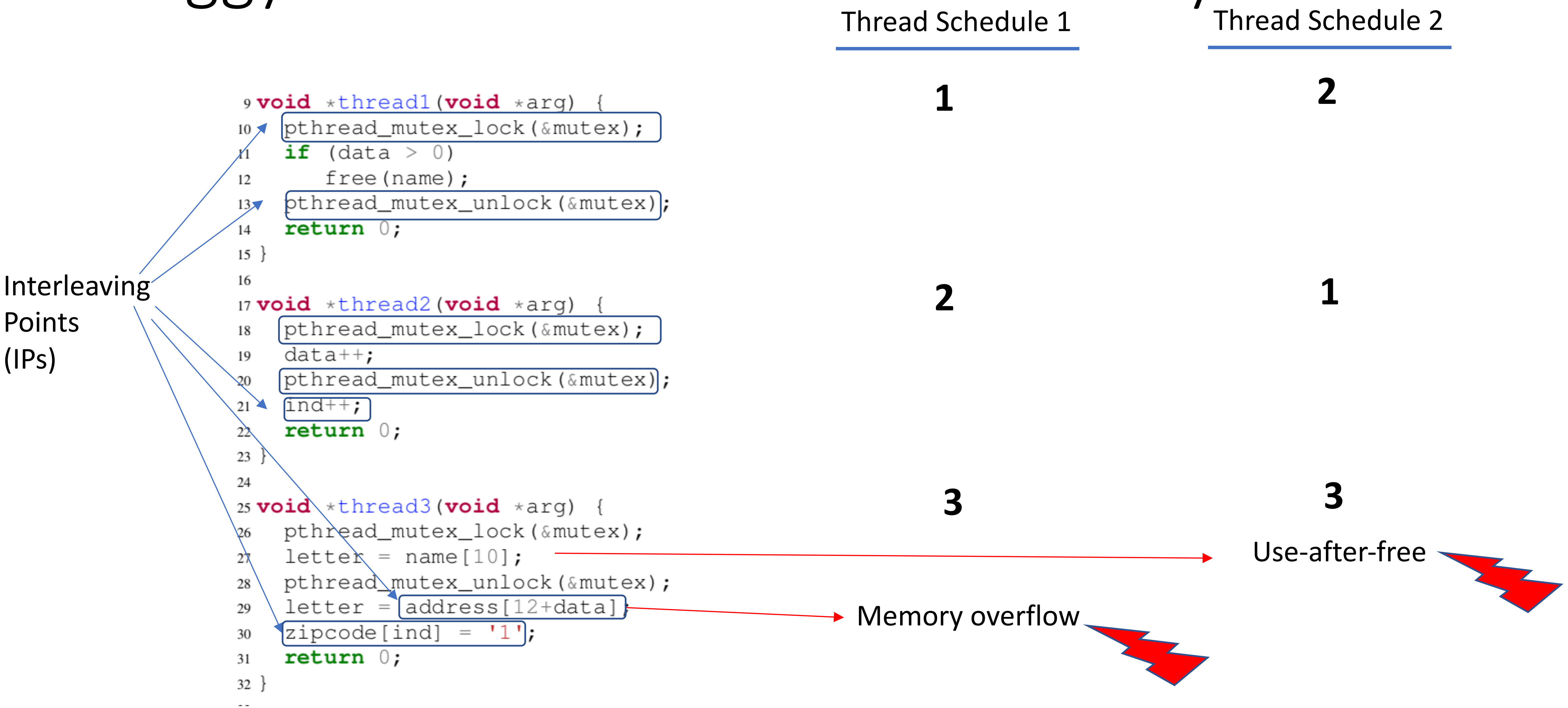
Instructions that define objects (data and ind) accessed in **memory access index expressions** become property relevant



```
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
     free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}
```

Used in memory access index expressions: data and ind

Buggy Thread Schedules Detected by SIFT



Thread Schedule 1	Thread Schedule 2
1	**2**
2	**1**
3	**3**

Interleaving Points (IPs)

```c
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
     free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}
```

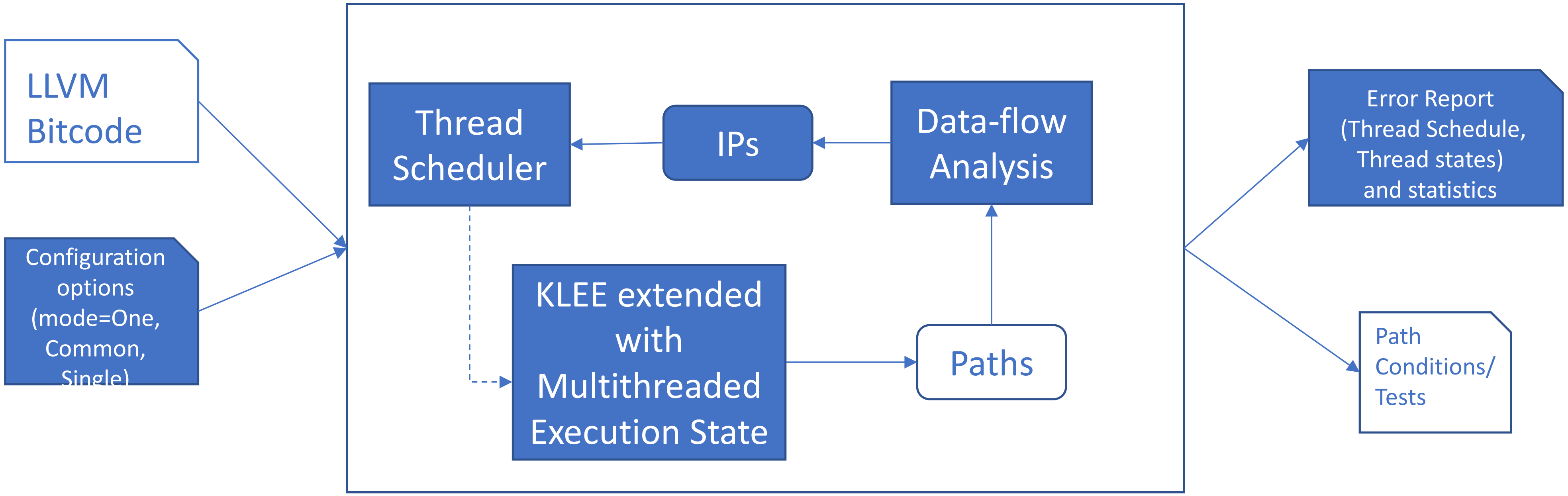
Use-after-free

Memory overflow

Optimizations

- Three modes for grouping the interleaving points (IPs)
  - *One*: Put all in a single set and generate schedules by considering every IP in this set
    - More likely to detect the error
    - May lead to too many thread interleaving scenarios
  - Common: Create partitions by grouping IPs that access common memory objects
    - May detect errors that involve scheduling decisions over a single memory object
    - Fewer scheduling scenarios than the One mode
  - *Single*: Create a separate partition for each IP
    - May detect errors that require a single error relevant context switch
    - May generate the least number of scheduling scenarios

SIFT Implementation


LLVM Bitcode

Configuration options (mode=One, Common, Single)

Thread Scheduler

IPs

Data-flow Analysis

KLEE extended with Multithreaded Execution State

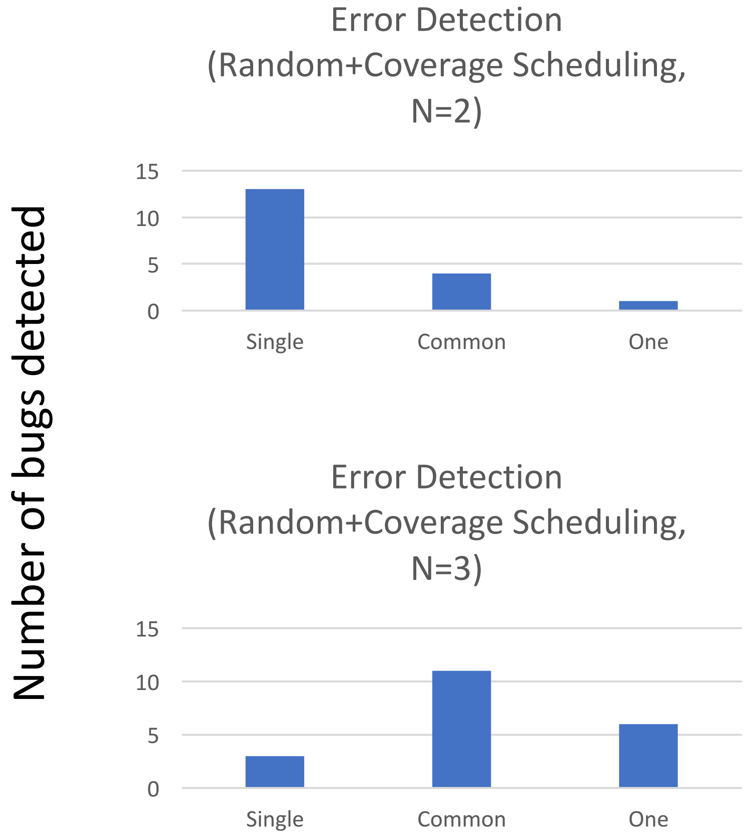
Paths

Error Report (Thread Schedule, Thread states) and statistics

Path Conditions/ Tests

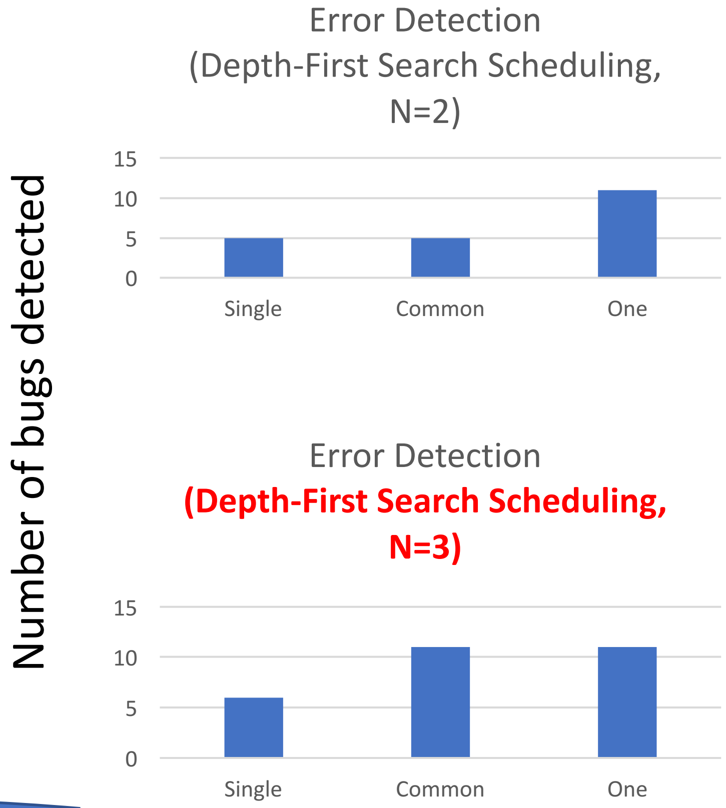
Extensions are shown in blue

Results on 10 CVE + 10 Svcomp benchmarks



Error Detection (Random+Coverage Scheduling, N=2)

Error Detection (Random+Coverage Scheduling, N=3)



Error Detection (Depth-First Search Scheduling, N=2)

Error Detection (Depth-First Search Scheduling, N=3)

Single	Common	One
> 0.07s < 19.96s	>0.07s <52.09s	>0.07s <24.62s

Timeout=500secs

SIFT can detect the bugs in all 10 CVE benchmarks whereas ConVul can detect in only 9 of them.

ConVul paper:
Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, Bin Liang:
Detecting concurrency memory corruption vulnerabilities.
ESEC/SIGSOFT FSE 2019: 706-717

Conclusion

- SIFT performs on-the-fly data-flow analysis to steer the thread schedule towards property violation
  - Memory safety + Custom assertions
  - https://github.com/sysrel/SIFT

- Improving scalability:
  - Integrate SIFT into dynamic analysis
  - Apply SIFT at the component-level similar to under-constrained symbolic execution

THANK YOU!