
# Detecting MPI Usage Anomalies via Partial Program Symbolic Execution

Fangke Ye, Jisheng Zhao, Vivek Sarkar

KLEE Workshop 2021

# Background

- Message Passing Interface (MPI) is a widely-used programming model for distributed-memory parallelism
- MPI programming is error-prone
  - MPI APIs are not expressed as structured program constructs
    - Easy to omit synchronizations for nonblocking communication API calls
  - Pointer aliasing and arithmetic on data buffers for communication for MPI applications written in C/C++
  - General difficulties in parallel programming
- Need tools to help debugging

# MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call

```
MPI_Request req[2];
uint32_t recvbuf[100], sendbuf[100];
 …
 MPI_Irecv(recvbuf, 10, MPI_LONG, x, 101, MPI_COMM_WORLD, &req[0]);
 …
 if (recvbuf[0] == x) {
  …
  MPI_Isend(sendbuf, 10, MPI_LONG, x+2, 103, MPI_COMM_WORLD, &req[1]);
  …

  MPI_Irecv(recvbuf, 10, MPI_LONG, x+1, 102, MPI_COMM_WORLD, &req[0]););
 }
 …
 MPI_Waitall(2, req, …);
```

1. datatype mismatch

2. data race

3. request overwriting, buffer overlap

4. unmatched wait: req[1]

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
if (rank == 0) {
    …
    MPI_Send(sendbuf, 10, MPI_INT, 1, …);
    …
} else { // rank == 1
    …
    MPI_Recv(recvbuf, 10, MPI_INT, 1, …);
    …
}
```

Rank 0 sends to rank 1

5. Unmatched P2P call

Rank 1 receives from rank 1
(should be rank 0 instead)



Georgia Tech
CREATING THE NEXT

# MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs
- Anomalies we target
  1. **Buffer Type Mismatch**
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call

```
MPI_Request req[2];
uint32_t recvbuf[100], sendbuf[100];
 ...
 MPI_Irecv(recvbuf, 10, MPI_LONG, x, 101, MPI_COMM_WORLD, &req[0]);
 ...
 if (recvbuf[0] == x) {
  ...
  MPI_Isend(sendbuf, 10, MPI_LONG, x+2, 103, MPI_COMM_WORLD, &req[1]);
  ...

  MPI_Irecv(recvbuf, 10, MPI_LONG, x+1, 102, MPI_COMM_WORLD, &req[0]););
 }
 ...
 MPI_Waitall(2, req, ...);
```

1. datatype mismatch

2. data race

3. request overwriting, buffer overlap

4. unmatched wait: req[1]

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
if (rank == 0) {
    …
    MPI_Send(sendbuf, 10, MPI_INT, 1, …);
    …
} else { // rank == 1
    …
    MPI_Recv(recvbuf, 10, MPI_INT, 1, …);
    …
}
```

Rank 0 sends to rank 1

5. Unmatched P2P call

Rank 1 receives from rank 1
(should be rank 0 instead)

# MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs
- Anomalies we target
  1. Buffer Type Mismatch
  2. **Buffer Data Race**
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call

```
MPI_Request req[2];
uint32_t recvbuf[100], sendbuf[100];
 …
MPI_Irecv(recvbuf, 10, MPI_LONG, x, 101, MPI_COMM_WORLD, &req[0]);
 …
if (recvbuf[0] == x) {
  …
  MPI_Isend(sendbuf, 10, MPI_LONG, x+2, 103, MPI_COMM_WORLD, &req[1]);
  …

  MPI_Irecv(recvbuf, 10, MPI_LONG, x+1, 102, MPI_COMM_WORLD, &req[0]););
}
…
MPI_Waitall(2, req, …);
```

1. datatype mismatch

2. data race

3. request overwriting, buffer overlap

4. unmatched wait: req[1]

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
if (rank == 0) {
    …
    MPI_Send(sendbuf, 10, MPI_INT, 1, …);
    …
} else { // rank == 1
    …
    MPI_Recv(recvbuf, 10, MPI_INT, 1, …);
    …
}
```

Rank 0 sends to rank 1

5. Unmatched P2P call

Rank 1 receives from rank 1
(should be rank 0 instead)

# MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. **Request Overwriting**
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call

```
MPI_Request req[2];
uint32_t recvbuf[100], sendbuf[100];
 ...
MPI_Irecv(recvbuf, 10, MPI_LONG, x, 101, MPI_COMM_WORLD, &req[0]);
 ...
if (recvbuf[0] == x) {
  ...
  MPI_Isend(sendbuf, 10, MPI_LONG, x+2, 103, MPI_COMM_WORLD, &req[1]);
  ...
  MPI_Irecv(recvbuf, 10, MPI_LONG, x+1, 102, MPI_COMM_WORLD, &req[0]);
}
...
MPI_Waitall(2, req, ...);
```

1. datatype mismatch

2. data race

3. request overwriting, buffer overlap

4. unmatched wait: req[1]

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
if (rank == 0) {
    …
    MPI_Send(sendbuf, 10, MPI_INT, 1, …);
    …
} else { // rank == 1
    …
    MPI_Recv(recvbuf, 10, MPI_INT, 1, …);
    …
}
```

Rank 0 sends to rank 1

5. Unmatched P2P call

Rank 1 receives from rank 1
(should be rank 0 instead)

# MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs
- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. **Unmatched Wait or Test**
  5. Unmatched Point-to-Point (P2P) Call

```
MPI_Request req[2];
uint32_t recvbuf[100], sendbuf[100];
 …
 MPI_Irecv(recvbuf, 10, MPI_LONG, x, 101, MPI_COMM_WORLD, &req[0]);
 …
 if (recvbuf[0] == x) {
  …
  MPI_Isend(sendbuf, 10, MPI_LONG, x+2, 103, MPI_COMM_WORLD, &req[1]);
  …
  MPI_Irecv(recvbuf, 10, MPI_LONG, x+1, 102, MPI_COMM_WORLD, &req[0]););
 }
 …
 MPI_Waitall(2, req, …);
```

1. datatype mismatch

2. data race

3. request overwriting, buffer overlap

4. unmatched wait: req[1]

*Branch not taken: req[1] not used*

*Wait for req[0] and req[1]*

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
if (rank == 0) {
    …
    MPI_Send(sendbuf, 10, MPI_INT, 1, …);
    …
} else { // rank == 1
    …
    MPI_Recv(recvbuf, 10, MPI_INT, 1, …);
    …
}
```

Rank 0 sends to rank 1

5. Unmatched P2P call

Rank 1 receives from rank 1 (should be rank 0 instead)

# MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs
- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. **Unmatched Point-to-Point (P2P) Call**

```
MPI_Request req[2];
uint32_t recvbuf[100], sendbuf[100];
 …
MPI_Irecv(recvbuf, 10, MPI_LONG, x, 101, MPI_COMM_WORLD, &req[0]);
 …
if (recvbuf[0] == x) {
  …
  MPI_Isend(sendbuf, 10, MPI_LONG, x+2, 103, MPI_COMM_WORLD, &req[1]);
  …
  MPI_Irecv(recvbuf, 10, MPI_LONG, x+1, 102, MPI_COMM_WORLD, &req[0]););
}
…
MPI_Waitall(2, req, …);
```

1. datatype mismatch

2. data race

3. request overwriting, buffer overlap

4. unmatched wait: req[1]

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
if (rank == 0) {
    …
    MPI_Send(sendbuf, 10, MPI_INT, 1, …);
    …
} else { // rank == 1
    …
    MPI_Recv(recvbuf, 10, MPI_INT, 1, …);
    …
}
```

Rank 0 sends to rank 1

5. Unmatched P2P call

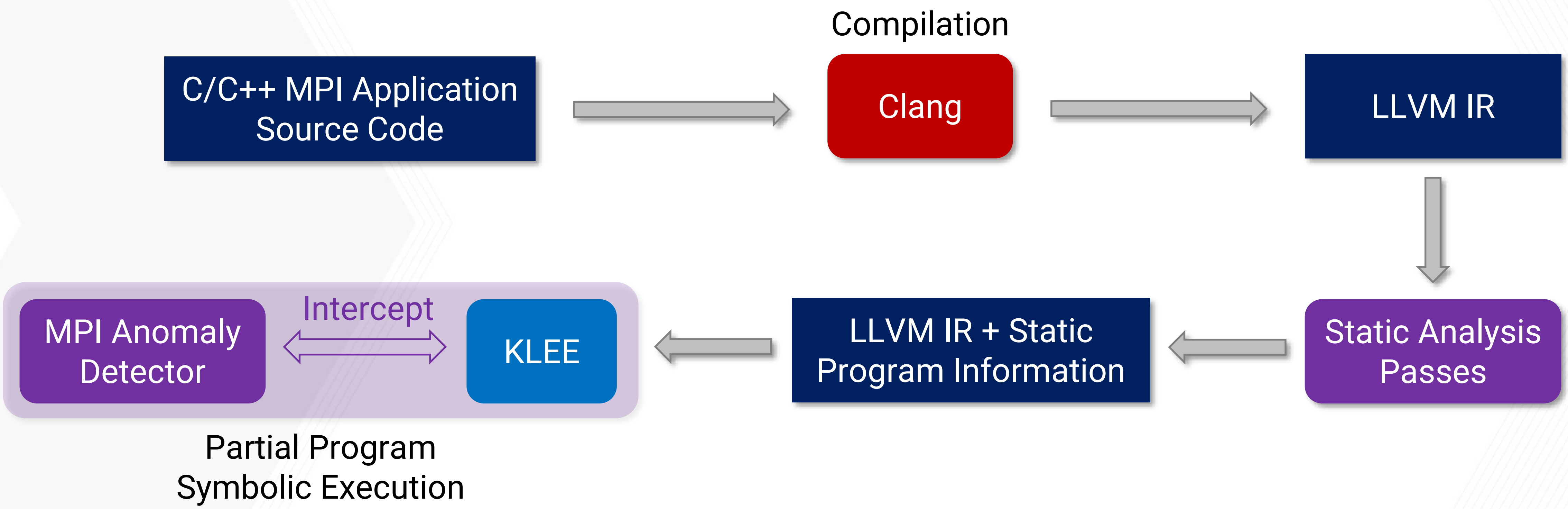
Rank 1 receives from rank 1
(should be rank 0 instead)

# Symbolic Execution for MPI Debugging

- Advantages
  - Precise reasoning of pointers in C/C++
  - Potential better coverage than debugging methods relying on a fixed set of concrete input
  - Better time and space efficiency than dynamically debugging for large-scale parallel programs

- Challenges
  - Modeling MPI API behaviors
  - Scalability

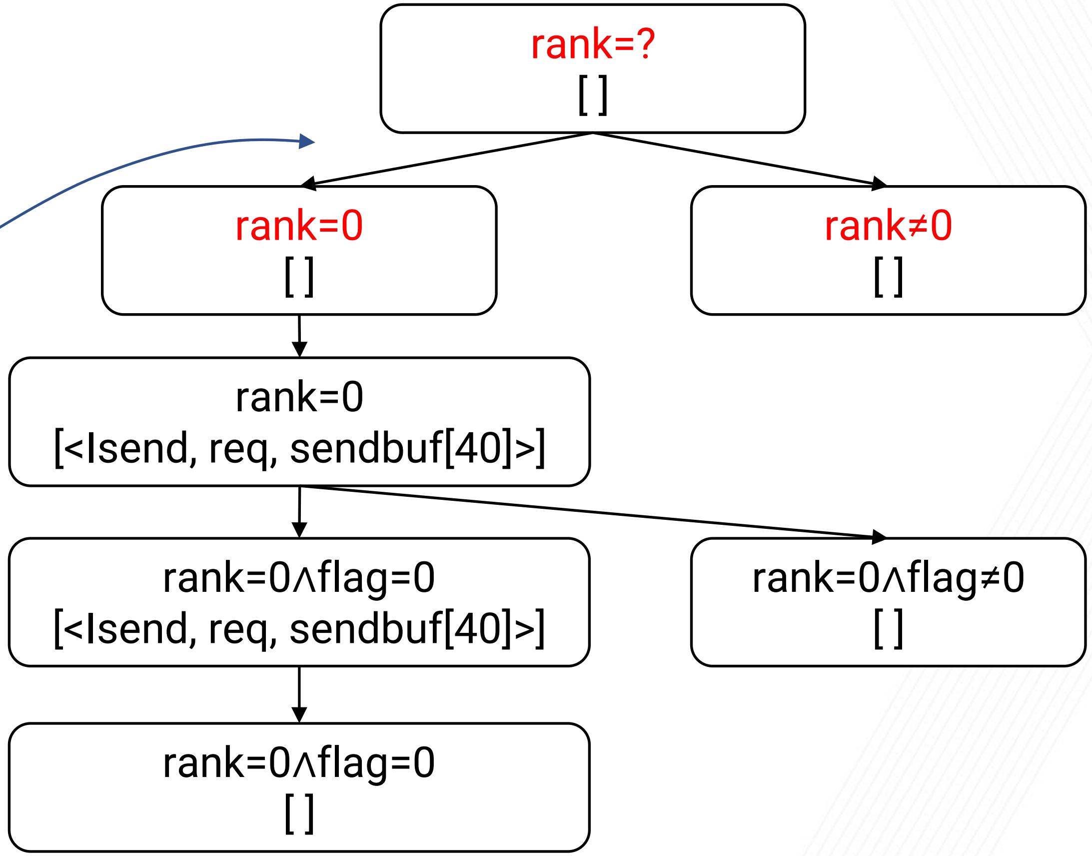
# Our Approach



# MPI Modeling – Ranks (Process Identifiers)

- Use a symbolic rank
- Fork the execution when rank is tested
  - Add a constraint on rank to the path condition
- Fill receive buffers with unconstrained symbolic values

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Isend(sendbuf, 10, MPI_INT, 1, 0,
              MPI_COMM_WORLD, &req);
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
    if (!flag)
        MPI_Wait(&req, MPI_STATUS_IGNORE);
}
```



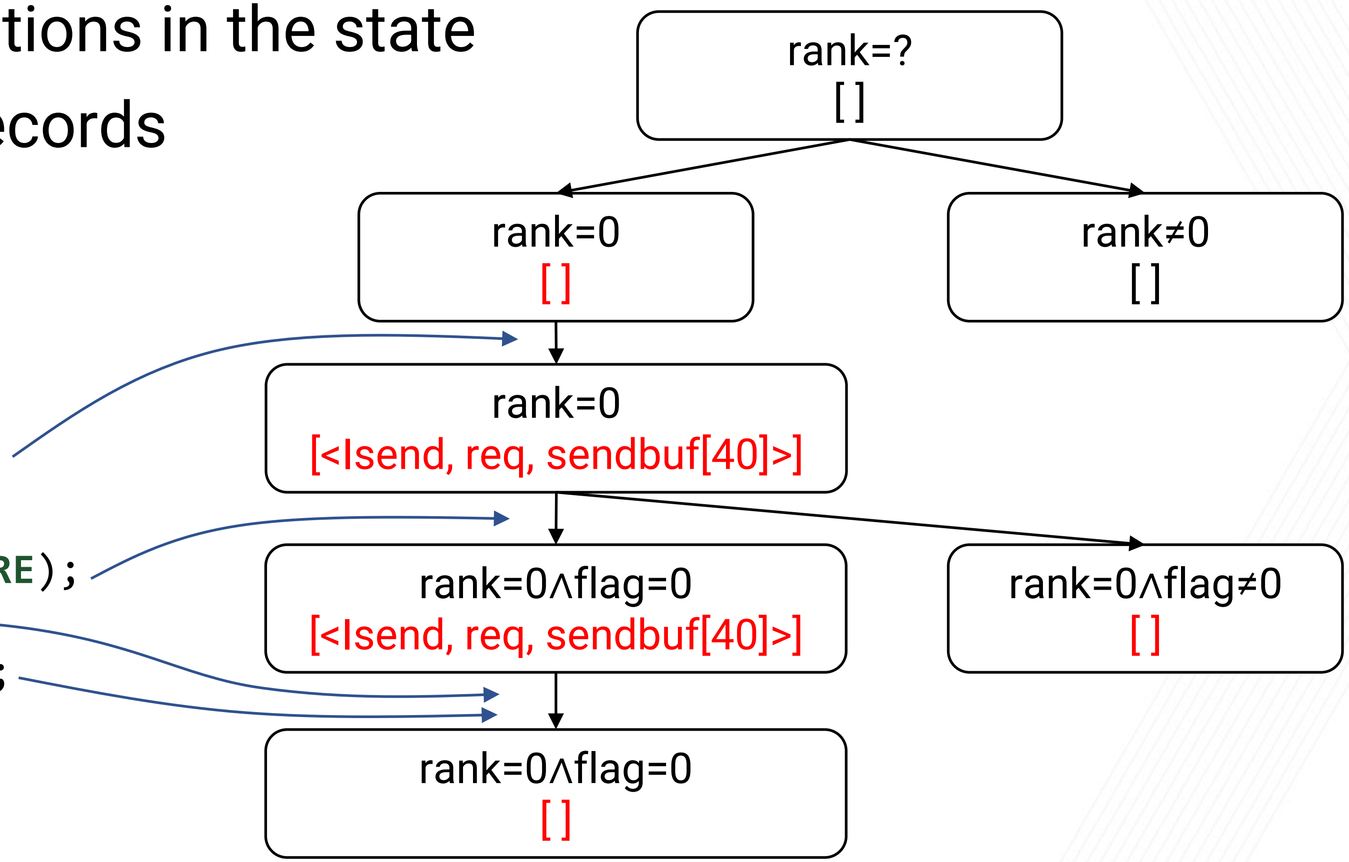



Georgia Tech
CREATING THE NEXT

# MPI Modeling – Nonblocking Operations

- Record ongoing nonblocking operations in the state
- Intercept MPI calls to update the records

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Isend(sendbuf, 10, MPI_INT, 1, 0,
              MPI_COMM_WORLD, &req);
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
    if (!flag)
        MPI_Wait(&req, MPI_STATUS_IGNORE);
}
```






Georgia Tech CREATING THE NEXT

# Anomaly Detection

- For anomalies related to nonblocking operations
  - Including Buffer Data Race, Request Overwriting, Unmatched Wait or Test
  - Use nonblocking operation records stored in the execution state
    - Check if the anomaly condition is satisfiable given the path condition

```
MPI_Irecv(recvbuf, 10, MPI_INT, x,
          101, MPI_COMM_WORLD, &req[0]);
recvbuf[i] = 0;
```

Path condition: $pc$
[<Irecv, &req[0], recvbuf[40]>]

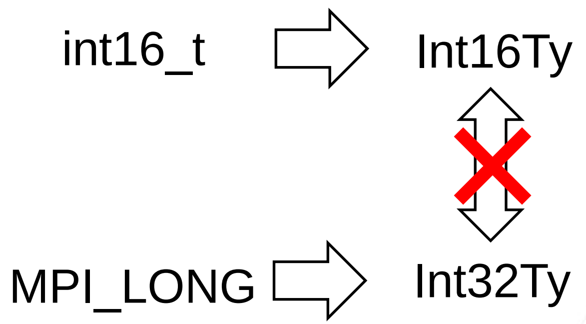
If $(pc \wedge recvbuf \leq \&recvbuf[i] < (char *)recvbuf + 40)$
is satisfiable

❌

# Anomaly Detection

- For other anomalies
  - Including Buffer Type Mismatch, Unmatched P2P Call
  - Use pre-computed static information to detect them
    - A map from MPI_Datatype to LLVM types
    - All P2P calls that are control dependent on each rank-related branch

```
uint16_t recvbuf[100];
MPI_Irecv(recvbuf, 10, MPI_LONG, x,
          101, MPI_COMM_WORLD, &req[0]);
```



# Improving the Scalability of Symbolic Execution

- Observation
  - Usually, a large portion of code in an MPI application is not relevant to communication
- Set limitations on execution
  - Max number of iterations per loop
  - Max fork depth
- Partial program symbolic execution
  - Start new executions at user-specified locations
  - Our implementation: select any function as the entrance of an execution

# Memory State Initialization

- The memory state is unknown before entering the entry function
- Lazy initialization [1, 2]
  - Allocate memory at dereferences
  - Fork the execution for multiple possible memory states
    - Eliminate some impossible states using pre-computed whole-program alias analysis results

Entry Function

```
int a[10];

void f(int *p, int i) {
  p[i] = 0;
}

void g() {
  int b[10];
  f(b, 0);
}

void h() {
  f(a, 0);
}
```



[1] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. *Generalized Symbolic Execution for Model Checking and Testing.* TACAS ’03.
[2] David A. Ramos and Dawson Engler. *Under-Constrained Symbolic Execution: Correctness Checking for Real Code*. USENIX Security ’15.

# Tracking Initialization States for Symbolic Pointers with Shadow Memory

- Not every pointer dereference needs lazy initialization
  - ***p** should not trigger lazy initialization

- Use shadow memory to track whether a symbolic pointer value has already been initialized

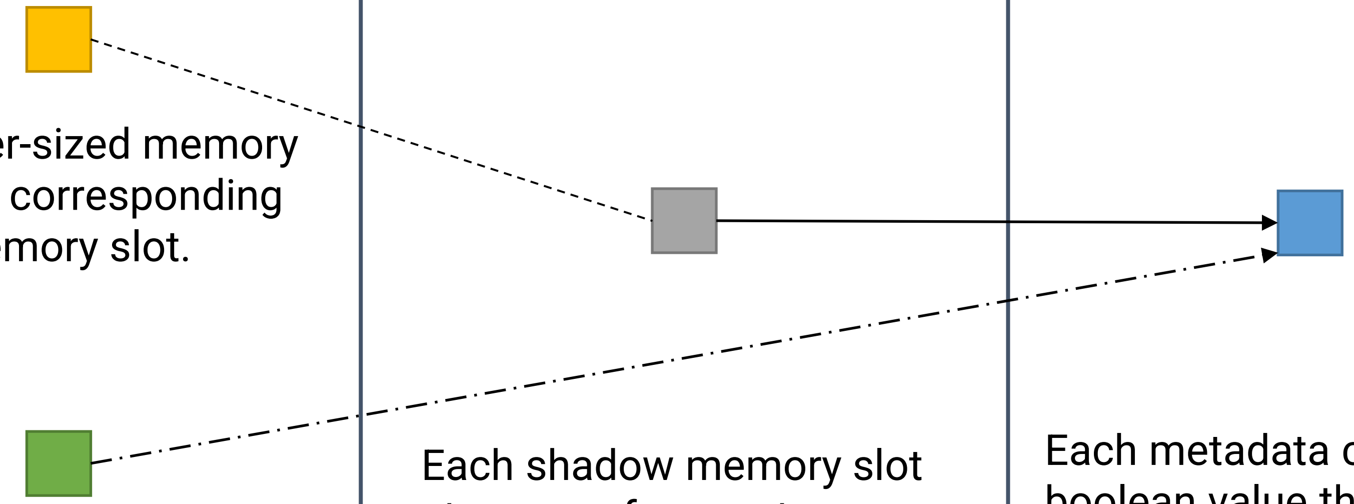
```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```

# Lazy Initialization with Shadow Memory

**Code**

```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```

**Application Memory & Local Variables**



Each pointer-sized memory block has a corresponding shadow memory slot.

Each local variable has a reference to a metadata cell.

**Shadow Memory**

Each shadow memory slot stores a reference to a metadata cell.

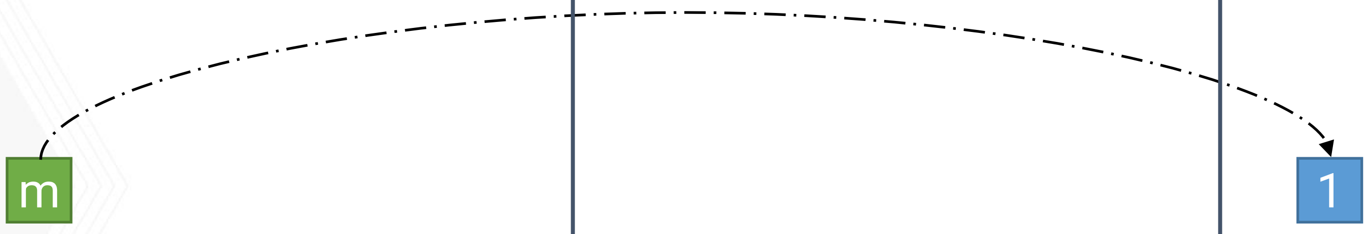
**Metadata**

Each metadata cell stores a boolean value that indicates whether lazy allocation is needed.

# Lazy Initialization with Shadow Memory

**Code** | **Application Memory & Local Variables** | **Shadow Memory** | **Metadata**

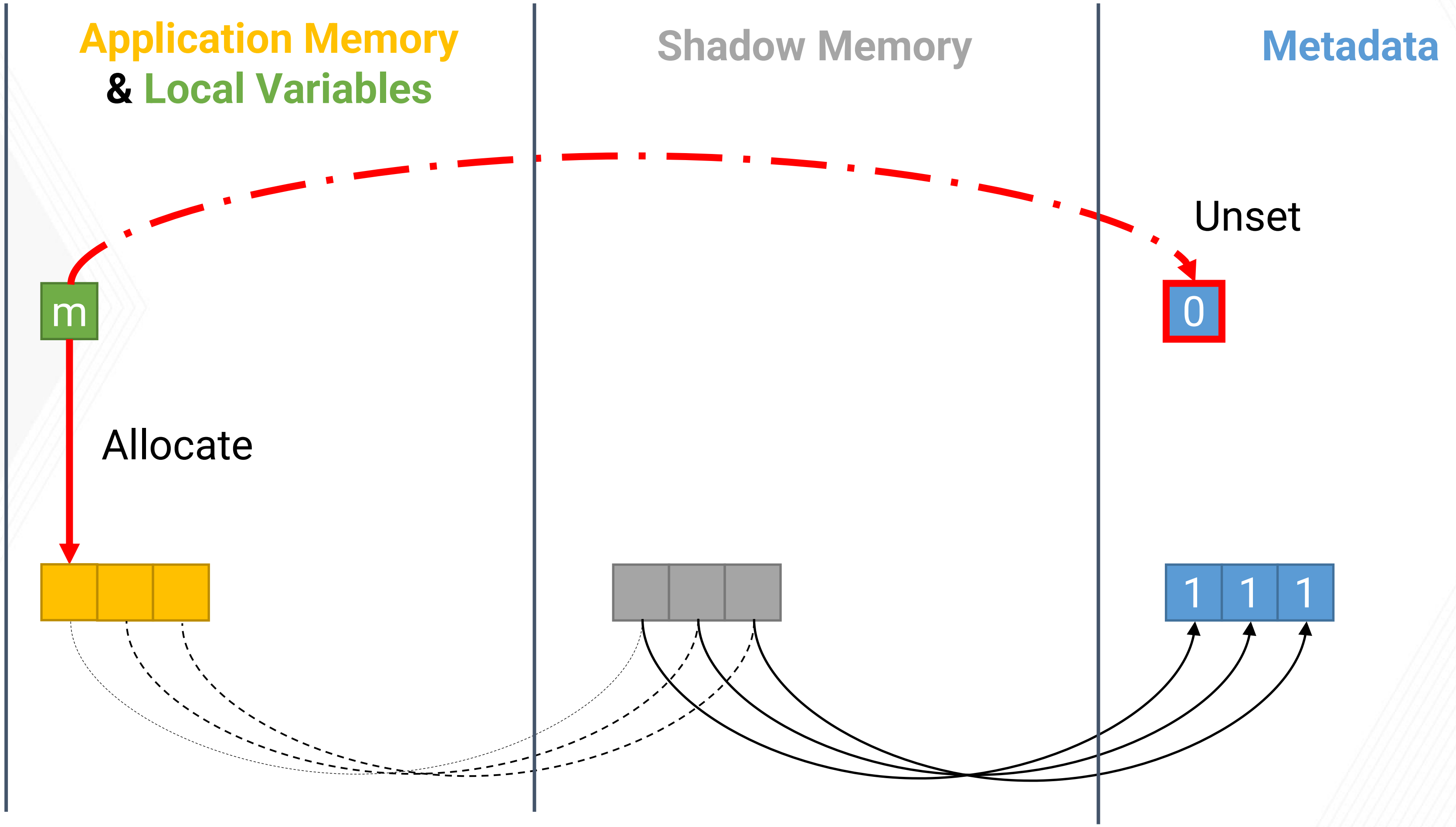
```c
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```



# Lazy Initialization with Shadow Memory

**Code**

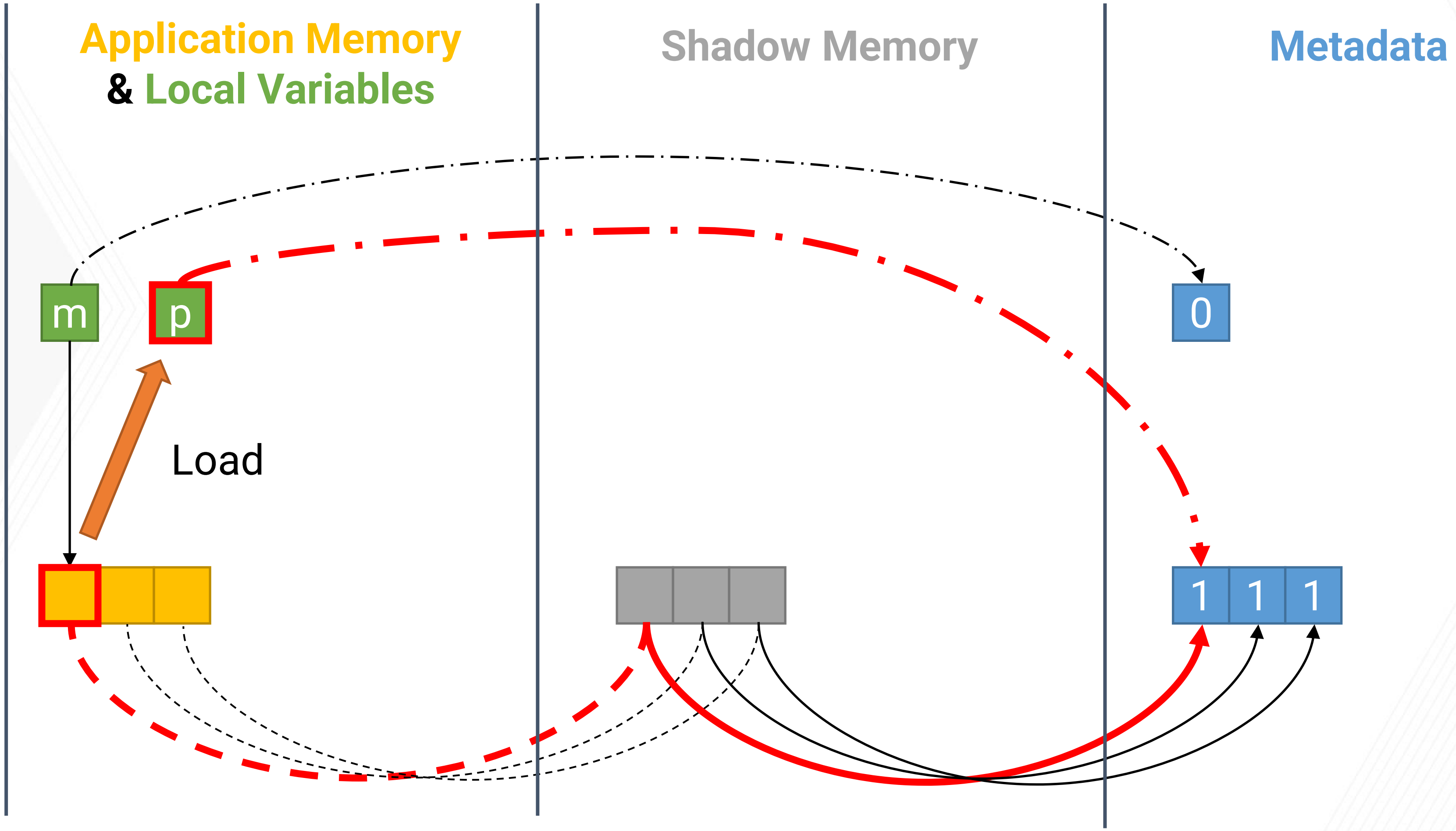
```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```



# Lazy Initialization with Shadow Memory

**Code**

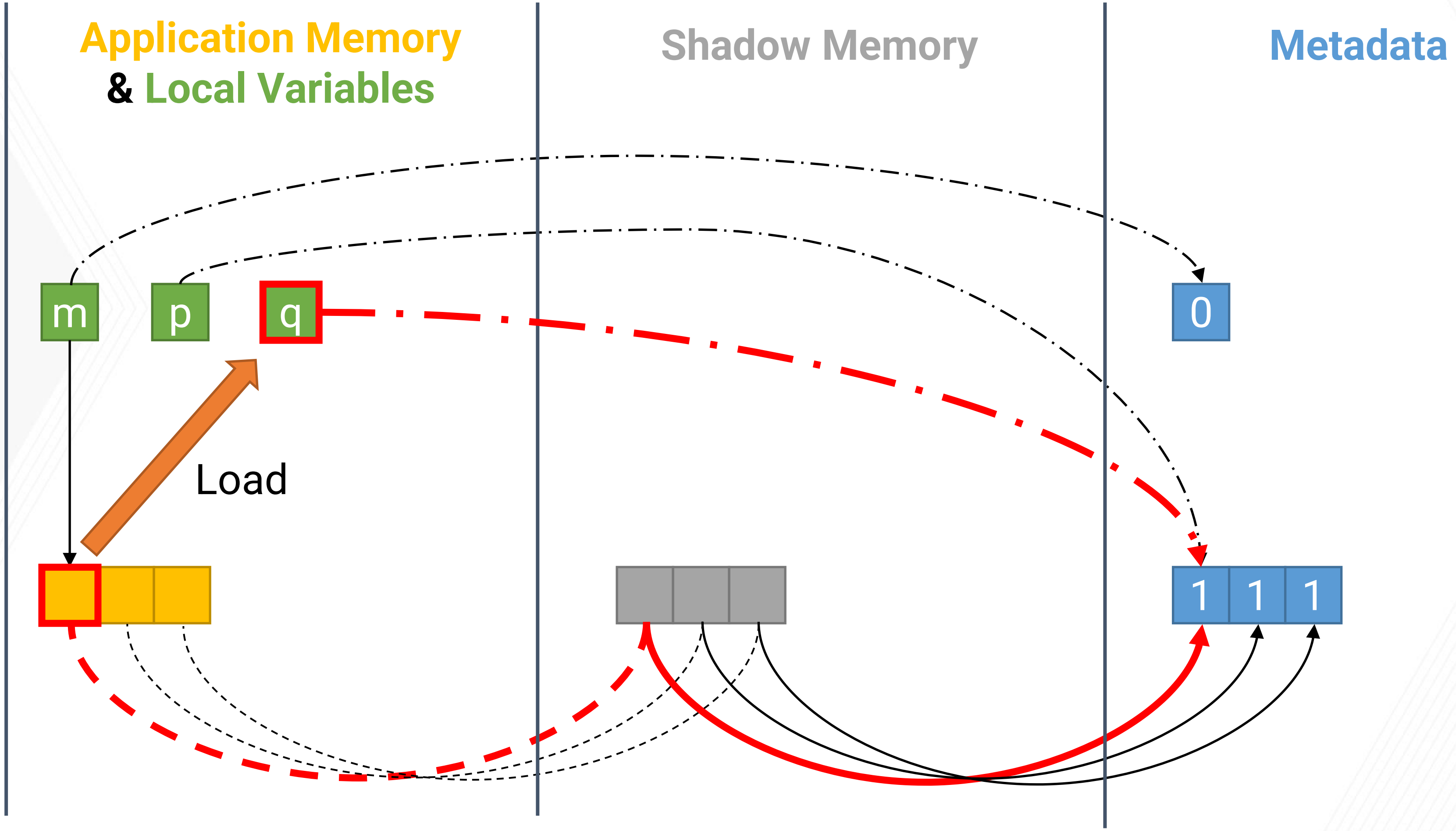
```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```



# Lazy Initialization with Shadow Memory

**Code**

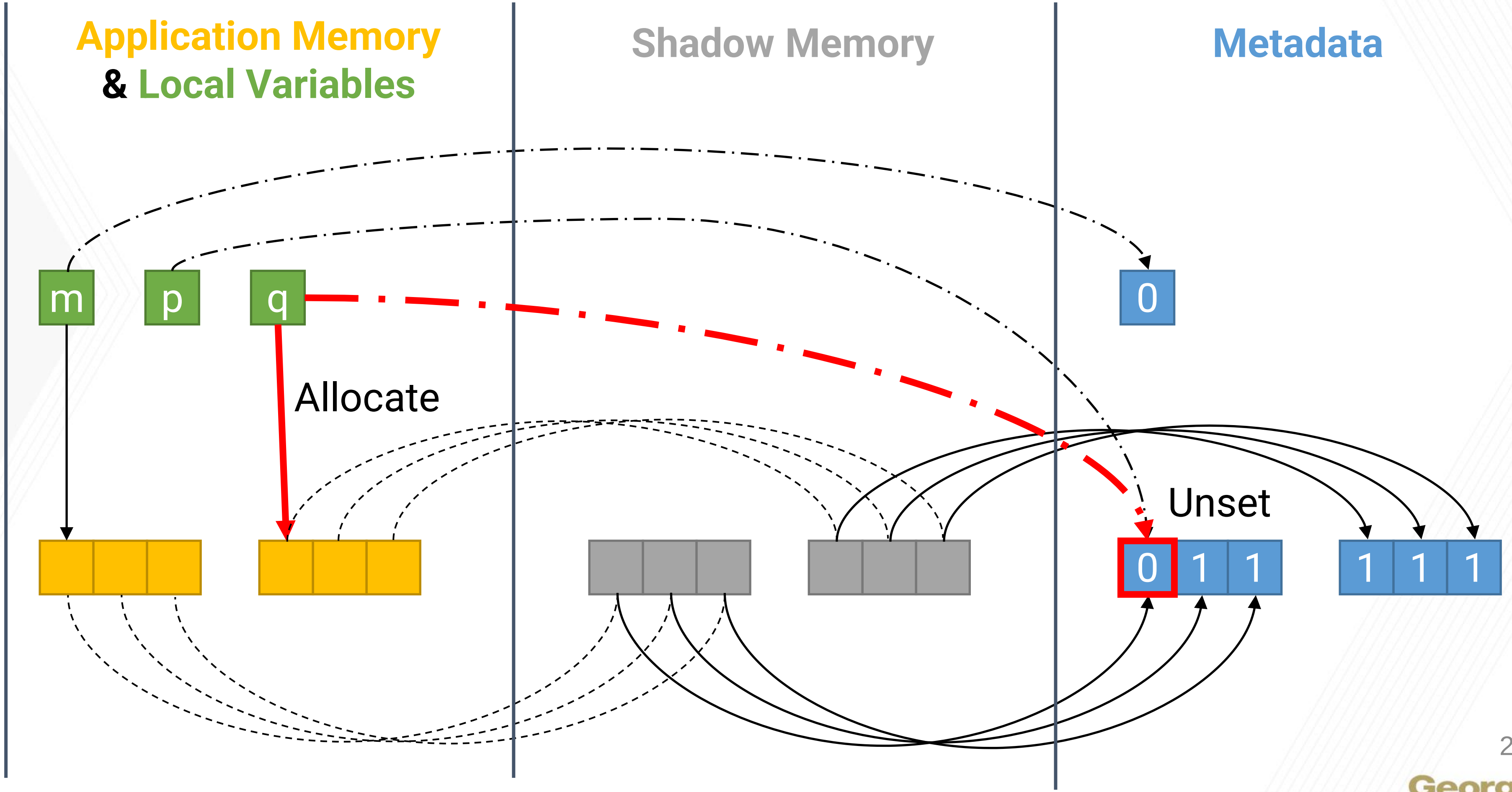
```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```



# Lazy Initialization with Shadow Memory

**Code**

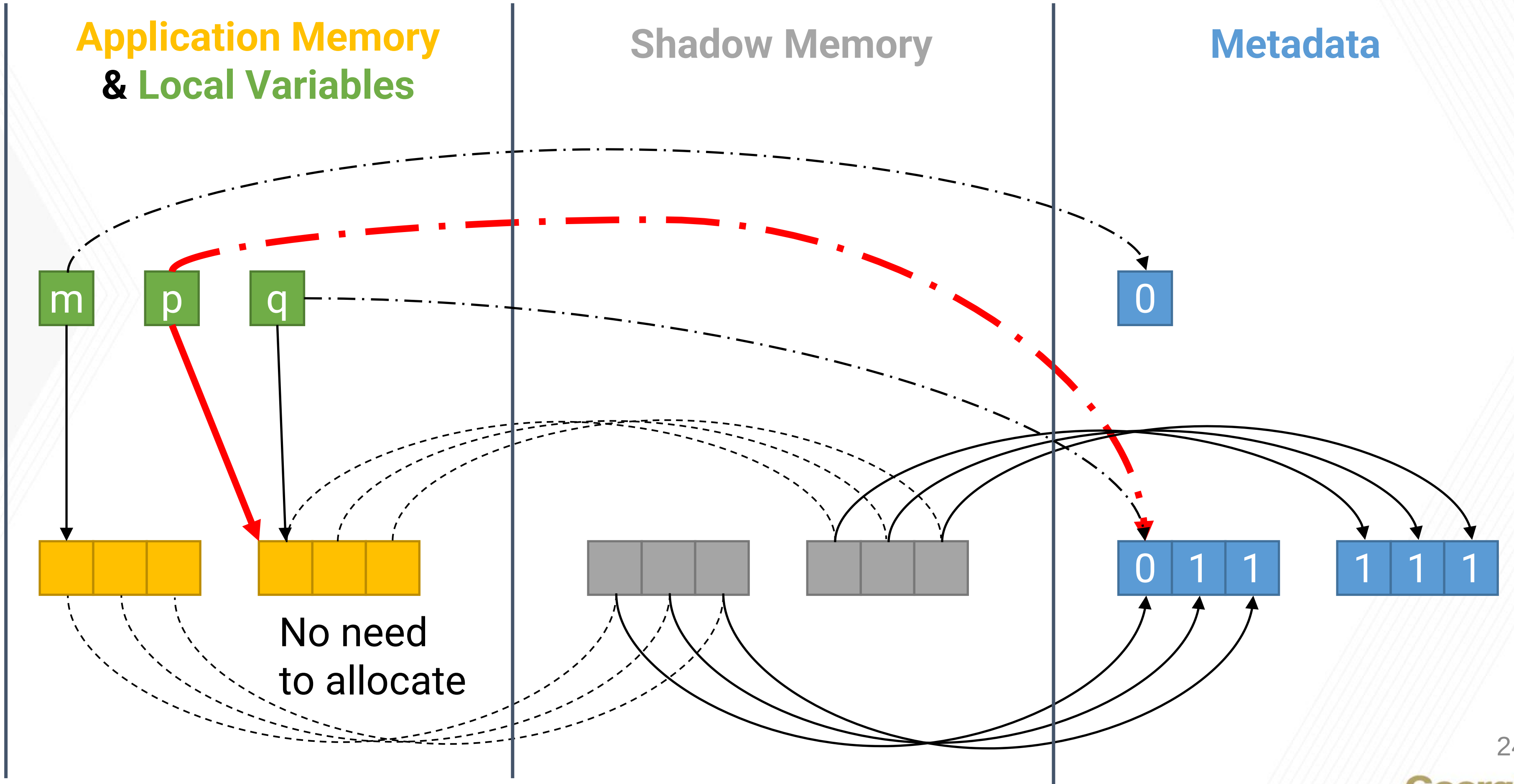
```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```



# Lazy Initialization with Shadow Memory

**Code**

```
int f(int **m) {
  int *p = m[0];
  int *q = m[0];
  *q = 0;
  return *p;
}
```



# Evaluation

- Benchmarks
  - 2 real-world applications: AMG2013, Athena
  - 2 Benchmarks from the NAS Parallel Benchmark Suite: NPB.IS, NPB.DT
  - A library implemented with MPI: OpenFFT
  - A benchmark application used in previous work: Sort [1]
- Comparison
  - Our approach: partial program symbolic execution (PSE)
  - Static analysis tool: MPI-Checker [2]
  - Dynamic analysis tool: MUST [3]

| Benchmark | Lines of Code |
|---|---|
| AMG2013 | 74,901 |
| Athena | 63,012 |
| NPB.IS | 6,498 |
| NPB.DT | 711 |
| OpenFFT | 892 |
| Sort | 127 |

[1] Zhezhe Chen, Xinyu Li, Jau-Yuan Chen, Hua Zhong, and Feng Qin. *SyncChecker: Detecting Synchronization Errors Between MPI Applications and Libraries*. IPDPS '12.
[2] Alexander Droste, Michael Kuhn, and Thomas Ludwig. *MPI-Checker: Static Analysis for MPI*. LLVM '15.
[3] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. *MPI Runtime Error Detection with MUST: Advances in Deadlock Detection*. SC '12.

# Evaluation – Effectiveness

| Benchmark | Anomaly Type | Number of Anomalies | Number of Anomalies Reported | | |
|---|---|---|---|---|---|
| | | | PSE | MPI-Checker | MUST |
| AMG2013 | Resuest Overwriting | 0 | 0 | 2 (FP) | 0 |
| | Unmatched Wait | 0 | 0 | 1 (FP) | 0 |
| Athena | Buffer Data Race | 4 | 4 (TP) | N/A | N/A |
| | Request Overwriting | 8 | 8 (TP) | 0 | 8 (TP) |
| Sort | Buffer Data Race | 1 | 1 (TP) | N/A | 1 (TP) |

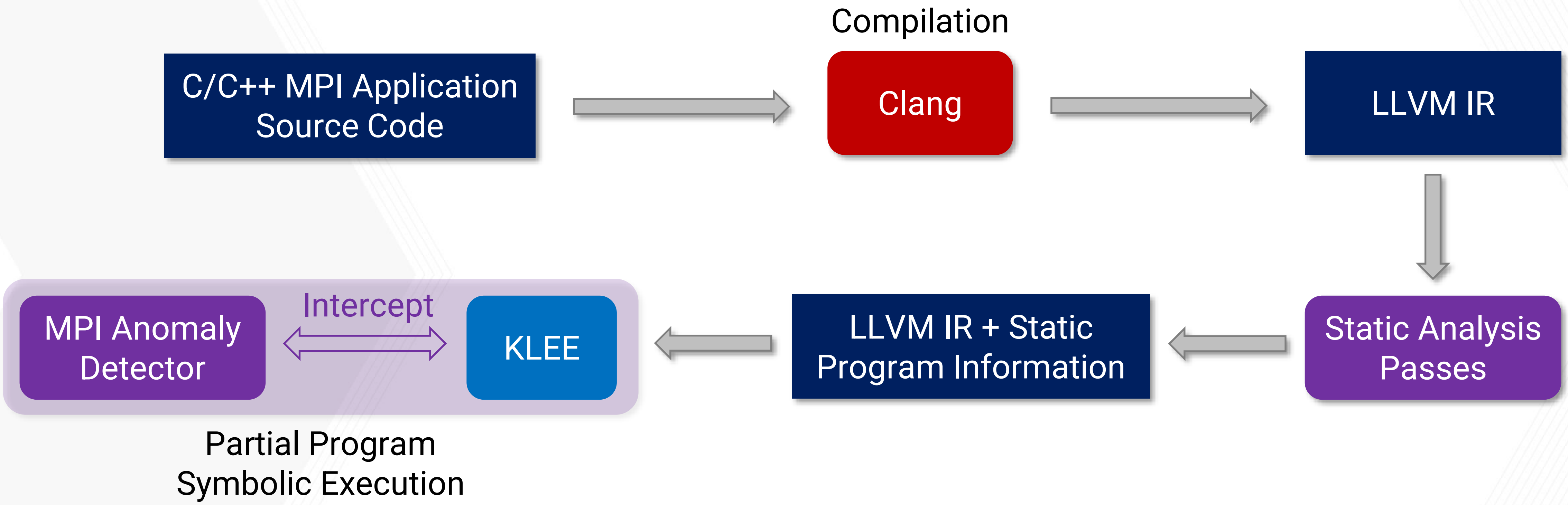
TP = True Positive
FP = False Positive
N/A: Not Supported

# Evaluation – Performance

| Benchmark | Time (s) | | | Memory Usage (MB) | | |
|---|---|---|---|---|---|---|
| | PSE | MPI-Checker | MUST | PSE | MPI-Checker | MUST |
| AMG2013 | 28.57* | 96.70 | 8.54 | 220 | 418 | 677 |
| Athena | 1960.14* | 27.13 | 119.19 | 277 | 333 | 700 |
| OpenFFT | 75.21* | 4.76 | N/A | 74 | 312 | N/A |
| NPB.IS | 206.21** | 1.00 | 9.15 | 1,926 | 85 | 654 |
| NPB.DT | 9.54** | 1.60 | 8.49 | 1,316 | 128 | 714 |
| Sort | 0.11** | 0.39 | 6.38 | 27 | 65 | 682 |

* Starting from a non-main function.
** Starting from the main function with concrete input and symbolic ranks.

# Conclusion




Source code available at

https://github.com/fkye/PSE-MPI