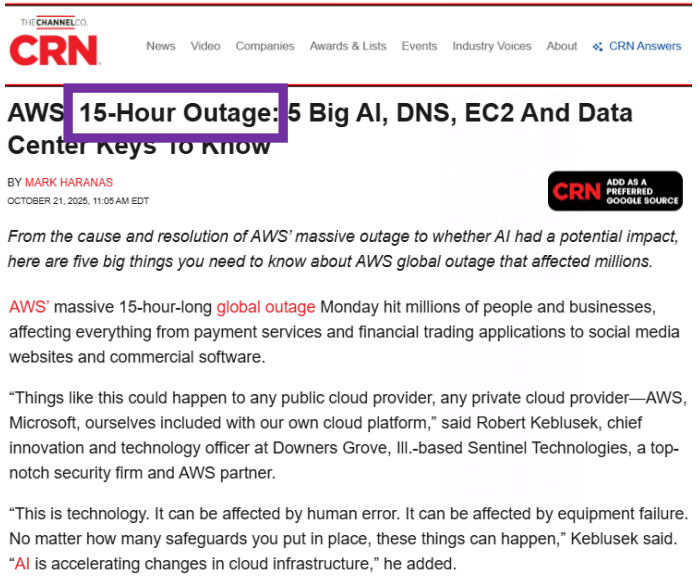
Themis: Detecting Distributed Concurrency Bugs through RPC-Driven Race-Directed Test Generation and Fuzzing

Hongchen Cao[1], Jingzhu He[1], Ting Dai[2], Guoliang Jin[3]

[1]ShanghaiTech University, [2]InsightFinder AI, [3]North Carolina State University

A Real-World Outage in 2025

Downtime



THE CHANNEL CO.
CRN

News Video Companies Awards & Lists Events Industry Voices About CRN Answers

AWS 15-Hour Outage: 5 Big AI, DNS, EC2 And Data Center Keys To Know

BY MARK HARANAS
OCTOBER 21, 2025, 11:05 AM EDT

CRN ADD AS A PREFERRED GOOGLE SOURCE

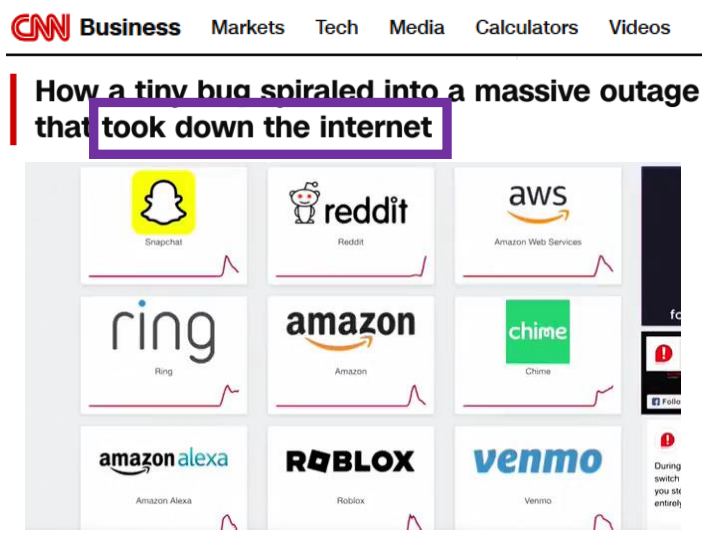
From the cause and resolution of AWS' massive outage to whether AI had a potential impact, here are five big things you need to know about AWS global outage that affected millions.

AWS' massive 15-hour-long global outage Monday hit millions of people and businesses, affecting everything from payment services and financial trading applications to social media websites and commercial software.

"Things like this could happen to any public cloud provider, any private cloud provider—AWS, Microsoft, ourselves included with our own cloud platform," said Robert Keblusek, chief innovation and technology officer at Downers Grove, Ill.-based Sentinel Technologies, a top-notch security firm and AWS partner.

"This is technology. It can be affected by human error. It can be affected by equipment failure. No matter how many safeguards you put in place, these things can happen," Keblusek said. "AI is accelerating changes in cloud infrastructure," he added.

Affected Services

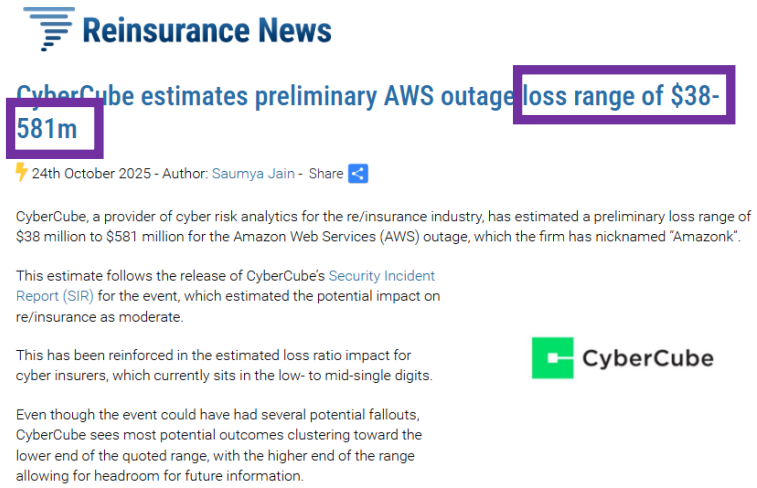


CNN Business Markets Tech Media Calculators Videos

How a tiny bug spiraled into a massive outage that took down the internet

Snapchat
reddit
Reddit
aws
Amazon Web Services
ring
Ring
amazon
Amazon
chime
Chime
amazon alexa
Amazon Alexa
ROBLOX
Roblox
venmo
Venmo

Estimated Losses



Reinsurance News

CyberCube estimates preliminary AWS outage loss range of $38-581m

24th October 2025 - Author: Saumya Jain - Share

CyberCube, a provider of cyber risk analytics for the re/insurance industry, has estimated a preliminary loss range of $38 million to $581 million for the Amazon Web Services (AWS) outage, which the firm has nicknamed "Amazonk".

This estimate follows the release of CyberCube's Security Incident Report (SIR) for the event, which estimated the potential impact on re/insurance as moderate.

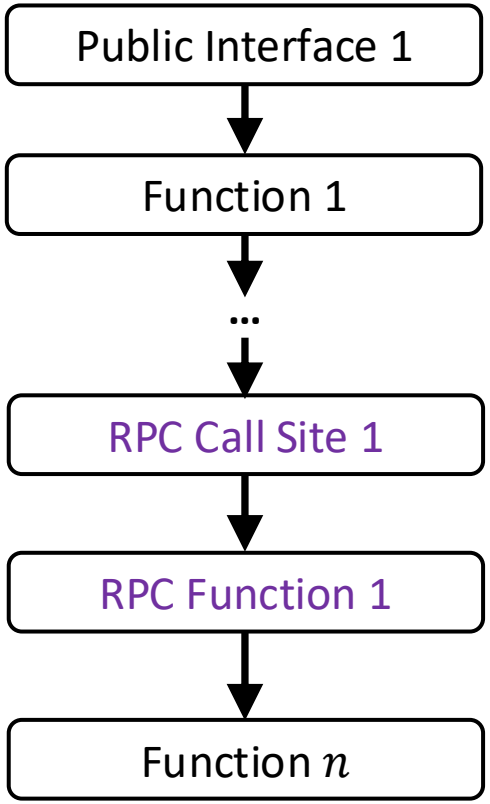
This has been reinforced in the estimated loss ratio impact for cyber insurers, which currently sits in the low- to mid-single digits.

CyberCube

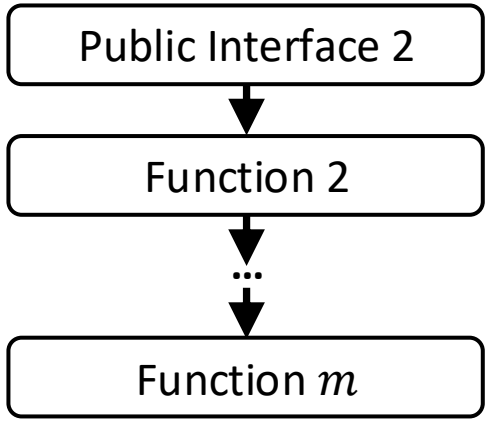
Even though the event could have had several potential fallouts, CyberCube sees most potential outcomes clustering toward the lower end of the quoted range, with the higher end of the range allowing for headroom for future information.

Distributed Concurrency Bugs

- Two concurrent execution flows accessing the same shared object
- At least one execution flow is triggered by inter-node communication (RPC)
- Causing incorrect behavior under certain access interleaving

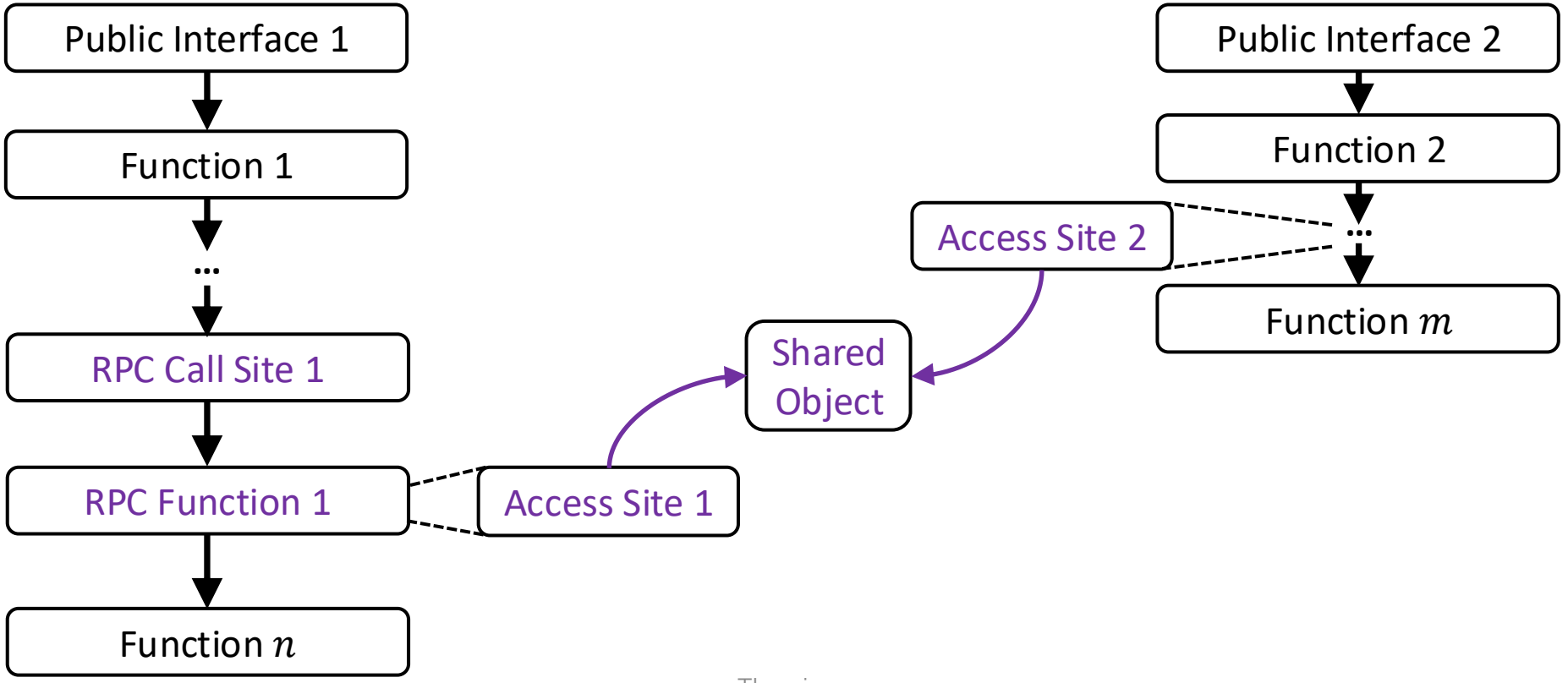


Widely used inter-node communication mechanism

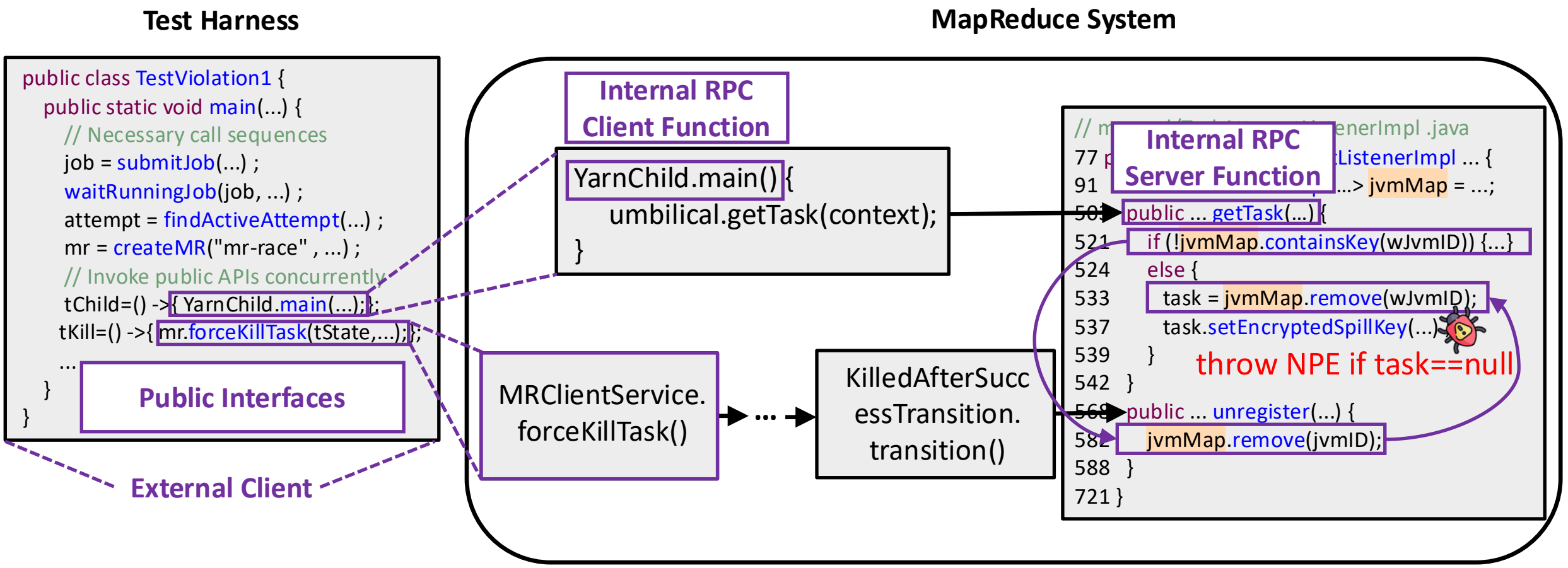


Distributed Concurrency Bugs

- Two concurrent execution flows accessing the same shared object
- At least one execution flow is triggered by inter-node communication (RPC)
- Causing incorrect behavior under certain access interleaving



A Newly Detected and Confirmed Bug – MapReduce 7507

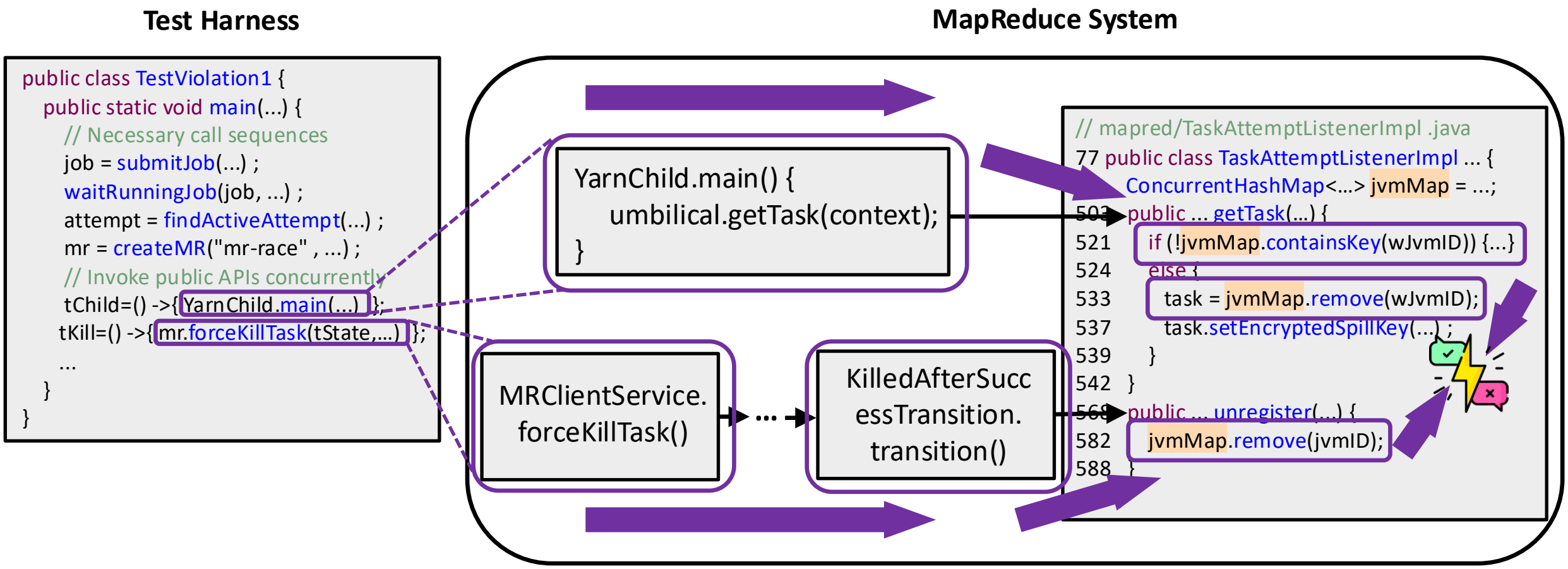


Static or Dynamic Analysis?

- Dynamic approaches (e.g., DCatch [ASPLOS '17], CloudRaid [FSE/ESEC '18]) rely on hand-crafted workloads or existing test suites

- Our Approach: Static analysis + validation
  - Static Analysis: Maximizing coverage for race detection
  - Dynamic Validation: Pruning benign races and false positives

Existing Top-Down Analysis for Static Race Detection With a Harness



Static Race Detection for Open Systems

- Problem: Static race detection for distributed concurrency bugs?
- Challenge: Static analysis for open systems
- Insight: Static RPC-driven race detection tailored for open systems
  - Assumed concurrency model where any pair of public interfaces can be concurrent
  - Variable-centric analysis focusing on RPC-reachable shared variables and their access site

Static RPC-Driven Race Detection Without a Harness

Test Harness

Test harness is unavailable

MapReduce System

```
YarnChild.main() {
    umbilical.getTask(context);
}
```

```
MRClientService.
forceKillTask()
```

…

```
KilledAfterSucc
essTransition.
transition()
```

```
// mapred/TaskAttemptListenerImpl .java
77 public class TaskAttemptListenerImpl ... {
91    ConcurrentHashMap<…> jvmMap = …;
503   public … getTask(…) {
521     if (!jvmMap.containsKey(wJvmID)) {…}
524     else {
533       task = jvmMap.remove(wJvmID);
537       task.setEncryptedSpillKey(…) ;
539     }
542   }
569   public … unregister(…) {
582     jvmMap.remove(jvmID);
588   }
721 }
```

Static RPC-Driven Race Detection Without a Harness

Test Harness

Assuming Concurrent Execution of Each Pair of Public Interfaces

MapReduce System

RPC Client and Server Function Pair

Shared Variable

```
YarnChild.main() {
    umbilical.getTask(context);
}
```

```
mapred/Tas                                va
public class                              ... {
  ConcurrentHashMap<...> jvmMap = ...;
503 public ... getTask(...) {
521   if (!jvmMap.containsKey(wJvmID)) {..}
524   else {
533     task = jvmMap.remove(wJvmID);
537     task.setEncryptedSpillKey(...) ;
539   }
542 }
569 public ... unregister(...) {
582   jvmMap.remove(jvmID);
588 }
```

Access Sites

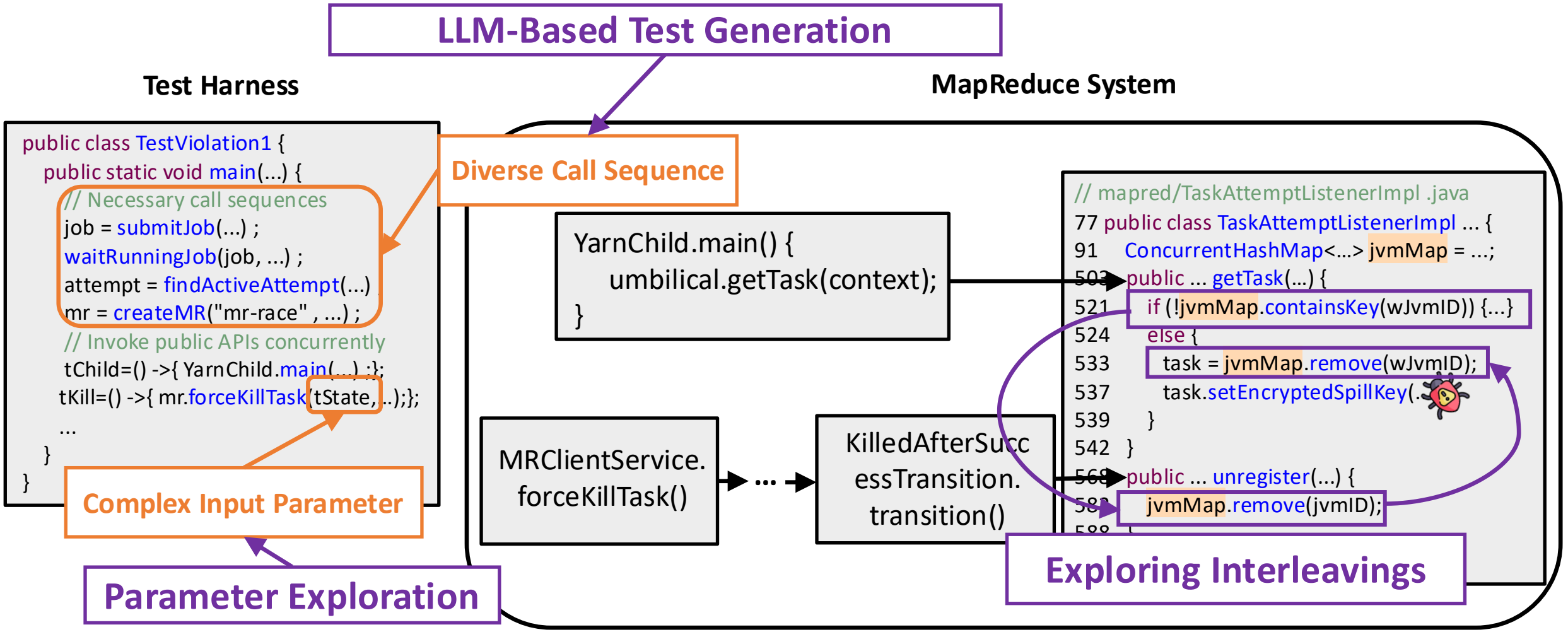
MRClientService.forceKillTask() → ... → KilledAfterSuccessTransition.transition()

Another Concurrent Call Chain

Constructing the Test Harness

- Problem: Construct a test harness to validate the static analysis result?
- Challenge: Combinatorial search space of call sequences and parameters
- Insight: Decoupling call sequence synthesis and parameter exploration

Decoupling Call Sequence Synthesis and Parameter Exploration



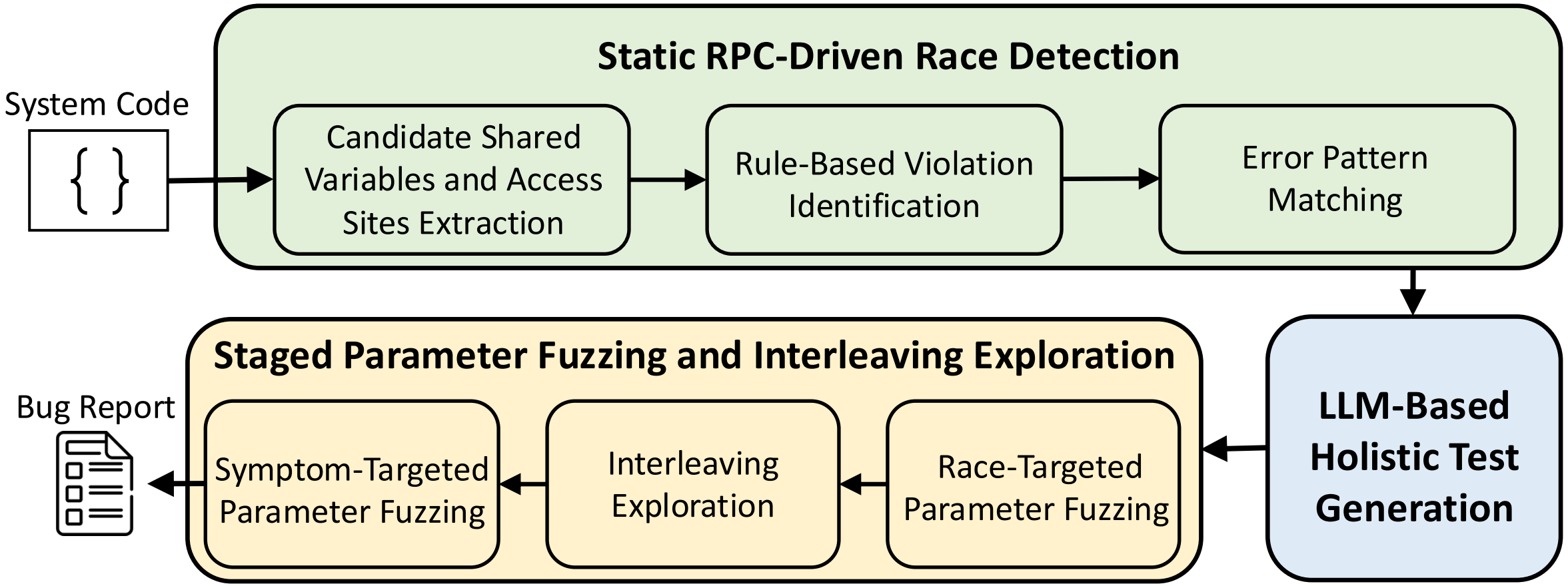
Towards Using LLMs

- Problem: Handle LLMs' hallucinations?

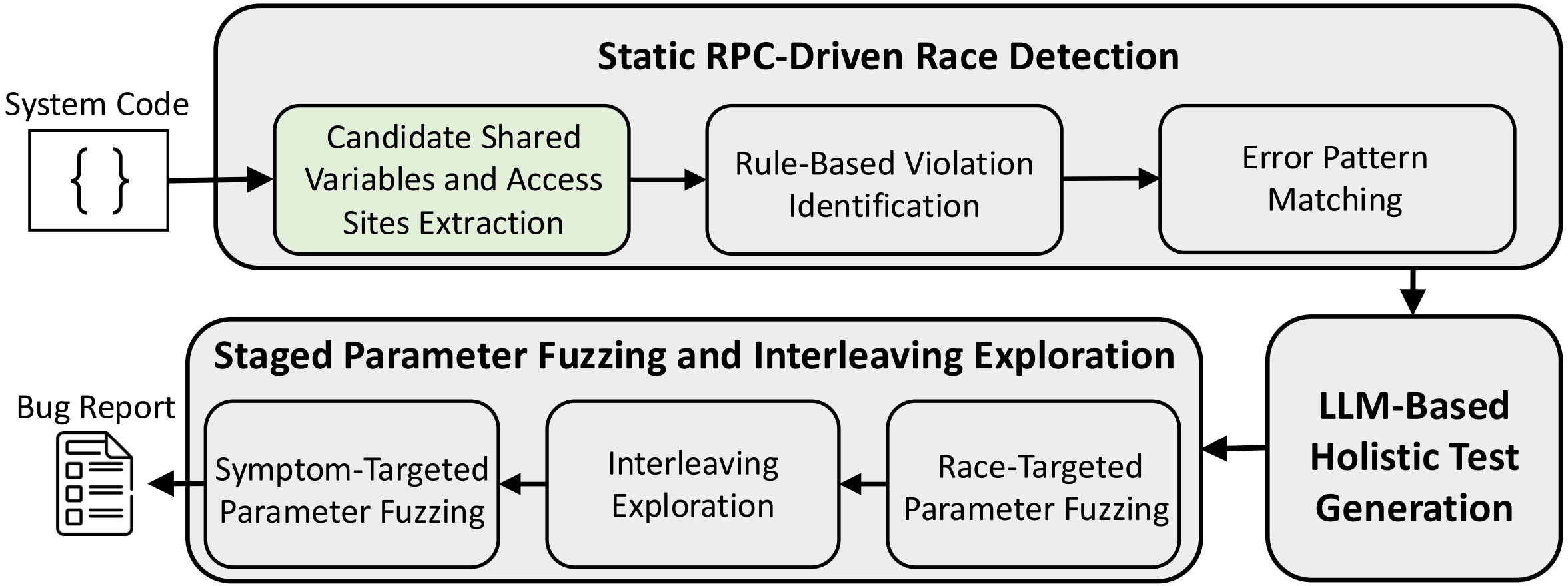
- Challenge: Be careful of using LLMs
  - The generated tests do not converge, missing target paths
  - Hard to determine the correctness of generated tests

- Insight: LLM + Guidance + Checking
  - Guidance of static analysis results
  - Checking by fuzzing to verify and refine the generated tests

Our Solution: Themis



Candidate Shared Variables and Access Sites Extraction



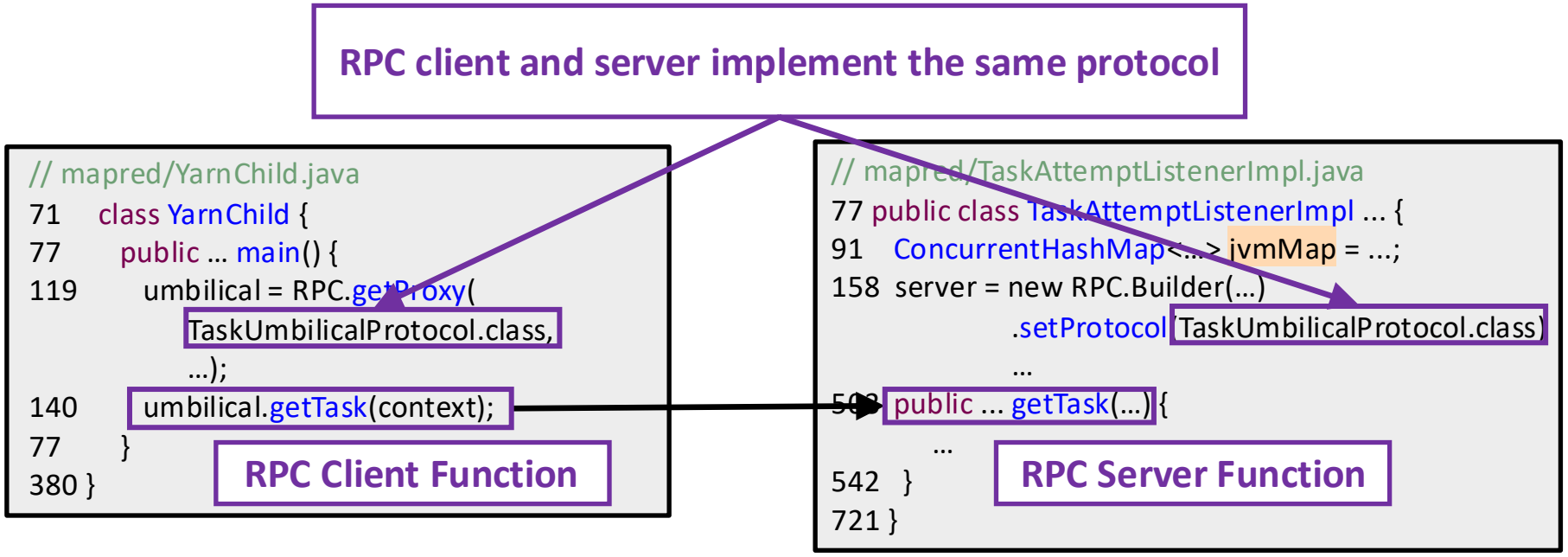
RPC Client–Server Pair Extraction

```
YarnChild.main() {
    umbilical.getTask(context);
}
```

```
// mapred/TaskAttemptListenerImpl.java
77 public class TaskAttemptListenerImpl ... {
91    ConcurrentHashMap<…> jvmMap = ...;
503   public ... getTask(…) {
          …
539       }
542   }
721 }
```

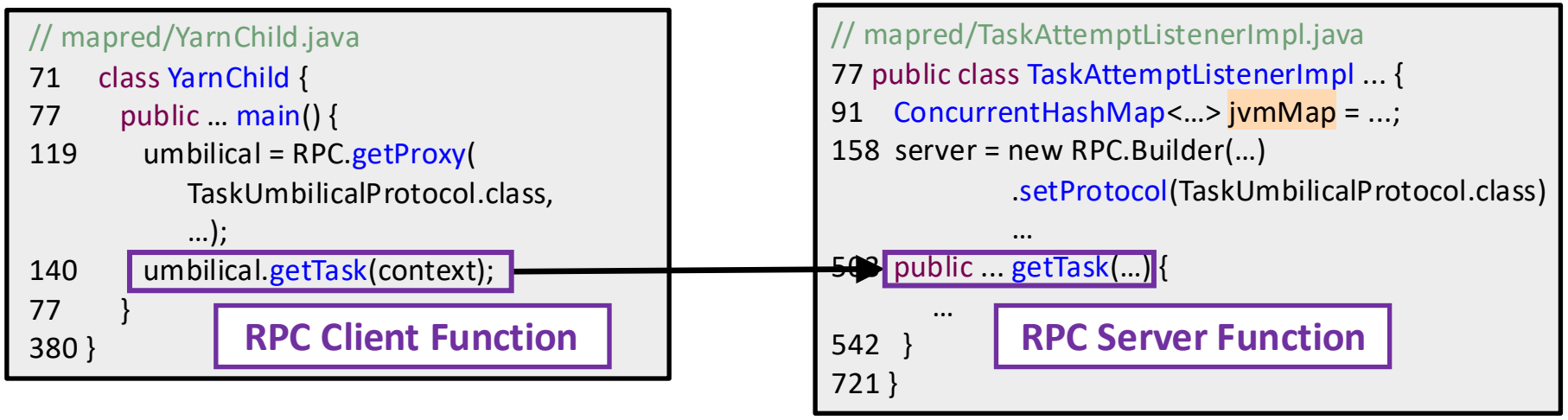
RPC Client–Server Pair Extraction

- Extracting candidate RPC client–server pairs that implement the same protocol



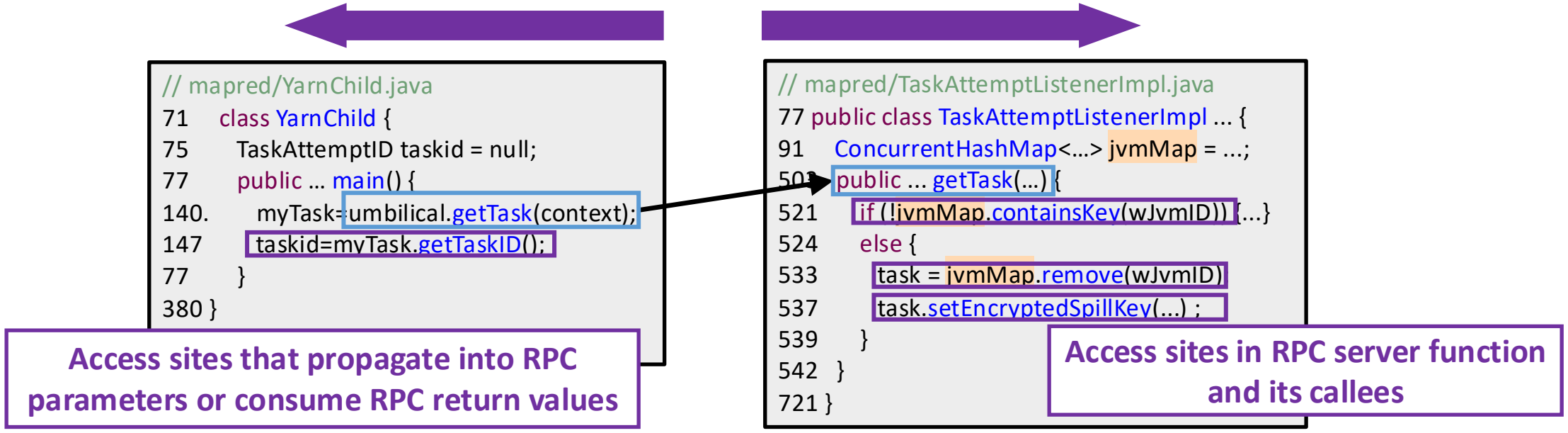
RPC Client–Server Pair Extraction

- Extracting candidate RPC client–server pairs that implement the same protocol
- Pruning RPC pairs with different configurable addresses



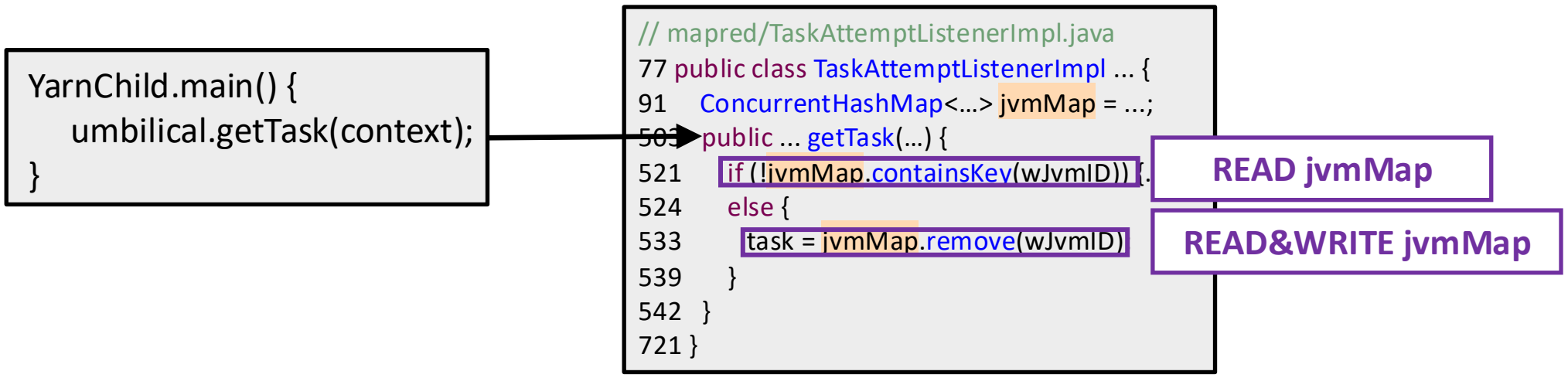
Access-Site Extraction

- Performing reachability analysis by traversing both upward and downward
- The downward traversal examines the server function and its callees
- The upward traversal examines the client function and its callers



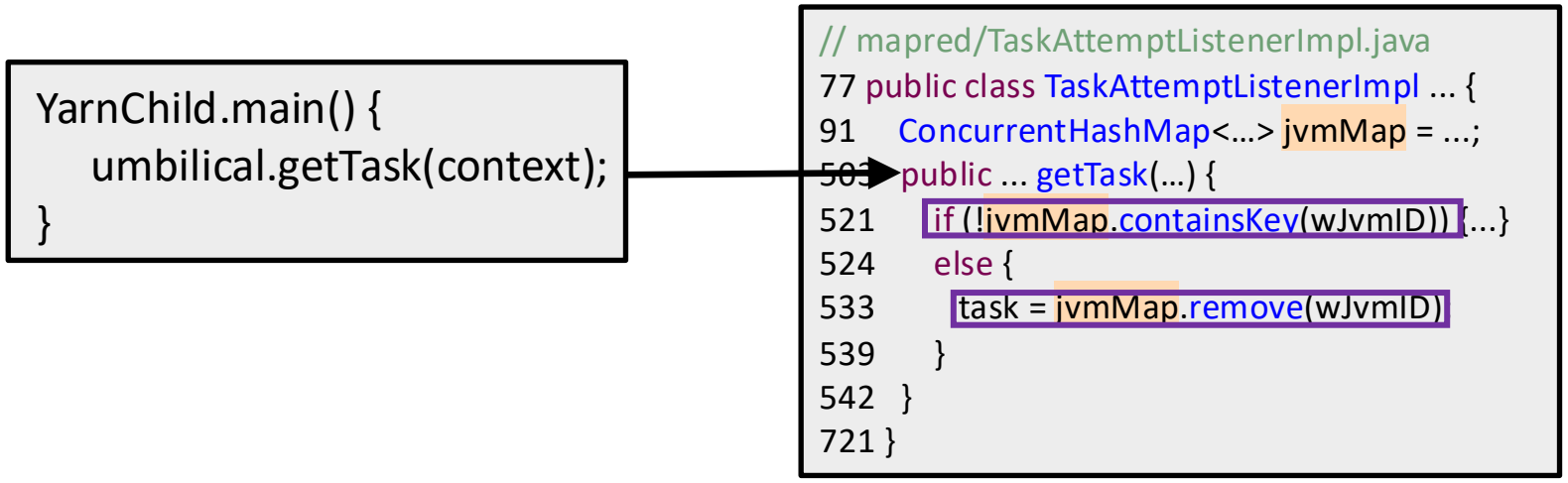
Variable Extraction from Access Sites

- Extract the variable and label it as a read or a write



Variable Extraction from Access Sites

- Extract the variable and label it as a read or a write
- Employing points-to analysis[1] to identify the referenced variables



[1] Lhoták, Ondřej, and Laurie Hendren. "Scaling Java points-to analysis using Spark." CC, 2003.

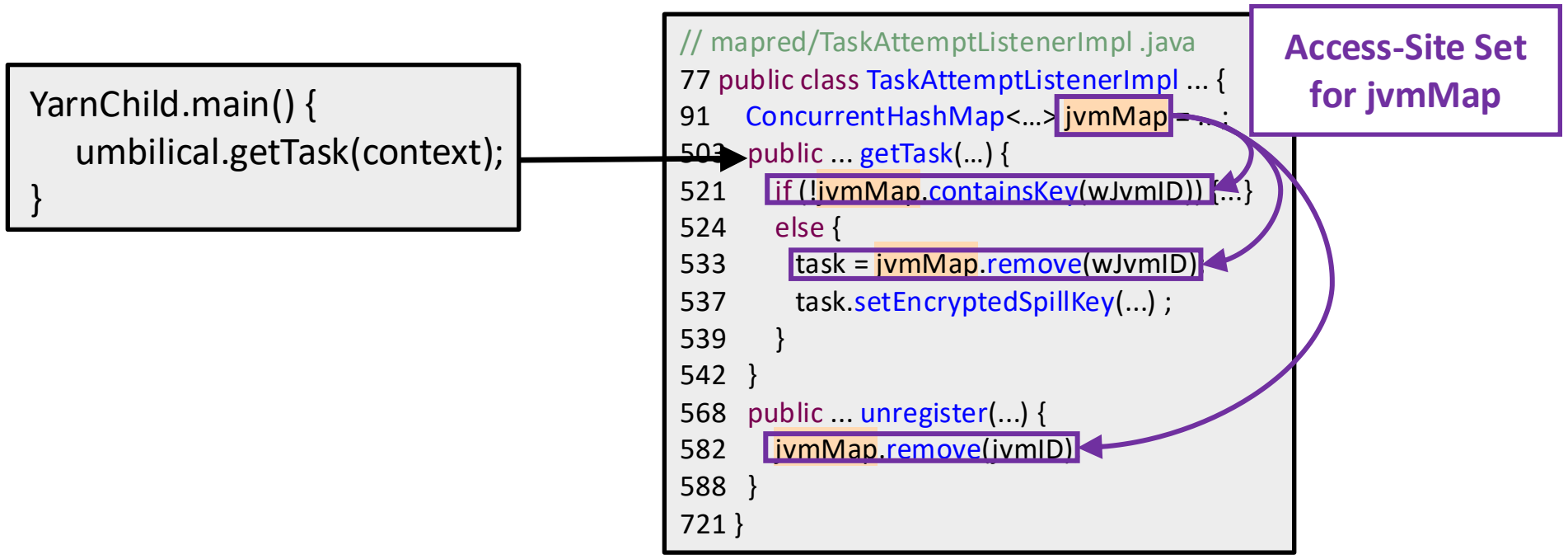
Candidate Shared Variable Filtering

- RPC Server Side
  - **Server-instance fields:** RPC server **concurrently** handles multiple client requests
  - Also tracking static fields, and IO/NIO objects

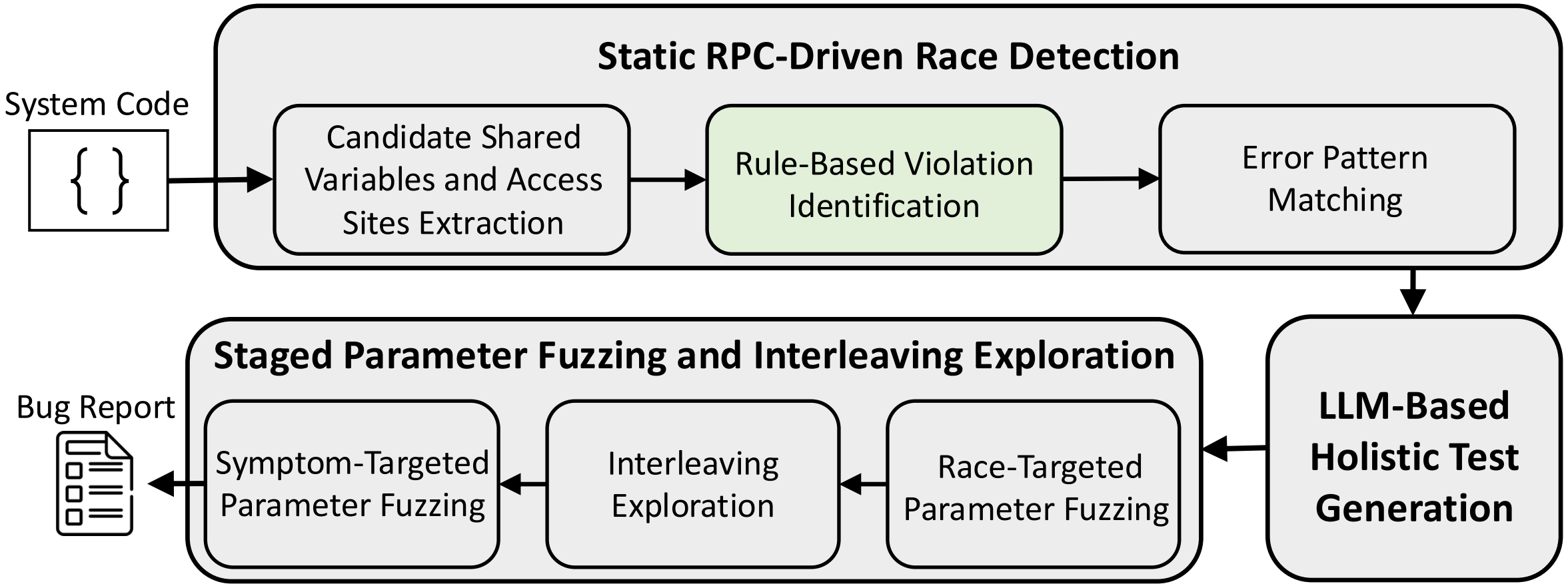
- RPC Client Side
  - Tracking all local, instance, and static variables, and IO/NIO objects

Per-Variable Access-Site Set Construction

- Examining all the access sites within the variable's declaration scope
- Applying points-to analysis to retain accesses that reference the same variable
- For IO/NIO objects, file path analysis is performed



Rule-Based Violation Identification



Rule-Based Violation Identification

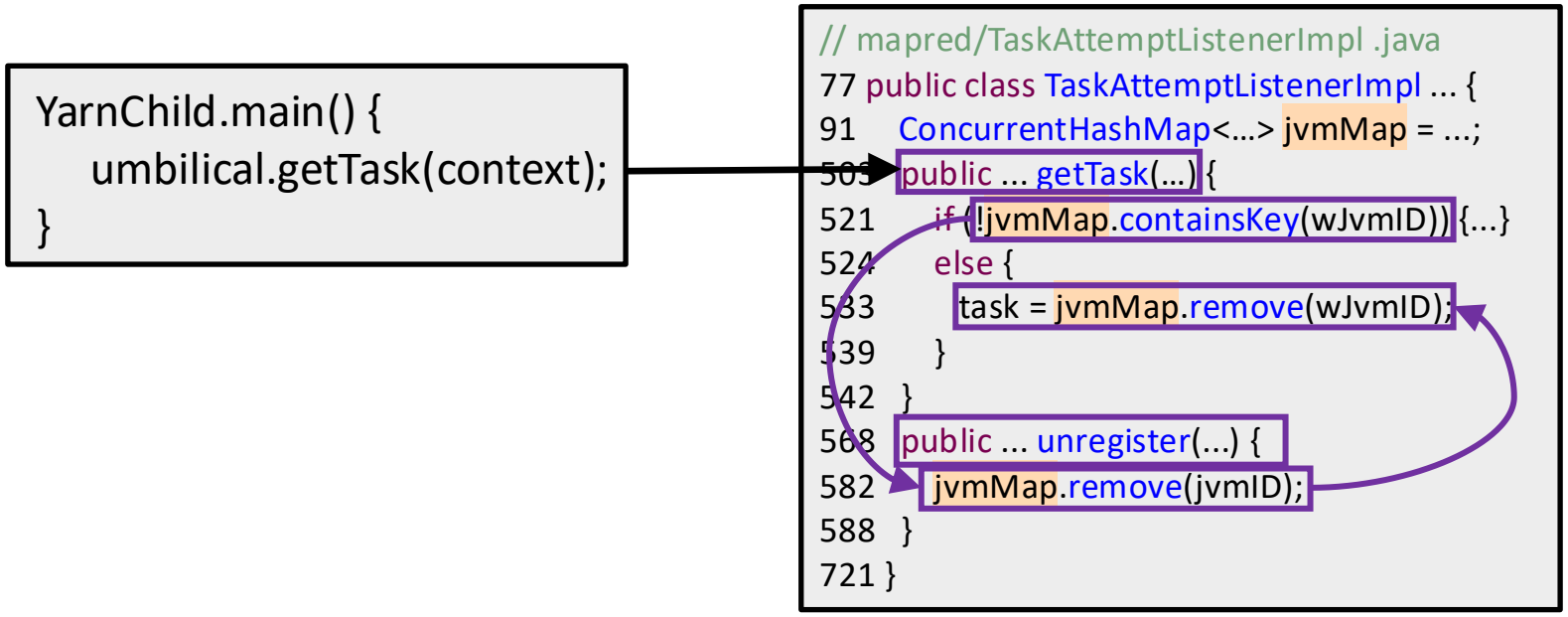
- Rule 1: Order violation across different functions
- Rule 2: Atomicity violation across different functions
- Rule 3: Self-atomicity violation within one function
- Rule 4: Order violation via asynchronous execution within one function
- Rule 5: Atomicity violation via asynchronous execution within one function

Server and Client Distinctions

- Rule 1: Order violation across different functions
- Rule 2: Atomicity violation across different functions
- Rule 3: Self-atomicity violation within one function
- Rule 4: Order violation via asynchronous execution within one function
- Rule 5: Atomicity violation via asynchronous execution within one function

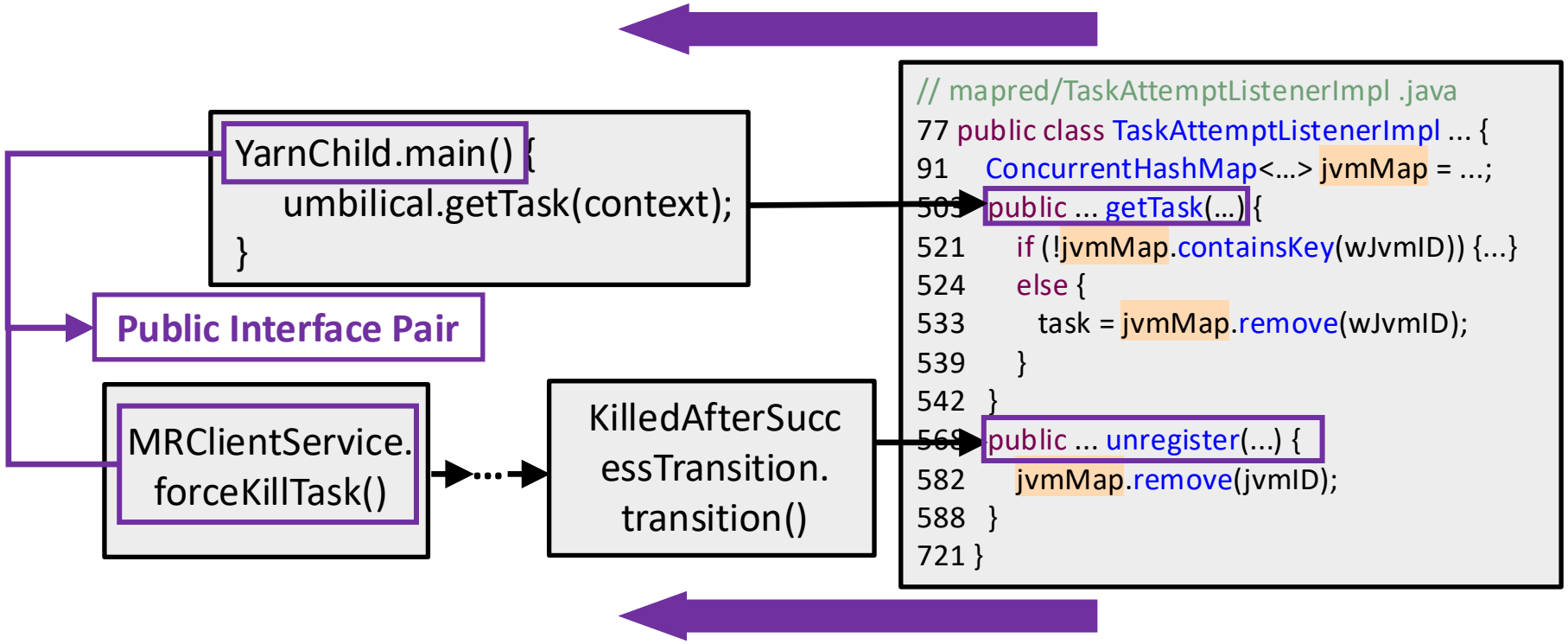
Rules 1-3 apply to both server and client code, while Rules 4 and 5 are specific to the client side

Detecting the Violation in the Example



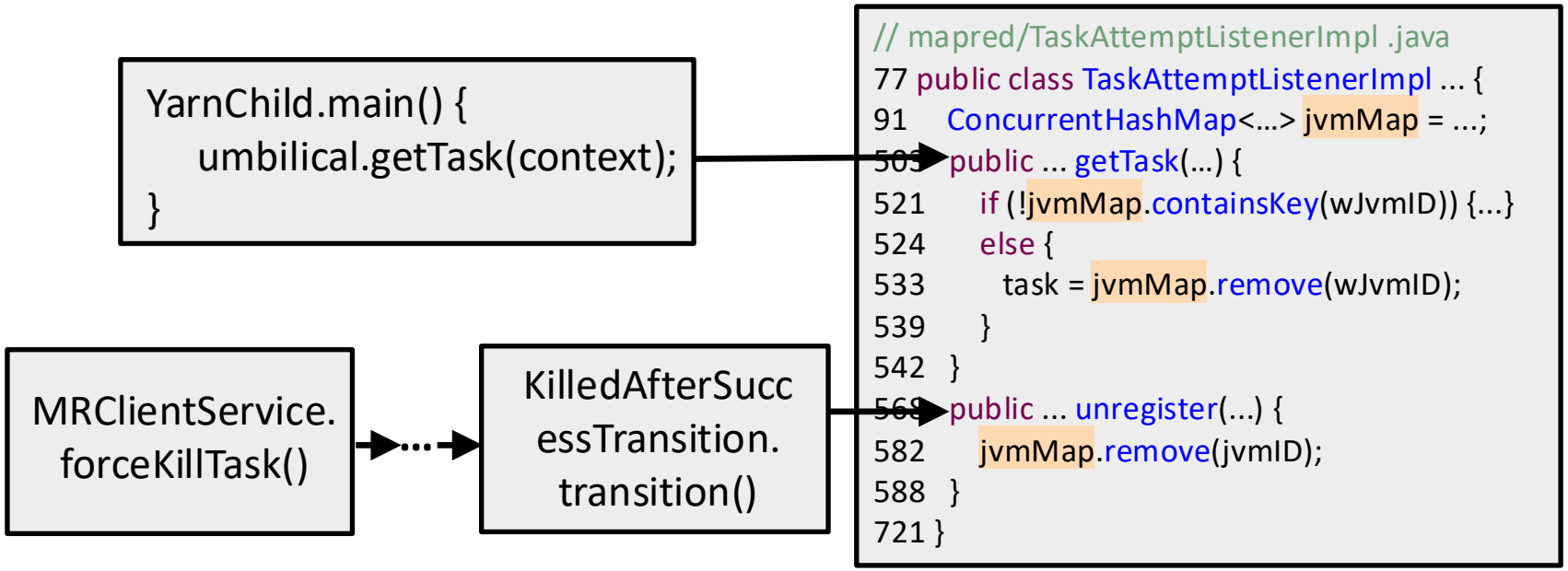
Public Interface Pair Extraction

- Rules 1–3: Two public interfaces for racy functions; for RPC call chains, client-side public interfaces are extracted



Public Interface Pair Extraction

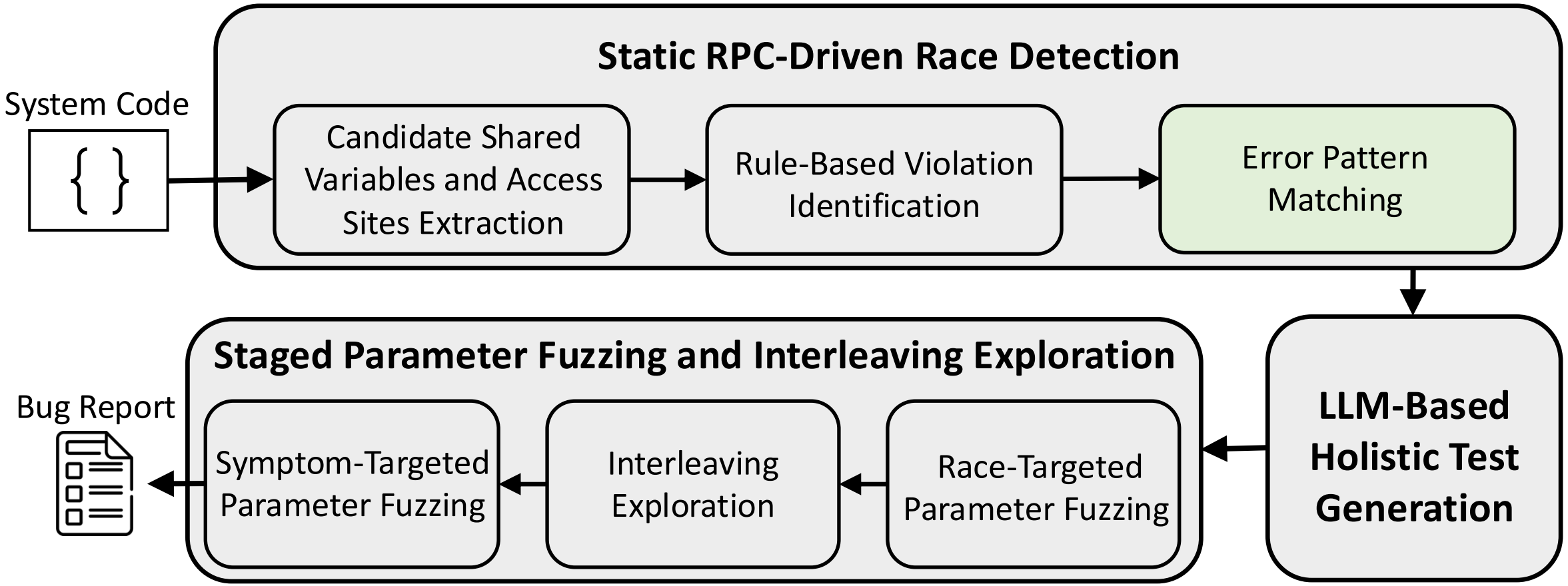
- Rules 1–3: Two public interfaces for racy functions; for RPC call chains, client-side public interfaces are extracted
- Rules 4 and 5: Only one public interface is needed because asynchronous execution guarantees concurrency



Synchronization Analysis for Atomicity Violations

- Pruning cases with Java synchronized keyword (method and variable)
- Ignoring application-defined synchronization (custom locks, condition variables)
- Deferring eliminating application-defined synchronization until the testing phase

Error Pattern Matching



Error Pattern Matching

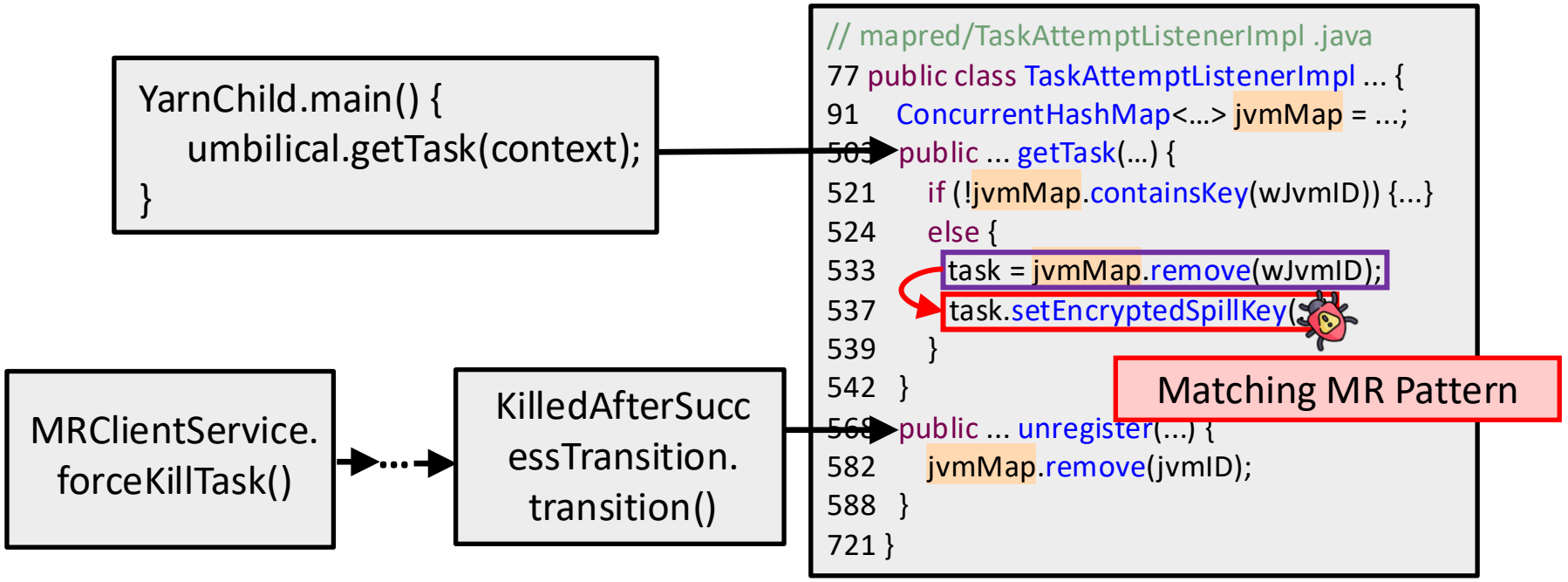
- Explicit Symptoms: Extracting failure statements propagated from races through control and data dependencies
  1. Uncaught exceptions: throw new YarnRuntimeException(…);
  2. Severe error logs: Log.error(“An error occurs …”);
  3. System abort: System.exit(-1);

- Explicit order violations (EO) and atomicity violations (EA)

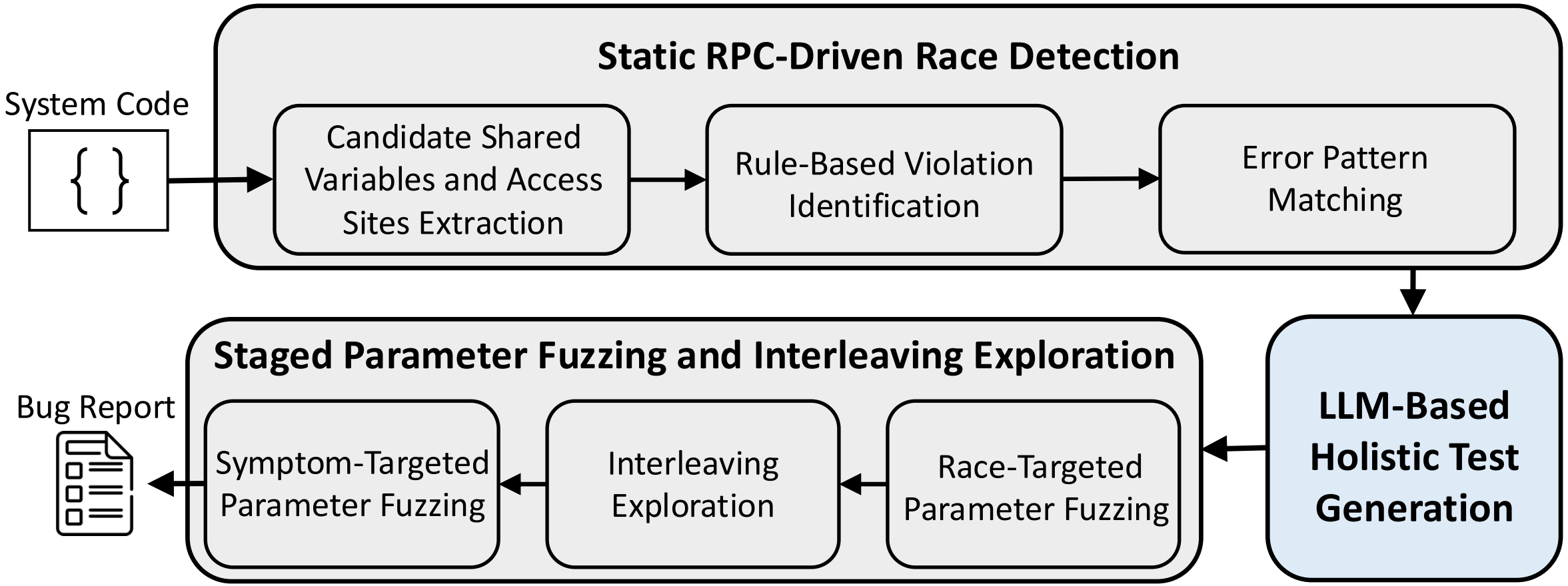
Error Pattern Matching

- Implicit Symptoms: Null pointer and out-of-bounds
  1. Null Pointer Dereference (NP)
  2. Out-of-Bounds Access (OB)
  3. Uninitialized Read (UR)
  4. Missing Retrieval Check (MR) : error propagation!
  5. Thread-unsafe Objects (TO)

Error Pattern Matching in the Example



LLM-Based Holistic Test Generation



LLM-Based Holistic Test Generation

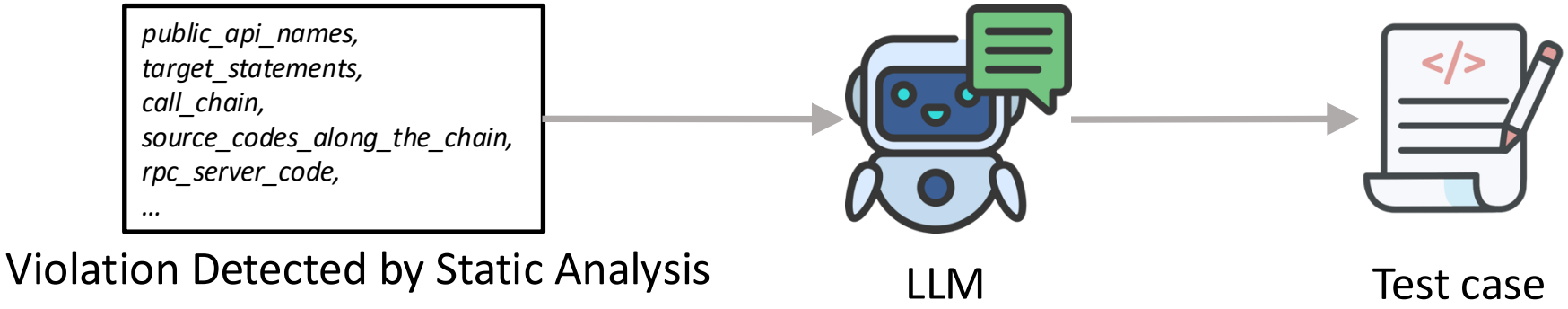
- **Goal**: Construct a test case that executes necessary call sequences and invokes public interfaces without runtime errors

LLM-Based Holistic Test Generation

- **Goal**: Construct a test case that executes necessary call sequences and invokes public interfaces without runtime errors
- **Public Interface Selection**: closest > top-level > others

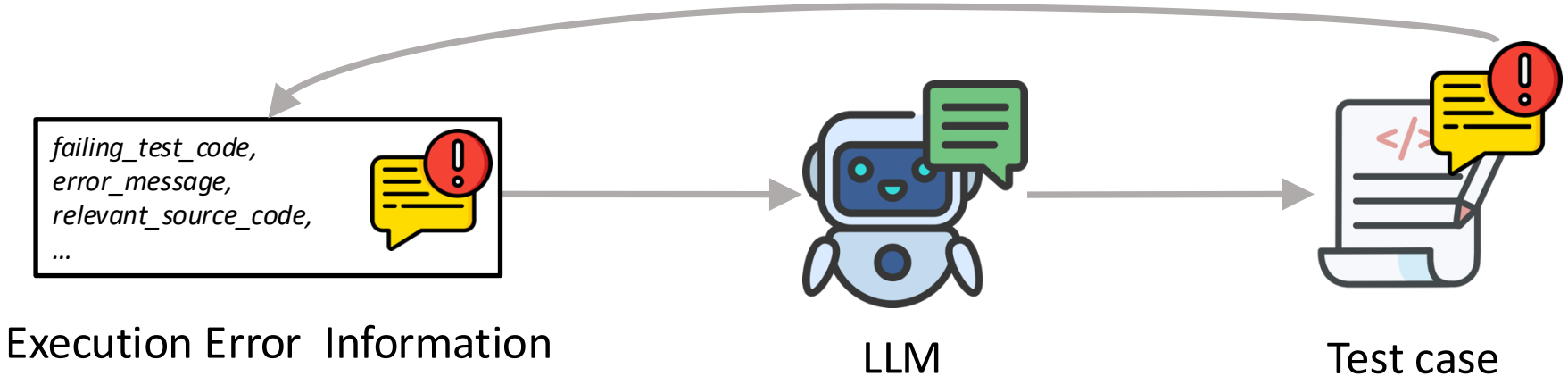
LLM-Based Holistic Test Generation

- **Goal**: Construct a test case that executes necessary call sequences and invokes public interfaces without runtime errors
- **Public Interface Selection**: closest > top-level > others
- **Workflow**: Generation-repair loop



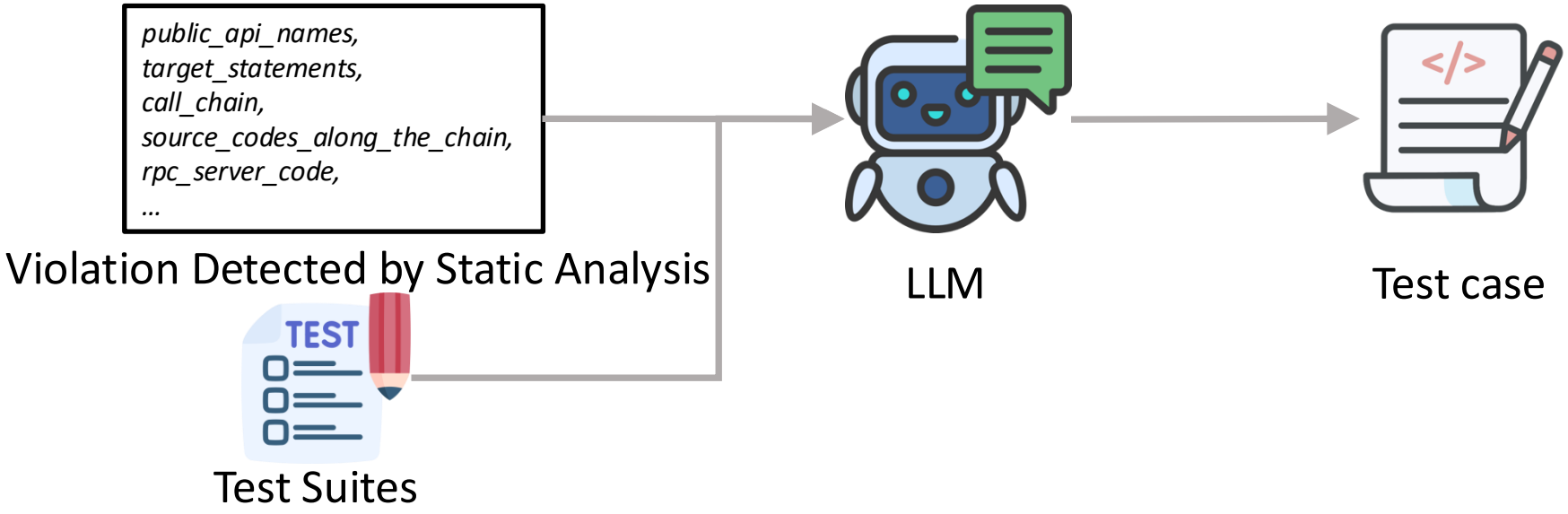
LLM-Based Holistic Test Generation

- **Goal**: Construct a test case that executes necessary call sequences and invokes public interfaces without runtime errors
- **Public Interface Selection**: closest > top-level > others
- **Workflow**: Generation-repair loop



LLM-Based Holistic Test Generation

- **Goal**: Construct a test case that executes necessary call sequences and invokes public interfaces without runtime errors
- **Public Interface Selection**: closest > top-level > others
- **Workflow**: Generation-repair loop

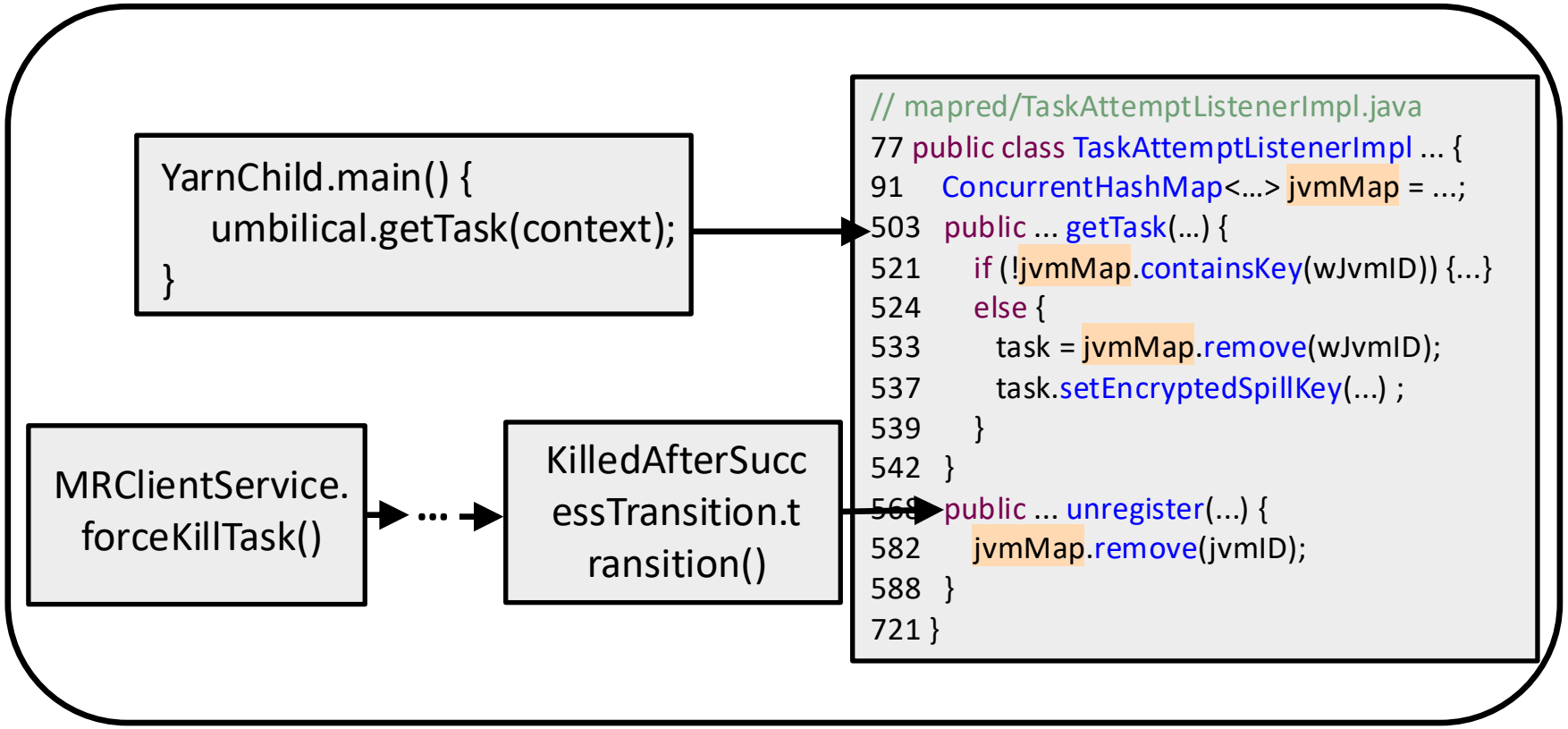


Generated Test Case with Incorrect Parameters

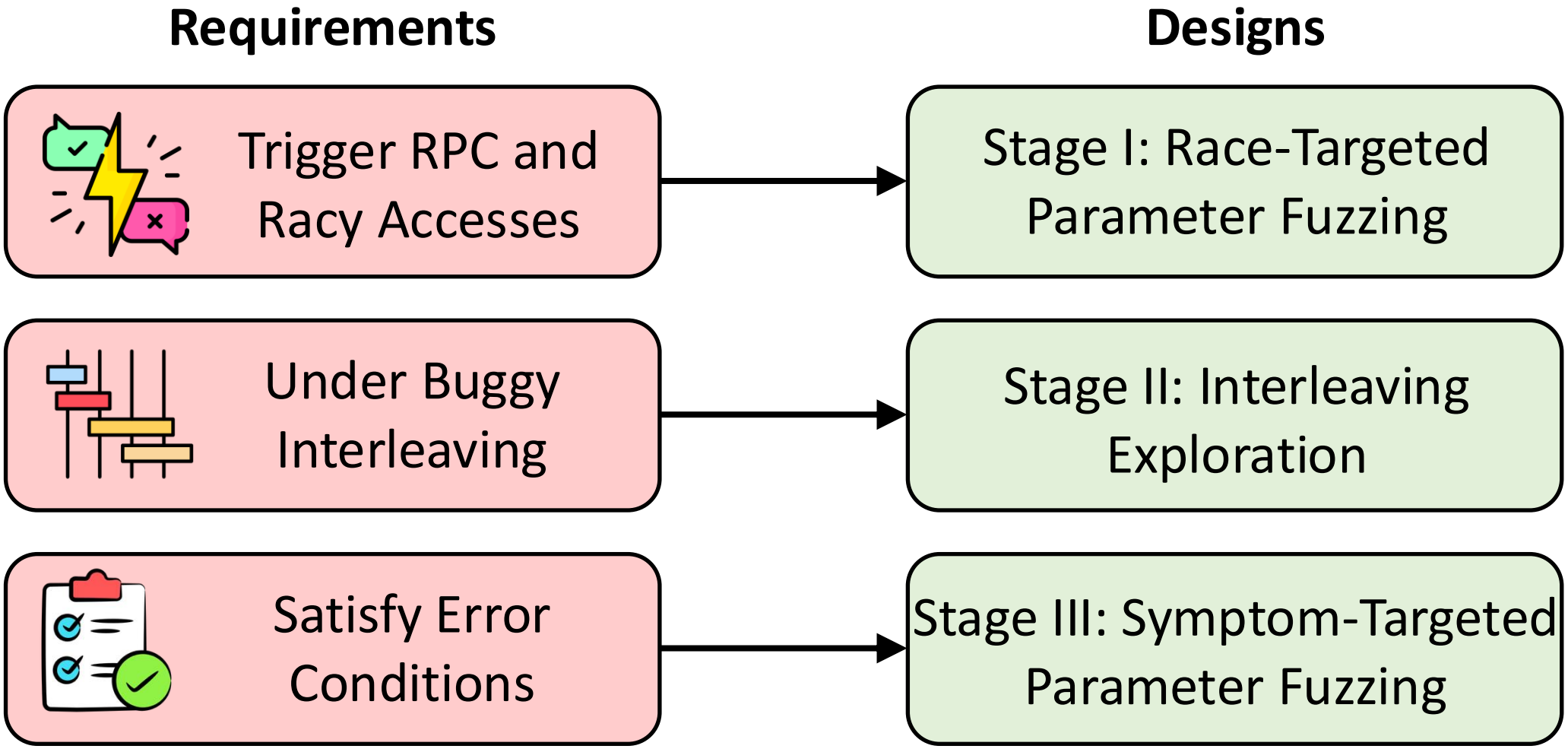
```
public class TestViolation1 {
   public static void main(...) {
      // Necessary call sequences
      job = submitJob(...) ;
      waitRunningJob(job, ...) ;
      attempt = findActiveAttempt(...) ;
      mr = createMR("mr-race" , ...) ;
      // Invoke public APIs concurrently
      tChild=() ->{ YarnChild.main(...) ;};
      tKill=() ->{ mr.forceKillTask(tState,..);};
      ...
   }
}
```

Incorrect Parameter

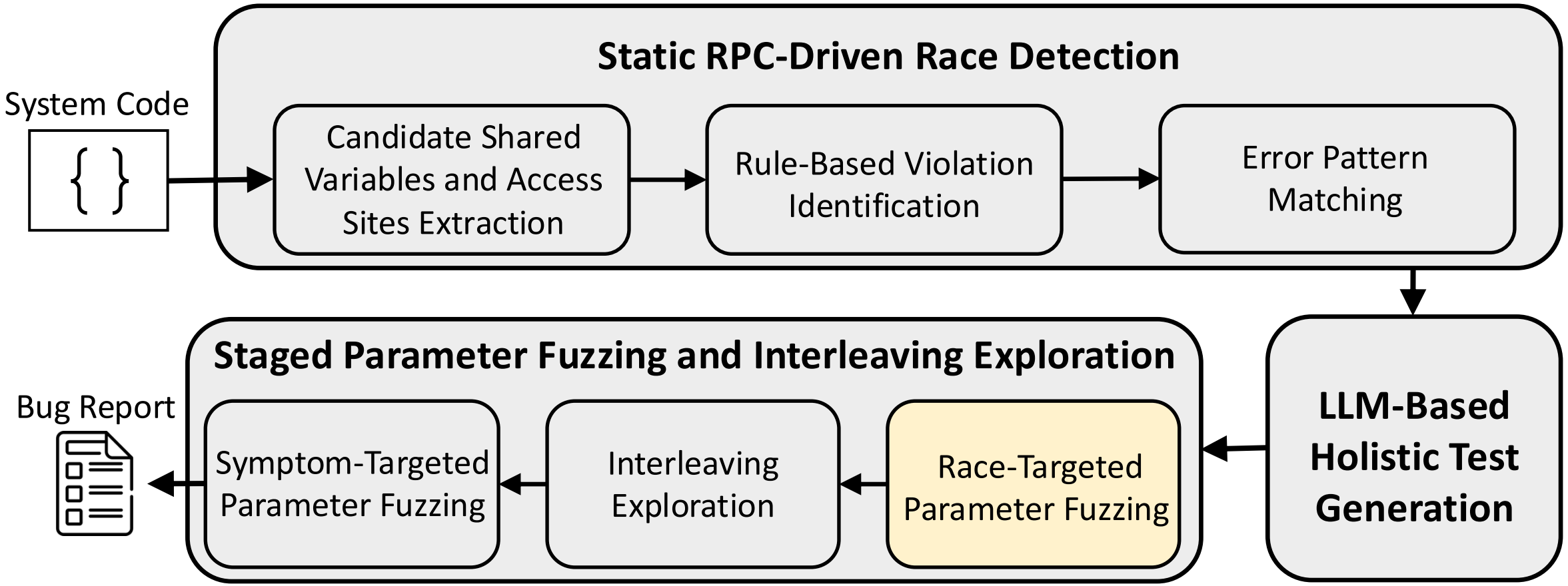
LLM-Generated Test Case



Staged Parameter Fuzzing and Interleaving Exploration

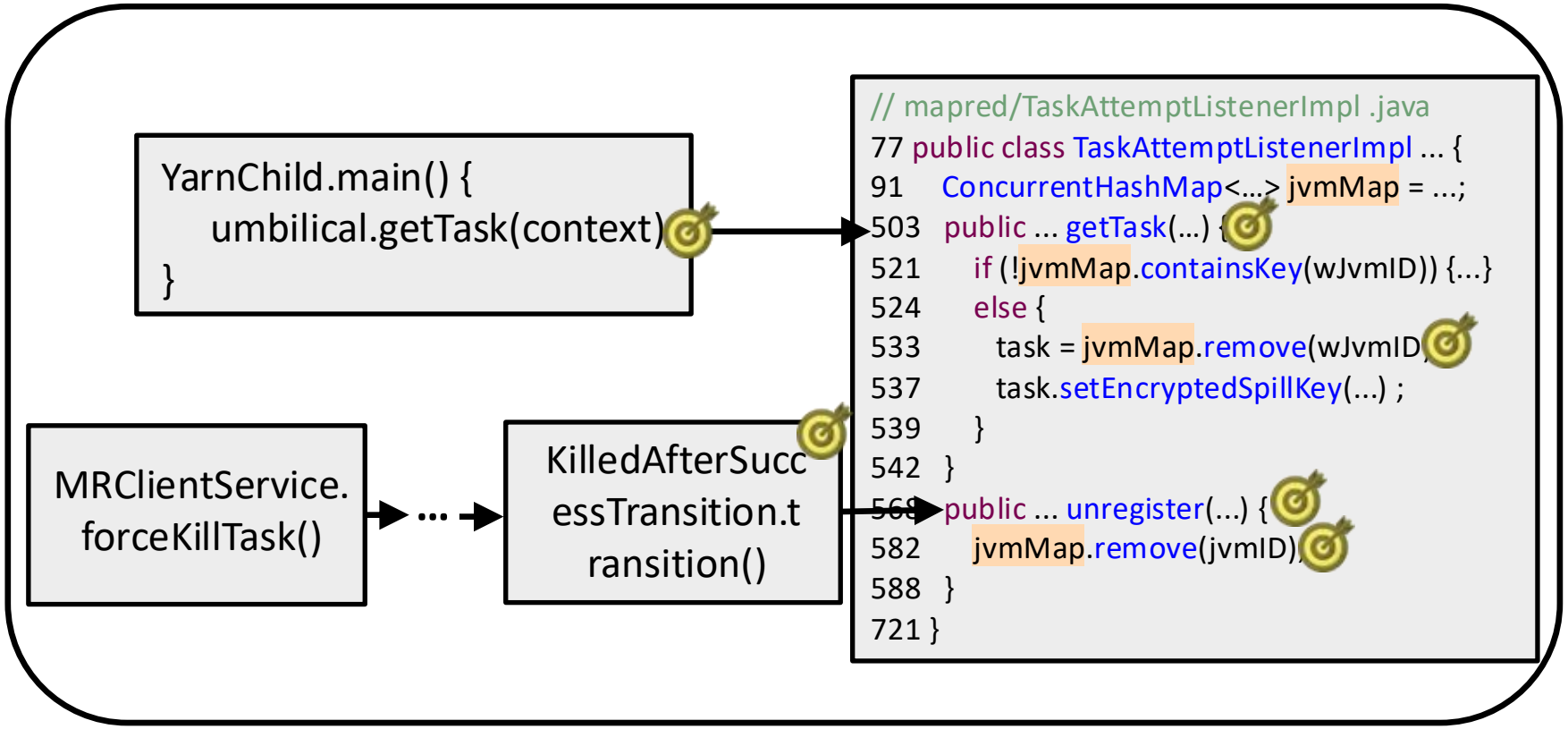


Stage I: Race-Targeted Parameter Fuzzing



Stage I: Setting Fuzzing Targets

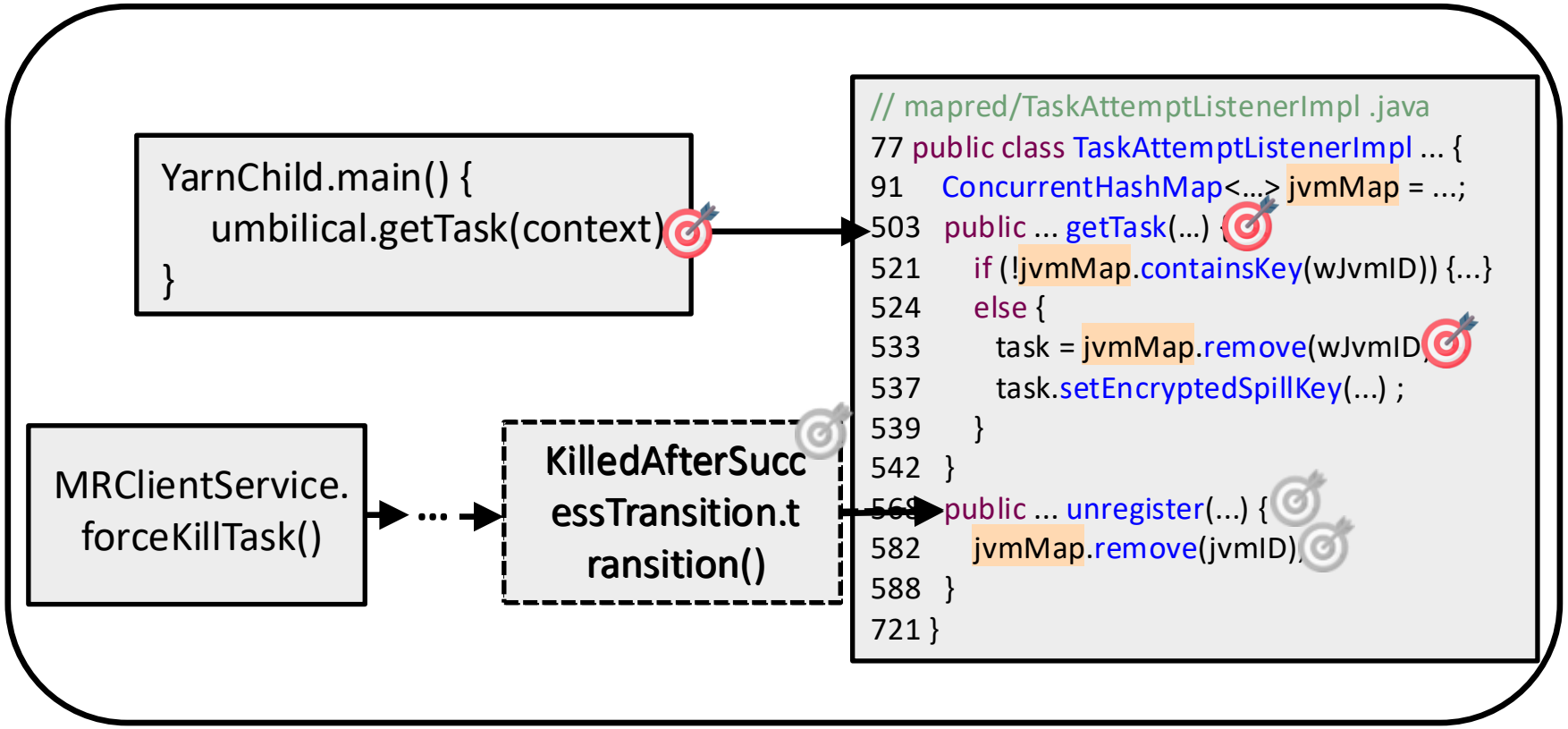
```
public class TestViolation1 {
   public static void main(...) {
      // Necessary call sequences
      job = submitJob(...) ;
      waitRunningJob(job, ...) ;
      attempt = findActiveAttempt(...) ;
      mr = createMR("mr-race" , ...) ;
      // Invoke public APIs concurrently
      tChild=() ->{ YarnChild.main(...) ;};
     tKill=() ->{ mr.forceKillTask(tState,...);};
     ...
   }
}
```



Stage I: Reaching the Racy Accesses

```
public class TestViolation1 {
   public static void main(...) {
      // Necessary call sequences
      job = submitJob(...) ;
      waitRunningJob(job, ...) ;
      attempt = findActiveAttempt(...) ;
      mr = createMR("mr-race" , ...) ;
      // Invoke public APIs concurrently
      tChild=() ->{ YarnChild.main(...) ;};
     tKill=() ->{ mr.forceKillTask(tState,...);};
      ...
   }
}
```

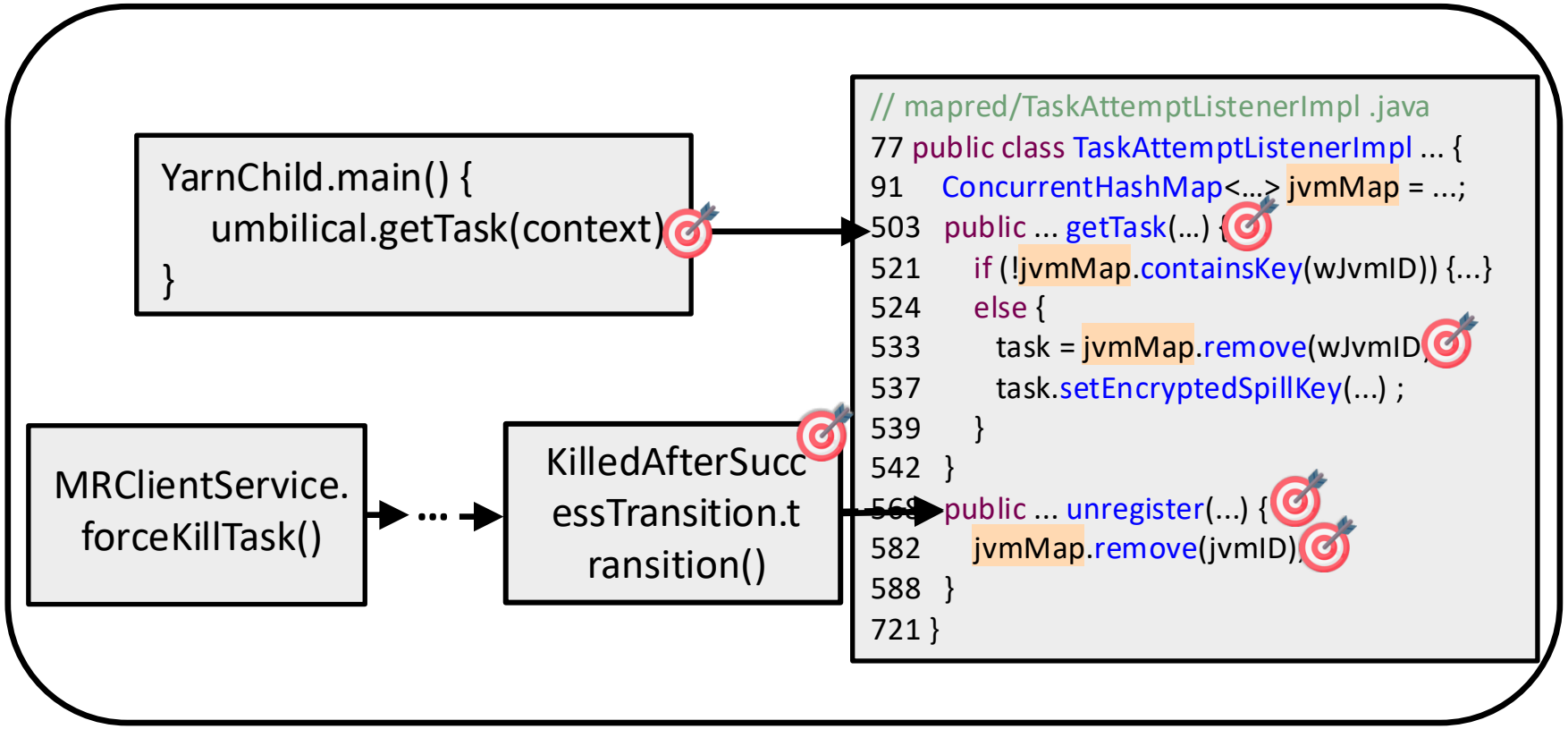
ASSIGNED



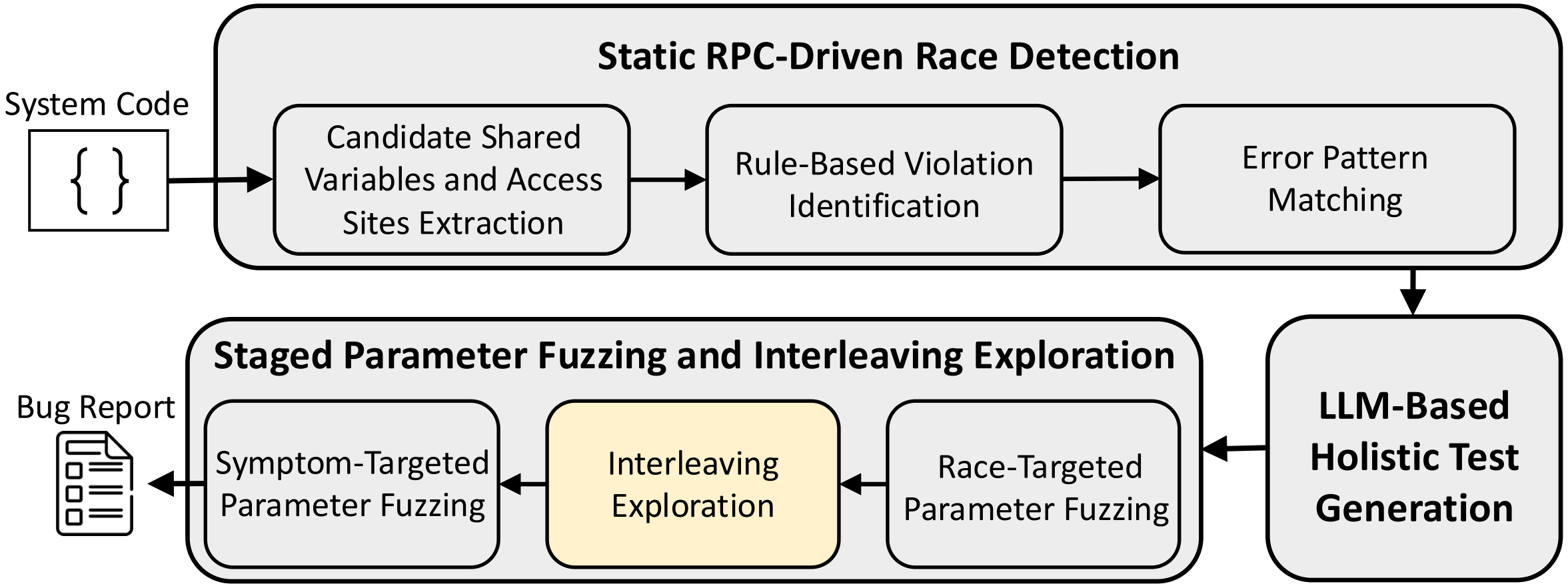
Stage I: Reaching the Racy Accesses

```
public class TestViolation1 {
   public static void main(...) {
      // Necessary call sequences
      job = submitJob(...) ;
      waitRunningJob(job, ...) ;
      attempt = findActiveAttempt(...) ;
      mr = createMR("mr-race" , ...) ;
      // Invoke public APIs concurrently
      tChild=() ->{ YarnChild.main(...) ;};
      tKill=() ->{ mr.forceKillTask(tState,...);};
      ...
}
```

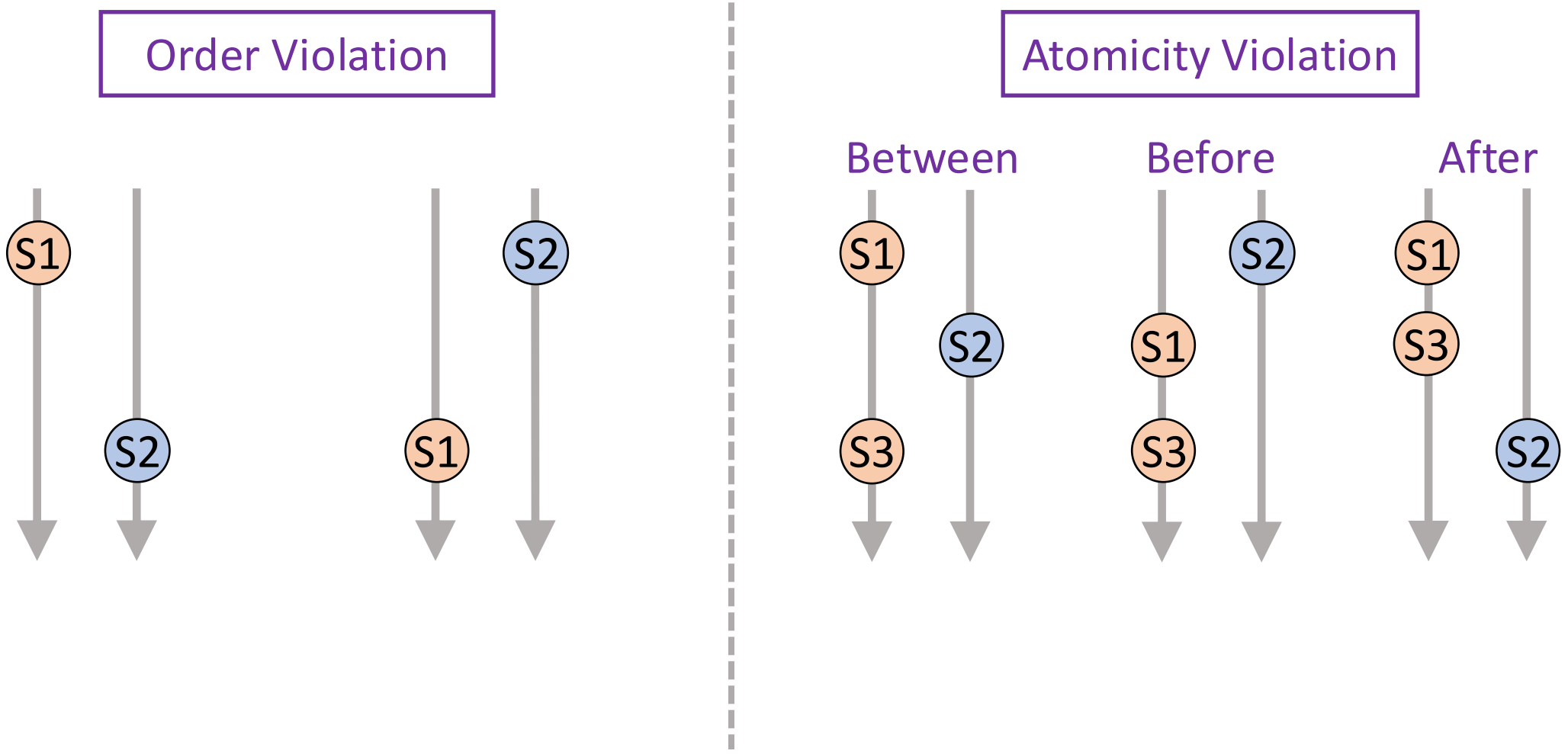
SUCCESS_FINISHING



Stage II: Interleaving Exploration

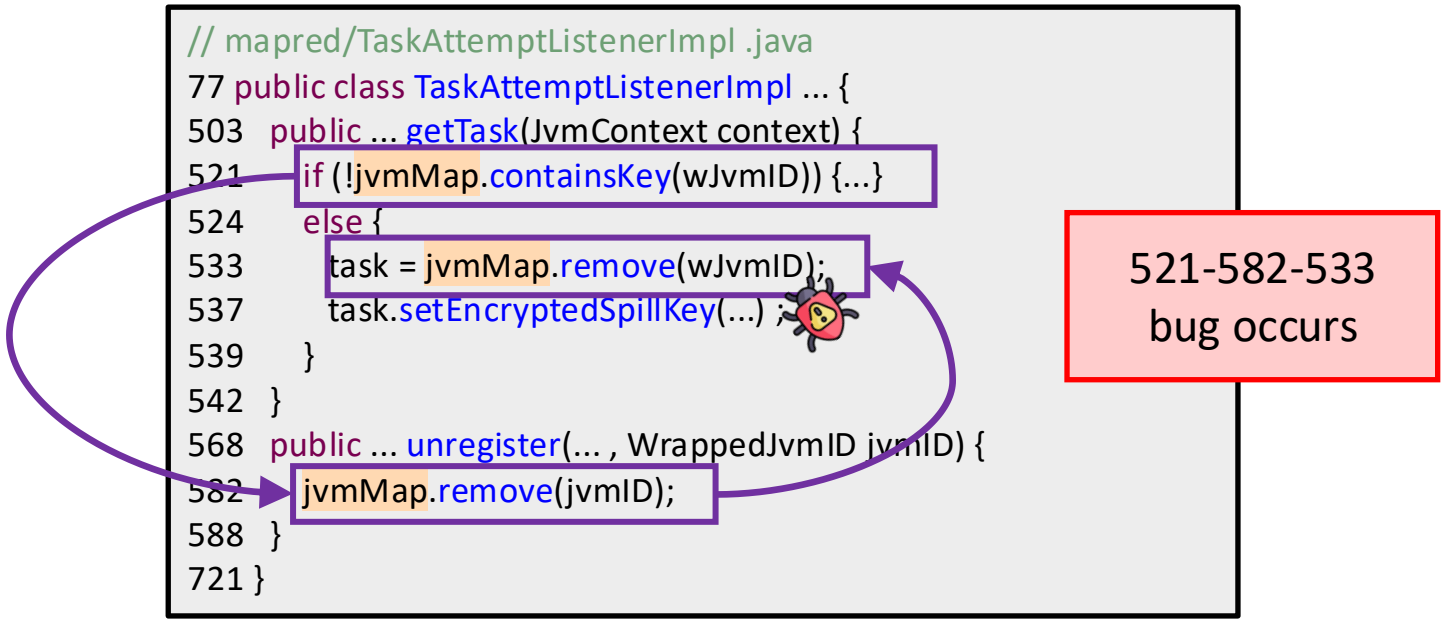


Stage II: Interleaving Exploration

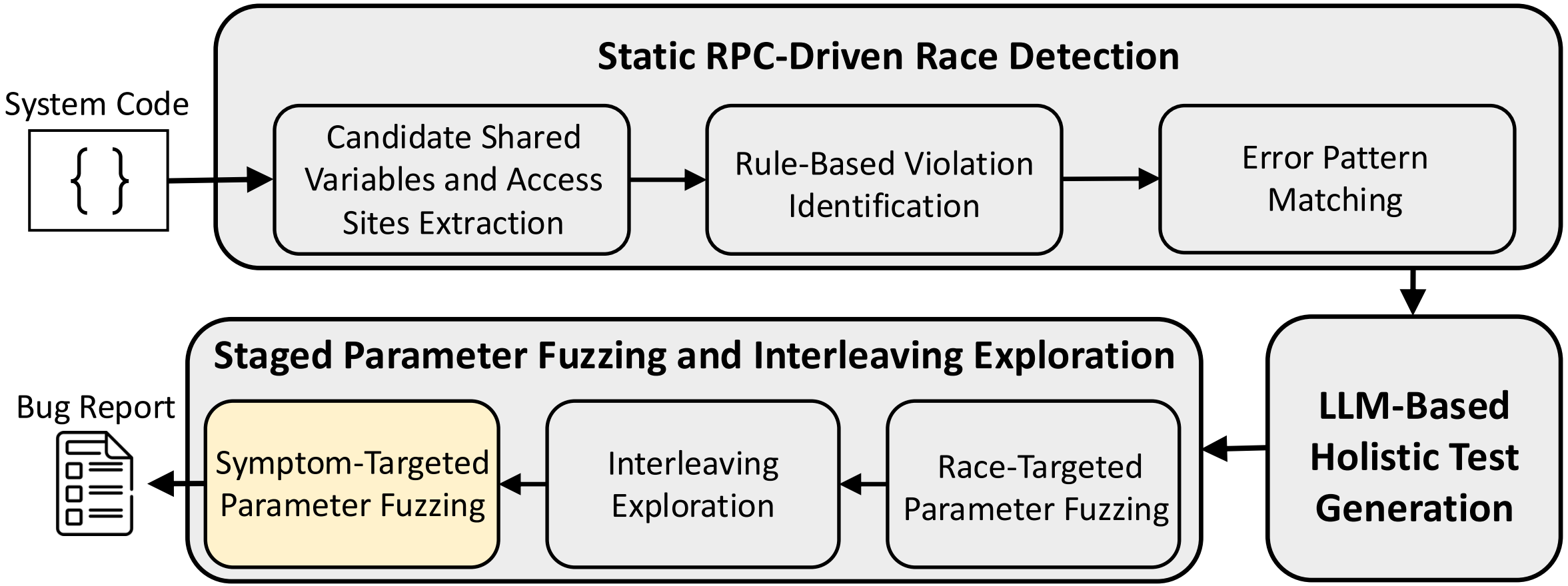


Stage II: Checking Behaviors

- Order: Symptom manifests under one specific order
- Atomicity: Symptom appears when one access occurs between the other two consecutive accesses



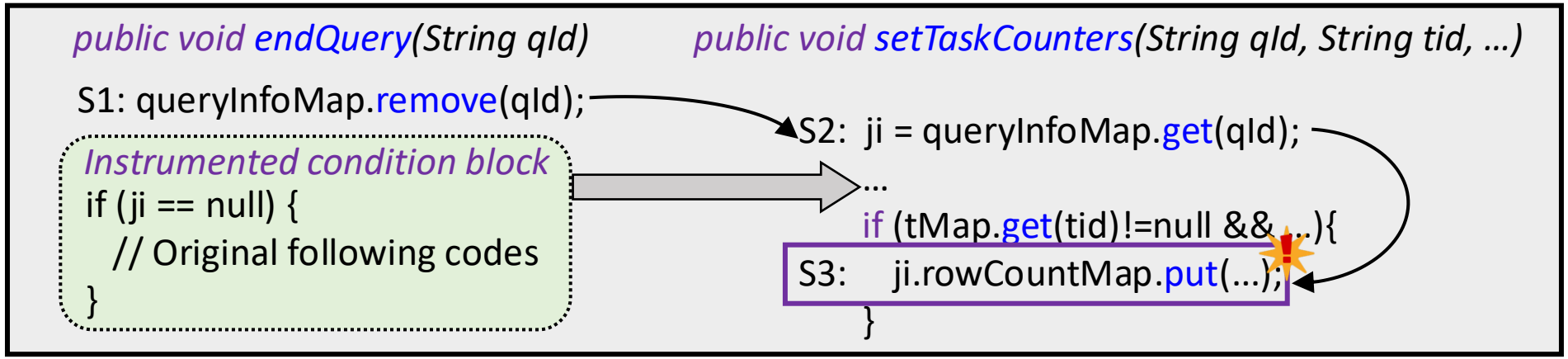
Stage III: Symptom-Targeted Parameter Fuzzing



Stage III: Symptom-Targeted Parameter Fuzzing

- Symptom statements as additional fuzzing targets
- Inherit seed pool and the parameters from Stage I
- Check whether racy accesses are reached, as in Stage I, and explore interleavings and check behaviors, as in Stage II

Transforming error-triggering conditions into explicit control flow branches



Evaluation Methodology

- Implementation
  1. The static analysis: Soot
  2. Test generation: Claude-4
  3. Directed fuzzing: SelectFuzz and JQF
- Benchmark
  1. Latest releases of **8** popular cloud systems: MapReduce, Yarn, HDFS, Hadoop, Hbase, Alluxio, Hive, Tez
  2. **6** known bugs from prior studies
- Metrics
  1. Number of violations (total and newly discovered)
  2. Number of bugs counts groups of violations that could be fixed by the same patch (total and newly discovered)

Evaluation Results

- **204** unique violations (**198** new), corresponding to **58** bugs (**52** new) and **6** have been confirmed

System	EO	EA	NP	OB	UR	MR	TO	Total
Alluxio	0	0	0	0	2	0	0	2
Hadoop	1	0	1	0	0	0	0	2
HBase	5[3]	3[1]	1	0	36	1	2	48[4]
HDFS	3	0	0	0	1	3	1	8
Hive	2	0	0	0	10	1	0	13
MapReduce	7[2]	3	3	2	19	3	2	39[2]
Tez	1	0	0	0	2	0	0	3
Yarn	9	0	4	0	71	2	3	89
Total	28[5]	6[1]	9	2	141	10	8	204[6]

Evaluation Results

- **204** unique violations (**198** new), corresponding to **58** bugs (**52** new) and **6** have been confirmed
- **170** out of 204 detected violations (**83.3%**) exhibit **implicit symptoms**

System	EO	EA	NP	OB	UR	MR	TO	Total
Alluxio	0	0	0	0	2	0	0	2
Hadoop	1	0	1	0	0	0	0	2
HBase	5[3]	3[1]	1	0	36	1	2	48[4]
HDFS	3	0	0	0	1	3	1	8
Hive	2	0	0	0	10	1	0	13
MapReduce	7[2]	3	3	2	19	3	2	39[2]
Tez	1	0	0	0	2	0	0	3
Yarn	9	0	4	0	71	2	3	89
Total	28[5]	6[1]	9	2	141	10	8	204[6]

Evaluation Results

- **204** unique violations (**198** new), corresponding to **58** bugs (**52** new) and **6** have been confirmed
- **170** out of 204 detected violations (**83.3%**) exhibit **implicit symptoms**
- **192** new violations (**49** new bugs) and **5** know bugs race on **server instance variables**

System	EO	EA	NP	OB	UR	MR	TO	Total
Alluxio	0	0	0	0	2	0	0	2
Hadoop	1	0	1	0	0	0	0	2
HBase	5[3]	3[1]	1	0	36	1	2	48[4]
HDFS	3	0	0	0	1	3	1	8
Hive	2	0	0	0	10	1	0	13
MapReduce	7[2]	3	3	2	19	3	2	39[2]
Tez	1	0	0	0	2	0	0	3
Yarn	9	0	4	0	71	2	3	89
Total	28[5]	6[1]	9	2	141	10	8	204[6]

Module-Specific Result Analysis

- Static Analysis
  1. **685** potential violation tuples
  2. After error pattern matching, **239** unique violations
- Test Generation
  1. Error-free test cases for **228** out of 239 violations
  2. **6 FNs** for the remaining 11 violations
- Parameter Fuzzing and Interleaving Exploration
  1. Exposes **204** true violations in total
  2. Stage I prunes **12 unreachable** violations and **5 FNs caused by insufficient call sequence**
  3. Stage II exposes **167** true violations and prunes **2 synchronization-protected cases**
  4. Stage III exposes **37** violations and **5 benign races** are pruned

Conclusion

- This paper presents Themis, a framework for detecting distributed concurrency bugs
- Themis integrates static race detection, LLM-based test harness generation, and parameter fuzzing with interleaving exploration to substantially improve detection coverage
- We apply Themis on eight widely used distributed systems and detect **58** bugs, including **52** previously unknown ones

Artifact

Thanks!

Contact: hejzh1@shanghaitech.edu.cn
caohch2023@shanghaitech.edu.cn