Static Analysis of Remote Procedure Call in Java Programs

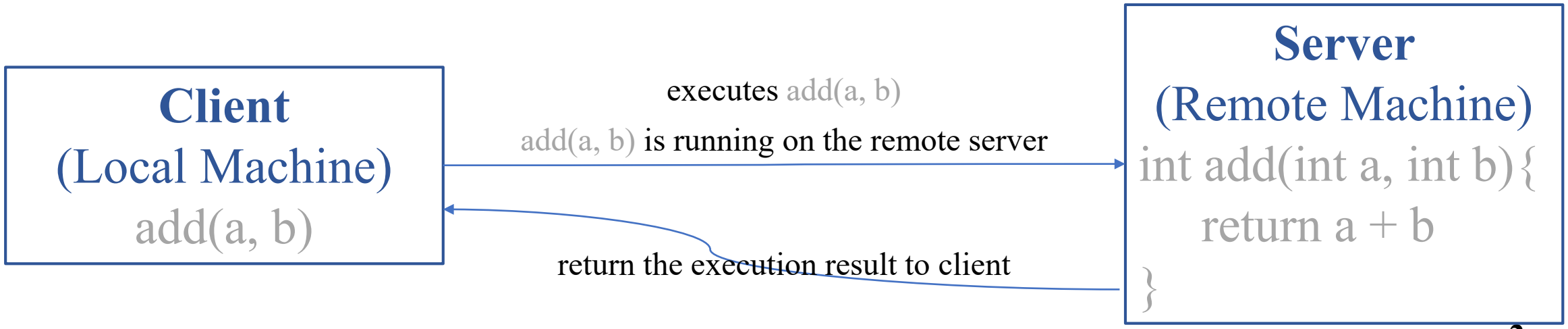
@ICSE2025

Baoquan Cui, Rong Qu, Zhen Tang, Jian Zhang

Institute of Software, Chinese Academy of Sciences (ISCAS)
University of Chinese Academy of Sciences (UCAS)
{cuibq, zj}@ios.ac.cn

Remote Procedure Calls (RPC)

- A program executes a procedure (subroutine) in a different address space
  - commonly on another computer on a shared computer network
- written as if it were a normal (local) procedure call
- without the programmer explicitly writing the details for the remote interaction



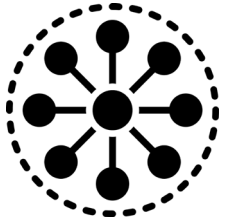
RPC in Java Programs

- Remote Procedure Calls (RPC) Widely Used

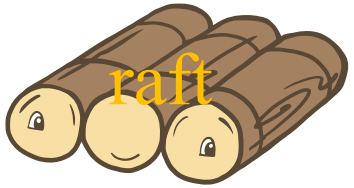
Distributed Data Store

Network Filesystem

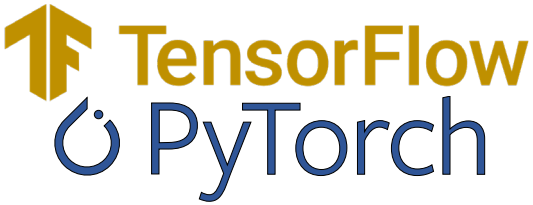
Data Analytics Framework



Cluster Orchestrator



Consensus Protocol



Deep Learning System

RPC in Java Programs

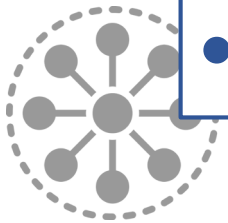
- Remote Procedure Calls Widely Used

Distributed Da

s Framework

In Google's datacenter, RPCs

- **generate >95% of application traffic[1]**
- **spend ~10% of its CPU cycles[2]**



Cluster Orchestrator

Consensus Protocol

orFlow

PyTorch

Deep Learning System

[1] Aequitas: Admission Control for Performance-Critical RPCs in Datacenters, SIGCOMM '22
[2] Profiling a Warehouse-Scale Computer, ISCA '15

RPC Example between Client & Server

0. Protocol between Server and Client

①Protocol

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

RPC Example between Client & Server

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

①Protocol

1. Server Side

```
8  // 1.1 Implement the protocol
9  public class CalculationImpl implements CalculationProtocol{
10     public int add(int a, int b){
11         SINK(a); // sink point
12         return a + b;
13     }
14 }
15 // 1.2 Bind a handler for the protocol and  start
16 String address = "127.0.0.1";
17 Server server = CREATESERVER(address);
18 CalculationProtocol handler = new  CalculationImpl();
19 server.bind(CalculationProtocol.class, handler);
20 server.startListen( (args)->{
21     int a,b = DESERIALIZE(args);
22     int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23     server.response(sum1);
24 });
```

② Implementation in Remote Server

③ Binding & Response

RPC Example between Client & Server

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

①Protocol

1. Server Side

```
8   // 1.1 Implement the protocol
9   public class CalculationImpl implements CalculationProtocol{
10      public int add(int a, int b){
11          SINK(a); // sink point
12          return a + b;
13      }
14  }
15  // 1.2 Bind a handler for the protocol and  start
16  String address = "127.0.0.1";
17  Server server = CREATESERVER(address);
18  CalculationProtocol handler = new  CalculationImpl();
19  server.bind(CalculationProtocol.class, handler);
20  server.startListen( (args)->{
21      int a,b = DESERIALIZE(args);
22      int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23      server.response(sum1);
24  });
```

② Implementation in Remote Server

③ Binding & Response

2. Client Side

```
27  // 2.1 create a client and connect to the server
28  Client client = CREATECLIENT();
29  String address = "127.0.0.1";
30  client.connect(address);
31  // 2.2 create an RPC caller instance
32  CalculationProtocol proxy =CREATERPCPROXY(
        CalculationProtocol.class, (args) -> {
33          String serializeObject = SERIALIZE(args);
34          return client.send(serializeObject);
35  }});
36  // 2.3 invoke an RPC method
37  int a = SOURCE(), b=5; // source point
38  int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
    the client, and obtain the  sum from the server remotely.
```

④ Call from Client

Challenges to Static Analysis

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

- Direct impact: impossible to determine which remote method responds to the local method

1. Server Side

```
8  // 1.1 Implement the protocol
9  public class CalculationImpl implements Calculat
10     public int add(int a, int b){
11         SINK(a); // sink point
12         return a + b;
13     }
14 }
15 // 1.2 Bind a handler for the protocol and  start
16 String address = "127.0.0.1";
17 Server server = CREATESERVER(address);
18 CalculationProtocol handler = new  CalculationImpl();
19 server.bind(CalculationProtocol.class,  handler);
20 server.startListen( (args)->{
21     int a,b = DESERIALIZE(args);
22     int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23     server.response(sum1);
24 });
```

```
34         return client.send(serializeObject);
35 }}});
36 // 2.3 invoke an RPC method
37 int a = SOURCE(), b=5; // source point
38 int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
   the client, and obtain the  sum from the server remotely.
```

Challenges to Static Analysis

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

- Direct impact: impossible to determine which remote method responds to the local method
- Indirect impact:
  - Source from the client (line 37)
  - Sink in the server (line 11)
  - Leak path is missed by taint analyzers

1. Server Side

```
8  // 1.1 Implement the protocol
9  public class CalculationImpl implements Calculat
10     public int add(int a, int b){
11         SINK(a); // sink point
12         return a + b;
13     }
14 }
15 // 1.2 Bind a handler for the protocol and  start
16 String address = "127.0.0.1";
17 Server server = CREATESERVER(address);
18 CalculationProtocol handler = new  CalculationI
19 server.bind(CalculationProtocol.class,  handler);
20 server.startListen( (args)->{
21     int a,b = DESERIALIZE(args);
22     int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23     server.response(sum1);
24 });
```

SINK

SOURCE

Client

```
36  // 2.3 invoke an RPC method
37  int a = SOURCE(), b=5; // source point
38  int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
    the client, and obtain the  sum from the server remotely.
```

RPC

- **Observations**

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

1. Server Side

```
8   // 1.1 Implement the protocol
9   public class CalculationImpl implements CalculationProtocol{
10      public int add(int a, int b){
11          SINK(a); // sink point
12          return a + b;
13      }
14  }
15  // 1.2 Bind a handler for the protocol and  start
16  String address = "127.0.0.1";
17  Server server = CREATESERVER(address);
18  CalculationProtocol handler = new  CalculationImpl();
19  server.bind(CalculationProtocol.class,  handler);
20  server.startListen( (args)->{
21      int a,b = DESERIALIZE(args);
22      int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23      server.response(sum1);
24  });
```

2. Client Side

```
27  // 2.1 create a client and connect to the server
28  Client client = CREATECLIENT();
29  String address = "127.0.0.1";
30  client.connect(address);
31  // 2.2 create an RPC caller instance
32  CalculationProtocol proxy =CREATERPCPROXY(
        CalculationProtocol.class, (args) -> {
33          String serializeObject = SERIALIZE(args);
34          return client.send(serializeObject);
35  }});
36  // 2.3 invoke an RPC method
37  int a = SOURCE(), b=5; // source point
38  int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
    the client, and obtain the  sum from the server remotely.
```

- The variable *proxy* points to the object *handler*

RPC

- **Observations**

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

1. Server Side

```
8   // 1.1 Implement the protocol
9   public class CalculationImpl implements CalculationProtocol{
10      public int add(int a, int b){
11          SINK(a); // sink point
12          return a + b;
13      }
14  }
15  // 1.2 Bind a handler for the protocol and  start
16  String address = "127.0.0.1";
17  Server server = CREATESERVER(address);
18  CalculationProtocol handler = new  CalculationImpl();
19  server.bind(CalculationProtocol.class,  handler);
20  server.startListen( (args) -> {
21      int a,b = DESERIALIZE(args);
22      int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23      server.response(sum1);
24  });
```

2. Client Side

```
27  // 2.1 create a client and connect to the server
28  Client client = CREATECLIENT();
29  String address = "127.0.0.1";
30  client.connect(address);
31  // 2.2 create an RPC caller instance
32  CalculationProtocol proxy =CREATERPCPROXY(
        CalculationProtocol.class, (args) -> {
33          String serializeObject = SERIALIZE(args);
34          return client.send(serializeObject);
35  }});
36  // 2.3 invoke an RPC method
37  int a = SOURCE(), b=5; // source point
38  int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
    the client, and obtain the "sum" from the server remotely.
```

- The variable *proxy* points to the object *handler*
- *Proxy.add* actually executes *handler.add*

RPC

- **Observations**

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

1. Server Side

```
8   // 1.1 Implement the protocol
9   public class CalculationImpl implements CalculationProtocol{
10      public int add(int a, int b){
11          SINK(a); // sink point
12          return a + b;
13      }
14  }
15  // 1.2 Bind a handler for the protocol and  start
16  String address = "127.0.0.1";
17  Server server = CREATESERVER(address);
18  CalculationProtocol handler = new  CalculationImpl();
19  server.bind(CalculationProtocol.class,  handler);
20  server.startListen( (args) -> {
21      int a,b = DESERIALIZE(args);
22      int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23      server.response(sum1);
24  });
```

```
27  // 2.1 create a client and connect to the server
28  Client client = CREATECLIENT();
29  String address = "127.0.0.1";
30  client.connect(address);
31  // 2.2 create an RPC caller instance
32  CalculationProtocol proxy =CREATERPCPROXY(
        CalculationProtocol.class, (args) -> {
33          String serializeObject = SERIALIZE(args);
34          return client.send(serializeObject);
35  }});
36  // 2.3 invoke an RPC method
37  int a = SOURCE(), b=5; // source point
38  int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
    the client, and obtain the "sum" from the server remotely.
```

- The variable *proxy* points to the object *handler*
- *Proxy.add* actually executes *handler.add*
- The variable sum_2 is the value sum_1 from the server after the execution

RPC

- Observations

0. Protocol between Server and Client

```
public interface CalculationProtocol { //  interface
    public int add(int a, int b);
}
```

1. Server Side

```
8   // 1.1 Implement the protocol
9   public class CalculationImpl implements CalculationProtocol{
10      public int add(int a, int b){
11          SINK(a); // sink point
12          return a + b;
13      }
14  }
15  // 1.2 Bind a handler for the protocol and  start
16  String address = "127.0.0.1";
17  Server server = CREATESERVER(address);
18  CalculationProtocol handler = new  CalculationImpl();
19  server.bind(CalculationProtocol.class, handler);
20  server.startListen( (args) -> {
21      int a,b = DESERIALIZE(args);
22      int sum1 = handler.add(a, b); //execute "add(a,b)" on Server
23      server.response(sum1);
24  });
```

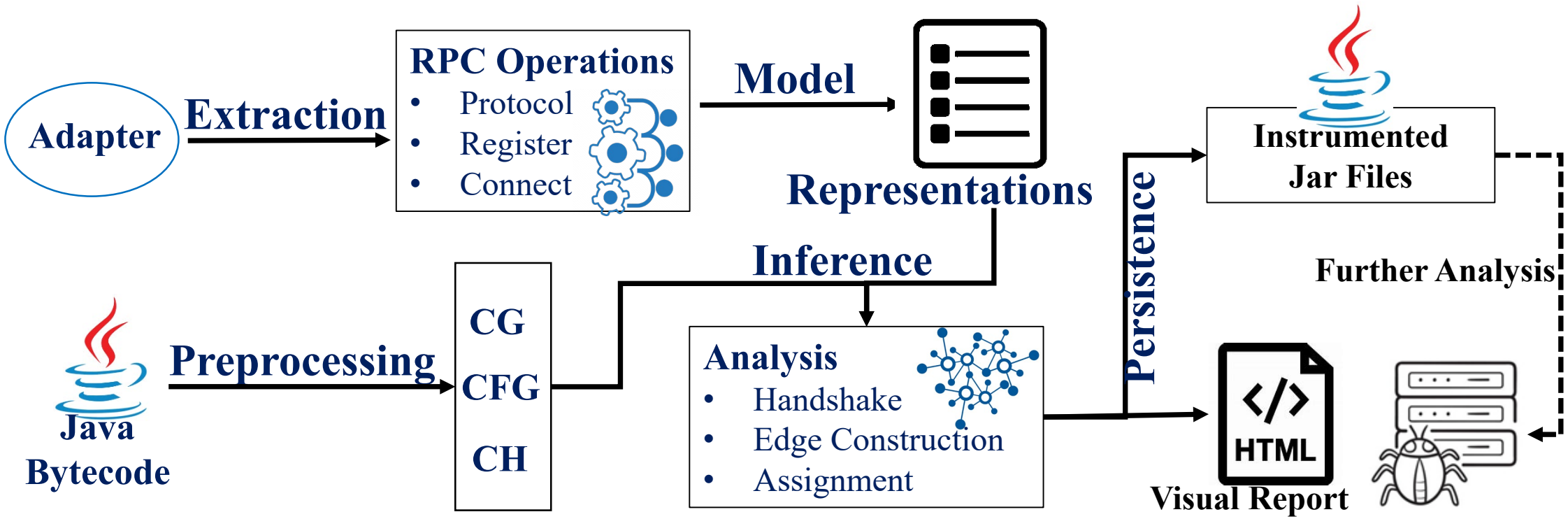
2. Client Side

```
27  // 2.1 create a client and connect to the server
28  Client client = CREATECLIENT();
29  String address = "127.0.0.1";
30  client.connect(address);
31  // 2.2 create an RPC caller instance
32  CalculationProtocol proxy =CREATERPCPROXY(
        CalculationProtocol.class, (args) -> {
33          String serializeObject = SERIALIZE(args);
34          return client.send(serializeObject);
35  }});
36  // 2.3 invoke an RPC method
37  int a = SOURCE(), b=5; // source point
38  int sum2 = proxy.add(a,b); //invoke "add(a,b)" method in
    the client, and obtain the "sum from the server remotely.
```

- Automatically builds the connections

- The variable *proxy* points to the object *handler*
- *Proxy.add* actually executes *handler.add*
- The variable sum_2 is the value sum_1 from the server after the execution

Overview


Adapter
Extraction
RPC Operations
• Protocol
• Register
• Connect
Model
Representations
Inference
Java
Bytecode
Preprocessing
CG
CFG
CH
Analysis
• Handshake
• Edge Construction
• Assignment
Persistence
Instrumented
Jar Files
Further Analysis
HTML
Visual Report

Static Analysis of Remote Procedure Call

- **Basic Representations**

Representation	Description
Call(*i*: *I*, *s*: *S*)	instruction *i* is a call to a method, whose signature is *s*
ActualArg(*i*: *I*, *n*: *N*, *v*: *V*)	at invocation *i*, the *n*-th parameter is local variable *v*. For virtual calls, the variable this is the first (0-th) element
FormalParam(*m*: *M*, *n*: *N*, *v*: *V*)	the variable *v* is the *n*-th formal parameter of the method *m*. The receiver is the 0-th parameter for virtual calls
AssignRetValue(*i*: *I*, *v*: *V*)	at invocation *i*, the value returned by the invocation is assigned to the local variable *v*
ReturnValue(*m*: *M*, *v*: *V*)	the variable *v* is the one (assumed single) returned by the method *m*
ObjType(*o*: *O*, *t*: *T*)	the object *o* has the type *t*
LookUp(*s*: *S*, *t*: *T*, *m*: *M*)	in type *t*, there exists a method *m* with the signature *s*; a sub-signature also works
VarPointsTo(*v*: *V*, *o*: *O*)	a variable *v* points to the object *o*
ParaListSize(*size*: *N, s*: *S*)	the number *size* indicates the size of the parameter list of the method whose signature is *s*
CallGraphEdge(*i*: *I, m*: *M*)	the method *m* is called at the instruction *i*

Static Analysis of Remote Procedure Call

- **Semantic Modeling of RPC Operations**

Representation	Description
PROTOCOLREGISTER(t_{proto}: T, $o_{handler}$: O, ip: N)	register the protocol type t_{proto} in the server with the IP address ip and bind the corresponding handling object $o_{handler}$
RPCCONNECT($o$proxy: O, ip: N)	connect the server with the IP address ip by the RPC caller o_{proxy}
REIFIEDRPCINSTANCE(t_{proto}: T, v: V)	v is the variable representing the proxy instance of the protocol type t_{proto}, which can launch an RPC invocation
SUBSIGNATURE(s: S, s_{sub}: S)	the s_{sub} is the sub-signature of the signature s, they have the same method declaration except for the information of the class they belong to
RPCCALLINFO(s: S, v_{proxy}: V, i: I)	a call instruction i invokes an RPC method m, whose instance caller is v_{proxy}
RPCOBJECTHANDLER(o_{proxy}: O, $o_{handler}$: O)	abstract RPC caller o_{proxy} has its invocation handled remotely by the corresponding method of the object $o_{handler}$ on the server with the same subsignature

Static Analysis of Remote Procedure Call

- **Adapter (Operations)**
  - **Register (Server)**
    - m_s <...ipc.RpcEngine: RPC.Server getServer(java.lang.Class,Java.lang.Object,...)>

    Call(i, $\boldsymbol{m_s}$), ActualArg (i, 1, v_0), ActualArg(i, 2, v_1),
    VarPointTo(v_0, t_{proto}), VarPointTo(v_1, o_{handler})
    → ProtocolRegister(t_{proto}, o_{handler}, -)
  - **Connect (Client)**
    - m_c <...ipc.RPC: <T> T waitForProxy(java.lang.Class,long,...)>

    Call(j, $\boldsymbol{m_c}$),
    ReturnVar(j, $v_{\textbf{proxy}}$), VarPointTo($v_{\textbf{proxy}}$, $o_{\textbf{proxy}}$)
    → RPCConnection(o_{proxy}, -)

Static Analysis of Remote Procedure Call

Points-to Analysis

$\textsc{VarPointsTo}(v_{proxy}, o_{handler})$,
$\textsc{RPCObjectHandler}(o_{proxy}, o_{handler})$
$\longleftarrow$
$\textsc{ReifiedRPCInstance}(t_{proto}, v_{proxy})$
$\textsc{RPCConnection}(o_{proxy}, ip)$,
$\textsc{ProtocolRegister}(t_{proto}, o_{handler}, ip)$,
$\longleftarrow$
$\textsc{Call}(i,$ "Server.register"$)$, $\textsc{ActualArg}(i, 1, t_{proto})$,
$\textsc{ActualArg}(i, 2, o_{handler})$, $\textsc{ActualArg}(i, 3, ip)$,
$\textsc{Call}(j,$ "Client.connect"$)$, $\textsc{ActualArg}(j, 1, o_{proxy})$,
$\textsc{ActualArg}(j, 2, ip)$, $\textsc{ObjType}(o_{proxy}, t_{proto})$
$\textsc{VarPointsTo}(v_{proxy}, o_{proxy})$,
$\textsc{VarPointsTo}(v_{handler}, o_{handler})$

(a) Handshake

RPC Edge Construction

$\textsc{CallGraphEdge}(i, m_{handler})$, $\textsc{VarPointsTo}(p_n, o_n)$,
$\textsc{RPCCallInfo}(s, v_{proxy}, i)$
$\longleftarrow$
$\textsc{ReifiedRPCInstance}(t_{proto}, v_{proxy})$
$\textsc{VarPointsTo}(v_{proxy}, o_{handler})$,
$\textsc{Call}(i, s =$ "the method invoked by the variable v_{proxy}"$)$,
$\textsc{SubSignature}(s, s_{sub})$, $\textsc{ObjType}(o_{handler}, t)$,
$\textsc{LookUp}(s_{sub}, t, m_{handler})$, $\textsc{ParaListSize}(size, s)$,
Each n from 0 to $(size - 1)$
$\textsc{FormalParam}(m_{handler}, n, p_n)$
$\textsc{ActualArg}(i, n, v_n)$, $\textsc{VarPointsTo}(v_n, o_n)$,

(b) RPC Edge Construction

Evaluation

- **Dataset (5 RPC Frameworks: Hadoop-common/gRPC/dubbo/RMI/Thrift)**

Fr.W	Project	Version	#Cls	#KLOC	#Fork	#Star
Hadoop	hadoop	3.4.0	122K	4,284	8.7K	14.4K
	hbase	3.0.0-beta-1	158K	5,949	3.3K	5.1K
	ozone	1.4.0	170K	6,112	474	772
	phoenix	2.5.0	193K	6,717	993	1K
	pravega	0.13.0	95K	3,220	404	2K
	tez	0.10.3	41K	1,498	415	463
gRPC-bench		C(64ac792)	20K	817K	3.8K	11.3K
dubbo-bench (157)		C(0ef8eae)	186K	6,575K	1.9K	2.2K
RMI	pax.exam (51)	4.13.5	20K	817K	100	84
	jmeter	5.6.3	43K	1,618K	2.1K	8.1K
	rmi-jndi (12)	C(bc82c67)	5K	173K	48	306
Thrift	cassandra	3.11.11	17K	585K	3.6K	8.6K

Evaluation

- **RPC Identification**

Project	#Pro	#CP	#H	#SP	#MP	#CMP	#RPC	#CRPC	#+E	#CE	T (s)	AT (s)
hadoop	111	53	97	37	37	20(59.5%)	171	121(70.4%)	768	535(69.7%)	6.3 (0.07%)	9,332
hbase	72	43	56	28	28	15(53.6%)	184	95(51.6%)	536	279(52.1%)	9.8 (0.31%)	3,155
ozone	76	35	55	35	35	19(54.3%)	238	142(59.7%)	670	362(54.0%)	8.9 (0.27%)	3,309
phoenix	30										8.2 (0.27%)	2,997
pravega	31										3.3 (0.03%)	11,607
tez	58										5.8 (0.17%)	3,447
gRPC-bench	5										1.3 (0.86%)	152
dubbo-bench	62										50.8 (0.41%)	12,335
pax exam	24										3.8 (0.22%)	1,723
jmeter	1										1.8 (0.56%)	324
rmi-jndi	17										3.2 (0.17%)	435
cassandra	5	5	5	5	5	5(100%)	27	21(77.8%)	27	21(77.8%)	3.1 (0.25%)	1,236
Total	492	334	398	263	263	166 (63.1%)	1,098	679 (61.8%)	2,578	1,549 (60.1%)	106.3 (0.21%)	50,052

- **263 protocols**
- **1,098 RPC calls**
- **2,578 call edges are added to the CG**

Evaluation

- **Real Caller-Callee**

Project	#Pro	#CP	#H	#SP	#MP	#CMP	#RPC	#CRPC	#+E	#CE	T (s)	AT (s)
hadoop	111	53	97	37	37	20(59.5%)	171	121(70.4%)	768	535(69.7%)	6.3 (0.07%)	9,332
hbase	72	43	56	28	28	15(53.6%)	184	95(51.6%)	536	279(52.1%)	9.8 (0.31%)	3,155
ozone	76	35	55	35	35	19(54.3%)	238	142(59.7%)	670	362(54.0%)	8.9 (0.27%)	3,309
phoenix	30	27	24	20	20	12(60.0%)	92	50(54.3%)	94	50(53.4%)	8.2 (0.27%)	2,997
pravega	31										3.3 (0.03%)	11,607
tez	58										5.8 (0.17%)	3,447
gRPC-bench	5										1.3 (0.86%)	152
dubbo-bench	62										50.8 (0.41%)	12,335
pax exam	24										3.8 (0.22%)	1,723
jmeter	1	1	1	1	1	1(100%)	3	2(66.7%)	3	2(66.7%)	1.8 (0.56%)	324
rmi-jndi	17	17	17	17	17	13 (76.5%)	17	13(76.5%)	17	13(76.5%)	3.2 (0.17%)	435
cassandra	5	5	5	5	5	5(100%)	27	21(77.8%)	27	21(77.8%)	3.1 (0.25%)	1,236
Total	492	334	398	263	263	166 (63.1%)	1,098	679 (61.8%)	2,578	1,549 (60.1%)	106.3 (0.21%)	50,052

About 60% of protocols/RPCs/edges are covered after running

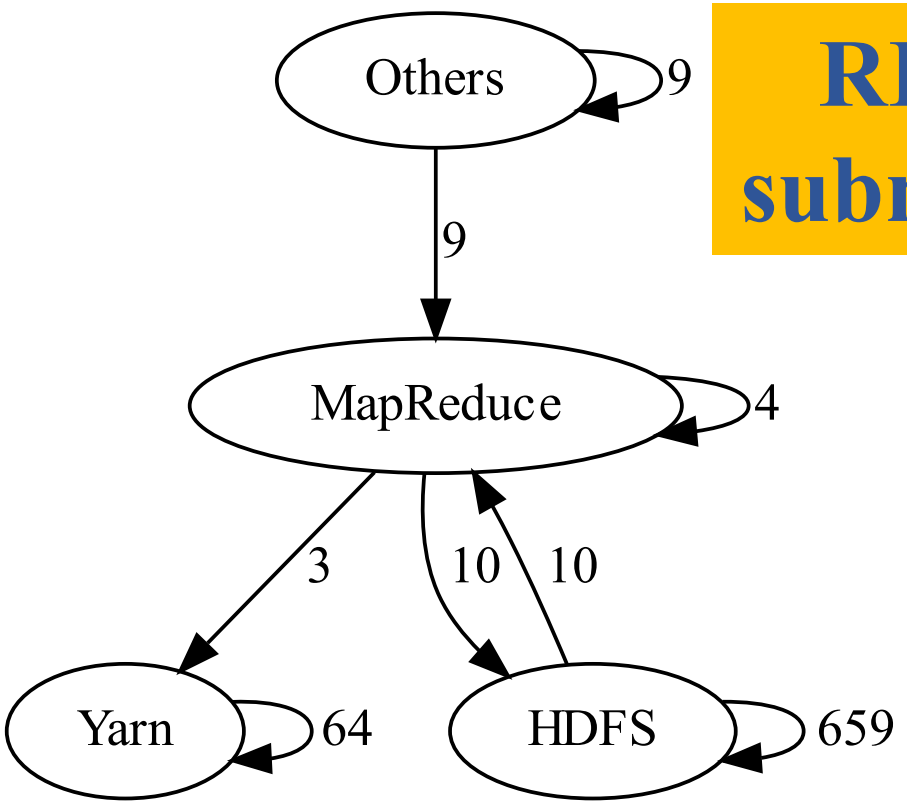
Evaluation

- False Positives (matched protocols in Hadoop)
  - Hadoop prefers to ProtoBuf instead of Writeable.
  - 37 matched protocols
    - 20 are covered (17 un covered)
      - 3 are Writeable from test cases
      - 17 are ProtoBuf
  - 38 Writeable protocols through text search

Assuming: all protocols that are not covered are false positives (false positive upper bound)

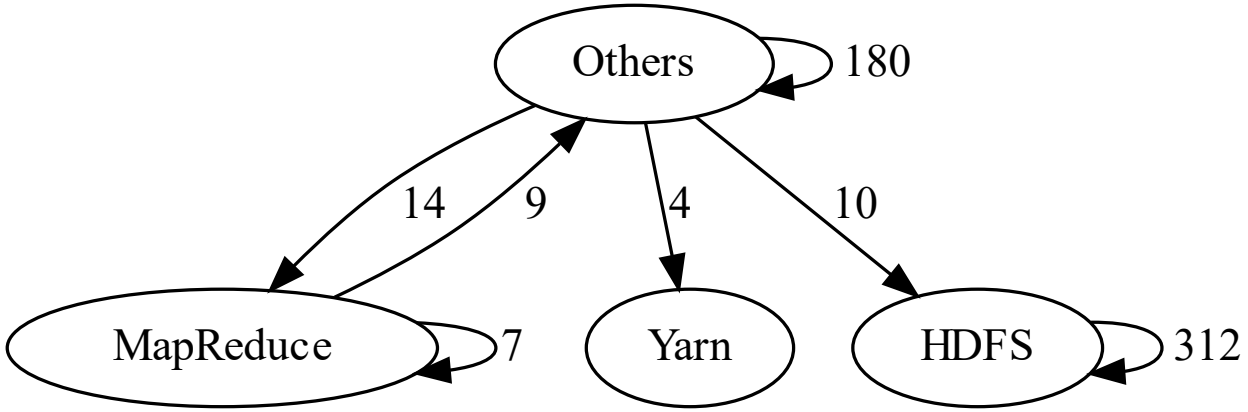
$$\text{FP}(\%) = \frac{37-20}{37+38-3} = 23.6\%$$

Case Study



Hadoop

RPC edges are built to make the submodules in the system connected



HBase

Evaluation

- **Benefits of RPC connectivity**
- **Taint Analyzer： FlowDroid**

Project	Leakage Path(Δ)	Memory(Δ) (GB)	Time(Δ) (s)
hadoop	12 (+4)	16.1 (+0.08)	21 (+4)
hbase	7 (+1)	9.3 (+0.04)	15 (+2)
ozone	1 (0)	20.7 (+0.05)	28 (+4)
phoenix	3 (0)	25.4 (+0.03)	31 (+2)
pravega	-	-	TO
tez	2 (+2)	10.6 (+0.06)	13 (+1)
Total	37 (+9)	82.1 (+0.26))	108 (+13)

Leak Path: +24.3 = 9/37
Memory Cost: +0.26%
Time Cost: +12%

Limitations

- IP addresses are taken into account in our approach, but it is not easy to actually analyze it statically (IP addresses are not considered in our experiments)
- The size of most of the actual programs with RPC are large, and analyzing them requires a lot of hardware resources
- RPC frameworks can support cross-language communication, but static analysis of cross-language programs is difficult
- Adapters building for the RPC framework require a priori expert knowledge
  - Deep learning or large language models may be able to get this knowledge automatically

Static Analysis of Remote Procedure Call in Java Programs

@ICSE2025

Baoquan Cui, Rong Qu, Zhen Tang, Jian Zhang

Institute of Software, Chinese Academy of Sciences (ISCAS)
University of Chinese Academy of Sciences (UCAS)
{cuibq, zj}@ios.ac.cn