Static Analysis of Remote Procedure Call in Java Programs

Baoquan Cui[1,2,3,4], Rong Qu[1,2,3,4], Zhen Tang[1,2,3,4] and Jian Zhang[1,2,3,4]
[1]State Key Laboratory of Computer Science, [2]Key Laboratory of System Software (CAS)
[3]Institute of Software, Chinese Academy of Sciences (CAS), [4]University of CAS, Beijing, China
Email: {cuibq, qurong, zj}@ios.ac.cn, tangzhen12@otcaix.iscas.ac.cn

***Abstract*—The Remote Procedure Call (RPC) is commonly used for inter-process communications over network, allowing a program to invoke a procedure in another address space even another machine as if it were a local call within the same address space. Its convenience comes from encapsulating network communication. However, for the same reason, it cannot be penetrated by current static analyzers. Since the RPC programs/frameworks play a more important role in various domains, the static analysis of RPC is significant and cannot be ignored.**

We have observed that many of the existing RPC frameworks/programs written in Java are based on explicit protocols, which makes them possible to be modelled for static analysis. The challenges are how to identify RPC operations in different frameworks/programs and how to automatically establish relationships between clients and servers. In this paper, we propose a novel approach, RPCBridge, which uses an adapter to unify the most basic operations during the RPC process. It models the RPC with logic rules in a straightforward and precise way based on its semantics, performs points-to analysis and constructs RPC edges in the call graph, making it more complete. The evaluation on real-world large-scale Java programs based on 5 common RPC frameworks shows that our approach can effectively capture the operations of the RPC (263 matched protocols and 1,098 RPCs), and construct critical links (2,578 edges in the call graph) between clients and servers, in which 60.1% are the true caller-callee pairs after execution. Our approach is expected to bring significant benefits (+24.3% leakage paths for the taint analyzer) for previously incompletely modelled code with a very little memory and time overhead, and connect the modules in a system, so that it can be statically analyzed more holistically.

***Index Terms*—Remote Procedure Call (RPC), Static Analysis, Points-to Analysis**

I. Introduction

The Remote Procedure Call (RPC) is a common mechanism for implementing network/process communication in network programming. When a computer program executes a procedure or subroutine in another address space (usually on another computer on a shared network), the remote address space call is coded as a local procedure call (LPC). This means that programmers write essentially the same code without worrying about whether the program is running locally or remotely, making the intricacies of network communication transparent to developers.

However, the convenience of using RPC in programs brings challenges to static analysis. It is difficult to construct a connected edge for the call graph (CG) for the RPC invocation, because it is usually difficult to figure out which method remotely responded to the RPC call after the parameter serialization and network API invocation. A disconnected or incomplete CG will lead to imprecise and unsound results and hinder the static analysis of the program. Modern static analyzers for Java programs are mature for the normal points-to analysis [1–3], call graph construction and value flow analysis, even for the dynamic features: reflection and dynamic proxy [4–9]. Unfortunately, they ignore the RPC and give it far less attention, even though it is prevalent in practice and becomes a valuable feature in network programming.

Traditionally, if a method invocation occurs via the network communication, such as the Socket or HTTP, static analysis will run into troubles, because (1) variables and objects are difficult to trace after serialization and deserialization. The type of an object may be changed after the data translation without any inheritance relationship, and the object as data may even be disrupted and reorganized; and (2) the local caller method and the remote callee method may not be linked, losing edges on the CG. The consequence is to reduce the ability of the static analyzer. For example, if a variable is tainted on the client and after being propagated via the RPC it is actually sunk (sink point) on the remote server, then this leakage path cannot be detected by the taint analyzer due to the unsound value flow analysis and the incomplete CG. There are even real exceptions [10, 11] in Hadoop [12], crashes [13, 14] in gRPC [15], and possible memory leaks [16, 17] on the RPC server which can only be triggered by the client.

However, for the RPC network communication, we have observed that many existing RPC frameworks/programs are based on explicit protocols, such as RMI [18], gRPC, Thrift [19], Dubbo [20], Hadoop and so on. They are explicitly or implicitly guided by the specifications [21–27] for RPC from the Request for Comments (RFC) Series [28]. This feature means that the client and server have a consensus protocol that supports the implementation and use of the RPC. Meanwhile, that makes it possible to perform static analysis of (1) the variables and objects involved: because it forces these objects and variables to be of the same type and (2) the connection between the caller method and the callee method: since they have the same method declarations (signatures) on the server and the client as specified in the protocol.

In this paper, we propose a novel approach, RPCBridge, to perform static analysis on the RPC programs written in Java, connecting the client and the server in one system, which will make the further analysis more holistic. However, how to automatically identify RPC operations in different frameworks/programs is a challenge. Another challenge is how to overcome the analysis of the complex network communication to establish a relationship between the client and the server and to determine which remote method is the actual one responding to a local RPC. For the first challenge, we use an adapter to unify the most basic operations during the RPC process, registration and connection, to be compatible with different frameworks. We then model the semantics of the operations under the RPC mechanism by using formal representations. For the other challenge, we build logical rules to automatically infer the virtual points-to relationships between the variables/objects from both the client and the server, to find the exact remote response for a local RPC, based on the representations. It is a virtual points-to relationship because these variables and objects are not in the same address space or on the same device actually. In the following sections, we will ignore this virtual limitation and treat them as if they were all in the same address space, just like the RPC mechanism. Points-to analysis represents a points-to relation $ptt \subseteq V \times O$ where V is the variable set and O is the heap abstraction set, forming the substrate of most inter-procedural static analysis. Thus, our approach can benefit other static analyzers, since it is a full capture of the behaviors on the object flow and the method invocation under the RPC mechanism. In addition, we provide auxiliary algorithms to persist the analysis result back into the program file as one of the outputs, making the use of `RPCBridge` non-intrusive for the further analysis. Finally, we use experiments to demonstrate the effectiveness of `RPCBridge` and the gains it can bring for the static analyzer, and use a case study to illustrate its critical contribution to the inter-module connection in a system.

In summary, our contributions are as follows:

- We propose `RPCBridge`, a novel static analysis approach to connect the client and the server under the RPC programs in one system. It introduces an adapter to unify RPC operations in different frameworks/programs, and models the semantics of the operations with formal representations. With the representations, it conducts inference for the points-to analysis on the objects/variables during the RPC process and constructs the RPC edge in the call graph. Based on the analysis result, it provides persistence algorithms for the further analysis.
- We evaluate `RPCBridge` on real-world large-scale open source programs, and the experiment shows that it can effectively capture the operations of the RPC in the programs, and construct critical links between the clients and the servers, benefiting taint analysis to connect the modules in a system together and making the analysis no longer in isolation.

II. Background

In this section, we will introduce the RPC mechanism including how it works and use a motivating example to demonstrate the difficulties it brings for traditional static analyzers, and the opportunities we see for the static analysis.

A. Remote Procedure Call

The RPC mechanism is a powerful technique for constructing distributed or client-server (C/S) based applications, allowing a program to invoke a procedure in another address space as if it were a local call within the same address space.

A client and a server are deployed on different machines, and a specified protocol is the cornerstone of collaboration between them. ① When the client starts a method invocation ($Call_i$), it invokes a client proxy, passing parameters after serialization in the usual way. ② Then client proxy passes the message to the transport layer via Socket or HTTP, sending it to the remote server machine. ③ On the server, a listener will receive the call, and all calls will be pushed into the call queue (or pool) waiting for processing by a scheduler. ④ The scheduler pops the call from the queue, and transfers the task to the corresponding handler ($handler_i$) according to the protocol after deserialization. The handler is the desired server routine and is executed as the regular procedure call. ⑤ When the handler completes, its return result will be transferred with serialization as the response. ⑥ Finally, the client proxy receives and deserializes the response and returns execution to the caller ($Call_i$).

Although this workflow may seem complicated, due to the support of the RPC mechanism, the data serialization and deserialization, the network connection and determination of the appropriate handler are standardized. That is to say, these processes are hidden and transparent to the developers. All they have to do is to focus on the protocol and its implementation, and call the RPC method in the protocol as if it were a local invocation.

B. Motivating Example

We will show a simplified example of using the RPC in Listing 1. In addition to instantiating the workflow above, it serves to emphasize that while the practical implementation of the RPC is sophisticated with data conversion and network communication, its use is simple. Whereas the former hinders a more comprehensive static analysis, the latter enlightens us for further analysis.

Consider the code in Listing 1, it contains three parts.

- **Protocol.** The protocol (`CalculationProtocol`, lines 3-5) is agreed upon by the server and the client. It is the prerequisite of the precise RPC communication between the local machine and the remote one. The protocol is an interface for calculation, declaring a method *add(...)* with integer parameters (*a* and *b*) and the integer return type.
- **Server Side.** Firstly, the server must implement the protocol, providing the specific execution of the task *add(...)* (lines 8-14). There is a sink point in the implementation, *SINK(a)*, for the variable *a* (line 11). Then, the server initializes itself with the IP address, instantiates a handler (**handler**) for the protocol and binds them together (lines, 15-19). Finally, it starts listening with a callback (simplified by a lambda expression, the syntax sugar for an anonymous function, denoted as (args)→{body}), which deserializes the argument into the parameters for invoking the method

Listing 1. A Motivating Example

```
1   0. Protocol between Server and Client
2
3   public interface CalculationProtocol { //
        interface
4      public int add(int a, int b);
5   }
6   1. Server Side
7
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  });
25  2. Client Side
26
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.
```

add(...) by the handler, and returns the calculation result (**sum**$_1$) to the requester (lines 20-24).

- **Client Side.** At the beginning, the client connects the server after its instantiation (lines 27-30). Later, it creates a proxy object which specifies the protocol to a proxy (**proxy**) and the processing callback (also simplified by a lambda expression) for the agent. When the callback is triggered, it serializes the arguments as a message to be sent to the remote server (lines 32-35). Then, the proxy invokes the method *add(...)* with arguments (*a* and *b*) to perform the computation task (lines 36-38) and obtains the result (**sum**$_2$), where variable *a* is derived from a source invocation (*i.e.*, a = SOURCE(), source point, line 37), and the task will be answered by the remote server after the handler execution (lines 21-23).

For the traditional static analyzer, it can perform the analysis on regular variables and objects, such as *a, b, address* and so on, and even can construct an edge into the CG from the proxy invocation (line 37) to the proxy callback (line 31) in the client [8]. However, the analysis comes to an abrupt end when the object is serialized and deserialized as data, especially during network communication. It makes the analysis break down on the variables, **handler**, **sum**$_1$, **sum**$_2$ and **proxy**. Subsequently, the client call and the server response is disconnected, leaving the CG incomplete and each module in the system to be analyzed in isolation. In addition, an incomplete CG can cause false negatives and false positives during static analysis, which is an implicit impact. For example, there is a taint sink point (line 11) in the implementation of the method *add(int,int)* and there is also a source point for the actual argument *a* of the invocation of method *add(int,int)*, but there is not an edge in the call graph to connect the method invocation and its real implementation, thus a leak path will be missed by the taint analysis. In addition to the taint analysis above, the mutual influence between the server and the client is inevitable and can easily cause crashes during the RPC. There are also some real bugs that can only be triggered from the RPC client, such as "Unexpected INTERNAL error propagated to application layer", which forces developers to retry pull operation polluting logs with errors/warnings along the way [13] and "StatusException" [14] in gRPC, repeated sub-directory creation during the RPC "getContentSummary" [11] in Hadoop, and possible memory leak [16, 17]. An isolated analysis of the client or the server will miss these defects.

In fact, based on the knowledge of RPC mechanism, we can easily observe that the variable **proxy** is the alias of the variable **handler**, since the former one is the caller and the latter one is the callee specified by the protocol agreed between the client and the server. In the same way, the variables **sum**$_1$ and **sum**$_2$, are aliases of each other as well. Our goal is to make the data serialization and deserialization, and network communication transparent to the static analyzer, just as the implementation of RPC is transparent to the developer. That is, to construct a points-to relationship between the variable **proxy** (line 32) and the object created in the location of the instantiation for the variable **handler** (line 18). Thus, an edge can be constructed, from the call site (line 38) to the actually executed method (lines 10-13), making the CG more complete and the analysis on the modules less isolated. In the next section, we will show how to model the semantics of the RPC and infer the relations automatically.

III. RPCBridge

Figure 1 shows the overview of our approach, RPCBridge. It takes the Java bytecode and an adapter as inputs. The adapter is an abstraction of the RPC operations to unify behaviors in RPC programs since different programs/frameworks have different implementation stacks of the RPC mechanism. It contains basic operations under the mechanism: registration and connection as described by the RPC specification [21]. With the basic operations, we model their semantics via the

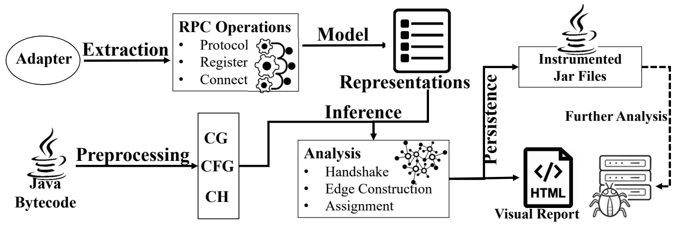


Fig. 1. Overview of `RPCBridge`

representations written by the language in Prolog style (Section III-A), then perform inference on the points-to analysis based on the representations (Section III-B). Finally, we output the result into a visual HTML report and persist it back to the input bytecode for further analysis (Section III-C).

A. Model

We use the language with the Prolog style for modelling since it is mature and easy to read for static analysis with a long history, providing a high-level abstraction of the semantic view [29]. In particular, the Java Virtual Machine (JVM) (SE 8 Edition) specification also uses it to describe the formal clause [30]. Our analysis is built on the top of the existing analysis approach, such as points-to analysis, data flow analysis and so on. That means, we focus on the extra analysis brought by RPC. Thus, in this section, we will introduce (1) the basic existing representation briefly, (2) the representation of modelling RPC invocation.

Basic Representation. At the beginning, the basic rule "$\mathbf{Z}(a,c) \leftarrow \mathbf{X}(a,b), \mathbf{Y}(b,c)$" means if the predicate $\mathbf{X}(a,b)$ and the predicate $\mathbf{Y}(b,c)$ both are true, then the predicate $\mathbf{Z}(a,c)$ is true can be inferred. The symbol "←" separates pre-predicates and post-predicates, and both of them may have more than one predicate. Figure 2 shows the meaning of the basic representation as described in [31]. The preliminaries are present as follows.

- *v* is a variable
- *o* is an object, the heap abstraction
- *m* is a method
- *s* is a signature of a method
- *t* is a type: a class or an interface
- *n* is a number
- *i* is an instruction call site, or the method invocation at the call site

Taking the representation, CallGraphEdge(*i*: *I*, *m*: *M*), as an example, it means the method *m* is invoked at the instruction i. The representation, VarPointsTo(*v*: *V*, *o*: *O*), means a variable *v* points to the object *o*. And, ParaListSize(*size*: *N*, *s*: *S*), computes the size of the parameter list of the method with the signature s.

Representation for RPC Invocation. For the RPC in Java programs, the protocol is usually an interface, and it may have characteristics to be identified, such as inheriting from a specific interface, or it may not, depending on the implementation. However, all RPC programs include the process of registration and connection, which is determined by its mechanism, no matter which platform they are implemented on. We model their semantics by the representations, denoted as ProtocolRegister(t_{proto}: *T*, $o_{handler}$: *O*, *ip*: *N*) and RPCConnection(o_{rpc}: *O*, i: *I*, *ip*: *N*) as shown in Figure 3. The former one indicates that the server registers the protocol type t_{proto} with the IP address *ip* and binds the corresponding handling object $o_{handler}$, and the latter one means the connection to the server with the IP address *ip* before the RPC call site *i* by the local proxy of the protocol t_{proto}. Although the implementation of these two semantics on different platforms does not exactly match two specified methods, we still use this description at a high-level abstraction for convenience, since this is a mere engineering concern (in the actual implementation of the approach, adapters for different platforms are needed). Moreover, the IP address is difficult to obtain during static analysis, we still retain it for the scalability of our approach. In the actual implementation of the approach, its value can be assigned to distinguish different servers or to be ignored by assuming that there is only one server.

The two representations recognize the protocol, its proxy instance in the local client and its handler in the remote server. The proxy instance is denoted as, ReifiedRPCInstance(t_{proto}: *T*, *v*: *V*), representing a local RPC proxy instance for the protocol type t_{proto}, which can launch an RPC invocation (since an interface cannot instantiate objects directly, application of the protocol needs to be implemented through a proxy). Then, the method invocation of the proxy instance is an RPC, denoted as RPCCallInfo(*s*: *S*, o_{proxy}: *O*, *i*: *I*). It is a specific sub-representation of Call(*i*, *s*) with the caller instance o_{proxy}. To deal with the matching between the local RPC method and the remotely executed one, we introduce a computation SubSignature(*s*: *S*, s_{sub}: *S*), to get the sub-signature of a method, removing the class information in it while retaining the method declaration, e.g., the signature is "*<CalculationProtocol: int add(int,int)>*" and its sub-signature is "*int add(int,int)*".

B. Inference

Based on the basic representations and the ones for RPCs, we can model the semantics of the RPC operations: register, connection, local invocation and the remote handler. Over this schema, the further analysis can be captured into three logical rules: (1) the points-to analysis on the local proxy variable and the remote handler (handshake); (2) the RPC edge construction for CG (RPC edge construction); (3) the assignment of the variables returned by the RPC call and the handler (assignment), each of which will be explained individually.

Rule 1: Handshake. We use the word "handshake" to express how the client and server connect under the RPC mechanism during static analysis, based on an agreed protocol. The reasoning rule of the handshake is shown in Figure 4(a). In words, the logic expressed by it says as follows:

(1) if there is a server register instruction, Call(*i*, "Server.register"), with three actual arguments: the agreed

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*

Fig. 2. Representation of Relations for the Program under Analysis and Their Meaning.

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

Fig. 3. Representation of Relations for RPCs.

protocol t_{proto}, the handler of the protocol $o_{handler}$ and the IP address ip, in turn, marking the completion of server registration of the protocol and the handler, *i.e.*, ProtocolRegister(t_{proto}, $o_{handler}$, ip); and

(2) similarly, if there is a client instruction to establish connection, Call(j, "Client.connect"), with two actual arguments: a local proxy o_{proxy} of the protocol instance, ObjType(o_{proxy}, t_{proto}), and the same IP address ip as the server above, marking a connection established by a client, *i.e.*, RPCConnection(o_{proxy}, j, ip) and a declaration of a local proxy instance o_{proxy} for the RPC protocol, *i.e.*, ReifiedRPCInstance(t_{proto}, o_{proxy}); then

(3) a handshake has been completed: the invocation from the local proxy o_{proxy} will be handled by the remote handler $o_{handler}$, *i.e.*, RPCObjectHandler($o_{proxy}, o_{handler}$), and obviously, the variable v_{proxy} of the local proxy will point to the object $o_{handler}$ since it originally points to the object o_{proxy}, *i.e.*, VarPointsTo(v_{proxy}, $o_{handler}$).

Thus, the analysis takes a crucial step, connecting the modules, the client and the server, together in a system. Again, the methods, *Server.register* and *Client.connect*, may not be matched exactly by the implementation in an RPC framework/program. If not, their semantics may be implemented by a combination of several methods, which should be handled by an adapter for a specific framework during the engineering of this approach.

Rule 2: RPC Edge Construction. With the inference above, it can be known that if the predicate VarPointsTo(v_{proxy}, $o_{handler}$) is true, the variable v_{proxy} will start the RPC method invocation which will be handled by the remote handler $o_{handler}$. So an edge can be added into the CG, from the method invocation instruction by the variable v_{proxy} to the corresponding method of the handler $o_{handler}$, which will be expressed in Figure 4(b). This step is necessary since there may not be an explicit class hierarchy (CH, inheritance or implementation) between proxies and handlers. Figure 4(b) simulates the Dynamic Method Dispatch in Java enabling the correct method to be called at runtime, based on the actual class of the object: when there is an RPC proxy instance v_{proxy} calling its RPC method at instruction i, *i.e.*, the predicate RPCCallInfo(s, v_{proxy}, i) is true, pointing to the remote handler $o_{handler}$, an edge will be added into the CG from the call site i to the method $m_{handler}$ of the handler $o_{handler}$ with the same sub-signature as the method invoked at call site i, while each actual argument at the location will be mapped to the formal parameter of the handler method $m_{handler}$.

Rule 3: Assignment. Similarly, we build the points-to relationship between the variable returned by the RPC invocation and the object output after the execution of the handler method, when there is an edge in the CG, from a local RPC call to the remote handler method, as shown in Figure 4(c).

The necessity of the steps of Rule 2 and Rule 3 (but not the necessity of all their details) will be further discussed in Section V.

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

(a) Handshake

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

(b) RPC Edge Construction

VarPointsTo(r, o)
$\longleftarrow$
  CallGraphEdge(i, $m_{handler}$),
  AssignRetValue(i, r),
  ReturnVar($m_{handler}$, $r_{handler}$),
  VarPoints($r_{handler}$, o)

(c) Assignment of Return Value

Fig. 4. Inference Rules

Name: $\dfrac{A\in(a\ldots zA\ldots Z\$<>)*}{\textbf{\textit{name}}(A)}$ **Type:** $\dfrac{\textbf{\textit{name}}(T)}{\textbf{\textit{type}}(T)}$

Variable: $\dfrac{\textbf{\textit{type}}(T) \quad \textbf{\textit{name}}(V)}{\textbf{\textit{var}}(T, V)}$ **Field:** $\dfrac{\textbf{\textit{type}}(T) \quad \textbf{\textit{name}}(V)}{\textbf{\textit{field}}(T, V)}$

Allocation: $\dfrac{\textbf{\textit{var}}(V) \quad \textbf{\textit{type}}(T)}{\textbf{\textit{instruction}}(\textbf{\textit{alloc}}(V,T))}$ **Return:** $\dfrac{\textbf{\textit{var}}(R)}{\textbf{\textit{instruction}}(\textbf{\textit{return}}(R))}$

Program: $\dfrac{\textbf{\textit{instruction}}(I) \quad \textbf{\textit{program}}(P)}{\textbf{\textit{program}}(I,P)}$ **Store:** $\dfrac{\textbf{\textit{var}}(V) \quad \textbf{\textit{field}}(F)}{\textbf{\textit{instruction}}(\textbf{\textit{store}}(F,V))}$

Expression: $\dfrac{\textbf{\textit{instruction}}(E)}{\textbf{\textit{instruction}}(\textbf{\textit{exp}}(E))}$ **Site:** $\dfrac{\textbf{\textit{exp}}(E)}{\textbf{\textit{site}}(E)}$

Class: $\dfrac{\textbf{\textit{name}}(Name) \quad \textbf{\textit{type}}(Super) \quad \textbf{\textit{var}}(Field) \quad \textbf{\textit{fun}}(Sign)}{\textbf{\textit{class}}(Name,Super,Field,Sign)}$

Function: $\dfrac{\textbf{\textit{name}}(Sign) \quad \textbf{\textit{var}}(Arg) \quad \textbf{\textit{program}}(Body)}{\textbf{\textit{fun}}(Sign,Arg,Body)}$

Invocation: $\dfrac{\textbf{\textit{var}}(Base) \quad \textbf{\textit{name}}(Sign) \quad \textbf{\textit{var}}(Arg) \quad \textbf{\textit{var}}(Ret)}{\textbf{\textit{instruction}}(\textbf{\textit{call}}(Base,Sign,Arg,Ret))}$

InsertExpr: $\dfrac{\textbf{\textit{exp}}(E_1) \quad \textbf{\textit{fun}}(Sign) \quad \textbf{\textit{exp}}(E_2)}{\textbf{\textit{insertExpr}}(Sign,E_1,[after],\textbf{\textit{site}}(E_2))}$

InsertVar: $\dfrac{\textbf{\textit{fun}}(Sign) \quad \textbf{\textit{var}}(V)}{\textbf{\textit{insertVar}}(Sign,V)}$ **InsertField:** $\dfrac{\textbf{\textit{class}}(Clz) \quad \textbf{\textit{field}}(F)}{\textbf{\textit{insertField}}(Clz,F)}$

InsertFun: $\dfrac{\textbf{\textit{class}}(Clz) \quad \textbf{\textit{fun}}(Sign)}{\textbf{\textit{insertFun}}(Clz,Sign)}$ **Annotation:** $\dfrac{\textbf{\textit{exp}}(E)}{\textbf{\textit{annotation}}(E,\ [@]Msg)}$

Fig. 5. Syntax for RPC Instrumentation.

Algorithm 1: Instrumentation in Server

```
Input: $handler$, $m_{alloc}$, $clz$
1  // insert field
2      $t \leftarrow$ type($handler$);
3      $location$ = alloc($handler$);
4      $line \leftarrow$ site($location$);
5      $f \leftarrow$ field($t$, name($handler$) + $line$);
6      insertField($clz$, $f$);
7  // insert store
8      $st \leftarrow$ store($f$, $handler$);
9      insertExpr($m_{alloc}$, $st$, “after”, $location$)
10 // insert getField method
11     $body \leftarrow$ return($f$);
12     $m \leftarrow$ fun(“get”+$f$.name, NONE, body);
13     insertFun(clz, $m$);
```

C. Persistence

Usually, an RPC program is large in size, with so many classes, methods and statements, which is a particularly complex software, making any analysis on it time-consuming. Therefore, in addition to visualizing the HTML output of the analysis results, we also provide algorithms for instrumenting to save the RPC analysis results into the program file (in Java, it is the bytecode), avoiding duplicate computation in further analysis.

Syntax. We use the syntax shown in Figure 5 to describe how the result is embedded into the program file. For each rule in it, the part below the horizontal line is the operation command and the above one is the restriction. The basic rules, **Name, Type, Variable, Field, Return, Program, Expression, Site, Class** and **Function**, are easy to understand. For the operations, **Allocation** and **Store**, we adopt the default meaning: assigning values from right to left (*V = new T or F = V*). The operations, **Invocation, InsertExpr, InsertVar, InsertField** and **InsertFun**, are core commands for the persistence, to produce an invocation, insert a local variable/expression into a method, and insert a field/method into a class. Since we stitch multiple modules together, it is not possible for one module to accurately trace the creation of an object in another module. Thus, the newly created fields and their get methods are both static (with the modifier: “static”) to facilitate static analysis without violating the language grammar. In particular, we define the operation **Annotation** to provide the additional specification for further analysis by the analyzers using annotations [32–36].

Instrumentation. With the syntax defined, we can perform the instrumentation conveniently, which can be unfolded into two aspects: the server side and the client side. Given an RPC call from the client to the server, the local invocation occurs in a method (called: local launch method, m_{ll}), with an RPC invocation (denoted as *rpc*<$m, args, ret$>, containing the method invoked, its arguments and the return variable), and the remote handler object ($handler$) is allocated in the method (m_{alloc}) of the class (clz) in the server.

The server must expose the object $handler$ for the access by the client via the interface. Algorithm 1 shows the process in server, taking the $handler$, m_{alloc} and clz as inputs, to create a field (lines 2-6) to store the object $handler$ (line 8), which is exposed via a get method (lines 11-13). We use the line number of the handler's allocation to avoid possible duplicate name conflicts (lines 4-5).

Then the client can access the handler via the get method of the intermediary field. Algorithm 2 shows how to localize the field: to define a local variable as the return value of the get method invocation (lines 1-6). Then it simulates a real response from the remote handler: create an invocation by

Algorithm 2: Instrumentation in Client

```
Input: $m_{ll}$, $rpc\langle$m,args,ret$\rangle$, $handler$, $clz$
$t \leftarrow$ type($handler$);
$line \leftarrow$ site(alloc($handler$));
$fieldName \leftarrow$ name($handler$ + $line$);
$localhandler \leftarrow$ var($t$, "local"+ $fieldName$);
// localhandler = clz.getmHandler$n$()
$load \leftarrow$ call($clz$, "get" + $fieldName$, NONE, $localhandler$)
insertExpr($m_{ll}$, $load$, "after", $rpc$)
// ret = proxy.m(args) $\rightarrow$ ret = localVar.m(args)
$remote \leftarrow$ call($localhandler$, $rpc$.m, $rpc$.args, $rpc$.ret)
insertExpr($m_{ll}$, $remote$, "after", $load$)
annotation($remote$, "@RPCActual")
annotation($rpc$, "@RPCVirtual")
```

1. Server

m_s: the actual method in Hadoop. Its signature is "$\langle$ ...ipc.RpcEngine: RPC.Server getServer(java.lang.Class,Java.lang.Object,...) $\rangle$".

$\textsc{ProtocolRegister}(t_{proto}, o_{handler}, \text{-})$

$\longleftarrow$

$\textsc{Call}(i, m_s)$, $\textsc{ActualArg}(i, 1, v_0)$, $\textsc{ActualArg}(i, 2, v_1)$,
$\textsc{VarPointsTo}(v_0, t_{proto})$, $\textsc{VarPointsTo}(v_1, o_{handler})$

2. Client

m_c: the actual method in Hadoop. Its signature is "<...ipc.RPC: <T> T waitForProxy(java.lang.Class,long,...) >".

$\textsc{RPCConnection}(\, o_{proxy}, \text{-})$

$\longleftarrow$

$\textsc{Call}(j, m_c)$,
$\textsc{ReturnVar}(j, v_{proxy})$, $\textsc{VarPointsTo}(v_{proxy}, o_{proxy})$

Fig. 6. Adapter for Hadoop Framework

the local handler with the RPC method and its arguments and add it into the method too. At the end, it annotates the RPC invocation with the annotation "@RPCActual" and the simulated response with the "@RPCVirtual" as specifications. Finally, a real channel appears in the code permanently to connect the client and the server together for further analysis, which is no longer isolated.

IV. Evaluation

We have developed a prototype tool also named RPCBridge based on our approach, which is built on the top of Soot [1] and its intermediate representation Jimple. The adapter is the mapping between real methods in the RPC framework and virtual methods mentioned in Section III-B. For simplicity, it ignores the value of the IP variable, assuming that there is only one server. This does not affect the scalability of our approach. The adapter for Hadoop is shown in Figure 6. The purpose of our work is to analyze the variables in the RPC process. The sensitivity of our approach is consistent with the pre-analysis strategy. For example, RPCBridge uses Soot and its points-to analysis configuration is Spark, which is a precise analysis. CHA can also be used, which is a sound and fast analysis but is not precise. We investigate the following research questions for RPCBridge:

- **RQ1:** How does RPCBridge support the static analysis of the RPC on the real-world RPC programs?
- **RQ2:** How effective is RPCBridge? Is the RPC edge constructed by RPCBridge the true caller-callee relation?
- **RQ3:** Can RPCBridge benefit the existing analyzer?

A. Setup

Benchmark. We present a micro-benchmark written by Java with the RPCs based on Hadoop, gRPC, Dubbo, RMI and Thrift RPC frameworks. For the Hadoop framework, we choose Hadoop itself, HBase [37], Ozone [38], Phoenix [39], Pravega [40] and Tez [41]. Hadoop is a platform that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is one of the most popular systems for processing large data sets [42–44]. Although Hadoop may contain a module, HBase [37], we also take HBase with latest version into account since they have different characteristics in terms of storage, applicability, flexibility to read-write, availability and scalability. For the gRPC framework, we choose the benchmark [45] provided by its community, representing the most common usage of gRPC. For the Dubbo framework, we also choose the benchmark [46] provided by its community, which contains 186 projects where 157 are successfully compiled by us. For the RMI framework, we have emailed the author of [47] asking whether the dataset or the implementation is available, and the response is "this project was 20 years ago and I do not have these benchmarks or any implementation code". Therefore we choose Pax Exam [48], which contains 52 projects where 51 are successfully compiled, jmeter [49], and Rmi-jndi [50] which contains 12 projects where all of them are successfully compiled. They have the most stars when we use the keywords "extends java.rmi.Remote" and "extends Remote", which is an interface serving to identify interfaces whose methods may be invoked from a non-local virtual machine, when searching on grep.app with the filter written in Java. For the Thrift framework, we choose Cassandra [51],which is an open source NoSQL distributed database. There are fewer options based on the gRPC and Thrift frameworks because cross-language is one of their features, and our prototyping tool is Java-oriented.

Table I lists the real-world, open source, large-scale programs using the frameworks with their versions. The version denoted as "C(ID)" represents the version with a commit ID. It also lists the numbers of classes (#Cls), the numbers of the kilo line of code (#KLOC), and the forks and stars on GitHub. All the following experiments run in a Linux device based on Linux 5.4.0-148-generic with 56 cores (Intel (R) Xeon (R) E5-2680) and 256G RAM. And the experiments are running under the environment of JDK-1.8. The tool and the dataset are available anonymously [1] and will be made public after the paper is accepted.

Metrics. For RQ1 and RQ2, the metrics are described as follows. **#Pro** represents the number of the creation of the proxy instances in the client, which is the RPC connection, $\textsc{RPCConnect}(\, o_{proxy}: O, \text{-})$; **#CP** represents the number of the protocols proxied in the client after deduplication; **#H** represents the number of registered RPC handlers, *i.e.*, $\textsc{ProtocolRegister}(t_{proto}: T, o_{handler}: O, \text{-})$; **#SP** represents the number of the protocols instantiated in the server after

[1] https://anonymous.4open.science/r/RPCBridge-F1E0

TABLE I
INFORMATION OF THE OPEN SOURCE PROGRAMS

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

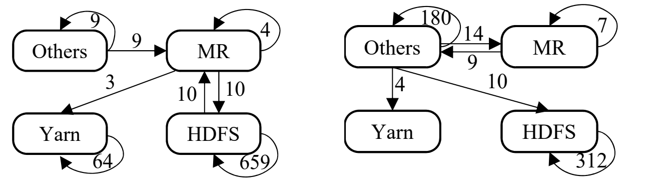
deduplication; **#MP** represents the intersection of protocols in the client and the server; **#CMP** represents the covered ones in **#MP** and its percentages (%) when running the project; **#RPC** represents the number of RPCs launched by the client via the proxies, which is RPCCallInfo(s: *S*, v_{proxy}: *V*, *i*: *I*); **#CRPC** represents the covered ones in **#RPC** and its percentages (%) when running the project; **#+E** represents the number of the incremental RPC edges constructed by our approach in Section III-B, which mean the successful handshake between the client and the server; **#CE** represents the covered ones in **#+E** and its percentages (%) when running the project; **A T & T** represent the analysis time of RPCBridge and the total time of the analysis including the preprocessing time, respectively; the percentage in brackets represents the proportion of analysis time of RPCBridge to total time.

B. RQ1: Support for the RPC Mechanism

Table II shows the analysis result of RPCBridge on the benchmark. The number of proxies, client protocols, handlers and matching protocols found in Hadoop is the largest among the benchmark, and the number of RPC edges constructed in it, is the largest as well. Ozone launches the most RPCs (238) in its clients while Pravega consumes the most time (11,607s). We have observed that the number of protocols in the server always equals to the number of the matched protocols. We think it is because that if there is a protocol registered in the server, there must be an RPC proxy of the protocol in the client, since every response must have a request; otherwise, it is not, because the responding server may be in the implementation of the top application provided by the third party, excluded from the framework. The fact that the number of #+Edge is larger than the number of #RPC indicates that an RPC may have multiple responses as we assume that there is only one server in the evaluation. Totally, RPCBridge finds 263 matched RPC protocols and constructs 2,578 RPC edges in CG. Its analysis time is only a really small part of the total time (Maximum: 0.31% in HBase, Minimum: 0.03% in Pravega, on Average: 0.12%).

Answer to RQ1. Our approach can effectively find RPC protocols (263) and construct RPC edges (2,578) in CG in real-world large-scale RPC programs with low time consumption.

Case Study. We will use a case study from Hadoop and HBase to illustrate the connectivity of RPCBridge for RPC programs. Simply, we divide them into four modules, MR (the set of classes with a prefix "*org.apache.hadoop.mapreduce*"), HDFS (prefix "*org.apache.hadoop.hdfs*"), Yarn (prefix "*org. apache.hadoop.yarn*"), and Others (the remaining classes) which can be seen as the top application based on the other three modules. And assume that the modules run on different machines with different addresses. Figure 7 shows the RPC interaction between the modules, generated automatically by our tool. The directed edge in it represents the RPC call and its response, and the number on the edge represents the amount of RPC call/response pairs. The spin node represents the RPC call that occurs inside it: in fact, it should be split and deployed on different machines. Totally, there are 768 RPCs and responses in the Hadoop system and 536 in the HBase system, connecting their modules together. RPCBridge constructs the critical invocations between modules for static analysis, making the CG of the RPC system a connected one to support further analysis. but in HBase, it needs to spend more attention on all the interaction with the other three modules .



(a) Hadoop (b) HBase

Fig. 7. RPC Interaction between Modules in (a) Hadoop and (b) HBase, Connected by RPCBridge.

C. RQ2: Effectiveness

To the best of our knowledge, there is no labeled dataset that allows us to calculate false positives and false negatives. To evaluate the effectiveness of RPCBridge, we use Soot for instrumentation to observe whether the RPC protocol and the RPC edge constructed by RPCBridge is the true caller-callee. We execute the programs and their test cases. Table II shows the coverage result after execution. For those projects, gRPC-bench, dubbo-bench, Pax Exam, jmeter and Rmi-jndi, the RPCs in them can be covered with almost 100% as their entrances are relatively simple. Although it looks like that there are many protocols and RPCs in dubbo-bench, Pax Exam and Rmi-jndi, they contain many subprojects as mentioned before, each of which is relatively simple, with a relatively obvious entry, server and client. For the very complex Hadoop-based projects, the coverage of the matched protocols ranges from 53.6% (HBase) to 60.0% (Phoenix) and the coverage of RPCs ranges from 54.3% (Phoenix) to 70.4% (Hadoop). Totally, on average, the matched protocols are covered by 63.1%, the RPCs are covered by 61.8% and the RPC edge constructed by RPCBridge are 60.8%. RPCs cannot be covered does not mean that our results are incorrect as many RPCs require complex pre-conditions to be triggered. For those protocols, RPCs detected and the RPC edge constructed by RPCBridge which have not been covered, to further explore whether they can be covered, it

TABLE II
RPC Operations Found by Our Approach

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	20	5	4	4	4(100%)	17	17(100%)	19	17(89.5%)	3.3 (0.03%)	11,607
tez	58	42	43	27	27	15(55.6%)	111	61(55.0%)	187	101(54.0%)	5.8 (0.17%)	3,447
gRPC-bench	5	5	5	5	5	5(100%)	18	16(88.9%)	18	16(88.9%)	1.3 (0.86%)	152
dubbo-bench	62	62	62	62	62	48(77.4%)	173	121(69.8%)	173	121(69.8%)	50.8 (0.41%)	12,335
pax exam	24	24	28	22	22	9(40.9%)	47	20(42.6%)	66	32(48.5%)	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

is necessary to understand more deeply and carefully design more complicated trigger conditions for these systems, which is beyond the scope of this paper. For the false positive analysis, we cannot check whether they are used or not actually. There is an evidence that Hadoop prefers Protocol Buffers over Java's Writeable for the serialization of the data in protocols. 28 matched protocol found in it by RPCBridge contains "PB", and 9 protocols does not which are used in test cases. The false negatives will be discussed in Section V.

Answer to RQ2. About 60% of the protocols and RPCs detected by RPCBridge are actively used, and also about 60% of the RPC edges constructed by RPCBridge can be covered by execution, indicating that they are the true caller-callee relationship.

D. RQ3: Benefits for the Existing Analyzer

To answer RQ3, we perform the taint analysis with FlowDroid on the Hadoop benchmark which is more complex and has a larger size than others, and compare the analysis result of the source bytecode with the bytecode instrumented after building the RPC edges. And the timeout is set to 5 hours. For the comparison, we record three metrics. The comparison results are shown in Table III. The symbol 'Δ' (also the number in brackets) in each column represents the change after the RPC connection. The data in Pravega is not recorded due to timeout. More leakage paths have been found in all projects except for Phoenix after the RPC connection. Totally, there are 9 (9/37=24.3%) leakage paths newly contributed by RPCBridge for the RPC connection. The memory overhead (+0.26%) of RPCBridge can be ignored. The increase in time consumption (13/108=12.1% on average) is more obvious compared to increase in memory consumption. The increase in time consumption proves that our connection for RPC explicitly impact taint analysis during the taint propagation exploration, as more nodes are connected and more paths are explored. That is to say, RPCBridge provides more paths for static analyzers to support further analysis.

Answer to RQ3. Our approach is expected to bring significant benefits (+24.3%) for the existing analysis tool, and introduces an increase in memory consumption (0.3%) and time consumption (12.1%).

TABLE III
Benefits and Overhead for Taint Analysis

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	11 (+2)	20.7 (+0.05)	28 (+4)
phoenix	3 (0)	25.4 (+0.03)	31 (+2)
pravega	-	-	TO
tez	4 (+2)	10.6 (+0.06)	13 (+1)
Total	37 (+9)	82.1 (+0.26)	108 (+13)

V. Threat to Validity

Our prototype tool is based on Soot, which, although powerful, currently supports processing bytecode versions lower than JDK 17, limiting its use. When running Soot, sometimes it will have some exceptions when parsing bytecode or constructing CG, which may affect the reproduction of experimental results. The adapter used in the experiment may not represent complete cases of creation and registration, although some of these methods are specified as: "*Get a proxy connection to a remote server*" [52] and "*Construct a server for a protocol implementation instance*" [53]. In addtion, the configuration of prepoints-to algorithm will affect the accuracy of our approach, causing false positives or negatives. Furthermore, our work focuses on modelling the typical semantics of the RPC operations (Unary RPC, where the client sends a single request and gets back a single response), but not all semantics of the operations from all frameworks, which may cause false negatives. For example, the framework gRPC [15] provides stream operations (Server streaming RPC, Client streaming RPC and Bidirectional streaming RPC) [54]. Our approach can model the semantics but additional adaptation may be required when constructing the call edge, since the request and response are not one-time, *i.e.*,not a Unary RPC, but are encapsulated in the stream: when a message arrives via the stream, a listener is needed to process it. The construction of the call edges becomes more complicated when using the

Bidirectional streaming RPC because "*Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order.*" [54]. This is why the steps of Rule 2 and Rule 3 in Section III-B are necessary, even though their details do not represent all situations.

VI. Related Work

We present related work as follows: one is on points-to analysis of various features in Java programs, and another is on the log analysis and RPC customization.

A. Points-to Analysis for Java Programs

The Java language has some dynamic features: some of them come from the language itself, such as reflection, dynamic proxy and generic; others come from the powerful framework, such as dependency injection and aspect-oriented programming, which are core technologies in the Spring framework. Some efforts have been made to further support static analysis for these features. Analysis for reflection is performed based on string constant searching, class identification, argument type matching, etc., either statically or dynamically [4–7, 55–58]. `Jasmine` is developed for static analysis to overcome the handicaps: dependency injection and aspect-oriented programming, in the Spring programs [59]. Fourtounis et al. model the semantics of dynamic proxy to improve the soundness of the static analysis [8]. Li et al. introduce generic-sensitive pointer analysis, which can serve as crucial context elements for effectively distinguishing contexts in Java programs [60]. He et al. propose IFDS-based, Container-Usage-Pattern-based and library summary-based approaches to make the points-to analysis faster, more fine-grained and more precise [61–65]. All of these works contribute to points-to analysis in terms of certain features, as does our work. Compared with them, we focus on the dynamic feature of the generic RPC in Java programs, usually based on a protocol, to overcome the challenges in static analysis for network communication and interaction on different devices.

Sharp et al. try to establish the points-to relationship for RMI-based Java applications [47], providing a flow- and context-insensitive points-to analysis for such applications. One of the issues of RMI is the high latency, and its most-common simple solution is to cash objects at the client which could lead to further issues such as distortion of consistency, blocking the widespread use of RMI [66]. Our work is a more general approach that targets existing RPC frameworks/programs through adapters under the RPC specification, including but not limited to RMI-, Hadoop-, and gRPC-based programs. In addition, our approach is based on existing mature points-to analysis, therefore it can be flow-sensitive, context-sensitive and even object-sensitive.

B. Log Analysis and RPC Customization

Due to the scale of the software and the obstacle that network communication brings to static analysis, dynamic analysis is an effective choice for program analysis with network communication. Many research works focus on log analysis, via log instrumentation, collection, abstraction and mining, for monitoring and failure diagnosis. For example, `lprof` [67] and `Pensieve` [68] try to connect modules of the distributed system based on log analysis: the former is a profiling tool that automatically reconstructs the execution flow of each request in a distributed application and the latter is a tool capable of reconstructing near-minimal failure reproduction steps from log files and system bytecode, automatically simulating failure localizaition of artificial debugging. A survey for log analysis has found log analyzers suffer from the ever-increasing volume, variety, and velocity of logs produced by modern software, and suspicious logs are often overwhelmed by logs generated during software normal executions, making them labor-intensive and error-prone [69]. Our approach can avoid the above problems while covering the analysis of code that is not executed or does not generate logs. Meanwhile, it does not execute the program under test, with very little dependency on the environment, deployment and resources. On the other hand, it may introduce false positives due to inaccurate modelling or different choice of strategies during the points-to analysis.

The RPC specifications actively promote the development and implementation of RPC and become part of the *de facto* standard, which is the convention followed by the RPC frameworks and the infrastructure for our work model. As mentioned in Section V, our modelling for RPC under that case can be reused but the construction of the call edge should be more careful when messages are delivered in any order [54] or by remote direct memory access (RDMA) [22, 25, 26]. Our approach requires adaptation when it comes to some customization of RPC [70–72]. Moreover, our experiments lead us to believe that our approach can also benefit vulnerability detection, security concern and so on during RPC process [73–79].

VII. Conclusion

RPC is widely used in real-world enterprise large-scale programs, allowing a program to invoke a remote procedure as if it were a local call. Although the RPC programs/frameworks play an increasingly important role in various domains, including telecommunications, distributed systems, cloud computing, and artificial intelligence, to the best of our knowledge, there is little work on their static analysis in nearly two decades.

In this paper, we propose a novel approach to model the semantics of the RPC operations, and perform reasoning for points-to analysis between the client variable and the object in the server based on the logical rules to connect the modules together and persist the analysis result for further reuse. We evaluate our approach on real-world large-scale programs, and the experiment shows that it can connect effectively different modules together in one system and provide significant benefits to upper-level analyzers. In the future, we will apply our approach to more RPC programs/frameworks written in different languages and investigate the stream operations during the RPC process.

References

[1] "Soot - A framework for analyzing and transforming Java and Android applications," https://soot-oss.github.io/soot.

[2] "WALA UserGuide: Pointer Analysis," http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis.

[3] "Doop - Framework for Java Pointer and Taint Analysis," https://github.com/plast-lab/doop.

[4] Y. Li, T. Tan, and J. Xue, "Understanding and Analyzing Java Reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, pp. 7:1–7:50, 2019. [Online]. Available: https://doi.org/10.1145/3295739

[5] V. B. Livshits, J. Whaley, and M. S. Lam, "Reflection Analysis for Java," in *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3780. Springer, 2005, pp. 139–160. [Online]. Available: https://doi.org/10.1007/11575467_11

[6] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of Java reflection: literature review and empirical study," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 507–518. [Online]. Available: https://doi.org/10.1109/ICSE.2017.53

[7] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer, "More Sound Static Handling of Java Reflection," in *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9458. Springer, 2015, pp. 485–503. [Online]. Available: https://doi.org/10.1007/978-3-319-26529-2_26

[8] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, "Static analysis of Java dynamic proxies," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, 2018, pp. 209–220. [Online]. Available: https://doi.org/10.1145/3213846.3213864

[9] B. Livshits, "Improving Software Security with Precise Static and Runtime Analysis," *Ph.D. Dissertation. Stanford University*, 2006.

[10] "Hadoop client unable to relogin because a remote DataNode has an incorrect krb5.conf," https://issues.apache.org/jira/browse/HADOOP-15378.

[11] "RBF: getContentSummary RPC compute sub-directory repeatedly," https://issues.apache.org/jira/browse/HDFS-16382.

[12] "The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing," https://hadoop.apache.org.

[13] "Unexpected INTERNAL error propagated to application layer," https://github.com/grpc/grpc-java/issues/9289.

[14] "StatusException: UNAVAILABLE on client when maxConnectionAge is set on server," https://github.com/grpc/grpc-java/issues/9566.

[15] "A high performance, open source universal RPC framework," https://grpc.io.

[16] "Possible memory leak in rpc layer," https://issues.apache.org/jira/browse/DRILL-4266.

[17] "Flatten query leads to out of memory in RPC layer," https://issues.apache.org/jira/browse/DRILL-6235.

[18] "RMI - The Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine," https://docs.oracle.com/Javase/tutorial/rmi/.

[19] "Thrift is a lightweight, language-independent software stack for point-to-point RPC implementation," https://thrift.apache.org.

[20] "Apache Dubbo is a high-performance, Java based, open source RPC framework," https://cn.dubbo.apache.org/en/.

[21] R. Thurlow, "RPC: Remote Procedure Call Protocol Specification Version 2," *RFC*, vol. 5531, pp. 1–63, 2009. [Online]. Available: https://doi.org/10.17487/RFC5531

[22] T. Talpey and B. Callaghan, "Remote Direct Memory Access Transport for Remote Procedure Call," *RFC*, vol. 5666, pp. 1–34, 2010. [Online]. Available: https://doi.org/10.17487/RFC5666

[23] B. Lengyel and M. Björklund, "Partial Lock Remote Procedure Call (RPC) for NETCONF," *RFC*, vol. 5717, pp. 1–23, 2009. [Online]. Available: https://doi.org/10.17487/RFC5717

[24] A. Adamson and N. Williams, "Remote Procedure Call (RPC) Security Version 3," *RFC*, vol. 7861, pp. 1–26, 2016. [Online]. Available: https://doi.org/10.17487/RFC7861

[25] C. Lever, W. A. Simpson, and T. Talpey, "Remote Direct Memory Access Transport for Remote Procedure Call Version 1," *RFC*, vol. 8166, pp. 1–55, 2017. [Online]. Available: https://doi.org/10.17487/RFC8166

[26] C. Lever, "Bidirectional Remote Procedure Call on RPC-over-RDMA Transports," *RFC*, vol. 8167, pp. 1–13, 2017. [Online]. Available: https://doi.org/10.17487/RFC8167

[27] T. Myklebust and C. Lever, "Towards Remote Procedure Call Encryption by Default," *RFC*, vol. 9289, pp. 1–21, 2022. [Online]. Available: https://doi.org/10.17487/RFC9289

[28] "RFC Editor," https://www.rfc-editor.org/.

[29] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*. ACM, 2005, pp. 1–12. [Online]. Available: https://doi.org/10.1145/1065167.1065169

[30] "The Java® Virtual Machine Specification (Java SE 8 Edition)," https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.10.1.

[31] Y. Smaragdakis and G. Balatsouras, "Pointer Analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: https://doi.org/10.1561/2500000014

[32] "PMD is an extensible cross-language static code analyzer," https://pmd.github.io/.

[33] D. Tang, A. Plsek, and J. Vitek, "Static checking of safety critical Java annotations," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010, Prague, Czech Republic, August 19-21, 2010*, ser. ACM International Conference Proceeding Series. ACM, 2010, pp. 148–154. [Online]. Available: https://doi.org/10.1145/1850771.1850792

[34] W. Pugh, "JSR 305: Annotations for Software Defect Detection," https://jcp.org/en/jsr/detail?id=305.

[35] P. Pinheiro, J. C. Viana, M. Ribeiro, L. Fernandes, F. C. Ferrari, R. Gheyi, and B. Fonseca, "Mutating code annotations: An empirical evaluation on Java and C# programs," *Sci. Comput. Program.*, vol. 191, p. 102418, 2020. [Online]. Available: https://doi.org/10.1016/j.scico.2020.102418

[36] N. Peru, "Annotation on array type should be properly handled," https://sonarsource.atlassian.net/browse/SONARJAVA-1420.

[37] "Apache HBase is an open-source, distributed, versioned, column-oriented store modeled after Google' Bigtable," https://github.com/apache/hbase.

[38] "Ozone is a scalable, redundant, and distributed object store for Hadoop and Cloud-native environments," https://github.com/apache/ozone.

[39] "Apache Phoenix is a SQL skin over HBase delivered as a client-embedded JDBC driver targeting low latency queries over HBase data," https://github.com/apache/phoenix.

[40] "APravega is an open source distributed storage service implementing Streams," https://github.com/pravega/pravega.

[41] "Apache Tez is a generic data-processing pipeline engine envisioned as a low-level engine for higher abstractions," https://github.com/apache/tez.

[42] Y. Gu and R. L. Grossman, "Toward Efficient and Simplified Distributed Data Intensive Computing," *IEEE Trans. Parallel Distributed Syst.*, vol. 22, no. 6, pp. 974–984, 2011. [Online]. Available: https://doi.org/10.1109/TPDS.2011.67

[43] S. Lee, S. Shakya, R. Sunderraman, and S. Belkasim, "Real Time Micro-blog Summarization Based on Hadoop/HBase," in *2013 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology, Atlanta, Georgia, USA, 17-20 November 2013, Workshop Proceedings*. IEEE Computer Society, 2013, pp. 46–49. [Online]. Available: https://doi.org/10.1109/WI-IAT.2013.148

[44] A. Zarei, S. Safari, M. Ahmadi, and F. Mardukhi, "Past, Present and Future of Hadoop: A Survey," *CoRR*, vol. abs/2202.13293, 2022. [Online]. Available: https://arxiv.org/abs/2202.13293

[45] "The "Queries Per Second Benchmark" allows you to get a quick overview of the throughput and latency characteristics of grpc." https://github.com/grpc/grpc-java/tree/master/benchmarks.

[46] "Samples for Apache Dubbo." https://github.com/apache/dubbo-samples.

[47] M. Sharp and A. Rountev, "Static Analysis of Object References in RMI-Based Java Software," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 664–681, 2006. [Online]. Available: https://doi.org/10.1109/TSE.2006.93

[48] "Pax Exam is a testing framework for OSGi." https://github.com/ops4j/org.ops4j.pax.exam2.

[49] "Apache JMeter open-source load testing tool for analyzing and measuring the performance of a variety of services." https://github.com/apache/

jmeter.

[50] "Demos for rmi, jndi, ldap, jrmp,jmx and jms." https://github.com/longofo/rmi-jndi-ldap-jrmp-jmx-jms.

[51] "Apache Cassandra is a highly-scalable partitioned row store." https://github.com/apache/cassandra.

[52] "Get a protocol proxy that contains a proxy connection to a remote server," https://github.com/apache/hadoop/blob/trunk/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/ipc/RPC.java#L644.

[53] "Construct a server for a protocol implementation instance," https://github.com/apache/hadoop/blob/trunk/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/ipc/RpcEngine.java#L122.

[54] "An introduction to key gRPC concepts, with an overview of gRPC architecture and RPC life cycle," https://grpc.io/docs/what-is-grpc/core-concepts/.

[55] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011, pp. 241–250. [Online]. Available: https://doi.org/10.1145/1985793.1985827

[56] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[57] Y. Zhang, Y. Li, T. Tan, and J. Xue, "Ripple: Reflection analysis for Android apps in incomplete information environments," *Softw. Pract. Exp.*, vol. 48, no. 8, pp. 1419–1437, 2018. [Online]. Available: https://doi.org/10.1002/spe.2577

[58] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind, "Fast online pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 2, p. 11, 2007. [Online]. Available: https://doi.org/10.1145/1216374.1216379

[59] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, "Jasmine: A Static Analysis Framework for Spring Core Technologies," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 60:1–60:13. [Online]. Available: https://doi.org/10.1145/3551349.3556910

[60] H. Li, T. Tan, Y. Li, J. Lu, H. Meng, L. Cao, Y. Huang, L. Li, L. Gao, P. Di, L. Lin, and C. Cui, "Generic Sensitivity: Generics-Guided Context Sensitivity for Pointer Analysis," *IEEE Trans. Software Eng.*, vol. 50, no. 5, pp. 1144–1162, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3377645

[61] D. He, Y. Gui, W. Li, Y. Tao, C. Zou, Y. Sui, and J. Xue, "A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, pp. 971–1000, 2023. [Online]. Available: https://doi.org/10.1145/3622832

[62] D. He, J. Lu, and J. Xue, "IFDS-based Context Debloating for Object-Sensitive Pointer Analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 101:1–101:44, 2023. [Online]. Available: https://doi.org/10.1145/3579641

[63] D. He, J. Lu, Y. Gao, and J. Xue, "Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis," *IEEE Trans. Software Eng.*, vol. 49, no. 2, pp. 719–742, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3162236

[64] J. Lu, D. He, W. Li, Y. Gao, and J. Xue, "Automatic Generation and Reuse of Precise Library Summaries for Object-Sensitive Pointer Analysis," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 736–747. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00039

[65] D. He, J. Lu, and J. Xue, "Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis," in *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, ser. LIPIcs, vol. 222. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 30:1–30:29. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

[66] A. O. Mustafa and M. A. Sayegh, "Message passing: Survey on rpc, rmi, and corba," 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235309138

[67] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A Non-intrusive Request Flow Profiler for Distributed Systems," in *OSDI*, 2014, pp. 629–644.

[68] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach," in *SOSP*, 2017, pp. 19–33.

[69] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A Survey on Automated Log Analysis for Reliability Engineering," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 130:1–130:37, 2022. [Online]. Available: https://doi.org/10.1145/3460345

[70] K. Seemakhupt, B. E. Stephens, S. M. Khan, S. Liu, H. M. G. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy, "A Cloud-Scale Characterization of Remote Procedure Calls," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 2023, pp. 498–514. [Online]. Available: https://doi.org/10.1145/3600006.3613156

[71] J. Wang, Y. Yang, J. Zhang, X. Yu, O. Alfarraj, and A. Tolba, "A data-aware remote procedure call method for big data systems," *Comput. Syst. Sci. Eng.*, vol. 35, no. 6, pp. 523–532, 2020. [Online]. Available: https://doi.org/10.32604/csse.2020.35.523

[72] X. Yan, A. P. Joa, B. Wong, B. Cassell, T. Szepesi, M. Naouach, and D. Y. Lam, "Specrpc: A general framework for performing speculative remote procedure calls," in *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*, P. Ferreira and L. Shrira, Eds. ACM, 2018, pp. 266–278. [Online]. Available: https://doi.org/10.1145/3274808.3274829

[73] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 2013, pp. 652–661. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606611

[74] L. Jokubauskas, J. Toldinas, and B. Lozinskis, "Detecting applications vulnerabilities using remote procedure calls," in *Proceedings of the 27th International Conference on Information Society and University Studies (IVUS 2022), Kaunas, Lithuania, May 12, 2022*, ser. CEUR Workshop Proceedings, vol. 3611. CEUR-WS.org, 2022, pp. 58–65. [Online]. Available: https://ceur-ws.org/Vol-3611/paper10.pdf

[75] A. Singh, B. Singh, and H. Joseph, "Vulnerability Analysis for RPC," in *Vulnerability Analysis and Defense for the Internet*. Springer, 2008, pp. 135–167.

[76] S. Marksteiner, H. Vallant, and K. Nahrgang, "Cyber security requirements engineering for low-voltage distribution smart grid architectures using threat modeling," *J. Inf. Secur. Appl.*, vol. 49, 2019. [Online]. Available: https://doi.org/10.1016/j.jisa.2019.102389

[77] E. Barlas and T. Bultan, "Netstub: a framework for verification of distributed java applications," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. ACM, 2007, pp. 24–33. [Online]. Available: https://doi.org/10.1145/1321631.1321638

[78] K. H. Lee, N. Sumner, X. Zhang, and P. Eugster, "Unified debugging of distributed systems with Recon," in *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 2011, pp. 85–96. [Online]. Available: https://doi.org/10.1109/DSN.2011.5958209

[79] B. Xin, P. T. Eugster, X. Zhang, and J. Yang, "Lightweight Task Graph Inference for Distributed Applications," in *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*. IEEE Computer Society, 2010, pp. 100–110. [Online]. Available: https://doi.org/10.1109/SRDS.2010.20