DisTA: Generic Dynamic Taint Tracking for Java-Based Distributed Systems

Dong Wang, Yu Gao, Wensheng Dou, Jun Wei
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China
{wangdong18, gaoyu15, wsdou, wj}@otcaix.iscas.ac.cn

***Abstract*—Dynamic taint tracking is a powerful information flow analysis approach, which can be applied in many analysis scenarios, e.g., debugging, testing, and security vulnerability detection. Most dynamic taint tracking approaches are designed for standalone systems, and cannot support inter-node taint tracking in distributed systems. Few inter-node taint tracking approaches are designed for specific distributed systems, e.g., Apache Spark, and require specific modifications to different distributed systems.**

In this paper, we present DisTA, a generic dynamic taint tracking tool for Java-based distributed systems. By instrumenting common network communication modules in Java, DisTA can perform inter-node taint tracking for different distributed systems with little manual efforts. We evaluate DisTA on five large-scale real-world distributed systems, e.g., ZooKeeper and Yarn, and require only 10 LOC launch script modification on average. The experimental results show that DisTA can accurately track all inter-node taints with a relatively low overhead.

***Index Terms*—taint tracking, data flow analysis, distributed systems**

I. Introduction

Dynamic taint tracking (DTA) [1] is a commonly-used technique in privacy leakage detection [2], SQL injection detection [3], program debugging [4], [5] and testing [2], [6], [7], etc. With DTA tools, developers first assign taints to certain specific data in the program, i.e., taint source points. Then, taints can propagate along with data during the program execution. Users can further set program points in the program, i.e., taint sink points, to check whether the data is tainted.

Nowadays, distributed systems, e.g, distributed computing frameworks [8], [9], storage systems [10], [11], synchronization services [12], [13], have become pervasive. These distributed systems are usually deployed on multiple machines (i.e., nodes) which communicate with each other by messages. Sometimes, multiple distributed systems, e.g., distributed computing frameworks and storage systems, also need to work together to fulfil specific user requirements. This hinders applying DTA in distributed systems, for example, tracking the privacy data in storage systems to monitor if the privacy data is leaked to untrusted nodes [14].

Existing DTA tools [15]–[21], e.g., Phosphor [22] and TaintDroid [2], mainly work on standalone systems, and can only perform *intra-node taint tracking*. They usually treat the communication among nodes as a blackbox, and cannot perform *inter-node taint tracking*. Further, they cannot perform taint tracking among multiple distributed systems, i.e., *cross-system taint tracking*.

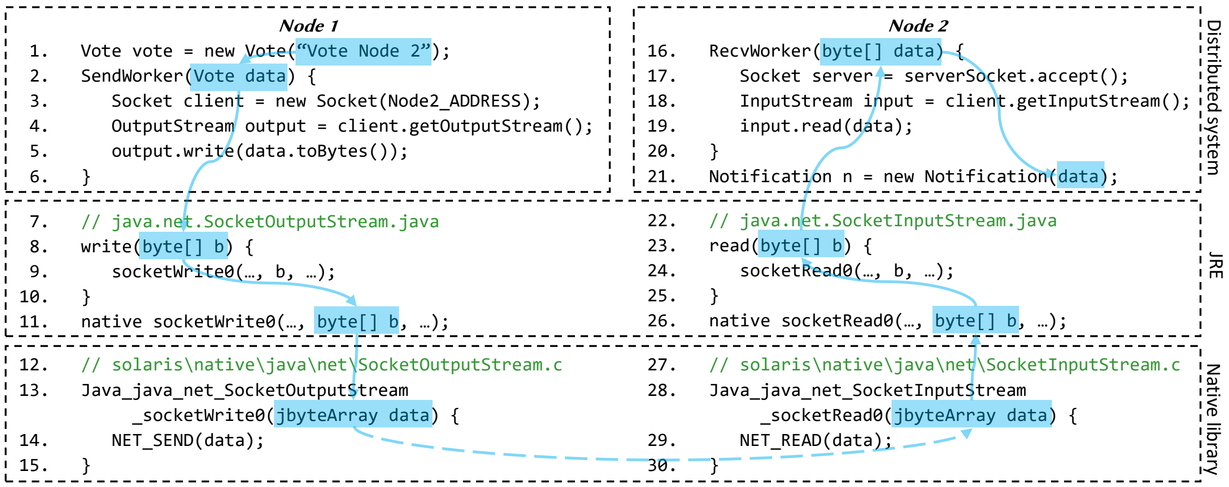
Few tools are designed for tracking taints in distributed systems, e.g., Taint-Exchange [23], Kakute [14], and FlowDist [24]. Taint-Exchange intercepts network related system calls to propagate taints with messages. Kakute [14] modifies Apache Spark's network communication methods, i.e., shuffle methods, and then propagates taints among nodes within RDD data. FlowDist [24] manually pairs the specified message-passing APIs for senders and receivers, e.g., *SocketChannel.read* and *SocketChannel.write*, to trace message transfer between nodes, and further analyzes inter-node data flows by static analysis.

However, these tools suffer from *soundness*, *precision* and *usability* issues. Taint-Exchange [23] can only support the single taint, i.e., mark the data as tainted or not, which prevents it being used in some scenarios, e.g., program debugging [5]. Besides, it cannot be applied on Java programs running in JVM. Kakute [14] can only work on Apache Spark, and cannot track other data except Spark RDDs. FlowDist [24] requires developers to use their experience to annotate message-passing API mapping, and write dozens of script files to drive its analysis. Furthermore, its static analysis can introduce precision issues.

In this paper, we present DisTA, a generic dynamic taint tracking tool for Java-based distributed systems. DisTA has the following three advantages over existing approaches.

- **Sound**. Network communication in Java-based distributed systems utilizes Java Native Interface, i.e., JNI, to bridge Java APIs and the underlying operating system. DisTA instruments network communication APIs at the JNI level, and avoids missing network communication in distributed systems. Thus, DisTA is sound for taint tracking involving network communication.
- **Precise**. All messages between nodes are finally transferred into bytes. To achieve high precision, DisTA perform inter-node taint tracking at the byte-level granularity for all network communication APIs.
- **Generic**. By instrumenting the common network communication modules in JRE, DisTA can be easily applied to different Java-based distributed systems without extra system-specific modifications. By supporting multi-taint tracking, DisTA can be used in multiple scenarios.

To evaluate DisTA, we build a micro benchmark which contains 30 test scenarios for common network communica-



```
1.   Vote vote = new Vote("Vote Node 2");
2.   SendWorker(Vote data) {
3.      Socket client = new Socket(Node2_ADDRESS);
4.      OutputStream output = client.getOutputStream();
5.      output.write(data.toBytes());
6.   }
7.   // java.net.SocketOutputStream.java
8.   write(byte[] b) {
9.      socketWrite0(…, b, …);
10.  }
11.  native socketWrite0(…, byte[] b, …);
12.  // solaris\native\java\net\SocketOutputStream.c
13.  Java_java_net_SocketOutputStream
         _socketWrite0(jbyteArray data) {
14.     NET_SEND(data);
15.  }

16.  RecvWorker(byte[] data) {
17.     Socket server = serverSocket.accept();
18.     InputStream input = client.getInputStream();
19.     input.read(data);
20.  }
21.  Notification n = new Notification(data);
22.  // java.net.SocketInputStream.java
23.  read(byte[] b) {
24.     socketRead0(…, b, …);
25.  }
26.  native socketRead0(…, byte[] b, …);
27.  // solaris\native\java\net\SocketInputStream.c
28.  Java_java_net_SocketInputStream
         _socketRead0(jbyteArray data) {
29.     NET_READ(data);
30.  }
```

Fig. 1. A simplified communication example in ZooKeeper. Blue blocks indicate the message data. Solid arrows show the taint flows within one node, and the dashed arrow shows the taint flows between two nodes through OS.

tion protocols. DisTA can precisely track all taints in these test scenarios. We further apply DisTA on five real-world distributed systems, i.e., ZooKeeper [12], MapReduce/Yarn [25], ActiveMQ [26], RocketMQ [27] and HBase [10]. To run DisTA on these systems, we only modify 10 LOC in the launch script of these distributed systems on average. The experimental results show that DisTA can successfully perform taint tracking for these distributed systems without over-tainting or losing any taints. The whole taint tracking process, including the intra-node and inter-node tracking, causes 4.23X overhead, while the pure intra-node tracking causes 3.92X overhead, meaning that it is not expensive in inter-node tracking. We have made DisTA and its experimental benchmarks publicly available at https://github.com/tcse-iscas/DisTA.

In summary, this paper makes the following contributions.

- We design and implement DisTA, a generic taint tracking tool for distributed systems. DisTA can be easily applied on different distributed systems with small manual efforts and be used in multiple taint tracking scenarios.
- We evaluate DisTA on the micro benchmark and real-world distributed systems, and the experimental results show that DisTA can effectively and efficiently perform inter-node taint tracking in diverse distributed systems.

II. Background and Motivation

In this section, we use a simplified example in ZooKeeper [12] to illustrate the network communication process in distributed systems (Section II-A). We further introduce intra-node taint tracking (Section II-B) and its limitation (Section II-C). Finally, we discuss and compare existing DTA tools for distributed systems (Section II-D).

A. Motivating Example

Figure 1 shows ZooKeeper's communication that $Node1$ votes for $Node2$. $Node1$ first constructs a *Vote* data (Line 1), and then passes it to the *SendWorker* (Line 2). In *SendWorker*, *write* method is called (Line 5) for writing *data* into the output stream. The implementation of *write* method (Line 8 - 10) is located in *SocketOutputStream.java*, which is a JRE class. In the method body, a JNI method *socketWrite0* is called. The corresponding native method (Line 13 - 15) of *socketWrite0* is implemented in C language, and it calls the method *NET_SEND* (Line 14). *NET_SEND* is a Linux system call method which can deliver the data *buf* to OS to send out of the node. Line 16 - 30 shows the code executed on $Node2$. It follows a similar process as $Node1$. We follow the taint flow and state the process from the lowest code level to the highest. The native method (Line 28 - 30) in *SocketInputStream.c* first invokes the system call method *NET_READ* to read data and store it in *data* (Line 29). Then the data is passed to the JNI method *socketRead0* and further to *read* in *SocketInputStream.java*. Finally, the data is passed to the *RecvWorker* (Line 16 - 20). *RecvWorker* reads the data from the input stream and stores it in an object *Notification* (Line 21).

```
  int a, b;
+ Taint a_t = new Taint("a_tag");
+ Taint b_t = new Taint("b_tag");
  int c = a + b;
+ Taint c_t = a_t.combine(b_t);
```

Fig. 2. Phosphor assigns a taint to every variable.

B. Taint Tracking for Standalone Programs

The example in Figure 1 shows two standalone programs: the program executed on $Node1$ (Line 1 - 15) and the other one executed on $Node2$ (Line 16 - 30). The solid arrows show the taint flows within the single node. Existing DTA tools can track these flows with good soundness and precision. We take Phosphor [22], the state-of-the-art DTA tool, as the example to state how these tools perform intra-node taint tracking.

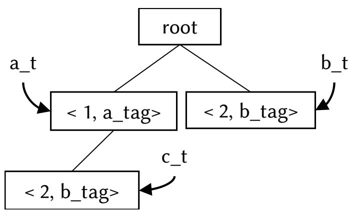


Fig. 3. Phosphor uses a singleton tree to store all taint tags.

```
1.   public int native socketRead0(byte[] data);
2.
3. + public ReturnTaint socketRead0(byte[] data,
   +     Taint data_t) {
4. +         ReturnTaint ret = new ReturnTaint();
5. +         ret.value = socketRead0(data);
6. +         ret.taint = data_t;
7. +         return ret;
8. + }
```

Fig. 4. Phosphor surrounds the native method with a wrapper.

Phosphor adds a shadow variable as the taint of every variable, and utilizes ASM [28] to instrument Java bytecode so that taints can propagate with data. We illustrate its designs in two aspects, i.e., taint storage and taint propagation.

Taint storage. Phosphor assigns a taint to each variable in the program. A taint is a set composed of multiple tags. Once developers set a variable as a taint source point, the corresponding taint is assigned with a unique tag. The code snippet in Figure 2 shows an example. The original code declares three variables a, b (Line 1) and c (Line 4). Phosphor assigns each variable with a shadow variable as its taint, i.e., a_t (Line 2) for a, b_t (Line 3) for b, and c_t (Line 5) for c, respectively. These taints are initialized as empty, i.e., no tags. In Figure 2 a and b are set as taint source points. Thus, their taints are assigned by two String tags a_tag and b_tag, respectively. Note that the value of the tag is set by developers. It can be a String as the example, or any other object.

Phosphor maintains a singleton tree to store all taint tags. As Figure 3 shows, when a new tag is assigned, a new node is added as the child of the root node. Every node is a tuple $< ID, Tag >$, where ID is the unique rank of the tag in the tree, and Tag is the value of the tag. The taint can refer to a tag in the tree, which implies that the taint is assigned with the tag. Related to the code snippet in Figure 2, there are two tags $< 1, a\_tag >$ and $< 2, b\_tag >$ as children of root. a_t refers to $< 1, a\_tag >$ and b_t refers to $< 2, b\_tag >$, implying that a_t has the tag a_tag (i.e., $a\_t = \{a\_tag\}$) and b_t has the tag b_tag (i.e., $b\_t = \{b\_tag\}$).

By utilizing the above taint storage strategy, Phosphor can save much memory usage. If two variables have the same taint tag, their taints can refer to the same node in the tree, thus avoiding storing the same tags repeatedly.

Taint propagation. Phosphor uses the combination of taints to represent the taint propagation process. Every variable assignment is a taint propagation process. In Figure 2, variable c is assigned by the result of a plus b. So its taint combines taint a_t and b_t (i.e., $c\_t = a\_t \bigcup b\_t$). Correspondingly, in Figure 3, the singleton tree adds a new node $< 2, b\_tag >$ as the child of $< 1, a\_tag >$, which implies that the tag of taint a_t is combined with the tag of taint b_t. This new node is referred by c_t. c_t has all tags on the path from the root to the node it refers to. Thus, c_t has both a_tag and b_tag (i.e., $c\_t = \{a\_tag, b\_tag\}$).

C. Limitations in Intra-node Taint Tracking

Phosphor performs taint tracking by instrumenting Java bytecode. Thus, it can track taint flows in Java code, but cannot handle with flows in the native code. Taking Figure 1 as an example, Phosphor cannot track taints in Line 12 - 15 and Line 28 - 30. To solve this problem, Phosphor surrounds JNI methods (e.g., *SocketWrite0*) with a wrapper, and directly assigns taints of parameters to the return value. As shown in Figure 4, for the original native method *socketRead0* (Line 1), Phosphor adds a wrapper method (Line 3 - 8). When the original method is called, this wrapper is called instead. In the wrapper body, a *ReturnTaint* object which can wrap the return value and its taint is created (Line 4). Then, the original native method is called to receive the message and store it in *data* object. However, the taint of the message is lost. Instead, Phosphor directly assigns the taint of the parameter *data_t* to the message. Obviously, it tracks a wrong taint flow, and makes the taint tracking unsound and imprecise.

D. Inter-node Taint Tracking

In general, building the inter-node taint tracking tool must focus on challenges of three factors:

1) **Soundness:** If a tool is not sound, then it may incorrectly drop taint information of the data. This usually happens when data in some network communication APIs are not tracked. Thus taints cannot propagate through these APIs.
2) **Precision:** If a tool is not precise, it may incorrectly add additional taint information to the data, i.e., over-tainting. In inter-node taint tracking, low precision is usually due to the coarse granularity in tracking message.
3) **Usability:** If a tool can only track taints in some specific systems or some specific scenarios, or requires lots of manual efforts when applied to different systems, we say it has poor usability.

Some tools have been developed for tracking taints in distributed systems, e.g., Taint-Exchange [23], Kakute [14], and FlowDist [24]. We explain them as follows.

Taint-Exchange [23] is a generic inter-node taint tracking tool for x86 binaries. It utilizes libdft [29] to perform intra-node taint tracking. To make taints propagate between nodes, Taint-Exchange intercepts Linux system calls such as $write()$, $send()$, $socket()$ and $accept()$ to wrap the taint information with the data and transfer them together through network. Since the system call level instrumentation does not require modifying the upper applications, Taint-Exchange can be applied in different programs. However, Taint-Exchange applies a bit for every data to mark whether it is tainted. Thus, it cannot taint different data by different taints and cannot be applied in some scenarios, e.g., program debugging [5]. Besides, Taint-Exchange is specially designed for C-based x86 programs, so that it cannot track taints in Java programs.

Kakute [14] works for Spark [9] applications, and aims to track RDDs in Spark. It leverages Phosphor [22] to track intra-node taint propagation. To perform inter-node taint tracking, it modifies shuffle APIs, which are specific to RDD transfer in Apache Spark. Thus, Kakute cannot be applied in other distributed systems that do not have shuffle APIs. Further, Kakute cannot track other data that do not relate to RDDs in Apache Spark, e.g., the password to login the Spark system. Thus, Kakute is unsound in some taint tracking scenarios. Through instrumenting the data structure of RDD, Kakute can easily mark fields in RDDs as tainted, and further tracks them at runtime. It is precise in RDD dependency analysis. However, it cannot be precise enough for all tracking scenarios.

FlowDist [24], [30] can support different distributed systems. It requires developers to provide the message-passing APIs used in distributed systems, so that FlowDist can map the message sending and receiving statements at runtime. After that, FlowDist performs offline static analysis to track statement-level data flows in the program, and merges intra-node data flows into a whole data flow graph based on message-passing APIs mapping information. By default, FlowDist only modifies 6 JRE APIs for network communication, including two APIs for socket communication (i.e., *Socket.getInputStream/getOutputStream*), two APIs for object serialization I/O (i.e., *ObjectInputStream.readObject/ObjectOutputStream.writeObject*), and two APIs for NIO (i.e., *SocketChannel.read/write*). However, there are over 100 APIs for network communication in JRE. FlowDist can drop the data flow information within these unmonitored APIs. Thus, it is unsound. FlowDist performs statement-level static analysis, which means it cannot distinguish different variables with runtime values, thus introducing imprecision issues. Besides, the combination of the static and dynamic analysis builds a barrier for users to easily use it.

In comparison, our approach DisTA can perform sound and precise taint tracking for different Java-based distributed systems. The architecture of DisTA is similar to Taint-Exchange. We utilize Phosphor to perform intra-node taint tracking, and instrument specific network communication APIs to make taints propagate between nodes. The differences between DisTA and Taint-Exchange locates at that we have instrumentation designs, e.g., byte level tracking, to make DisTA sound and precise, and runtime designs to solve practical problems in scenarios requiring multiple taints. All these designs make DisTA be a generic and easy-to-use taint tracking tool for different distributed systems.

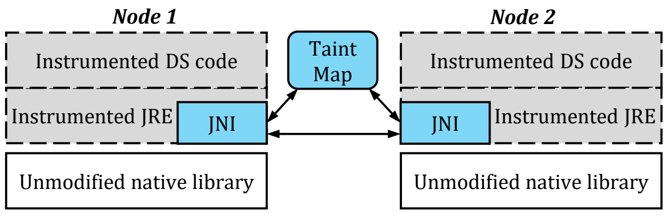


Fig. 5. The overview of DisTA.

III. DisTA Design

Figure 5 shows an overview of DisTA. First, DisTA instruments all Java bytecode. The distributed system code and part of JRE code are instrumented by Phosphor (Grey dashed box areas), while DisTA further instruments the network communication JNI methods (Blue solid box areas). Then, at runtime, the instrumented system code is executed with the instrumented JRE. The taints of messages are sent out by the instrumented JNI methods, and stored in an intermediate component named Taint Map. The other node receives taints from Taint Map and reassigns them to the received messages.

The following sections illustrate how we design DisTA, including the instrumentation level and tracking granularity (Section III-A), the selection of methods for instrumentation (Section III-B), and the instrumentation implementation details (Section III-C). At last, we introduce DisTA's runtime execution (Section III-D).

A. Instrumentation Level and Tracking Granularity

To perform inter-node taint tracking in distributed systems, we need to instrument network communication methods. As shown in Figure 1, we can instrument these methods at different levels, e.g., distributed system code and JRE methods. Instrumenting at the distributed system code level (e.g., *SendWorker* and *RecvWorker* in Figure 1) can make our approach not applied for other systems, since different systems usually use different network communication methods. Instrumenting at JRE level can solve the problem. However, there are hundreds of network communication methods in JRE. These methods can adopt TCP and UDP communication, and TCP communication can use different I/O stream classes, e.g., pure byte I/O, object I/O, buffered I/O, NIO and AIO. Every I/O class has several even dozens of different I/O methods. Instrumenting all these methods is an enormous task.

In DisTA, we instrument JNI methods, which are at the bottom level among all JRE methods. All network communication methods in JRE must send / receive message to / from OS through JNI methods, so DisTA can track taint flows in different JRE methods by instrumenting limited JNI methods. Moreover, the message data in most JNI methods are in the type of the byte array, which gives us the opportunity to track taint in the single byte level granularity.

B. Instrumented Methods

To find out which methods should be instrumented, we inspect all JNI methods in Java source code (HotSpot Open-

```
 1.   native send(Data d);
 2.   native receive(Data d);

 3. + send(Data d, Taint d_t) {
 4. +    Data wrappedData = serialize(d, d_t);
 5. +    send(wrappedPacket);
 6. + }
 7. + receive(Data d, Taint d_t){
 8. +    Data wrappedData = Data.emptyData();
 9. +    int ret = receive(wrappedData);
10. +    deseralize(wrappedData, d, d_t);
11. + }
```

Fig. 6. A simplified illustration of instrumenting JNI method.

JDK 1.8), and further judge if they are used in network communication. Generally, a network communication method should have the word *read* / *write* or *receive* / *send* in its name. Among these methods, some are specially designed for file I/O, which are not our focus. We further check the method information on Javadoc and exclude these methods.

As a result, we get 13 JNI methods in 5 classes. Two methods in *SocketInputStream* and *SocketOutputStream* are used for TCP communication. Three methods in *PlainDatagramSocketImpl* are used for UDP communication. Eight methods in *FileDispacherImpl* and *DatagramDispatcherImpl* are used to implement NIO and AIO communication. Note that, although *FileDispatcherImpl* looks like a file I/O specific class, it is actually extended by *SocketDispatcherImpl* which is used in NIO and AIO communication in Linux. Thus, *FileDispatcherImpl*'s four methods are included.

C. Instrumentation Details

Our basic idea for instrumentation is simple and intuitive. We add a wrapper method for each method. When it is called, the wrapper is called instead. For senders, we combine the message with its taint and send them out by the original method. For receivers, we invoke the original method to receive the message and then divide it into the data and taint.

Figure 6 shows a simplified example of the instrumentation. *send* (Line 1) and *receive* (Line 2) are a pair of JNI methods. Method *send* can send data d out of the node, and *receive* can receive data and store it in the parameter d. When these two methods are called, we replace them by our two modified methods (Line 3 - 11). In the wrapper method *send*, we serialize data d and its taint d_t and wrap them in $wrappedData$ (Line 4). Then we send them out by the original JNI method (Line 5). In *receive*, we first construct an empty $Data$ object $wrappedData$ (Line 8). Then we call the original JNI method *receive* to receive the message and store it in $wrappedData$ (Line 9). At last, we deserialize it and assign the data and taint to the parameter d and d_t.

Ideally, the type of message data in network communication should be a byte or a byte array, so that we can track the distinct taint of every single byte and make the tracking precise. However, not all data types in JNI methods are as we expected. Depending on the data type of the method parameter, we categorize these methods into three types: stream oriented methods, packet oriented methods and direct buffer oriented methods. They are instrumented in three different ways.

```
 1.   class DatagramPacket {
 2.      byte[] data;
 3. +    Taints[] taints;
 4.   }
 5.   native send(DatagramPacket packet);
 6.   native receive0(DatagramPacket packet);

 7. + send(DatagramPacket packet, Taint packet_t) {
 8. +    byte[] data = packet.getData();
 9. +    Taint[] taints = packet.getTaints();
10. +    byte[] serialBytes = serialize(data, taints);
11. +    DatagramPacket wrappedPacket =
              new DatagramPacket(serialBytes);
12. +    send(wrappedPacket);
13. + }
14. + receive0(DatagramPacket packet, Taint packet_t) {
15. +    DatagramPacket wrappedPacket =
              new DatagramPacket();
16. +    receive0(wrappedPacket);
17. +    packet.setData(wrappedPacket.getData());
18. +    packet.setTaints(wrappedPacket.getTaints());
19. + }
```

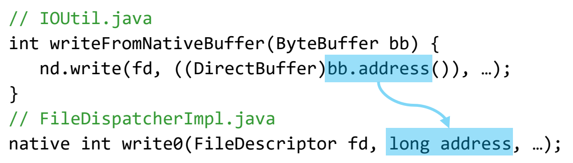
Fig. 7. Instrumentation for packet oriented methods.

Type 1: Stream oriented methods. TCP message related JNI methods are in this type. They use stream I/O methods to read / write data. The message data in these methods is in the type of byte array or the single byte. Thus, we can directly use the instrumentation way shown in Figure 6 for them.

Type 2: Packet oriented methods. UDP message related JNI methods belong to this type. They usually uses a packet object to wrap the message data in. We must access into the packet before serializing / deserializing the data and taint.

Figure 7 shows an example of instrumenting two UDP methods *send* and *receive0* in *PlainDatagramSocketImpl*. Line 1 - 4 shows *DatagramPacket* class, which stores the message data in the field $data$. After instrumentation, we add a field $taints$ to store taints for every distinct byte data. Line 5 and Line 6 are the original JNI methods. They use *DatagramPacket* object rather than the byte array to store data. Line 7 - 13 shows the wrapper of *send*. First, we fetch out the data (Line 8) and its taints (Line 9), and wrapped them in *serialBytes*. Then we initiate a new packet object to carry the wrapped bytes (Line 11) and send them out by the original JNI method (Line 12). Note that we do not directly replace *packet*'s *data* field by serialized bytes, because *packet* may be used by the following code. Changing its field may affect the execution semantics. For the receiver method, we create a new *DatagramPacket* object (Line 15) to receive the full bytes (Line 16), and then deserialize them into the data and taints. At last, they are placed in the fields of the original parameter object.

Type 3: Direct buffer oriented methods. This type of methods most commonly occurs in NIO communication which uses *DirectBuffer* to store the message data. *DirectBuffer* is a special class that manages a memory block out of Java heap. It does not directly store an object or bytes carrying the message data, but the data's address in the physical memory. When a



```
// IOUtil.java
int writeFromNativeBuffer(ByteBuffer bb) {
   nd.write(fd, ((DirectBuffer)bb.address()), …);
}
// FileDispatcherImpl.java
native int write0(FileDescriptor fd, long address, …);
```

Fig. 8. Direct buffer oriented methods only provide the memory address of data rather than the data itself and its taints.

TABLE I
PARTIAL INSTRUMENTED METHODS AND THEIR TYPES

Class	Method	Type
SocketInputStream	socketRead0	1
SocketOutputStream	socketWrite0	1
LinuxVirtualMachine	read	1
	write	1
PlainDatagramSocket	send	2
	receive0	2
	peek	1
DirectByteBuffer	get	3
	put	3
IOUtil	writeFromNativeBuffer	3
	readIntoNativeBuffer	3
WindowsAsynchronous-SocketChannelImpl	implRead	3
	implWrite	3

node sends out data by *DirectBuffer*, it directly accesses the data by the memory address, rather than copying a heap object to OS by network related native method. This helps improve I/O efficiency in Java programs.

However, the use of *DirectBuffer* hinders us directly fetching the data and taints in the parameters of a JNI method invocation. Taking the code in Figure 8 as the example, we can only find a long format address in the native method *write0*. Thus, the previous instrumentation ways cannot work here. Then we have a look on the caller of the method, i.e., *writeFromNativeBuffer* in the class *IOUtil*. Here is a *DirectBuffer* object but still no available taint information, because *DirectBuffer* does not store data objects as well as taints associated with the data.

To solve this problem, we first modify *DirectBuffer* class. We add a taint array field in *DirectBuffer* to store all taints of the data, and modify every put / get method in the class, so that when the developer writes / reads the data to / from the memory, the associated taint can be stored to / fetched from the added taint array field.

Second, we instrument all callers of direct buffer oriented native methods such as *wrtieFromNativeBuffer* in Figure 8. It is similar to what we do for packet oriented methods. For write methods, we allocate a new larger DirectBuffer, and fetch the data out from the original DirectBuffer. Then we serialize the data with its taints stored in the added taint array field, and write the serialized results into the new buffer. Finally, we invoke the native method on the new buffer to send out the data and taints. Correspondingly, we allocate a larger DirectBuffer for read methods, and read the serialized data and taints by the buffer. Then we deserialize and put them into the original DirectBuffer object.

Finally, we instrument 23 methods in total. Table I shows a portion.

D. DisTA Runtime

After instrumentation, taints can propagate in distributed systems at run time. When a taint propagates within the single node, we denote it as a *local taint*. When it is transferred between nodes, it is a *global taint*. We design a component named *Taint Map* to store and transfer all global taints. Taint Map is an independent process which can communicate with all nodes, and maintain a map structure to store all global taints and their *Global IDs*.

Figure 9 shows how the Taint Map processes global taints. There are two bytes on Node 1, i.e., *b1* and *b2*, to be sent to Node 2, while Node 2 only receives one byte data *b1*. Both *b1* and *b2* are tainted by *t1*. Along with *b1*, *t1* must be transferred to Node 2. The whole transfer process can be divided into 5 steps. ① Node 1 sends the taint *t1* to Taint Map and request a Global ID. Taint Map allocates a unique number as Global ID for every global taint, e.g., *1* for *t1*. ② Node 1 receives the Global ID, and store it in its local taint storage, which is a tree that we introduced in Section II-B. Note that, since we have got the Global ID for *t1*, and *b2*'s taint is also *t1*, Node 1 does not need to request a Global ID again if it sends *b2* out later. ③ Node 1 serializes every data byte with its taint into a byte array, and transfers it to Node2. Here we do not directly serialize the taint, but put the Global ID after the data byte and transfer it. ④ Node 2 receives the data and the taint's Global ID, and requests to Taint Map to get the taint. ⑤ Node 2 receives the taint, and store it in its local taint storage.

1) Taint Tag Design: Compared the taint tag structure used by Phosphor, i.e., $< ID, Tag >$ introduced in Section II-B, we adds two additional fields, i.e., $LocalID$ and $GlobalID$. Thus, the taint tag of DisTA is a quad, i.e., $< ID, Tag, LocalID, GlobalID >$.

$LocalID$ is used to solve the problem of tag conflict. In distributed systems, different nodes can run the same code. That means, when we set some variables in the system as the taint source points, a same variable can be tainted with the same tag value on different nodes. If a tag propagates to another node and the node has already generated the same tag, they can be in conflict. Take Figure 9 as the example. If $Node2$ generates an *a_tag* before communicating with $Node1$, then this tag will be in conflict with the *a_tag* from $Node1$. If they are not distinguished correctly, it will make the taint tracking imprecise since we do not know where the taint is from. To solve this problem, we add a field *Local ID* for every taint tag. It consists of node's IP and JVM's process ID. Thus, nodes are aware of where tags are generated and distinguish them even when they have the same tag value.

$GlobalID$ is used to mark every unique taint in the inter-node taint tracking process. It is set as zero when a taint tag is generated at one node, and assigned a unique positive integer by Taint Map when transferred to another node.

2) Taint Map: We design Taint Map to solve two problems in inter-node taint tracking: large bandwidth usage when

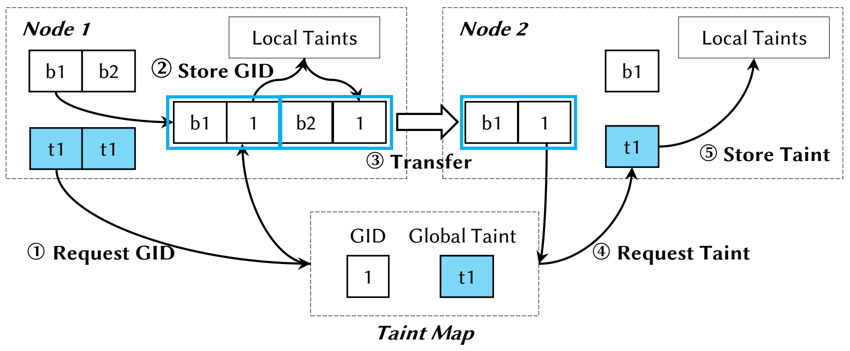


Fig. 9. Taints go through 5 steps in inter-node taint tracking.

repeatably transferring serialized taints, and mismatched serialized taint length between senders and receivers.

Large bandwidth usage. A serialized taint with one tag can be over 200 bytes. What's worse, a taint can have multiple tags. As the number of tags increases, the length of the byte array increases linearly. The serialized bytes array can cause far more than 200X bandwidth overhead.

Mismatched serialized taint length. In message-passing, the receiver will allocate a fixed length empty byte array for storing the data bytes received. However, the length of the empty byte array does not always exactly equal the number of bytes sent by the sender. As Figure 9 shows, $Node1$ sends two bytes $b1$ and $b2$, while $Node2$ only receives $b1$. To receive the serialized taint simultaneously, the length of the allocated byte array must be enlarged. Note that the length of the serialized taint is not fixed. Thus, we cannot simply enlarge the byte array space by a fixed amount. It means that the receiver probably cannot receive the full serialized taint bytes, which can fail the following deserialization step.

With Taint Map and Global ID, we can transfer taints in a fixed length byte array. Thus, we can enlarge the allocated byte array on the receiver and do not need to worry about receiving an incomplete taint. On the other hand, every node only needs to communicate with Taint Map to transfer every global taint for only one time, and nodes only transfer serialized Global ID to each other. Thus, the bandwidth overhead caused by transferring global taints is acceptable, which depends on the length of the Global ID.

On the other hand, Taint Map may become bottleneck, since it runs as a single-point component that can be accessed by all nodes to request Global IDs for global taints. As the number of the global taints increases, the requests to Taint Map for allocating Global IDs also increase. The limit on the throughput of Taint Map may cause performance degradation in inter-node taint tracking. However, our evaluation results in Section V-F shows that the performance degradation is acceptable.

IV. Implementation

DisTA is implemented in 2,045 lines of code (LOC). Among them, 1,591 LOC are instrumentation related code. Most of them are ASM [28] instructions. As mentioned above, we instrument 23 methods. That means every method requires 69 LOC for instrumentation on average. To perform runtime instrumentation, we deploy the instrumentation code as a Java agent [31] attached with the target system.

Taint Map is implemented in 202 LOC. It is a simple map structure which can communicate with all nodes. In practice, Taint Map can be replaced by other mature K-V store systems such as ZooKeeper [12] and etcd [32] to improve its performance. On the other hand, it can be improved by some reliable designs, e.g., adding a standby node to handle with the single point failure. In this work, we only make the simplest implementation, since DisTA is designed for in-house analysis and testing for now, but not for production.

Besides, we modify Phosphor in 252 LOC to implement our global taint tag structure and support the inter-node taint serialization / deserialization.

V. Evaluation

Our evaluation addresses the following research questions:

RQ1: Is DisTA sound and precise in inter-node taint tracking?

RQ2: Is DisTA easy to use?

RQ3: How is DisTA's performance overhead?

To answer the above questions, we implement 30 test cases with different commonly used communication protocols and APIs (Section V-A) as the micro benchmark, and collect several network communication scenarios in 5 real-world distributed systems (Section V-B) for evaluation.

A. Micro Benchmark

As shown in Table II, we implement 30 test cases for different network communication APIs and protocols. All of them are frequently used in Java network communication. Three cases come from a third-party network application framework Netty [33], and the others use JRE standard APIs. Among them, JRE Socket has multiple test cases, because users can invoke different I/O interfaces in different stream classes to read / write different kinds of data. For example, *writeObject* in class *ObjectOutputStream* is specially for writing an object to the stream. Other communication ways do not have multiple cases, since they only have the single I/O implementation.

TABLE II
MICRO BENCHMARK

Name	Description	# cases
JRE Socket	JRE standard TCP	22
JRE Datagram	JRE standard UDP	1
JRE SocketChannel	JRE NIO TCP	1
JRE DatagramChannel	JRE NIO UDP	1
JRE AsyncSocketChannel	JRE AIO	1
JRE HTTP	JRE HTTP	1
Netty Socket	3rd-party TCP	1
Netty DatagramSocket	3rd-party UDP	1
Netty HTTP	3rd-party HTTP	1
Total		30

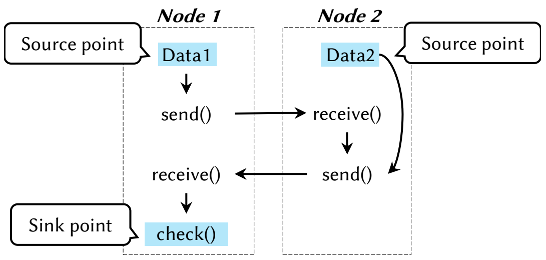


Fig. 10. Taint tracking scenario for micro benchmark.

Workloads. We design the workloads for the micro benchmark as shown in Figure 10. First, $Node1$ sends *Data* to $Node2$. $Node2$ receives it and combines it with another *Data*. Then the combined data is sent back to $Node1$. Finally, $Node1$ checks the received data by method *check*(). The data transferred in network are specially designed for each test case. Specifically, we control the size of total data around 10MB. For JRE socket cases that use different stream methods to read / write different kinds of data, the input data are also different. They can be a large int array or an object with a long text String field. For JRE Datagram, and channel type protocols, i.e., JRE SocketChannel, JRE DatagramChannel and three Netty protocols, they directly use *DatagramPacket* or *ByteBuffer* to read / write data. We design the same data, i.e., a 10MB size byte array. For HTTP type protocols, i.e., JRE HTTP and Netty HTTP, we design an HTML page containing large amounts of text contents.

Taint tracking scenario. We set *Data1* and *Data2* on $Node1$ and $Node2$ as source points, and the method *check*() as the sink point. The *check* method is the same for all protocols, while *Data1* and *Data2* are specifically designed. At the sink point, DisTA should obtain two taints of *Data1* and *Data2*.

B. Real-World Distributed Systems

As shown in Table III, we select 5 popular real-world distributed systems, i.e., ZooKeeper [12], MapReduce/Yarn [25], ActiveMQ [26], RocketMQ [27] and HBase [10] as evaluation subjects. These systems represent different kinds of distributed systems: ZooKeeper for coordination systems, MapReduce/-Yarn for computing frameworks, ActiveMQ and RocketMQ for message middlewares, and HBase for databases. These systems use different communication protocols. ZooKeeper uses JRE standard TCP APIs and Netty library. MapReduce/-Yarn uses JRE NIO and Yarn RPC. ActiveMQ and RocketMQ supports many kinds of protocols including standard TCP, UDP, NIO, as well as HTTP/HTTPS, WebSocket and STOMP [34] protocols. HBase uses standard JRE NIO and Google's protobuf RPC [35].

TABLE III
REAL-WORLD DISTRIBUTED SYSTEMS

System	Workload
ZooKeeper	Leader election
MapReduce/Yarn	Execute Pi
ActiveMQ	Message distribution
RocketMQ	Message distribution
HBase+ZooKeeper	Query data

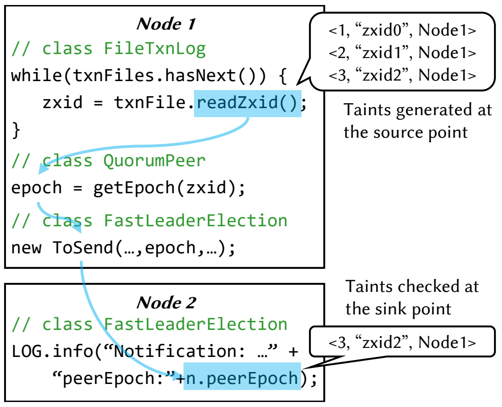
TABLE IV
TAINT TRACKING SCENARIOS FOR SYSTEMS

Type	System	# Sources	# Sinks
Specific data trace (SDT)	ZooKeeper	3	1
	MapReduce/Yarn	1	1
	ActiveMQ	1	1
	RocketMQ	1	1
	HBase+ZooKeeper	1	1
System I/O monitor (SIM)	ZooKeeper	18	347
	MapReduce/Yarn	20	2,235
	ActiveMQ	6	371
	RocketMQ	9	408
	HBase+ZooKeeper	35	1,166

Workloads. We design one workload for each distributed system. We select the leader election process for ZooKeeper, a job to calculate the value of Pi for MapReduce, long text message distribution for ActiveMQ and RocketMQ, and getting data from a table for HBase. The 5 workloads are all common ones in these systems, as shown in Column Workload in Table III. Note that HBase's workload must run within two systems, i.e., HBase and ZooKeeper. Thus, this workload can be considered a cross-system taint tracking scenario.

Taint tracking scenarios. As Table IV shows, we design two types of taint tracking scenarios, i.e., *specific data trace* (SDT) and *system input/output monitor* (SIM). SDT scenarios are common in *program debugging* [4], [5]. It marks the specific data such as a vote in the leader election as tainted, to trace how it propagates in the system. In these scenarios, the number of taints is usually small and determinate. SIM scenarios are common in *data leakage detection* [24]. In these scenarios, the DTA tool marks data input functions, e.g., reading from a configuration file, as source points, and data output functions, e.g., log print statements, as sink points. Compared with SDT scenarios, the taints number of SIM is relatively large and indeterminate.

In SDT scenarios, we track important variables of the workload. For ZooKeeper, we select variable *Vote* as source point. During the leader election process, all nodes instantiate lots of *Vote* objects, but we only select 3 variables which are first transferred into the network. For MapReduce/Yarn,


Node 1

```
// class FileTxnLog
while(txnFiles.hasNext()) {
   zxid = txnFile.readZxid();
}
// class QuorumPeer
epoch = getEpoch(zxid);
// class FastLeaderElection
new ToSend(…,epoch,…);
```

Node 2

```
// class FastLeaderElection
LOG.info("Notification: …" +
   "peerEpoch:"+n.peerEpoch);
```

Fig. 11. A simplified taint tracking example in ZooKeeper.

we select *ApplicationID* of the job generated on the client as the source point. For ActiveMQ, we select a *TextMessage* variable representing the long text message as the source point. RocketMQ is similar to ActiveMQ. We select a *Message* variable. For HBase, we set a *TableName* variable as the source. For sink points, we select the method *checkLeader* as the sink of ZooKeeper. It is invoked on a follower when the leader is selected. For other systems, since all cases are the end-to-end request type, we set variables representing request results or methods to get the request result as sink points. They are *getApplicationReport* method in MapReduce/Yarn, an ActiveMQ's *Message* variable and an RocketMQ's *MessageExt* variable received on the message consumer, and an HBase's *Result* variable containing the data rows.

For SIM scenarios, we uniformly set file reading methods as source points for all systems. These files can be configuration files or data files, which may contain sensitive data. Once the method is invoked at runtime, we mark the return value as tainted. We set *LOG.info* method as sink points for all systems, and check if any log statement prints a tainted variable.

Cluster setting. For ZooKeeper, we deploy 1 node for Leader and 2 nodes for Follower. For MapReduce/Yarn, we deploy 1 node for ResourceManager, 1 node for NodeManager and 1 node for Task Container. For ActiveMQ and RocketMQ, we deploy them as three peer nodes. For HBase, we deploy 1 HMaster node and 2 HRegionServer nodes, and each node is equipped with a ZooKeeper process. Except ZooKeeper, all other systems have an extra node to run their clients.

C. Experimental Setting

For each node in Section V-A and Section V-B, we use a virtual machine in VMware Workstation Pro 16.1.0 to run it. Each VM is equipped with 2 CPU cores and 8GB memory. All VMs run on a host machine which has Intel Core(TM) i9-9900 CPU and 64GB of RAM.

D. Soundness and Precision

To evaluate the soundness and precision of the taint tracking, for each case in Table II and Table III, we check at sink points if any taint is dropped or appears unexpectedly.

We take the SIM scenario in ZooKeeper in Figure 11 as the example to illustrate the checking process. When a ZooKeeper node starts, it reads from existing transaction log files to get the largest transaction ID. In the while loop, Node 1 reads three files. Thus, three different taints are generated at this source point. During the workload execution, DisTA finds that a log print statement on Node 2 has checked a taint, which is the taint tagged as *zxid2* from Node 1. Then, we inspect code to figure out two questions: *Why zxid2 can propagate to this sink point*, and *why other taints generated at the same source point do not propagate to this sink point*? We find that only the transaction ID in the last file is assigned to the variable *zxid*, and further assigned to *epoch* and sent to Node 2. Thus, only the taint generated by the last file read method invocation can propagate to Node 2, while others are only generated and never propagated. For this sink point on Node 2, we can determine that there are no unexpected taints.

For the micro benchmarks and SDT scenarios in real-world distributed systems, we check every sink point, since the taint number is small and the propagation process is clear. For SIM scenarios, there are too many taints generated, and the taint propagation is too complex. Thus, we only randomly select a fraction of sink points to check. We observe that DisTA can accurately track all taints at these sink points. Therefore, we draw the conclusion for RQ1: *DisTA is precise for inter-node taint tracking, and sound for common communication protocols.*

E. Usability

We evaluate DisTA's usability by check if it can track taints at sink points without much extra instrumentation and specification.

To run DisTA in distributed systems, we first need to instrument JRE, which can be automatically performed by DisTA, by using the following instruction: **java -jar DisTA.jar JAVA_HOME jre-inst**. To instrument distributed systems, we can run **java -jar DisTA.jar SYSTEM_HOME**, and DisTA can instrument all class files and jar assembly files in the distributed system directory $SYSTEM\_HOME$. We can also choose not to manually instrument the distributed system code, since DisTA can automatically instrument uninstrumented code at run time.

To run the instrumented system code on the instrumented JRE, we only need to add two JVM flags to the original Java execution command: **-Xbootclasspath/a:DistTA.jar** to add DisTA runtime library to the classpath, and **-javaagent:DistTA.jar** to automatically modify uninstrumented libraries. For example, we only modify 3 LOC in ZooKeeper's environment configuration script file *zkEnv.sh* to configure DisTA.

```
JAVA="$INST_JAVA_HOME/bin/java"
SERVER_JVMFLAGS="-Xbootclasspath/a:DisTA.jar -
    javaagent:DisTA.jar=taintSources=
    $SOURCE_FILE,taintSinks=$SINK_FILE"
CLIENT_JVMFLAGS="-Xbootclasspath/a:DisTA.jar -
    javaagent:DisTA.jar=taintSources=
    $SOURCE_FILE,taintSinks=$SINK_FILE"
```

TABLE V
RUNTIME OVERHEAD FOR MICRO BENCHMARK

Case	Original (ms)	Phosphor		DisTA	
		Time (ms)	Overhead (X)	Time (ms)	Overhead (X)
JRE Socket-Best	3,644	7,532	2.07	8,932	2.45
JRE Socket-Worst	3,266	12,762	3.91	18,976	5.81
JRE Socket-Avg	4,119	10,391	2.52	16,833	4.09
JRE Datagram	7,532	25,811	3.43	30,540	4.05
JRE SocketChannel	8,410	25,000	2,97	27,644	3.29
JRE DatagramChannel	9,050	27,055	2.99	28,901	3.19
JRE AsyncSocketChannel	8,733	24,971	2.86	26,357	3.02
JRE HTTP	2,714	4,072	1.50	5,810	2.14
Netty Socket	5,196	12,813	2.47	17,402	3.35
Netty DatagramSocket	5,782	14,097	2.44	23,618	4.08
Netty HTTP	3,155	15,535	4.93	19,600	6.21
Average	4,706	12,599	2.62	18,341	3.95

TABLE VI
RUNTIME OVERHEAD FOR REAL-WORLD DISTRIBUTED SYSTEMS

System	Original (ms)	Phosphor-SDT		DisTA-SDT		Phosphor-SIM		DisTA-SIM	
		Time (ms)	Overhead	Time (ms)	Overhead	Time (ms)	Overhead	Time (ms)	Overhead
ZooKeeper	3,117	9,707	3.11	12,741	4.09	9,813	3.15	13,502	4.33
MapReduce/Yarn	28,824	108,199	3.75	108,770	3.77	115,449	4.01	115,974	4.02
ActiveMQ	4,065	19,100	4.70	20,311	5.00	19,535	4.81	20,599	5.07
RocketMQ	3,799	18,553	4.88	19,714	5.19	20,197	5.32	21,203	5.58
HBase+ZooKeeper	26,845	105,735	3.94	120,065	4.47	109,831	4.09	128,396	4.78
Average	13,330	52,259	3.92	56,320	4.23	54,965	4.12	59,935	4.76

At Line 1, we replace the Java environment by the instrumented JRE. Line 2 and 3 are JVM flags for ZooKeeper server and client respectively. Note that we add two files here, i.e., the source and sink files, which contains the user specification for taint source and sink points. They are specified in the form of Java method descriptors. When a method is specified as a taint source point, its return value is tainted. When a method is specified as a taint sink point, we check if its parameters are tainted before its method body execution.

On average, we modify 10 LOC in launch scripts for systems in Table III, and do not need to modify or inspect source code in these distributed systems. Compared with DisTA, other tools such as FlowDist [24] require much more efforts to work. FlowDist instruments different APIs for different systems. Furthermore, it requires users to perform instrumentation for 4 times and 3 different kinds of analysis. For example, users must run FlowDist for 4 times to instrument the source / sink points setting logic, method, branch and instruction level taint propagation code respectively. This makes the taint tracking process quite complicated.

Based on these results, we draw the following conclusion for RQ2: *DisTA can be easily applied on different distributed systems*.

F. Overhead

To evaluate the overhead of DisTA, we run each case three times and record its execution time. At the first time, we run the case without tracking any taints, i.e, the original execution. Then, we run the case on Phosphor, i.e., only intra-node taint tracking. Last, we run the case on DisTA, i.e., both intra-node and inter-node taint tracking.

Note that we do not consider evaluating the network and memory overhead. As introduced in Section III-D, DisTA transfers a fixed length byte array (4 bytes in default) storing Global ID for every data byte. Thus, DisTA should introduce about 5X network overhead. For the memory aspect, DisTA directly utilizes Phosphor's taint store design. Thus, it should introduce the similar memory overhead as Phosphor. Since Phosphor has evaluated its memory overhead (1X - 8X, 2.7X average), we do not evaluate it again.

Micro benchmark. The evaluation results for micro benchmarks are shown in Table V. Note that JRE Socket consists of 22 cases, so we list the best (JRE Socket-Best) and worst (JRE Socket-Worst) scenarios as well as the average (JRE Socket-Avg) values. Compared with the original execution time (Column Phosphor/Time), DisTA causes 2.14X overhead at best, and 6.21X at worst (Column DisTA/Overhead). It seems a huge overhead. However, comparing with the 3.95X overhead caused by DisTA (Column DisTA/Overhead) and Phosphor's 2.62X overhead (Column Phosphor/Overhead), we find out the pure inter-node taint tracking does not bring in much overhead.

Real-world distributed systems. The results are shown in Table VI. Compared with results in the micro benchmark, taint tracking in real-world distributed systems causes higher overhead in both Phosphor (3.92X and 4.12X) and DisTA (4.23X and 4.76X). We think the results are reasonable, since real-world systems are much more complex. Compared with the intra-node taint tracking, DisTA causes a relatively small extra overhead in inter-node taint tracking. For SDT scenarios, it causes 0.31X (4.23X - 3.92X). For SIM scenarios, it causes 0.64X (4.76X - 4.12X).

SDT vs SIM. By comparing the evaluation results in taint tracking in SDT (Specific data trace) scenarios and SIM (System input / output monitor) scenarios, we can figure out the performance degradation caused by Taint Map. In SDT scenarios, the overhead is 4.23X in average, while it is 4.76X in SIM scenarios. Then, we compare the number of global taints recorded in Taint Map in both kinds of scenarios. In SDT scenarios, the minimum number of global taints is one, and the maximum is six. In comparison, the minimum number of global taints is 54, and the maximum is 327 in SMT scenarios. We notice that *the overhead does not increase significantly with the number of global taints increases*.

Based on the above results and analysis, we draw the conclusion for RQ3: *DisTA introduces slight overhead comparing to the intra-node taint tracking, and is scalable for multiple taints*.

VI. Discussion

Support for specific JNI methods. In Section III-B, we mainly consider standard network communication JNI methods in JRE. However, distributed system developers can design their own native communication libraries and corresponding JNI methods, in which the taint cannot be directly tracked by DisTA. To support these methods, users can follow the three instrumentation ways and extend our instrumentation interfaces to instrument them.

Implicit flows handling. How to handle implicit flows (control flows) is an important problem for taint tracking tools. For DisTA, we only guarantee the correctness of taint tracking in inter-node taint tracking, i.e., from the message sending JNI method to the receiving JNI method. As to the correctness of the taint propagation within the single node from the source / sink points to the instrumented JNI methods, it is guaranteed by the intra-node taint tracking tool Phosphor [22]. Considering that Phosphor is still not perfect on implicit flows handling, we declare that DisTA inherits the limitation on Phosphor.

Comparison with other tools. We do not perform any comparison experiment. We introduce three different taint tracking tools in Section II-D. Taint-Exchange [23] is for x86 binaries, it cannot be applied to Java programs. Kakute [14] is specific for Spark, and aims to the RDD tracking scenario. The best comparison subject is FlowDist [24]. However, its static analysis implementation is too complex to run. We failed to reproduce its experiments.

Threats to validity. The main threats to our experiments are related to the representativeness of our selected test cases. We design a number of network communication scenarios as our micro benchmark. These cases include the standard JRE APIs and third-party communication libraries. All the communication protocols and APIs in them are widely used in Java ecosystem. For systems in Table III, all of them are widely used and cover a diverse set of architectures, i.e., leader-follower and peer-to-peer, and network protocols, e.g., Netty, YarnRPC, and HTTP. Thus, we believe our selected cases in both the micro benchmark and real-world distributed systems are representative.

VII. Related works

In this section, we discuss related works that are not discussed in previous sections.

Static taint tracking tools. Static taint tracking is a powerful approach for security tasks such as privacy leak detection, because it can cover all possible paths in code theoretically. Moreover, it has no impact on runtime performance. Researchers have presented several static tools for small scale programs. STILL [36] can detect exploit code in web request. FlowDroid [37] can precisely detect data leaks within Android applications. IccTA [38] extends FlowDroid to detect privacy leaks between multiple components. However, network communication in distributed systems is much more complex and non-deterministic. The dynamic message data and uncertain sending / receiving timing can make static analysis imprecise and unsound.

Dynamic taint tracking tools in different platforms. Dytan [16], libdft [29], TaintEraser [21], TaintPipe [39], and NeuTaint [40] can track taints for C-based programs. These tools do not consider taint propagation in network communication. Thus, they cannot be applied for distributed systems. Titian [41] is a prior work than Kakute [14] which is also specifically designed for data tracking in Spark. It is not portable for other systems. TaintDroid [2] customizes Android's specific Binder framework to track IPC messages between applications. It is designed for Android platform, so it cannot be applied to Java-based distributed systems, either.

VIII. Conclusion

Existing DTA tools cannot support inter-node taint tracking for distributed systems, or are designed for specific distributed systems and require specific modifications. We develop DisTA, a generic dynamic taint tracking tool for Java-based distributed systems. DisTA aims to be sound and precise in taint tracking, and easy to use. It instruments common network communication modules in JRE at the JNI level, and tracks taints in the single byte granularity. The experimental results on both the micro benchmark and real-world distributed systems show that DisTA achieves all its goals.

IX. Acknowledge

We thank the anonymous reviewers and our shepherd, Jun Xu, for their constructive suggestions. This work was partially supported by National Natural Science Foundation of China (62072444, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJSSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142).

REFERENCES

[1] J. Newsome and D. X. Song, “Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[2] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[3] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Improving application security with data flow assertions,” in *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 291–304.

[4] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha, “Debugging model-transformation failures using dynamic tainting,” in *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2010, pp. 26–51.

[5] M. Ganai, D. Lee, and A. Gupta, “DTAM: Dynamic taint analysis of multi-threaded programs for relevancy,” in *Proceedings of ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[6] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. Lippmann, “Coverage maximization using dynamic taint tracing,” MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, Tech. Rep., 2007.

[7] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 317–331.

[8] (2008) Apache hadoop mapreduce. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[9] (2015) Apache spark. [Online]. Available: https://spark.apache.org/

[10] (2007) Apache hbase. [Online]. Available: https://hbase.apache.org/

[11] (2016) Apache cassandra. [Online]. Available: https://cassandra.apache.org/

[12] (2010) Apache zookeeper. [Online]. Available: https://zookeeper.apache.org/

[13] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proceedings USENIX Annual Technical Conference (ATC)*, 2014, pp. 305–319.

[14] J. Jiang, S. Zhao, D. Alsayed, Y. Wang, H. Cui, F. Liang, and Z. Gu, “Kakute: A precise, unified information flow analysis system for big-data security,” in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2017, pp. 79–90.

[15] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, “Understanding data lifetime via whole system simulation,” in *USENIX Security Symposium*, 2004, pp. 321–336.

[16] J. Clause, W. Li, and A. Orso, “Dytan: A generic dynamic taint analysis framework,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2007, pp. 196–206.

[17] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, “Privacy oracle: A system for finding application leaks with black box differential testing,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008, pp. 279–288.

[18] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 85–96, 2004.

[19] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, “RIFLE: An architectural framework for user-centric information-flow security,” in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2004, pp. 243–254.

[20] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 116–127.

[21] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, “TaintEraser: Protecting sensitive data leaks using application-level taint tracking,” *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[22] J. Bell and G. Kaiser, “Phosphor: Illuminating dynamic data flow in commodity jvms,” in *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA)*, 2014, pp. 83–101.

[23] A. Zavou, G. Portokalidis, and A. D. Keromytis, “Taint-exchange: A generic system for cross-process and cross-host taint tracking,” in *Proceedings of International Workshop on Security*, 2011, pp. 113–128.

[24] X. Fu and H. Cai, “FlowDist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems,” in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 2093–2110.

[25] (2008) Apache hadoop yarn. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html/

[26] (2019) Apache activemq. [Online]. Available: https://activemq.apache.org/

[27] (2012) RocketMQ. [Online]. Available: https://rocketmq.apache.org/

[28] (2002) ASM. [Online]. Available: https://asm.ow2.io/

[29] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “libdft: Practical dynamic data flow tracking for commodity systems,” in *Proceedings of ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2012, pp. 121–132.

[30] X. Fu and H. Cai, “A dynamic taint analyzer for distributed systems,” in *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 1115–1119.

[31] (2020) Package java.lang.instrument. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

[32] (2013) etcd. [Online]. Available: https://etcd.io/

[33] (2003) Netty project. [Online]. Available: https://netty.io/

[34] (2012) STOMP: The Simple Text Oriented Messaging Protocol. [Online]. Available: https://stomp.github.io/

[35] (2008) Protocol buffers. [Online]. Available: https://developers.google.com/protocol-buffers

[36] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, “STILL: Exploit code detection via static taint and initialization analyses,” in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 289–298.

[37] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 259–269.

[38] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, “Iccta: Detecting inter-component privacy leaks in android apps,” in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2015, pp. 280–291.

[39] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, “Taintpipe: Pipelined symbolic taint analysis,” in *Proceedings of USENIX Security Symposium*, 2015, pp. 65–80.

[40] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, “Neutaint: Efficient dynamic taint analysis with neural networks,” in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1527–1543.

[41] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie, “Titian: Data provenance support in spark,” vol. 9, no. 3, p. 216, 2015.