FLOWDIST: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems

Xiaoqin Fu
Washington State University
xiaoqin.fu@wsu.edu

Haipeng Cai ✉
Washington State University
haipeng.cai@wsu.edu

Abstract

Dynamic information flow analysis (DIFA) supports various security applications such as malware analysis and vulnerability discovery. Yet traditional DIFA approaches have limited utility for distributed software due to *applicability*, *portability*, and *scalability* barriers. We present FLOWDIST, a DIFA for common distributed software that overcomes these challenges. FLOWDIST works at purely application level to avoid platform customizations hence achieve high *portability*. It infers implicit, interprocess dependencies from global partially ordered execution events to address *applicability* to distributed software. Most of all, it introduces a *multi-staged refinement-based* scheme for application-level DIFA, where an otherwise expensive data flow analysis is reduced by method-level results from a cheap pre-analysis, to achieve high *scalability* while remaining effective. Our evaluation of FLOWDIST on 12 real-world distributed systems against two peer tools revealed its superior effectiveness with practical efficiency and scalability. It has found 18 known and 24 *new* vulnerabilities, with 17 confirmed and 2 fixed. We also present and evaluate two alternative designs of FLOWDIST for both design justification and diverse subject accommodations.

1 Introduction

Tracking/checking dynamic information flow underlies various security applications (e.g., [73, 94, 96, 110, 116]). It addresses a general source-sink problem for a program execution, in which a *source* is where confidential or untrusted (i.e., *sensitive*) information is produced and flows into the program, while a *sink* consumes the information and makes it flow out of the program execution [40]. Due to its focused reasoning about actual executions, this approach has precision merits over statically inferring information flow.

One technique realizing the approach is to compute the chains of dynamic control/data dependencies hence infer full information flow paths between given sources and sinks during the execution (e.g., [94–96, 115]). We refer to this technique as dynamic information flow analysis (DIFA).

An alternative technique is to apply a tag to (i.e., *taint*) the data entering the program via the sources, propagate the taint tag during the execution, and check the data at the sinks against the presence of the tag (e.g., [42, 45, 59, 60, 64, 77, 80, 82, 83, 97, 102, 112, 114, 117, 120, 121, 124–126]). We refer to this technique as dynamic taint analysis (DTA). Unlike DIFA, DTA does not compute full information flow paths. DIFA thus provides better support in usage scenarios that require more detailed flow information (e.g., diagnosing data leaks by inspecting the full flow paths).

Yet current DIFAs are hardly applicable to multi-process programs, such as distributed systems (e.g., Voldemort [29], a distributed key-value store). The reason is that they rely on explicit dependencies (via references and/or invocations) among code entities, dismissing implicit dependencies across processes [67]. On the other hand, distributed systems widely serve critical application domains (e.g., banking, medical, social media), thus their security is of paramount importance.

Only a few existing DIFA/DTA tools (e.g., [80, 117]) overcame the *applicability challenge* by working at system level with platform customizations. However, keeping the customizations up with diverse and rapidly evolving platforms would be time-consuming and even infeasible, which constitutes a *portability challenge*. A purely application-level analysis would eliminate the need for platform customizations. Yet such an analysis faces a *scalability challenge* for two reasons. First, application-level dynamic analysis is known to generally incur substantial overheads. Second, working at a fine granularity for desirable precision, as well as the typically large size and great complexity of distributed software, adds further to the analysis costs.

In this paper, we present FLOWDIST, a purely application-level DIFA that addresses all the three challenges to work practically with common distributed software. The practicality goal here subsumes two specific aims: *scalability*—FLOWDIST should be scalable to real-world distributed systems, and *effectiveness*—it should be effective for discovering known and unknown (new) vulnerabilities in such systems at a reasonable level of accuracy. Our key insights for fulfilling these aims are as follows:

- Since a fine-grained information flow path is subsumed by a corresponding coarser-grained path, a cheap pre-analysis computing the latter can narrow down the scope of the former which may be quite expensive. This way, the overall analysis cost can be largely reduced without effectiveness loss, fulfilling the scalability aim.
- As the collection and use of various forms of program data come with different cost and effectiveness contributions to the cost-effectiveness of the entire analysis, carefully combining these data can help attain a practical level of accuracy while maintaining efficiency, which fulfills the effectiveness aim without sacrificing scalability.

Following these insights, FLOWDIST introduces a *multi-staged refinement-based* scheme for DIFA to attain high scalability, where a pre-analysis computes method-level information flow paths approximately but rapidly, followed by a fine-grained analysis that computes statement-level flow paths precisely as guided by the method-level results. Then, FLOWDIST adopts a hybrid scheme using various forms of data (i.e., method-level execution events, static dependencies and dynamic coverage both at statement level) to balance its cost and effectiveness. FLOWDIST addresses the portability and applicability challenges by working at purely application level while inferring implicit, interprocess dependencies from happens-before relations among executed methods across processes by partially ordering key execution (*entry*, *returned-into*) events of those methods. The slight compromise of precision (due to the method-level granularity) of the interprocess part of the DIFA is compensated by precise, ultimately statement-level analysis results within each process (i.e., *intraprocess* part of the DIFA), resulting in a practical level of accuracy overall.

To further understand the methodology for scalable, application-level DIFA, we have also developed two alternative designs of FLOWDIST: FLOWDIST*sim* and FLOWDIST*mul*. The first performs more static analysis while the second performs more dynamic analysis, both further reducing the overall analysis overhead under certain conditions. With these variants, FLOWDIST accommodates diverse user needs in providing the best cost-effectiveness tradeoffs for different kinds of distributed systems.

We implemented FLOWDIST and the two alternative designs for Java and applied them to 12 distributed systems of diverse scales, architectures, and domains, all of which are real-world systems. For various operational scenarios of these systems, FLOWDIST exhibited highly promising analysis accuracy and efficiency. For the given lists of sources/sinks (default ones in our study), FLOWDIST computes information flow paths between all possible source-sink pairs. For each subject execution, we sampled 20 flow paths when there were more; otherwise, we sampled all paths reported. FLOWDIST attained perfect precision and recall per our manual validation.

On average, FLOWDIST took 19 minutes for its one-off analyses for all possible information flow path queries (i.e., source-sink pairs) with respect to a given source/sink configuration and 13 seconds for each query, while incurring less than 1x run-time slowdown and negligible storage costs. We further validated the practical usefulness of FLOWDIST by using it to identify real vulnerability cases reported previously in public vulnerability databases (e.g, CVEs [31]). Out of 24 cases studied, FLOWDIST found 18, with the other 6 being missed because the respective vulnerabilities were not covered by the executions considered. It also revealed 24 new vulnerability cases in several of the studied industry-scale systems, of which 17 have been confirmed and 2 fixed already by the developers. In contrast to the only two state-of-the-art peer tools for Java that we could compare with, (one dynamic [47] and one static [76]), FLOWDIST exhibited superior effectiveness with practical efficiency and high scalability. None of the baselines found any of the existing and new vulnerabilities that FLOWDIST discovered.

Through FLOWDIST, we demonstrate a *general, refinement-based methodology* for cost-effective and scalable DIFA at purely application level, which can enable a number of applications beyond the scope of information flow security (e.g., system understanding and performance diagnosis) and the domain of distributed software (e.g., single-process concurrent programs). Our contributions and novelties are:

- The first purely application-level DIFA for common distributed software, FLOWDIST, which features a hybrid fine-grained data flow analysis that instantiates a multi-staged refinement-based methodology for DIFA to holistically overcome applicability, portability, scalability, and cost-effectiveness barriers with peer approaches (§3).
- Alternative designs of FLOWDIST that further explore the design methodology for DIFA to best accommodate distributed software of diverse scale/complexity (§4).
- An open-source implementation of FLOWDIST for Java that works with distributed software systems of various architectures and application domains (§5).
- Extensive empirical evaluations of FLOWDIST that show its practical effectiveness and scalability, as well as superior capabilities in vulnerability discovery, over two state-of-the-art approaches (§6).

The FLOWDIST artifact is available here [65], including the source code, experimental scripts, (installation, configuration, and usage) documentation, and relevant data sets.

2 Background and Motivation

We introduce distributed software systems and define the problem of DIFA for these systems as opposed to DTA. A real-world example is then given to motivate our work.

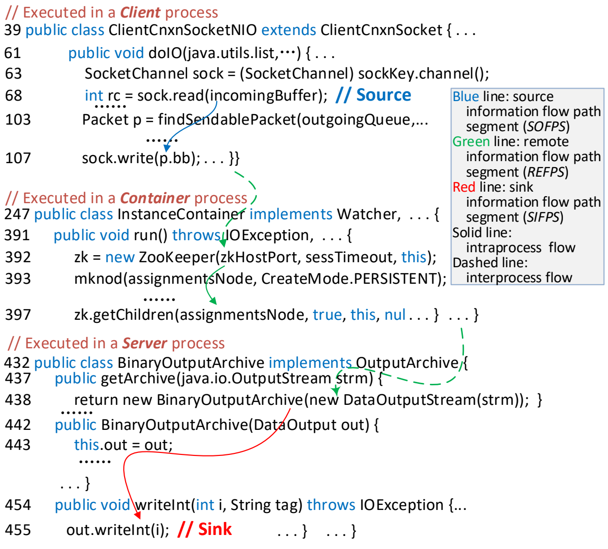
Distributed software systems. Driven by increasing demands for computational performance and scalability, increasingly more real-world software systems today are *distributed* by design [61]. We address systems for general-purpose distributed computing as defined in [61], noted as *common* distributed systems, as opposed to those of special types (e.g., RMI-based [113] or event-based [99]). In common distributed software, components located at networked computers communicate and coordinate their actions only by passing messages, while running concurrently in multiple (distributed) processes without a global clock.

Due to this decoupling, dependencies among distributed components (processes), noted as *inter-component* (*interprocess*) dependencies, are *implicit* [53]. Sensitive information can flow across decoupled components/processes via these implicit dependencies, leading to, among other issues, information flow security vulnerabilities that are missed by analyses based on explicit dependencies (as are most current techniques). Next, we use a real-world example to illustrate the need for analyzing such information flows.

DIFA versus DTA. These are two related techniques for tracking/checking dynamic information flows. While they have been treated equivalently and named exchangeably [60], we differentiate them (1) by their inner workings as mentioned earlier—DIFA works by computing dynamic dependencies while DTA works via data tainting and taint propagation, and (2) by their results—DIFA provides full information flow paths while DTA just tells which data is tainted.

On the other hand, both DIFA and DTA solve a source-sink problem, concerning information flow between given sources and sinks. For DIFA, we define a *source* as a function (call) producing information of interest (e.g., sensitive data) that flows into the program, and a *sink* as a function (call) that consumes the information and makes it flow out of the program. We refer to an exercised program path from a source to a sink as a (dynamic) *sensitive information flow* path. For multi-process programs (e.g,. distributed software systems), we divide an interprocess information flow path into three segments: *source information flow path segment* (*SOFPS*) and *sink information flow path segment* (*SIFPS*), consisting of only statements within the process that executes the source and the sink, respectively, and *remote information flow path segment* (*REFPS*) consisting of all other statements.

Motivating example. Figure 1 shows an excerpt from Apache ZooKeeper (v3.4.11), a popular distributed coordination service, where the sensitive flow is responsible for CVE-2018-8012 [32]. It revealed that when an Apache ZooKeeper server starts and attempts to join a quorum, there is no enforced authentication. As shown, the data-leaking flow exercised in the relevant execution crossed three processes: The sensitive data (a security key) was read into *incomingBuff* in class `ClientCnxnSocketNIO` of a *Client* process (at the *Source*), passed through class `InstanceContainer` of a *Container* process, and reached class `BinaryOutputArchive` of a *Server* process where the data leaked out of the system (at the *Sink*). This leakage caused an authentication failure when an endpoint attempted to join a quorum, which thus might propagate fake changes to the leader node of ZooKeeper.



```
// Executed in a Client process
39 public class ClientCnxnSocketNIO extends ClientCnxnSocket { . . .
61        public void doIO(java.utils.list,⋯) { . . .
63           SocketChannel sock = (SocketChannel) sockKey.channel();
68           int rc = sock.read(incomingBuffer);  // Source
              ......
103          Packet p = findSendablePacket(outgoingQueue,...
                          ......
107          sock.write(p.bb); . . . }}

// Executed in a Container process
247 public class InstanceContainer implements Watcher,  . . . {
391     public void run() throws IOException,  . . . {
392        zk = new ZooKeeper(zkHostPort, sessTimeout, this);
393        mknod(assignmentsNode, CreateMode.PERSISTENT);
                      ......
397        zk.getChildren(assignmentsNode, true, this, nul . . . }  . . . }

// Executed in a Server process
432 public class BinaryOutputArchive implements OutputArchive {
437     public getArchive(java.io.OutputStream strm) {
438        return new BinaryOutputArchive(new DataOutputStream(strm));  }
       ......
442     public BinaryOutputArchive(DataOutput out) {
443        this.out = out;
           ......
       . . . }
454     public void writeInt(int i, String tag) throws IOException {...
455       out.writeInt(i);  // Sink        . . . }   . . . }
```

Blue line: source information flow path segment (*SOFPS*)
Green line: remote information flow path segment (*REFPS*)
Red line: sink information flow path segment (*SIFPS*)
Solid line: intraprocess flow
Dashed line: interprocess flow

Figure 1: A case of sensitive information flow (marked by arrowed lines) in ZooKeeper across its three components (processes).

Suppose this case, along with the system execution that revealed it, is reported to a developer for diagnosis, to whom no platform customization is feasible. Purely application-level DIFA/DTA tools exist (e.g., [35]), which only track flows within the same processes (plus most of such tools only work for C/C++ programs). There are analyses that resolve data flows across decoupled components (e.g., [74, 119]), yet they do not work for common distributed software; and they are static hence would lead to excessive imprecision. We will demonstrate how FLOWDIST addresses these challenges.

3 Approach

This section elaborates FLOWDIST, starting with an overview, followed by design details.

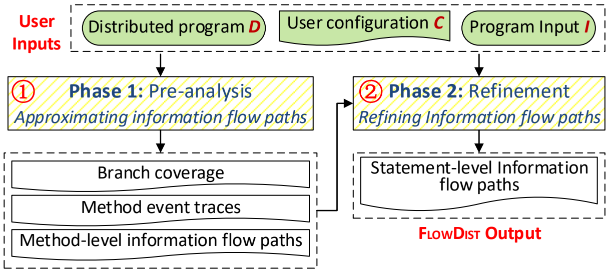


Figure 2: FLOWDIST overview: the cheap pre-analysis computes coarse flow paths to reduce the expensive fine-grained analysis which refines the coarse result.

3.1 Overview

Figure 2 depicts the overall workflow and two phases of FLOWDIST. The high-level idea is to achieve the scalability and effectiveness aims via a multi-staged refinement-based DIFA design, as guided by our key insights as outlined earlier.

FlowDist takes three **inputs**: the distributed program *D* under analysis, the run-time input *I* that drives the specific concrete execution of *D*, and a user configuration *C* that specifies the sources and sinks as common inputs required by DIFA/DTA. *Optionally*, a list of message-passing APIs that FlowDist recognizes in order to monitor interprocess communication (i.e., message-passing) events may be given in *C*. If the user does not specify this list, common message-passing APIs in the language (e.g., Java) SDK would be considered by default (as listed in [65]/Message_PassingAPIList.txt and in the Appendix).

With these inputs, FlowDist profiles method execution events and branch coverage hence computes method-level information flow paths in its pre-analysis phase (**Phase 1**) to narrow down the scope of later analyses that may be highly expensive (hence impede the overall scalability of FlowDist) otherwise. Then, in the refinement phase (**Phase 2**), FlowDist refines the method-level paths via a hybrid analysis of dynamic dependencies, using the method events and branch coverage. This phase produces statement-level flow paths as the eventual **output** of FlowDist.

Working example. To illustrate how FlowDist works, we will use the case of Figure 1 as a working example: running ZooKeeper against a system test, querying information flow paths between two methods (`java.nio.channels.SocketChannel read(java.nio.ByteBuffer)` and `java.io.DataOutput writeInt(int)`) as a source/sink pair (i.e., flow path query). For brevity, we will omit from our illustrations the callees not shown in the figure.

3.2 Pre-analysis (Phase 1)

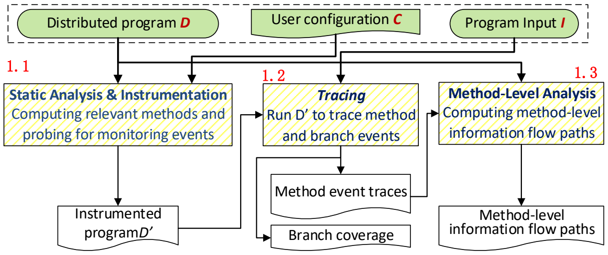

Distributed program D
User configuration C
Program Input I
1. 1
1. 2
1. 3
Static Analysis & Instrumentation
Computing relevant methods and probing for monitoring events
Tracing
Run D' to trace method and branch events
Method-Level Analysis
Computing method-level information flow paths
Instrumented programD'
Method event traces
Branch coverage
Method-level information flow paths

Figure 3: The process of FlowDist pre-analysis (Phase 1).

To enable a cost-effective DIFA, FlowDist uses (1) several forms of dynamic data and (2) static dependencies. Computing these, especially (2), can be too expensive to scale to large-scale systems, as we have empirically validated. The pre-analysis aims to *reduce the overall cost by narrowing down the scope of such computations*, in three steps as shown in Figure 3 and detailed below.

Static analysis and instrumentation (1.1). FlowDist utilizes three kinds of dynamic data in its hybrid analysis of dynamic dependencies to achieve a good cost-effectiveness balance, as inspired by prior work [70]: (1) two kinds of method execution events—*entry* (i.e., program control entering a method) and *returned-into* (i.e., program control returning from a callee into a caller), (2) two kinds of message-passing events—*sending/receiving a message*, and (3) branch coverage events—*a branch being exercised*.

To this end, we instrument *D* to probe for these events, with respect to the given or default message-passing APIs. Since only the methods on a static control flow path between a source-sink pair are likely to occur on a dynamic information flow path between the same pair, we only need to probe for the events of those methods, referred to as *relevant methods*. Accordingly, only branches in a relevant method (i.e., *relevant branches*) need to be probed. Thus, we start by constructing the interprocedural control flow graph (ICFG) of each distributed component in *D* and treat each message-sending and message-receiving API callsite in the component as an additional sink and source, respectively. Then, any method through which a sink is control-flow reachable from a source on the ICFG is identified as a relevant method.

Tracing (1.2). In this step, the instrumented program D' is executed against the program input *I*, during which the three kinds of events are traced (at instance level but only for relevant methods and branches). Given the absence of a global timing mechanism in common distributed software, FlowDist uses the Lamport time-stamping (LTS) [105] algorithm to derive the global partial ordering of the two kinds of method execution events, to derive the happens-before relations required for interprocess dependence inference. With LTS, each process maintains a logic clock locally, which may be updated by, or used to update, the local clocks of other (communicating) processes. The synchronization is realized by attaching the current values of local logic clocks to the messages transmitted among processes. Then, the synchronized logic clocks are used to time-stamp the method-execution events hence maintain the global partial ordering of all such events during *D*'s execution.

For each of the *n* processes (P_i) in the execution, besides the trace (T_i) of the partially-ordered, time-stamped method execution events, a mapping ($p2fm[i]$) is produced to keep the timestamp ($p2fm[i][j]$) of each message-passing event of P_i receiving the first message from a process P_j ($i, j \in [1,n], j \neq i$). This mapping is used to enhance the precision of the interprocess dependence inference.

Method-level analysis (1.3). With the event traces and mapping from Step 1.2, FlowDist then identifies (from *D*) the list *SO* (resp., *SI*) of the enclosing methods of each source (resp., sink) in *C* and computes the method-level paths according to Algorithm 1. The key idea is to combine method-level control flow and process-level data flow for a dynamic method-level dependence approximation.

The algorithm searches paths *ps* by traversing the *n* per-process traces (lines 2-11). In each trace T_i, the set S_d of covered source-enclosing methods is obtained (line 3). No path would start in P_i (corresponding to T_i) if there is no source executed in P_i (line 4). Otherwise, for each method *q* in S_d, the algorithm attempts to identify paths starting at *q* by

Algorithm 1 Computing method-level flow paths

let SO and SI be the list of source and sink enclosing methods, respectively
let T_i be time-stamped method execution event trace in process P_i, $i \in [1,n]$

```
 1: $ps = \emptyset$        // initialize the set of all method-level paths between the given pair
 2: for $i$=1 to $n$ do        // traverse the $n$ processes of the given execution
 3:     $S_d = \{s | s \in SO \wedge s \in T_i\}$
 4:     if $S_d == \emptyset$ then continue
 5:     for each method $q \in S_d$ do        // first infer intraprocess dependencies
 6:         $DS(q) = \{m | m \in T_i \wedge fe(q) \leq lr(m)\}$
 7:         for $j$=1 to $n$ do        // then infer interprocess dependencies
 8:             if $i==j \vee p2fm[i][j]==null$ then continue
 9:             $DS(q) \cup= \{m | m \in T_j \wedge fe(q) \leq p2fm[i][j] \leq lr(m)\}$
10:         if $DS(q) \cap SI == \emptyset$ then continue
11:         $ps \cup= \{< m1, ..., m_k > | m1 == q \wedge m_k \in SI \wedge$
      $\forall_{i<j,\ i,j \in [1,k]} fe(m_i) \leq lr(m_j) \wedge \forall_{i \in [1,k]} m_i \in DS(q)\}$
12: return $ps$
```

computing its dynamic dependence set $DS(q)$ (lines 5-10).

Let $fe(m)$ and $lr(m)$ be the timestamp of the first entry and last returned-into event of a method m, respectively. First, the *local* (intraprocess) dependencies are identified (line 6) according to the happens-before relation between q and each other method m executed in P_i (which is treated as a *local* process). The rationale is that a method $m2$ is not dependent on a method $m1$ if $m2$ has never executed after $m1$ [51].

Then, dependencies in every other (*remote*) process P_j are identified (lines 7-9). If P_j never sent a message (line 8) or the timing of message passing implies no dependence, relevant methods in P_j are dismissed; otherwise, they are added to $DS(q)$ (line 9). The rationale is that, suppose $m1$ and $m2$ execute in two processes $p1$ and $p2$, respectively, $m2$ depends on $m1$ only if $p2$ receives at least one message before $lr(m2)$ that is sent (directly or transitively) from $p1$ after $fe(m1)$.

Once $DS(q)$ is computed, if it includes a sink-enclosing method m_k (line 10), the partial ordering of relevant methods in $DS(q)$ forms an information flow path from q to m_k. All such paths are gathered into ps (line 11) and returned (line 12).

Illustration. For the working example, FLOWDIST first identifies relevant methods (e.g., `doIO` in the client component and `getArchive` in the server component) and branches in them to probe for. After the tracing, the methods in the resulting traces are ..., `doIO`, `run`, `getArchive`, `BinaryOutputArchive`, `writeInt`, ... Then, in Algorithm 1, all these methods are included in dependent set of `doIO`, the only source-enclosing method here. The global partially-ordered sequence between this method and `writeInt` (the only sink-enclosing method here) gives the method-level information flow path: `doIO` → `run` → `getArchive` → `BinaryOutputArchive` → `writeInt`.

3.3 Refinement (Phase 2)

In this phase, FLOWDIST aims to produce fine-grained information flow paths by refining the coarse (method-level) results computed in Phase 1, in three steps as shown in Figure 4 and detailed below. To this end, it leverages program data of two modalities (*static* and *dynamic*) and two granularity levels (*method* and *statement*). The primary motivation for this highly hybrid design is that *it may offer a practically competitive balance between the analysis precision and the total analysis cost* [52].

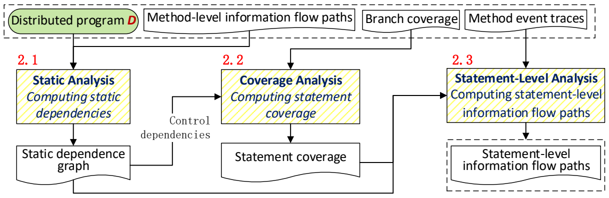


Figure 4: The process of FLOWDIST refinement (Phase 2).

Static analysis (2.1). A key enabler for FLOWDIST to attain its design goal is that it utilizes fine-grained (statement-level) static dependencies, represented as a graph. More importantly, the graph only needs to be *partial* (as opposed to a whole-system dependence graph [78])—only static dependencies involving methods on the flow paths from the pre-analysis are computed. The rationale is that *any method via which the sensitive information originated at a source s reaches a sink t must be on a method-level information flow path between the enclosing method of s and that of t.* The main idea here is that, while performing data/control flow analysis for computing varied kinds of dependencies (as detailed below), the analysis stops interprocedural propagation of relevant flow facts whenever it encounters a method that is not on the method-level path.

Specifically, FLOWDIST computes traditional control and data dependencies [78] within each thread, followed by computing dependencies across threads including threading-induced control (*ready* and *synchronization* [72]) and data (*interference* [101]) dependencies. The static dependence analysis here is chosen to be context-insensitive because its results are only used in Step 2.3 where the method-execution events used will provide the necessary contexts; further, its interprocedural analysis part is chosen to be flow-insensitive because those events are ordered (by their timestamps). The intraprocedural analysis part remains flow-sensitive, though, as those events are at method level. These choices reduce the total analysis cost of FLOWDIST.

The static dependencies *across* distributed components are not computed due to their implicit nature. Thus, the resulting dependence graph of the entire system consists of disconnected subgraphs (each for one component/process, as illustrated in Figure 5). FLOWDIST builds each subgraph by considering all possible entry points of D, without purposely separating/recognizing the components.

Coverage analysis (2.2). This step aims to generate per-process statement coverage, the only fine-grained dynamic data used to refine the hybrid analysis in Step 2.3. This is done by referring to the static dependencies from

Algorithm 2 Computing statement-level flow paths

```
let $SC$ be the set of statements covered across all processes
let $T_i$ be time-stamped method execution event trace in process $P_i$, $i \in [1,n]$
let $sDG$ be the partial static dependence graph
let $<s,t>$ be a source-sink callsite pair between which paths are computed
let $outlets$ be the list of all outlets
let $inlets$ be the list of all inlets
1: $SOFPS = \emptyset$, $REFPS = \emptyset$, $SIFPS = \emptyset$, $intraFP = \emptyset$
2: merge $T_i$, $i \in [1,n]$ into a global partially ordered sequence $ES$
3: $dDG$ = buildDyndepGraph($sDG$, $<s,t>$, $ES$)          // hybrid analysis
4: $dDG'$ = pruneDyndepGraph($dDG$, $SC$)          // statement-level pruning
5: $SOFPS$ = findPaths($dDG'$, $\{s\}$, $outlets$, $tr(s)$)
6: for $i$= to $n$ do          // compute remote segments of interprocess paths
7:     $intraFP \cup=$ findPaths($dDG'$,$\{s\}$,$\{t\}$,$S_j$)          // intraprocess paths
8:     if $tr(s) == T_i \vee tr(t) == T_i$ then continue
9:     $REFPS \cup=$ findPaths($dDG'$, $inlets$, $outlets$, $T_i$)
10: $SIFPS$ = findPaths($dDG'$, $inlets$, $\{t\}$, $tr(t)$)
11: return [spliceSegs($SOFPS$, $REFPS$, $SIFPS$), $intraFP$]
```

Step 2.1 and branch coverage from Step 1.2—statements control dependent on a covered branch are considered covered. Importantly, during this inference, only methods on any method-level path found in Phase 1 are considered. The insight is that *statements in other methods will not appear on the final (statement-level) information flow paths*.

Statement-level analysis (2.3). With the covered statements (SC), per-process method event traces (T_i), and partial static dependence graph (sDG) of D, FLOWDIST now computes statement-level information flow paths between each source-sink callsite pair ($<s,t>$) with Algorithm 2. It identifies the callsites of message sending and receiving APIs (within the methods on the method-level flow paths), referred to as *outlets* and *inlets*, indicating where information flows out from and into each process, respectively.

First, per-process sequences of method events are merged as a whole-system event sequence ES (line 2) ordered by event timestamps. Then, the subroutine buildDyndepGraph constructs a dynamic dependence graph dDG (line 3) by referring to the static dependencies in sDG while traversing ES, using a hybrid dependence analysis inspired by DIVER [51]. The key idea is summarized as follows. First, interprocedural dependencies in sDG are categorized into two classes: *adjacent* (due to parameter or return-value passing) and *posterior* (due to the def-use associations of heap variables and control dependencies). Next, when scanning ES, a static dependence of a method $m2$ on another method $m1$ is activated (hence added to dDG) if (1) that dependence is adjacent and $m2$ happens immediately after $m1$ in ES or (2) the dependence is posterior and $m2$ happens anywhere after $m1$ in ES. The analysis treats all static *intraprocedural* dependencies in a method that is in ES as activated and adds them to dDG. And the graph construction starts with s and only includes dependencies that reach t.

The resulting dependencies (in dDG) would be imprecise at statement level. Thus, FLOWDIST proceeds with a subroutine pruneDyndepGraph which prunes spurious dependencies in dDG per the statement coverage SC: nodes corresponding to unexercised statements and their associated edges are removed from the graph, resulting in the pruned graph dDG' (line 4). While the pruning may still leave spurious dynamic dependencies [39] in dDG', we make this choice to contain the overall analysis cost of FLOWDIST to gain in scalability.

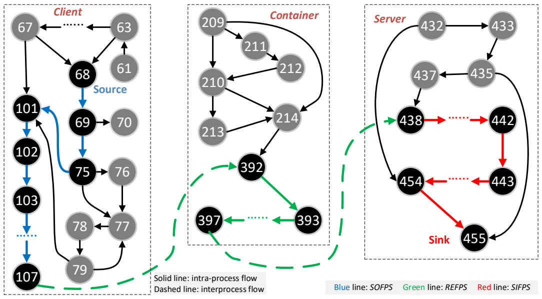


Figure 5: The partial static dependence graph created in (the Step 2.1 of) Phase 2 for the working example.

With the dDG', the algorithm then computes both intraprocess and interprocess information flow paths with findPaths(G,X,Y,T), a subroutine that finds paths from any statement in X to any statement in Y on a graph G while only considering nodes in T. Intraprocess paths (*intraFP*) in each process are computed by simply traversing dDG' (line 7).

For any interprocess flow path, however, the sink is not explicitly reachable from the source on dDG' because it (as a projection of sDG) remains disconnected. FLOWDIST computes the three segments separately (§2). First, the segment within the source (s)'s process (*SOFPS*) is computed via a traversal on dDG' (line 5) that retrieves paths from s to a relevant outlet within that process's trace—$tr(x)$ denotes the trace that includes an event of the method that encloses a statement x. The segment within the sink (t)'s process (*SIFPS*) is computed similarly (line 10), but by searching paths from any inlet to t. The remaining segment (*REFPS*) is searched within each process (lines 6-9) other than the one that encloses s or t (line 8). The search is again realized through a traversal on dDG', looking for paths from any inlet to any outlet within the process. Finally, these segments are spliced into interprocess information flow paths with the subroutine spliceSegs, according to the timestamps of relevant inlets/outlets. The splicing works such that there are not any events between the end of an *SOFPS* and the start of an *REFPS*, nor between the end of the *REFPS* and the start of an *SIFPS* as per the global partially ordered sequence ES. With the intraprocess paths, these spliced interprocess paths are then returned as the output of this algorithm (line 11).

Illustration. For the working example, Figure 5 depicts the dDG (i.e., before pruning), including three subgraphs each for one of the three processes *Client* (left), *Container* (middle), and *Server* (right)—only the part for the code of Figure 1 is shown. FLOWDIST then infers the covered statements from the branch coverage obtained in Phase 1: 68, ..., 103, ..., 107, 392, 393, ..., 397, ..., 438, ..., 442, 443, ..., 454, 455.

The dark solid nodes indicate covered statements on the corresponding dDG', among which {107,397} are outlets and {392,438} are inlets—in the example code, the actual message-sending/receiving APIs are invoked within some of the relevant methods shown (e.g., ZooKeeper(...) at line 392). The grey nodes are those pruned away per the statement coverage. After executing Algorithm 2, *SOFPS*=<68, 69, 75, 101, 102, 103, ..., 107>, *REFPS*=<392, 393, ..., 397>, and *SIFPS*=<438, ..., 442, 443, ..., 454, 455>. Splicing these segments leads to the entire path <68, 103, ..., 107, 392, 393, ..., 397, 438, 442, 443, ..., 454, 455>, as also highlighted in arrowed lines of Figure 1. *intraFP*==∅ as there is no intraprocedural information flow path between the source-sink pair in this example.

4 Alternative Designs

The default design of FlowDist as presented above targets common distributed systems in general. To more systematically explore the multi-staged refinement-based methodology for DIFA, we have developed two alternative designs: FlowDist*sim* and FlowDist*mul*. They may offer even greater cost-effectiveness and scalability for systems that meet certain conditions, by further reducing analysis costs while without compromising soundness and precision.

FlowDist*sim*: In the Step 1.1 of FlowDist, the goal of the static analysis (i.e., ICFG construction) is to reduce the instrumentation scope hence the costs of tracing method and branch events. Yet with certain systems, probing for and tracing all such events is cheap, and the cost incurred by this static analysis itself may outweigh the cost reduction. Optimized for systems meeting these conditions, FlowDist*sim* skips the static analysis, and simply instruments all methods and branches in *D* in this step, with the rest being the same as FlowDist.

FlowDist*mul*: With some systems, the FlowDist*sim* design is well justified. Yet probing for and then tracing all method and branch events in *D* incurs substantial costs. To reduce these costs, we introduce an intermediate phase, with two more changes, to FlowDist*sim*. The idea is to have a multi-staged refinement-based design in Phase 1 itself. First, the new Phase 1 only probes for and traces the first entry and last returned-into events of each method, and then computes method-level flow paths from those events. The intermediate phase then probes for and traces the coverage of branches in, and all instances of both kinds of events of, methods on such paths. Lastly, the Step 2.2 is removed from Phase 2.

Since FlowDist*mul* requires multiple executions of the same system against the same input (in the first and intermediate phases), this design is optimized for systems with deterministic executions—inconsistencies between the two executions could compromise the soundness of the DIFA as a whole. Another condition is that the cost reduction outweighs the costs incurred by the intermediate phase.

According to the rationale of each alternative design, FlowDist*sim* is expected to perform the best for small/simple systems with non-deterministic executions, while FlowDist*mul* is the best for such systems without non-deterministic executions. For large/complex systems, FlowDist would perform the best. These contrasts are justified by the *conditions* (as described above) under which either alternative design is motivated and best fits; when none of those conditions are met, FlowDist is superior in general.

5 Implementation and Limitations

We implemented FlowDist and its alternative designs for Java based on Soot [88] while reusing our dependence analyzers [48, 49]. Our tools take Java bytecode directly and account for data/control flows due to exception-handling constructs and reflection. For computing threading-induced dependencies, we reused relevant parts of Indus [108]. Additional implementation details can be found in [66, 68].

Due to their common inability to *fully* analyze dynamic language features (e.g., complex cases of reflection, native code) the static analyses in our tools are soundy [92] but not sound [79]. Since they compute information flow paths based on dynamic dependencies projected from static ones, while considering a specific system execution only, our tools may suffer from false negatives (akin to under-tainting in DTA).

Our tools do not address the problem of identifying the sources/sinks of interest, which are assumed to be given in the default source/sink lists or specified differently by users. Also, as with any dynamic analysis, the analyses in our tools are limited to the program parts that are exercised at runtime. Thus, their capabilities of discovering a bug rely on that (1) the relevant source and sink are specified and (2) the source and sink are covered by the run-time inputs considered. Moreover, considering the security context in specific usage scenarios (e.g., external protection mechanisms applied to the source or sink), our tools may suffer from false positives as they do not analyze, nor have access to, those external/context factors.

Finally, our tools require static instrumentation, thus they may not suit scenarios where the system cannot be modified. Additional limitations of FlowDist*sim* and FlowDist*mul* are those implied by the respective system conditions discussed earlier (e.g., the system execution is deterministic).

6 Evaluation

Our evaluation was guided by the following questions:

RQ1 *How effective is* FlowDist *in terms of its precision?*
RQ2 *How efficient is* FlowDist *in terms of its costs?*
RQ3 *How scalable is* FlowDist*?*

Table 1: Subject distributed programs and test inputs used

Subject	#SLOC	#Method	Scenario	Tests
NIOEcho	412	27	Client-Server	Integration
MultiChat	470	37	Peer-to-Peer	Integration
ADEN	4,385	260	Peer-to-Peer	Integration
Raining Sockets	6,711	319	Client-Server	Integration
OpenChord	9,244	736	Peer-to-Peer	Integration
Thrift	14,510	1,941	Client-Server	Integration
xSocket	15,760	2,209	Peer-to-Peer	Integration
ZooKeeper	62,194	5,383	Client-Server	Integration
			N-tier	Load
			N-tier	System
RocketMQ	105,444	6,198	N-tier	Integration
			N-tier	System
Voldemort	115,310	20,406	Client-Server	Integration
			N-tier	Load
			N-tier	System
Netty	167,961	12,389	N-tier	Integration
HSQLDB	326,678	10,095	Client-Server	Integration
			N-tier	System

RQ4 *Can* FLOWDIST *find real-world vulnerabilities?*
RQ5 *Can* FLOWDIST *discover new vulnerabilities?*
RQ6 *How does* FLOWDIST *compare to the state of the art?*
RQ7 *How well do the alternative designs perform?*

6.1 Experiment Setup

As shown in Table 1, we used 12 Java distributed systems as subjects. The subject sizes are measured by numbers of non-blank non-comment Java source code lines (*#SLOC*), numbers of methods defined in the subject (*#Method*), and execution scenarios (*Scenario*) including client-server, peer-to-peer, and n-tier. The last column lists the kinds of tests available to us, from which the run-time inputs are drawn.

NioEcho [27] provides an echoing service for any message sent by clients. MultiChat [26] is a chat service broadcasting messages received from one client to others. ADEN [25] offers a UDP-based alternative to TCP sockets. Raining Sockets [24] is a non-blocking and sockets-based framework. OpenChord [28] is a peer-to-peer network service. Thrift [33] is a framework for developing scalable cross-language services. xSocket [34] is an NIO-based library for building high-performance computing (HPC) software. ZooKeeper [30] is a coordination service achieving consistency and synchronization in distributed systems. RocketMQ [38] is a distributed messaging platform. Voldemort [29] is a distributed key-value store underlying LinkedIn's services. Netty [37] is a framework for rapid HPC application development. HSQLDB (HyperSQL DataBase) [36] is an SQL relational database system.

We chose these subjects to cover various scales, application domains, architectures, and mechanisms for message passing. The system and load tests were part of the software packages downloaded from the respective project websites. The integration tests were created manually as per the official documentation of each subject with concrete inputs. Both valid and invalid inputs were considered. For each of these tests, we ran two to five processes each on a different machine per the typical use of each subject.

In each integration test, we started several server/client instances and performed various operations, to cover main subject features. Particularly for ADEN, Raining Sockets, Thrift, xSocket, and Netty, which are frameworks/libraries, we developed an application for each to cover its major functional features and then exercised each of the applications. The following are brief descriptions of operations and test inputs involved in each integration test.

- NioEcho: We started a server and a client, sent random text messages from the client to the server, and then waited for the echo of each message.
- MultiChat: We started a server and three clients. From one client we sent random text messages to the server which broadcasted them to all other clients.
- ADEN: We started two nodes each of which sends messages to and receives messages from the other node.
- Raining Sockets: We started a server and a client, and then the client sent text messages to the server.
- OpenChord: We first started three nodes A, B, and C. Then, we performed following operations: On node A, create an overlay network; on the other nodes B and C, join the network; on the node C, insert a new data entry to the network; on the node A, search and then remove the data entry; Lastly, on the node B, list all data entries.
- Thrift: With a server and a client, a calculator application was developed. The client sent some basic arithmetic operations (addition, subtraction, multiplication, and division of two numbers, in order) to the server and got the calculation results from the server.
- xSocket: Two nodes were started and then each sends messages to the other node.
- ZooKeeper: Our operations were: create two nodes, search them, look up their attributes, update their data association, and remove these two nodes.
- RocketMQ: There are four components: a name server, a broker, a producer, and a consumer. The server provides reading and writing service and records full routing information. The broker stores messages. The producer sends messages to the broker. The customer receives messages from the broker.
- Voldemort: We performed the following operations in order: add a key-value pair, find the key for its value, remove the key, and retrieve the pair.
- Netty: We develop a 3-tier application with three nodes. The first node read an email list from a file and then sent relevant emails to the second node. Next, the second node encrypted the emails using the RSA algorithm and then sent them to the third node. Lastly, the third node used Postfix to send emails received.
- HSQLDB: We started a database server and a client. Then, the client sent a SQL query to the server and then received the SQL result from the server.

We evaluated FlowDist via its implementation for Java, thus we set the sources and sinks (found in [65]/data) based on our understanding of security-related APIs in the Java SDK, as default. We used the list of message-passing APIs (§3.1) in the Java SDK to cover Java Socket I/O, ObjectStream I/O, and Java NIO APIs (as listed in [65]/Message_PassingAPIList.txt).

6.2 Experimental Methodology

Given the default user configuration, we considered pair-wise pairing of all sources and sinks as queries against each subject execution. Due to the absence of ground truth, for each query we manually checked the (statement-level) information flow paths produced by FlowDist to compute precision.

Specifically, for each path, we tracked the dependencies of the source; then we considered the path a true positive if we reached the sink without encountering any sanitization via the path, and a false positive otherwise. FlowDist does not support sanitization at the moment—its current implementation does not check if a resulting flow path contains sanitizing operations. Yet among the paths we examined, we did not find sanitized ones. Also, we manually constructed the ground truth for three subjects to evaluate recall. In each case of manual analysis, the two authors and a non-author CS graduate student each inspected independently; then they cross-validated and confirmed the result when all three concurred. The manual check was time-consuming, thus we randomly sampled only 20 paths when there were more (otherwise we checked them all). We avoided taking more than one path between each pair to reduce biases.

Regarding efficiency, we computed FlowDist's time and storage costs for each query and reported the average-case numbers over all the queries per execution, in addition to run-time slowdowns and static analysis costs. To evaluate scalability, we used linear regression to model how those numbers vary with changing code and trace sizes.

We are not aware of a prior DIFA/DTA, nor a fine-grained dynamic data flow analysis that could serve the same purpose, that works with diverse real-world distributed systems. Thus, we compare FlowDist with Phosphor [47] and Joana [76], the state-of-the-art dynamic and static taint analyzers for single-process Java software, respectively. Our study considered only this single baseline DIFA/DTA because our extensive search for such tools and contact with the authors of relevant papers ended up with no more comparable tools to include (as further discussed in §7). We chose to include Joana to see how DIFA/DTA tools are compared with static ones. The machines we used were all Ubuntu 16.04.3 LTS workstations with an Intel E7-4860 2.27GHz CPU and 32GB DMI RAM.

6.3 Results and Analysis

We now discuss our results for each RQ. We focus on major findings while discussing the key implications and insights.

Table 2: Numbers of intraprocess (*Ir*) source/sink pairs (*Pr*) and information flow paths (*Ps*), versus interprocess (*Int*) ones

Execution	#IrPr	#IrPs	#IntPr	#IntPs	IntPs/AllPs
NioEcho	66	21	12	6	22.22%
MultiChat	42	0	12	0	0.00%
ADEN	0	0	5	0	0.00%
Raining Sockets	12	3	0	0	0.00%
OpenChord	14	0	24	0	0.00%
Thrift	4	0	4	3	100.00%
xSocket	10	8	26	2	20.00%
Zookeeper Integration	9	0	33	0	0.00%
Zookeeper Load	1086	1	6522	64	98.46%
Zookeeper System	124	0	1116	46	100.00%
RocketMQ Integration	19	23	46	17	42.50%
RocketMQ System	24	0	187	50	100.00%
Voldemort Integration	198	30	193	138	82.14%
Voldemort Load	6	0	6	0	0.00%
Voldemort System	80	30	77	42	58.33%
Netty	9	3	7	2	40.00%
HSQLDB Integration	140	10	668	0	0.00%
HSQLDB System	7	2	11	4	66.67%

6.3.1 RQ1: Effectiveness (Precision/Recall)

Table 2 shows the number of source-sink pairs covered in each execution (i.e., subject-test type) and that of information flow paths between the pairs, separately for intraprocess and interprocess paths. For each source/sink given in the configuration *C*, FlowDist treated each of its exercised callsites as a separate source/sink in counting the pairs and computing the paths. The last column shows the percentage of interprocess paths over all information flow paths per execution. The rows for executions without any information flow paths found are greyed.

The numbers of exercised source/sink pairs and information flow paths varied widely and were generally independent of subject size and input type. In 5 of the 18 cases (executions), FlowDist found no sensitive flow (e.g., for Voldemort-Load). In the other 13 cases, the paths were all true positives. Between any of the pairs in Thrift, Voldemort-Load, and Netty—the cases with smallest total numbers of pairs, we found no path beyond those found by FlowDist. Thus, the precision and recall were both 100% for the manually validated samples.

The majority (74% on average) of *all* of the reported paths were *interprocess* ones—in 7 cases the percentage was above 50% and in 3 cases 100%. This implies that, by only analyzing dynamic information flows *within* individual processes, a conventional DIFA/DTA would miss most of the sensitive flows in distributed program executions. This result also provides an alternative measure of recall of FlowDist versus single-process DIFA/DTA, and indicates the much higher recall of our approach.

More generally, while our evaluation on recall was limited due to the lack of ground truth and the impracticality of manually curating it for all queries (especially for large systems with complex executions), high recall (hence a low false negative rate) is crucial, especially in the context of finding security vulnerabilities. Meanwhile, we note that, relative to a static approach, the generally lower recall of a dynamic technique like ours is mainly attributed to the limited coverage of run-time inputs considered. On the other hand, a dynamic analysis is expected in nature to focus on the particular inputs (hence the specific executions) *given* by users. Thus, the input coverage problem is considered orthogonal to the design of a dynamic analysis [79]. With respect to the given executions, both our manual validation for RQ1 and evaluations against real vulnerability cases for following RQs confirmed that FLOWDIST found all of the information flow paths and related vulnerabilities, suggesting no false negatives for those executions.

In addition, the precision and recall of a hybrid analysis (as is the Step 2.3 of FLOWDIST) often compete with each other [85]. However, in our approach, we strive for precision improvement over a purely dynamic dependence analysis based on method-level control flows, by conservatively pruning static dependencies with those exercised control flows. This conservative nature leads to the ability of FLOWDIST to retain recall when gaining in precision.

Interprocess flow analysis is essential for a DIFA/DTA of common distributed systems. Manual validation suggested FLOWDIST*'s very-high precision and promising recall.*

6.3.2 RQ2: Efficiency (Time/Storage Costs)

Table 3 gives the breakdowns of the time and storage costs of FLOWDIST over its two phases and further over the steps of each phase. The time costs include those for static analysis (and instrumentation if any) (*St.*), profiling (*Run*), and on average for computing the (method- or statement-level) paths between each source-sink pair (*Query*). The second column lists the original run time (*Norm Run*) of each execution, from which profiling overheads were computed as runtime slowdown ratios (*Slowdown*). The eighth column shows the time for coverage analysis (*Co.*). The last column is the total storage cost (*Storage*) for all phases per execution—for storing the traces of method and branch events in Phase 1, statement coverage and partial static dependence graph in Phase 2, as well as the instrumented program. The overall averages (across all executions) are given in the bottom row.

On average over the 18 cases (executions), FLOWDIST took 19 minutes for all one-off analyses, including the time for all static analyses, instrumentation, and coverage analysis. We considered them *one-off* because their results are shared by all queries with respect to a given subject execution and source/sink configuration. In particular, the *partial* dependence analysis (as guided by the method-level paths from Phase 1) was significantly more efficient than a whole-system analysis (without a pre-analysis). For instance, per our additional experiments, the latter did not even finish in 12 hours with otherwise the same setup against Voldemort.

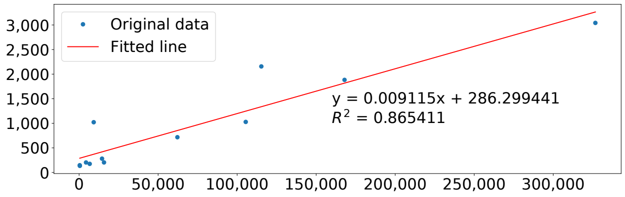


Figure 6: The total analysis time (seconds, y axis) versus subject size (#SLOC, x axis) of all subjects (integration test).

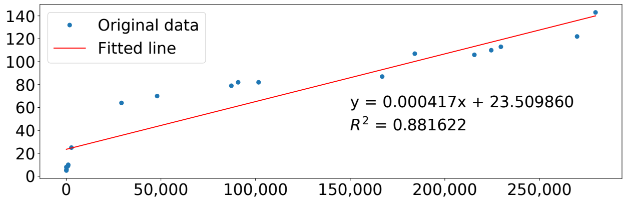


Figure 7: The run-time slowdowns (%, y axis) versus #method execution event instances (x axis) of all subject executions.

For profiling, FLOWDIST caused an average of 68% slowdown calculated as (T_i-T_o)/T_o where T_i and T_o is the run time of the instrumented and original program, respectively.

The time cost for querying each source/sink pair was 13 seconds on average, with a maximum of 50 seconds seen by HSQLDB-System mainly because of its static dependence complexity. Note that this cost was dominated by building the dynamic dependence graph from its static counterpart and an instance-level method execution event sequence (Algorithm 2), whose time expense depends on the scale of the graph and the length of the sequence.

The storage costs of FLOWDIST were all insignificant.

FLOWDIST *is promisingly efficient and scalable to large systems, taking on average 19 minutes by one-off analyses and 13 seconds to query for a source/sink pair while causing <1x slowdown and a negligible storage cost.*

6.3.3 RQ3: Scalability

We first look at how FLOWDIST scaled to subjects of growing sizes in terms of its total time cost (the sum of one-off analysis time, profiling costs, and the time for querying all possible source/sink pairs), against integration tests since every subject has such a test. Figure 6 shows the fitting curve, along with the determination coefficient $R^2 \in$[0,1] which indicates how close the data are to the curve. The closer R^2 is to 1, the better the fitting is. As shown, FLOWDIST's time cost grew linearly.

We then look at the scalability of FLOWDIST in terms of its runtime slowdown, for all 18 executions each characterized by the length of the instance-level method execution event sequence in it as a run-time complexity measure. In the same format as Figure 6, Figure 7 shows the fitting curve with R^2>0.88, indicating that FLOWDIST scaled gracefully to large-scale systems in terms of the runtime overhead.

Table 3: Time (in seconds) and storage (in MB) costs of FLOWDIST

Executions	Norm	Phase 1 Time				Phase 2 Time			Storage
	Run	St.	Run	Slowdown	Query	St.	Co.	Query	
NioEcho	39	53	41	5.16%	0.2	50	1	1.0	1.6
MultiChat	26	55	28	6.12%	0.2	50	1	0.1	1.0
ADEN	21	117	23	10.23%	0.3	59	3	0.3	4.0
Raining Sockets.	6	40	6	7.67%	0.3	122	6	0.4	14.5
OpenChord	54	177	59	8.54%	0.3	740	41	4.7	26.7
Thrift	8	146	10	24.83%	0.5	79	45	0.6	26.1
xSocket	11	101	19	63.99%	0.5	70	14	0.1	29.3
Zookeeper Integration	71	292	121	70.16%	0.5	193	108	1.8	231.2
Zookeeper Load	99	292	177	78.83%	0.6	137	67	2.0	404.0
Zookeeper System	98	292	178	81.87%	0.5	250	93	1.1	417.5
RocketMQ Integration	105	56	196	87.05%	0.6	704	49	21.5	291.0
RocketMQ System	339	156	753	122.09%	0.6	727	52	34.0	463.2
Voldemort Integration	28	1206	58	106.06%	0.6	566	317	9.1	560.4
Voldemort Load	11	1206	23	113.37%	0.6	435	260	14.4	523.1
Voldemort System	31	1206	65	109.81%	0.6	618	344	22.2	545.1
Netty	12	1132	22	81.65%	0.6	381	317	30.1	417.6
HSQLDB Integration	9	659	19	107.46%	0.7	2227	96	41.5	591.1
HSQLDB System	15	684	36	142.71%	0.7	2771	408	49.7	733.7
Overall Average	**55**	**437**	**102**	**68.20%**	**0.5**	**565**	**124**	**13.0**	**293.4**

Both the total analysis time and runtime overhead of FLOWDIST *grew linearly with the growth of subject and trace sizes, suggesting its high scalability in practice.*

6.3.4 RQ4: Finding Real-World Vulnerabilities

We searched real-world vulnerabilities from varied sources (e.g., bug repositories and CVE reports) on our subjects and then selected those on information flow security. We identified one or more vulnerabilities for 7 of our studied subjects, as shown in Table 4. For each of these subjects, cases along with reference links are listed, with marks indicating which was found and which was missed. The last column gives the numbers of false negatives (*#FN*).

We started with the information flow paths computed in our experiments for RQ1 and RQ2 (i.e., the paths between all the sources and sinks in the default lists). Next, for each of the known vulnerabilities, we narrowed the search down to the paths between the source/sink that are most relevant to the vulnerability according to its bug report/description, while navigating the associated subject's code to gain more confidence. Finally, we considered that FLOWDIST found the vulnerability case if any of those paths is responsible for the vulnerability as per the bug report/description.

FLOWDIST successfully found most of the cases for all these 7 subjects but Netty. 5 cases for Netty and 1 for Voldemort were missed by FLOWDIST. The reason, as we verified, was that the missed vulnerabilities were not exercised during the executions we considered—we did not purposely select run-time inputs to cover the vulnerabilities but just used those available to us to represent the operational scenarios of these systems. We note that *for all the 18 successful cases the underlying information flow paths were interprocess ones*.

FLOWDIST *found 18 out of 24 vulnerability cases related to our subjects, all on interprocess flow paths. The other 6 were missed as the respective vulnerabilities were not covered by the executions analyzed.*

6.3.5 RQ5: Discovering New Vulnerabilities

From the information flow paths found by FLOWDIST, we identified 24 new vulnerabilities related to 8 of our subjects, as listed in Table 5. We reported these to the respective developers, with 17 having been confirmed and 2 already fixed so far. It is important to note that FLOWDIST *does not need any bug reports or the like to find known or new vulnerabilities/bugs*—it just computes all information flow paths between the specified *or default* input sources and sinks in the given execution for vulnerability inspection, albeit using such reports that include particular sources/sinks/executions of interest would facilitate the inspection.

Full details on these 24 cases are documented in [65]. Next, we illustrate with one fixed case and one confirmed case.

Case 1. In the Netty-Integration execution, this fixed case is a data leak induced by logging via exceptional control flow, as depicted in Figure 8. The sensitive data (object *selectionKey*) was read in class `AbstractNioChannel` of the *Nio* process (at the *source*), passed through class `SingleThreadEventExecutor` of

Table 4: Known vulnerabilities detected by FlowDist

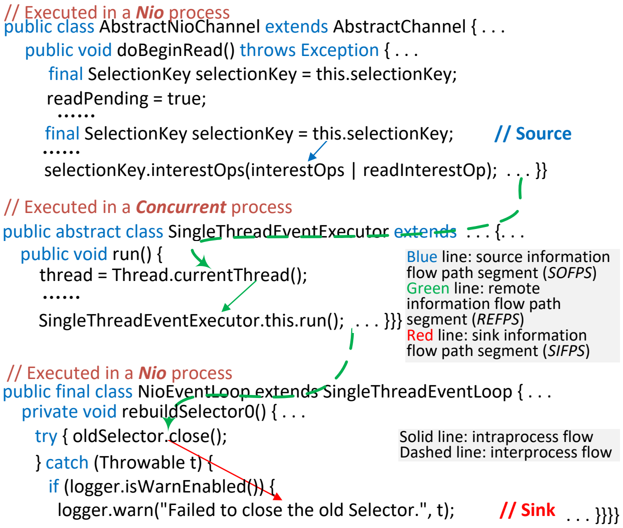
Subject	Vulnerability	Reference	Found	#Case	#FN
HSQLDB	CVE-2005-3280	[1]	✓	1	0
Netty	CVE-2014-0193	[3]	✗	10	5
	CVE-2014-3488	[4]	✗		
	CVE-2015-2156	[5]	✗		
	CVE-2016-4970	[7]	✗		
	Issue 8869	[10]	✗		
	Issue 9112	[11]	✓		
	Issue 9229	[12]	✓		
	Issue 9243	[13]	✓		
	Issue 9291	[14]	✓		
	Issue 9362	[15]	✓		
RocketMQ	CVE-2019-17572	[9]	✓	1	0
Thrift	CVE-2015-3254	[6]	✓	1	0
Voldemort	Issue 101	[16]	✓	6	1
	Issue 381	[20]	✓		
	Issue 387	[21]	✓		
	Issue 352	[17]	✓		
	Issue 378	[19]	✓		
	Issue 377	[18]	✗		
xSocket	Bug 21	[22]	✓	1	0
ZooKeeper	CVE-2014-0085	[2]	✓	4	0
	Bug 2569	[23]	✓		
	CVE-2018-8012	[32]	✓		
	CVE-2019-0201	[8]	✓		

Table 5: New vulnerabilities discovered by FlowDist

Subject	#Fixed	#Confirmed	#Pending
HSQLDB	0	5	2
Netty	1	1	0
Raining Sockets	0	1	0
RocketMQ	0	4	0
Thrift	0	5	0
Voldemort	0	0	4
xSocket	0	0	1
Zookeeper	1	1	0

the *Concurrent* process, and reached class `NioEventLoop` of the *Nio* process where the data went out of the system (at the *sink*). The throwable object *t* exposed *selectionKey* in the log, with which a client registers a socket channel and connects to the server. An adversary can exploit this leaked data to launch denial-of-service (DoS) attacks against the server. A single-process DIFA/DTA would have missed the *interprocess* information flow here hence this vulnerability.

Case 2. During the Thrift-Integration execution, we found again a logging-induced data leak, but in normal control flows, as depicted in Figure 9. At the *source*, an user input was read into *buf* in class `TIOStreamTransport` of the *Transport* process, passed through class `CalculatorClient` of the *Calculator* process, and flowed back into class `TSaslTransport` of the *Transport* process where the data went out of the system (at the *sink*). Any sensitive data (e.g., personal identification information) included in the user input would be leaked into the log, hence possibly enable intrusions into the system or cause losses. This vulnerability would be missed by existing application-level DIFA/DTA too since it occurs also via an *interprocess* information flow.



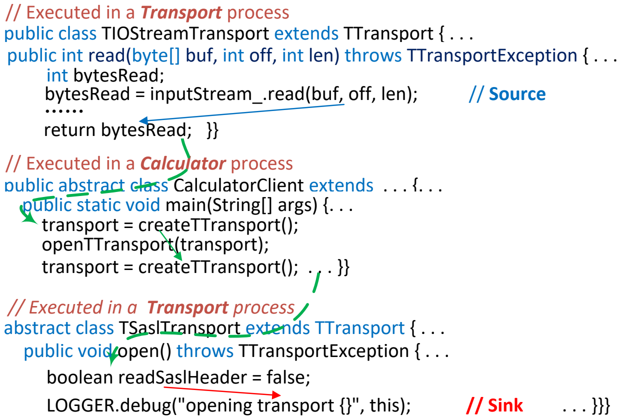
```
// Executed in a Nio process
public class AbstractNioChannel extends AbstractChannel { . . .
   public void doBeginRead() throws Exception { . . .
      final SelectionKey selectionKey = this.selectionKey;
      readPending = true;
        ......
      final SelectionKey selectionKey = this.selectionKey;     // Source
      ......
      selectionKey.interestOps(interestOps | readInterestOp);  . . . }}

// Executed in a Concurrent process
public abstract class SingleThreadEventExecutor extends  . . . {. . .
   public void run() {
      thread = Thread.currentThread();
      ......
      SingleThreadEventExecutor.this.run();  . . . }}}

// Executed in a Nio process
public final class NioEventLoop extends SingleThreadEventLoop { . . .
   private void rebuildSelector0() { . . .
     try { oldSelector.close();
     } catch (Throwable t) {
       if (logger.isWarnEnabled()) {
        logger.warn("Failed to close the old Selector.", t);        // Sink   . . . }}}}
```

Blue line: source information flow path segment (SOFPS)
Green line: remote information flow path segment (REFPS)
Red line: sink information flow path segment (SIFPS)
Solid line: intraprocess flow
Dashed line: interprocess flow

Figure 8: New vulnerabilities discovered: Case 1.



```
// Executed in a Transport process
public class TIOStreamTransport extends TTransport { . . .
 public int read(byte[] buf, int off, int len) throws TTransportException { . . .
      int bytesRead;
      bytesRead = inputStream_.read(buf, off, len);        // Source
      ......
      return bytesRead;   }}

// Executed in a Calculator process
public abstract class CalculatorClient extends  . . . {. . .
   public static void main(String[] args) {. . .
     transport = createTTransport();
     openTTransport(transport);
     transport = createTTransport();  . . . }}

// Executed in a  Transport process
abstract class TSaslTransport extends TTransport { . . .
   public void open() throws TTransportException { . . .
      boolean readSaslHeader = false;
      LOGGER.debug("opening transport {}", this);        // Sink       . . . }}}
```

Figure 9: New vulnerabilities discovered: Case 2.

> FlowDist *discovered 24 new vulnerabilities in 8 real-world distributed systems, with 17 confirmed and 2 fixed, suggesting its promising capability in this regard.*

Additional analysis. Not every information flow path reported by FlowDist represents a real vulnerability. Thus, additional analysis is expected for bug confirmation.

For a known vulnerability, once the relevant source and sink are identified as described earlier (§6.3.4), the vulnerability is readily confirmed as per the bug report/description after FlowDist found a path between the source and the sink.

For a new vulnerability, found from given sources/sinks, the additional analysis/effort is to confirm it by checking the relevant paths FlowDist produced. A path from a source *s* to a sink *t* reported may not always be a really critical bug *to the user*: for instance, the data retrieved at *s* may not actually be considered sensitive by the user even if *t* represents a data-leaking operation, or the sink is considered critical (e.g., making a branch decision) but *s* retrieves data

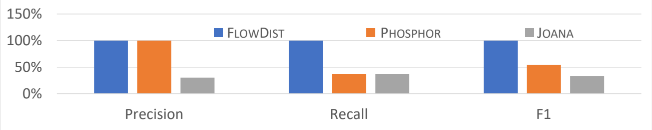


Figure 10: The accuracy of FlowDist versus the baselines.

not from any user input. It is also possible that in the user's specific application scenario there are some external security protection mechanisms (e.g., logging sensitive data into protected logs). Such security context factors are not currently considered by FlowDist itself (and it is hard to do so). Thus, confirmation is generally a necessary additional step.

False positives/negatives. Due to the presence of various security context factors, only part of the information flow paths reported by FlowDist will be confirmed as real bugs as described above; others are *false positives* from a vulnerability-discovery's point of view. Note that in RQ1 we reported a zero false positive rate, which was from the perspective of DIFA reporting true dynamic dependencies.

Given that all the known vulnerabilities were reproduced on interprocess flows according to the results for RQ4, in our experiments for RQ5 we focused on interprocess paths to discover new vulnerabilities. From a total of 323 unique reports, by carefully considering security context factors, we confirmed 209 bugs. Further confirmation with developers went slow, thus we only reported 24 most critical (in our view) ones by the time of writing this paper. Yet others are also valid/non-trivial. Thus, the overall false positive rate for security context was (323-209)/323=35%.

As a dynamic analysis, FlowDist cannot discover vulnerabilities that are not covered by the executions it analyzes, which naturally causes *false negatives*. Since the entire set of true vulnerabilities is unknown for our subject systems, we could not quantify the false negative rate of FlowDist for security context with respect to our dataset.

6.3.6 RQ6: Baseline Comparisons

Our baseline Phosphor instruments a standard JVM such that taint tags set and retrieved in (unit) test cases can be propagated during the execution of a given application on the instrumented JVM [47]. This requires sources and sinks to be in the test code. Thus, we needed to write a dedicated unit test for each source-sink pair per subject—the original test cases (e.g., system tests) associated with our subjects do not contain the sources/sinks considered in our comparisons.

In each of these dedicated tests, we first tainted the source data (variable) at the test entry, then triggered the original subject execution, and finally checked the taint-tag at the sink upon the test exit. We realized these taint tagging and checking operations using Phosphor APIs as per the variable type. These test cases are in [65]/PhosphorTest. We spent *4 to 10 hours to develop such dedicated tests for each of our subjects*. By design, Phosphor does not compute taint flow paths. Thus, for each source-sink pair, we considered that Phosphor found a taint flow between the pair if the sink contained the taint tag set at the source.

The other baseline Joana [71] identifies vulnerabilities (e.g., sensitive information leaks) in a given Java program through a static dependence analysis. It requires entry points, sources, and sinks explicitly specified by users *through annotations* in the program. We spent *1 to 3 hours to set such annotations for each of our subjects*. Joana does not report flow paths either, but only the sinks reachable from any annotated sources. To enable comparison, we considered that it found an information flow path between a source and a sink if it reports the sink when we annotated the source.

We note that a few cross-process DIFA/DTA tools do exist, yet to the best of our knowledge no such tools working for common distributed Java software like our subjects are available: For example, Kakute [80] works only with data-intensive applications based on a particular framework Spark while Taint-Exchange [125] (like Cloudfence [104] and Cloudopsy [124]) only works for C/C++ software. And all such tools are not purely application-level like ours.

Effectiveness. Figure 10 contrasts FlowDist with the baselines in terms of effectiveness for all the source/sink pairs in Thrift, Voldemort-Load, and Netty—we only considered these executions as we were able to (manually) produce the ground truth only for them (as for RQ1). Both baselines captured all the true *intraprocess* paths found by FlowDist but missed all the interprocess ones. Thus, they had the same but low (37.5%) recall; for the same reason, *none of them found any of the known and new vulnerabilities* (which were all on *interprocess* paths) as FlowDist did. Joana reported many additional paths that were not covered in the executions considered. With respect to the ground-truth paths (all being *dynamic*), those additional paths were false positives, leading to very low (30%) precision of Joana. As a result, FlowDist had a much higher F1 accuracy (100%) than Phosphor (54.6%) and Joana (33.3%).

It should also be noted that many of the vulnerabilities found by FlowDist were confirmed not just according to the source-sink reachability but by checking the complete, detailed flow paths as offered by a DIFA. DTA techniques like Joana and Phosphor would not sufficiently support such confirmations (even when working across processes to address interprocess flows), because they do not provide the path details needed. This helps justify using DIFA over DTA.

Efficiency. For the above effectiveness results, Phosphor and Joana took 1.38 and 0.43 seconds on average, respectively, for each source/sink pair, lower than FlowDist's querying cost (13 seconds on average). FlowDist also incurred a higher average storage cost (293.4MB) than Phosphor (21.2MB) and Joana (35.2MB). The reason is that FlowDist performed more, heavier analyses (e.g., probing, building the dependence graph, profiling instance-level method events) than the baselines (e.g., Joana only statically checked the source code). These extra costs of FlowDist were moderate and should be paid

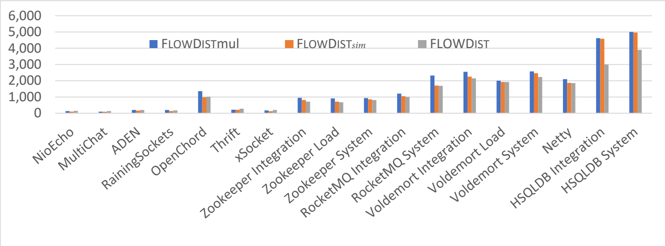


Figure 11: The total time costs (in seconds) of FLOWDIST*mul* and FLOWDIST*sim* against FLOWDIST for all subject executions.

off by its much higher effectiveness. Critically, it did not incur the substantial manual (e.g., test case development or source annotation) effort as the baselines require.

Discussion. Our goal with FLOWDIST is to achieve practical *applicability*, *portability*, *scalability*, and *cost-effectiveness* together for *DIFA of distributed software* instead of just better *DTA efficiency* for *single-process programs*. In addition, FLOWDIST works at an application level and computes full information flow paths (as opposed to taint checking only as by our baselines). Thus, we expected it to incur higher overheads than system-level DTA approaches (e.g., PHOSPHOR). The baselines need platform customization and/or substantial manual (test development or source annotation) effort that FLOWDIST avoids. The full information flow paths, which the baselines do not provide, are valuable for detailed security diagnoses. FLOWDIST thus complements the baselines by *making different tradeoffs* (e.g., portability versus efficiency).

FLOWDIST *achieved much higher effectiveness at reasonable costs over two state-of-the-art peer tools, yet without manual setup effort. None of the baselines found any of the known and new vulnerability as* FLOWDIST *did due to their failure to analyze* interprocess *flows.*

6.3.7 RQ7: Alternative Design Comparisons

To compare FLOWDIST to the two alternative designs, we repeated the experiments for RQ1 and RQ2 with FLOWDIST*sim* and FLOWDIST*mul*. We confirmed that these three tools produced the same information flow paths, hence their equivalence in effectiveness—while FLOWDIST*mul* generally suffers from non-determinism in the analyzed executions, it was not affected by such issues in our study.

Also as expected, the best performer among the three varied for different systems in terms of efficiency. Figure 11 shows the contrasts in the total analysis time of each tool for each of the 18 executions studied. For relatively large systems (ZooKeeper and larger), FLOWDIST was constantly the most efficient. For these systems the time saved due to the reduced instrumentation and profiling scope in the pre-analysis noticeably outweighed the time cost of the static analysis itself that enabled the reduction—thus, FLOWDIST won over FLOWDIST*sim*. Meanwhile, the time saved due to the reduced scope of profiling instance-level method events was outweighed by the extra time incurred by additional executions (with tracing) of the subject (in the intermediate phase)—thus, FLOWDIST won over FLOWDIST*mul*.

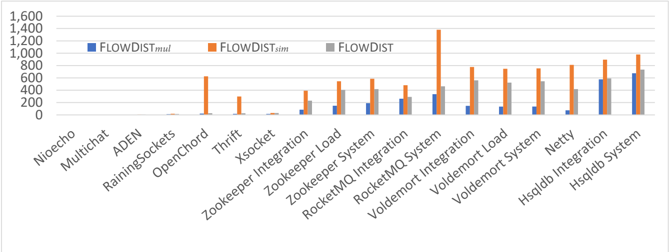


Figure 12: The storage costs (in MB) of FLOWDIST*mul* and FLOWDIST*sim* against FLOWDIST for all subject executions.

Table 6: Recommendations on DIFA/DTA tool selection

System type			With non-deterministic executions?	
			Yes	**No**
Distributed (multi-process)	Common	Small	FLOWDIST*sim*	FLOWDIST*sim* or FLOWDIST*mul*
		Large	FLOWDIST	FLOWDIST
	Specialized		Kakute [80] (for Spark [123])	
			Pileus [117] (for OpenStack [111]),...	
Single-process			PHOSPHOR [47], JOANA [76],...	

These outweighing contrasts were reversed for small systems (those smaller than ZooKeeper), which explains why for those systems the alternative designs won (albeit the difference between FLOWDIST*sim* and FLOWDIST*mul* was small). Here we differentiate systems as small and large not only by code size but also trace size.

Comparison on storage costs revealed insignificant differences, as shown in Figure 12. FLOWDIST*sim* needed the most storage spaces while FLOWDIST*mul* had the least storage requirements. And the storage costs incurred by FLOWDIST (default design) were in between. The reason is that FLOWDIST*sim* traces all instance-level method and branch events in the subject execution during the pre-analysis phase. In contrast, FLOWDIST traces relevant methods and branches only. On the other hand, FLOWDIST*mul* just records the first entry and last returned-into events in the pre-analysis phase, and then only traces methods on the method-level flow paths found in the pre-analysis and branches in those methods.

These findings led us to the recommendations on choosing the right tool for a particular system, as shown in Table 6. Overall, FLOWDIST best suits large-scale common distributed systems, regardless of the executions analyzed being non-deterministic or not. For small common distributed systems, either FLOWDIST*sim* or FLOWDIST*mul* may be a great choice if the target execution is known to be deterministic; otherwise, FLOWDIST*mul* would be opted out. We also put in a few peer tools that suite other types of (specialized distributed or single-process) systems, to highlight again that our work complements them.

The two alternative designs can complement FLOWDIST *in suiting smaller systems, while the three together complement existing DIFA/DTA tools in dealing with common distributed systems.*

6.4 Regarding the Vulnerabilities Discovered

The previously known vulnerabilities discovered by FlowDist have been documented in detail on respective CVE pages as seen in Table 4. The documentations include how the vulnerabilities have been disclosed and addressed.

Regarding each of the 24 new vulnerabilities discovered by FlowDist, we have contacted the developers of respective systems. By the time of this paper submission, all of these have been reported to the system vendors, although some of them have not been confirmed yet (i.e., for HSQLDB, Raining Sockets, Voldemort, and xSocket), possibly because the developers have not been active recently. Others have all been confirmed, among which two have been fixed. The details on each of these 24 vulnerabilities are documented in [65]/newVulnerabilities/Vulnerabilities.docx.

7 Related Work

Most previous information flow analyses are purely *static* (e.g., [50, 76, 100, 118]), including well-known works for Android (e.g., FlowDroid [41], IccTA [89], Amandroid [119], DroidSafe [74], and HornDroid [54]). These approaches suffer from imprecision issues common to purely static analysis, which is also commonly unsound due to dynamic constructs (e.g., reflection and dynamic code loading) in modern languages [92]. With distributed programs, these issues are exacerbated due to implicit dependencies among distributed (decoupled) components. Next, we discuss prior works closely related to ours (i.e., relevant to DIFA/DTA), which are *dynamic* in nature and target specific program executions by design (hence orthogonal to common problems like run-time input quality and limited coverage).

Conventional DIFA/DTA. Like TaintDroid [64], TaintMan [121] customizes the Android OS to track whole-system information flow at runtime. Panorama [120] performs system-side dynamic information flow tracking for Windows malware analysis, through dynamic instrumentation as Dytan [60] and TaintEraser [126]. In [77], a dynamic taint analysis was used for intrusion detection via a custom Linux security module. Juturna [93] employs bytecode augmentation and modified Java API classes, similar to Phosphor instrumenting JVM, for taint tracking. These approaches require customized run-time platforms, like a few others [59, 62, 116] using specialized hardware, to perform DTA. In [43, 44, 83], the authors proposed language semantics for dynamic taint analysis of JavaScript code. LabelFlow [58] works as an extension of PHP to implement security policies in web applications. Like many other DTA tools [35, 42, 45, 57, 97, 102, 107, 114], these approaches do not work with common distributed software as they only track information flows in single threads/processes.

In contrast, FlowDist is a purely application-level DIFA. It does not require modifying original run-time platforms nor specific frameworks/emulators. Importantly, it tracks dynamic information flow (across processes), which is out of the applicability scope of most peer approaches.

Cross-process DIFA/DTA. Only a few existing techniques address information flows across processes. Kakute [80] tracks field-level data flow with unified APIs for reference propagation and tag sharing. Based on Phosphor, it needs to customize (instrument) its runtime platform (i.e., JVM). And it focuses on Spark [123] applications only, not working with common distributed software. Similarly, Pileus [117] targets the applications on a special cloud platform OpenStack [111]. Taint-Exchange [125] is a framework for cross-host taint tracking, using libdft [84] to transfer taint information through sockets and pipes. Like Cloudfence [104] and Cloudopsy [124], Taint-Exchange relies on a customized platform (Pin) and targets C/C++ software.

In contrast, FlowDist works generally with *common* distributed systems, without any change to the original run-time platform while offering full information flow paths. We are not aware of a prior DIFA working for common distributed software: Kakute [80] and Pileus [117] are DTA and work only for specialized distributed systems—DTA is conceptually differentiated from DIFA (§2); other relevant approaches are either DTA or not working with common distributed systems. The key conceptual differences between FlowDist and peer approaches lie in our multi-staged, refinement-based methodology for DIFA and in FlowDist explicitly addressing *interprocess* information flow.

Dynamic dependence analysis for distributed programs. A number of dynamic slicing algorithms [46, 55, 63, 69, 75, 81, 86, 98] have been developed. In particular, prior work [46] defines varied kinds of dependencies induced by interprocess communication. However, the approach was not implemented to work on real-world distributed software, and its algorithmic nature implies scalability barriers. A major focus of FlowDist is to deal with the overhead of fine-grained dynamic dependence analysis so as to scale to large real-world distributed systems. The method-level dependence analysis in the pre-analysis of FlowDist was inspired by DistIA [53]. In comparison, FlowDist targets a finer-grained and much more precise data-flow analysis at statement level with high efficiency and scalability.

Reasoning about happens-before relations by addressing global timing via partial ordering based on logic clocks is a standard technique in concurrent program analysis. This technique has been used in testing concurrent programs and distributed systems [90, 103, 122]. For example, DCatch [90] detects concurrency bugs by checking a distributed execution against a set of happens-before relation rules. FlowDist also leverages happens-before relations, but among method execution events partially ordered through message-passing events and for inferring interprocess dependencies.

Language-based information flow control. Jif [106] extends Java to address information flow security via augmenting the language with features that are related to security. It supports security labels to help users specify confidentiality/integrity policies. Furthermore, as a platform and language for building secure distributed systems, Fabric [91] extends Jif to support distributed transactions and programming. It has several mechanisms, such as access control and information flow control, to prevent untrusted nodes from violating integrity and confidentiality. Other language-based information flow control approaches [56, 87, 109] have also been proposed.

In essence, these approaches offer ways of *constructing* an information-flow-secure system. Thus, to benefit from them, developers need to build the system in a specialized manner (e.g., using the Fabric language). Also, the security capabilities they offer depend on the accuracy of the policies specified. In contrast, FLOWDIST does not impose these burdens to developers and it analyzes *existing* distributed systems already built without any knowledge about itself. It also provides detailed code-level information flow paths that those language-based tools typically do not offer. Finally, the core of FLOWDIST is a dynamic data flow analysis, which can empower applications beyond those on security (e.g., testing, debugging, program understanding, performance analysis) that the language-based approaches do not readily support.

8 Conclusion

We presented FLOWDIST, a purely application-level dynamic information flow analysis for common distributed systems. To enable a practical solution to computing full information flow paths in large-scale systems, FLOWDIST overcomes multiple technical challenges via a *multi-staged refinement-based* analysis methodology. This methodology itself is applicable beyond information flow analysis and distributed systems.

Extensive evaluation of FLOWDIST and its two alternative designs showed that our approach scaled well to large-scale distributed systems with generally small run-time overhead. We also demonstrated its capabilities in discovering known and new vulnerabilities in diverse real-world systems, and its superiority over state-of-the-art peer techniques.

Acknowledgments

We thank the anonymous reviewers for constructive comments and our shepherd Engin Kirda for very helpful guidance. This work was supported by NSF through grant CCF-1936522.

References

[1] CVE-2005-3280. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3280.

[2] CVE-2014-0085. https://www.cvedetails.com/cve/CVE-2014-0085/.

[3] CVE-2014-0193. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0193.

[4] CVE-2014-3488. https://nvd.nist.gov/vuln/detail/CVE-2014-3488.

[5] CVE-2015-2156. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2156.

[6] CVE-2015-3254. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3254.

[7] CVE-2016-4970. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4970.

[8] CVE-2018-8012. https://nvd.nist.gov/vuln/detail/CVE-2019-0201.

[9] CVE-2019-17572. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17572.

[10] Netty/8869. http://github.com/netty/netty/issues/8869.

[11] Netty/9112. http://github.com/netty/netty/issues/9112.

[12] Netty/9229. http://github.com/netty/netty/issues/9229.

[13] Netty/9243. http://github.com/netty/netty/issues/9243.

[14] Netty/9291. http://github.com/netty/netty/issues/9291.

[15] Netty/9362. http://github.com/netty/netty/issues/9362.

[16] Voldemort/101. https://github.com/voldemort/voldemort/issues/101.

[17] Voldemort/352. https://code.google.com/archive/p/project-voldemort/issues/352.

[18] Voldemort/377. https://code.google.com/archive/p/project-voldemort/issues/377.

[19] Voldemort/378. https://code.google.com/archive/p/project-voldemort/issues/378.

[20] Voldemort/381. https://code.google.com/archive/p/project-voldemort/issues/381.

[21] Voldemort/387. https://code.google.com/archive/p/project-voldemort/issues/387.

[22] xSocket/21. https://sourceforge.net/p/xsocket/bugs/21/.

[23] ZooKeeper/2569. https://issues.apache.org/jira/browse/ZOOKEEPER-2569.

[24] RainingSockets. https://tinyurl.com/566hetmd, 2004.

[25] ADEN. https://tinyurl.com/h5wrhaka, 2013.

[26] MultiChat. https://tinyurl.com/nfdbwkxb, 2015.

[27] NioEcho. https://tinyurl.com/bwu5psvh, 2015.

[28] Open Chord. https://tinyurl.com/a33zm9ec, 2015.

[29] Voldemort. https://github.com/voldemort, 2015.

[30] ZooKeeper. https://zookeeper.apache.org/, 2015.

[31] CVE. https://cve.mitre.org/, 2018.

[32] CVE-2018-8012. https://nvd.nist.gov/vuln/detail/CVE-2018-8012, 2018.

[33] Thrift. https://thrift.apache.org/, 2018.

[34] xSocket. http://xsocket.org/, 2018.

[35] DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html, 2019.

[36] HyperSQL. http://hsqldb.org/, 2020.

[37] Netty. https://netty.io/index.html, 2020.

[38] RocketMQ. https://rocketmq.apache.org/, 2020.

[39] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[40] Abdullah Mujawib Alashjaee, Salahaldeen Duraibi, and Jia Song. Dynamic Taint Analysis Tools: A Review. *IJCSS*, 13(6):231, 2019.

[41] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, pages 259–269, 2014.

[42] Mohammadreza Ashouri and Christoph Kreitz. Hybrid Taint Flow Analysis in Scala. In *SSCI*, pages 657–663, 2019.

[43] Thomas H Austin and Cormac Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *PLAS*, pages 113–124, 2009.

[44] Thomas H Austin and Cormac Flanagan. Permissive Dynamic Information Flow Analysis. In *PLAS*, pages 1–12, 2010.

[45] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In *S&P*, pages 490–504, 2019.

[46] Soubhagya Sankar Barpanda and Durga Prasad Mohapatra. Dynamic slicing of distributed object-oriented programs. *IET Software*, 5(5):425–433, 2011.

[47] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *OOPSLA*, pages 83–101, 2014.

[48] Haipeng Cai. Hybrid program dependence approximation for effective dynamic impact prediction. *TSE*, 44(4):334–364, 2018.

[49] Haipeng Cai and Xiaoqin Fu. D2ABS: A framework for dynamic dependence analysis of distributed programs. Technical report, 2019.

[50] Haipeng Cai and John Jenkins. Leveraging historical versions of android apps for efficient and precise taint analysis. In *MSR*, pages 265–269, 2018.

[51] Haipeng Cai and Raul Santelices. Diver: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *ASE*, pages 343–348, 2014.

[52] Haipeng Cai, Raul Santelices, and Douglas Thain. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *TOSEM*, 25(2):1–50, 2016.

[53] Haipeng Cai and Douglas Thain. DistIA: A Cost-Effective Dynamic Impact Analysis for Distributed Programs. In *ASE*, pages 344–355, 2016.

[54] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In *EuroS&P*, pages 47–62, 2016.

[55] Jingde Cheng. Dependence Analysis of Parallel and Distributed Programs and Its Applications. In *Advances in Parallel and Distributed Computing*, pages 370–377, 1997.

[56] Winnie Cheng, Dan RK Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *USENIX ATC*, pages 139–151, 2012.

[57] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *ICC*, pages 749–754, 2006.

[58] Georgios Chinis, Polyvios Pratikakis, Sotiris Ioannidis, and Elias Athanasopoulos. Practical Information Flow for Legacy Web Applications. In *OOPSLA*, pages 17–28, 2013.

[59] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, pages 321–336, 2004.

[60] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *ISSTA*, pages 196–206, 2007.

[61] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th edition, 2011.

[62] Jedidiah R Crandall and Frederic T Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO*, pages 221–232, 2004.

[63] Evelyn Duesterwald, Rajiv Gupta, and M Soffa. Distributed slicing and partial re-execution for distributed programs. In *Languages and Compilers for Parallel Computing*, pages 497–511. 1993.

[64] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *TOCS*, 32(2):5, 2014.

[65] Xiaoqin Fu and Haipeng Cai. FlowDist Artifact. https://bitbucket.org/wsucailab/flowdist.

[66] Xiaoqin Fu and Haipeng Cai. A dynamic taint analyzer for distributed systems. In *FSE*, pages 1115–1119, 2019.

[67] Xiaoqin Fu and Haipeng Cai. Measuring interprocess communications in distributed systems. In *ICPC*, pages 323–334, 2019.

[68] Xiaoqin Fu and Haipeng Cai. Scaling application-level dynamic taint analysis to enterprise-scale distributed systems. In *ICSE Companion*, pages 270–271, 2020.

[69] Xiaoqin Fu, Haipeng Cai, and Li Li. Dads: dynamic slicing continuously-running distributed programs with budget constraints. In *FSE*, pages 1566–1570, 2020.

[70] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning. *TOSEM*, 30(1):1–45, 2020.

[71] Dennis Giffhorn and Christian Hammer. Precise Analysis of Java Programs using JOANA. In *SCAM*, pages 267–268, 2008.

[72] Dennis Giffhorn and Christian Hammer. Precise Slicing of Concurrent Programs. *Automated Software Engineering*, 16(2):197–234, 2009.

[73] Mehran Goli, Muhammad Hassan, Daniel Große, and Rolf Drechsler. Security validation of VP-based SoCs using dynamic information flow tracking. *it-Information Technology*, 61(1):45–58, 2019.

[74] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *NDSS*, volume 15, page 110, 2015.

[75] Diganta Goswami and Rajib Mall. Dynamic Slicing of Concurrent Programs. In *HiPC*, pages 15–26. 2000.

[76] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Working Conference on Programming Languages*, pages 123–138, 2013.

[77] Christophe Hauser, Frédéric Tronel, Colin Fidge, and Ludovic Mé. Intrusion detection in distributed systems, an approach based on taint marking. In *ICC*, pages 1962–1967, 2013.

[78] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *TOPLAS*, 12(1):26–60, 1990.

[79] Daniel Jackson and Martin Rinard. Software Analysis: A Roadmap. In *ICSE*, pages 133–145, 2000.

[80] Jianyu Jiang, Shixiong Zhao, Danish Alsayed, Yuexuan Wang, Heming Cui, Feng Liang, and Zhaoquan Gu. Kakute: A Precise, Unified Information Flow Analysis System for Big-data Security. In *ACSAC*, pages 79–90, 2017.

[81] Mariam Kamkar and Patrik Krajina. Dynamic Slicing of Distributed Programs. In *ICSM*, pages 222–229, 1995.

[82] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[83] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *TSE*, 46(12):1364–1379, 2018.

[84] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *VEE*, pages 121–132, 2012.

[85] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit Flows: Can't Live With 'Em, Can't Live Without 'Em. In *ICISSP*, pages 56–70, 2008.

[86] Bogdan Korel and Roger Ferguson. Dynamic Slicing of Distributed Programs. *Applied Math. and Computer Science*, 2(2):199–215, 1992.

[87] Elisavet Kozyri, Owen Arden, Andrew C Myers, and Fred B Schneider. Jrif: reactive information flow control for java. In *Foundations of Security, Protocols, and Equational Reasoning*, pages 70–88. 2019.

[88] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *CETUS*, volume 15, 2011.

[89] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, pages 280–291, 2015.

[90] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. *ASPLOS*, 45(1):677–691, 2017.

[91] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *SOSP*, pages 321–334, 2009.

[92] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *CACM*, 58(2):44–46, 2015.

[93] Florian D Loch, Martin Johns, Martin Hecker, Martin Mohr, and Gregor Snelting. Hybrid Taint Analysis for Java EE. In *SAC*, pages 1716–1725, 2020.

[94] Wes Masri and Andy Podgurski. Application-based anomaly intrusion detection with dynamic information flow analysis. *Computers & Security*, 27(5-6):176–187, 2008.

[95] Wes Masri and Andy Podgurski. Algorithms and tool support for dynamic information flow analysis. *IST*, 51(2):385–404, 2009.

[96] Wes Masri, Andy Podgurski, and David Leon. Detecting and Debugging Insecure Information Flows. In *ISSRE*, pages 198–209, 2004.

[97] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX Security*, pages 65–80, 2015.

[98] Durga P Mohapatra, Rajeev Kumar, Rajib Mall, DS Kumar, and Mayank Bhasin. Distributed dynamic slicing of Java programs. *JSS*, 79(12):1661–1678, 2006.

[99] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer Science & Business Media, 2006.

[100] Andrew C Myers. JFlow: Practical Mostly-Static Information Flow Control . In *POPL*, pages 228–241, 1999.

[101] Mangala Gowri Nanda and S Ramesh. Slicing Concurrent Programs. In *ISSTA*, pages 180–190, 2000.

[102] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, volume 5, pages 1–17, 2005.

[103] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace Aware Random Testing for Distributed Systems. *Proceedings of ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[104] Vasilis Pappas, Vasileios P Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D Keromytis. CloudFence: Data Flow Tracking as a Cloud Service. In *RAID*, pages 411–431, 2013.

[105] Manoj Plakal, Daniel J Sorin, Anne E Condon, and Mark D Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *SPAA*, pages 67–76, 1998.

[106] Kyle Pullicino. Jif: Language-based Information-flow Security in Java. *arXiv preprint arXiv:1412.8639*, 2014.

[107] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO*, pages 135–148, 2006.

[108] Venkatesh Prasad Ranganath and John Hatcliff. Slicing Concurrent Java Programs using Indus and Kaveri. *STTT*, 9(5-6):489–504, 2007.

[109] Bruno PS Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo. Hybrid static-runtime information flow and declassification enforcement. *TIFS*, 8(8):1294–1305, 2013.

[110] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *S&P*, pages 317–331, 2010.

[111] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. OpenStack: Toward an Open-source Solution for Cloud Computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[112] Venkatesh Gauri Shankar, Gaurav Somani, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. AndroTaint: An Efficient Android Malware Detection Framework using Dynamic Taint Analysis. In *ISEA Asia security and privacy*, pages 1–13, 2017.

[113] Mariana Sharp and Atanas Rountev. Static Analysis of Object References in RMI-Based Java Software. *TSE*, 32(9):664–681, 2006.

[114] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In *S&P*, pages 1527–1543, 2020.

[115] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic Dependency Monitoring to Secure Information Flow. In *CSF*, pages 203–217, 2007.

[116] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. pages 85–96, 2004.

[117] Yuqiong Sun, Giuseppe Petracca, Xinyang Ge, and Trent Jaeger. Pileus: Protecting User Resources from Vulnerable Cloud Services. In *ACSAC*, pages 52–64, 2016.

[118] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. In *ACSAC*, pages 289–298, 2008.

[119] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *CCS*, pages 1329–1341, 2014.

[120] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *CCS*, pages 116–127, 2007.

[121] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices. *TDSC*, 17(1):209–222, 2017.

[122] Xinhao Yuan and Junfeng Yang. Effective Concurrency Testing for Distributed Systems. In *ASPLOS*, pages 1141–1156, 2020.

[123] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[124] Angeliki Zavou. *Information Flow Auditing In the Cloud*. PhD thesis, Columbia University, 2015.

[125] Angeliki Zavou, Georgios Portokalidis, and Angelos D Keromytis. Taint-Exchange: a Generic System for Cross-process and Cross-host Taint Tracking. In *IWSEC*, pages 113–128, 2011.

[126] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.