HAL
archives-ouvertes.fr

SHRIFT System-Wide HybRid Information Flow Tracking

Enrico Lovat, Alexander Fromm, Martin Mohr, Alexander Pretschner

▶ **To cite this version:**

Enrico Lovat, Alexander Fromm, Martin Mohr, Alexander Pretschner. SHRIFT System-Wide HybRid Information Flow Tracking. 30th IFIP International Information Security Conference (SEC), May 2015, Hamburg, Germany. pp.371-385, 10.1007/978-3-319-18467-8_25 . hal-01345128

HAL Id: hal-01345128
https://hal.inria.fr/hal-01345128

Submitted on 13 Jul 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

SHRIFT
System-wide HybRid Information Flow Tracking

Enrico Lovat[1], Alexander Fromm[1], Martin Mohr[2], and Alexander Pretschner[1]

[1] Technische Universität München, Garching bei München, Germany
firstname.lastname@cs.tum.edu
[2] Karlsruhe Institute of Technology
martin.mohr@kit.edu

Abstract. Using data flow tracking technology, one can observe how data flows from inputs (sources) to outputs (sinks) of a software system. It has been proposed [**?**] to do runtime data flow tracking at various layers simultaneously (operating system, application, data base, window manager, etc.), and connect the monitors' observations to exploit semantic information about the layers to make analyses more precise. This has implications on performance—multiple monitors running in parallel—and on methodology—there needs to be one dedicated monitor per layer.
We address both aspects of the problem. We replace a *runtime* monitor at a layer L by its *statically* computed input-output dependencies. At runtime, these relations are used by monitors at other layers to model flows of data through L, thus allowing cross-layer system-wide tracking. We achieve this in three steps: (1) static analysis of the application at layer L, (2) instrumentation of the application's source and sink instructions and (3) runtime execution of the instrumented application in combination with monitors at other layers. The result allows for system-wide tracking of data dissemination, across and through multiple applications. We implement our solution at the Java Bytecode level, and connect it to a runtime OS-level monitor. In terms of precision and performance, we outperform binary-level approaches *and* can exploit high-level semantics.

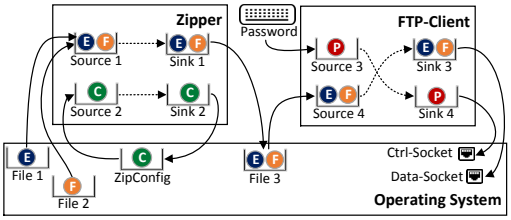
1 Introduction

Information flow analyses try to answer the question of whether or not data will potentially flow, or has potentially flowed, from inputs (*sources*) to outputs (*sinks*) of a certain system. Different analyses cater to different kind of source-sink dependencies, mainly distinguishing between *explicit* information flows (data-flow dependencies or *data flows*) and *implicit* information flows (like e.g. dependencies caused solely by control-flow). Data flow tracking solutions are generally tailored to one particular level of abstraction, like source code, byte code, machine code, or the operating system level (cf. §**??**).

Recently, data flow tracking technologies have been augmented by concepts of distributed data usage control [**?**,**?**,**?**,**?**,**?**] and performed at *multiple layers of abstraction*, to the end of expressing and enforcing more complex policies (e.g.

"any representation of this picture must be deleted after thirty days"). Multi-layer monitoring is important to preserve the *high-level semantics* of objects (e.g. "a mail") and events (e.g. "forward"), which is otherwise hard to capture at lower levels. But this benefit does not come for free: even a small number of monitors running in parallel may seriously compromise the performance of the overall system, and dedicated high-level monitors are not always available for every domain. In this case, the usual solution is to rely on conservative estimations provided by lower layers. For instance, if a dedicated monitor for a process is not available, an OS-level monitor would have to treat the process as a "black box" and assume that every sensitive data it got in touch with is propagated to every future output. This solution likely introduces many false positives and in this sense grossly overapproximates the set of potential information flows.

We propose SHRIFT, an approach to mitigate this issue. The core idea behind SHRIFT is to replace the runtime monitoring of how data flows through a process (or its *black-box* overapproximation) by consultations of a *statically precomputed mapping* between its inputs and outputs.



Running Example: *A company enforces the policy* "upon logout, delete every local copy of customer data" *to prevent clerks to work with outdated material. Upon every login, a clerk must download from a central server a fresh version of the customer data he is interested in. In this setting, a clerk uses the* Zipper *application to compress multiple customer data (**E**, **F**) into a single archive file (`File 3`), which he then sends to the company server using* Ftp-Client.

In this example, a data-flow tracking system can help tracking down every copy of to-be-deleted customer data in the system. However, if the tracking is imprecise (too many false positives), additional important resources may be accidentally deleted as well. For example, `ZipConfig` (Zipper's configuration file), which is updated during every run of Zipper, could be mistakenly marked as containing data `E` and deleted upon logout, making Zipper unusable in the future. Similar concerns also apply to the Ftp-Client: FTP works with two channels, one for commands, and one for payload. In a black-box monitoring situation, once sensitive data is read, every write to any of the two channels may be possibly carrying sensitive information, and, as such, it should propagate the taint to the socket connection, and possibly to the recipient side. In this case, the credentials (marked as `P` in the figure), sent via the command channel, and the database in which they are stored on the server side would also be marked as "to-be-deleted".

Our approach improves the precision of information-flow tracking *system-wide*, i.e. through and in-between different processes/applications, like the flow of data **E** and **F** through the Zipper application (`Source 1`→ `Sink 1`) into `File 3` and then through the Ftp-Client application (`Source 4` → `Sink 3`)

till the payload channel, with lower execution overhead than other dynamic monitors for comparable scenarios (cf. §**??**).

Problem: Concurrently running multiple monitors at different layers of abstraction allows to exploit high-level semantic information (e.g., "screenshot" or "mail") but is performance-wise expensive and requires dedicated monitoring technologies for every layer/application. On the other hand, relying only on estimates provided by other layers (e.g., the above black-box approach) improves performance but comes at the cost of (possibly significant) precision loss.

Solution: We propose a dynamic monitoring approach for generic processes that replaces runtime intra-process data flow tracking by consultations of a statically computed taint-propagation table. Such a monitor is more performant than equivalent runtime monitors for the same application and more precise than the OS-level overapproximation adopted when such a monitor is not available.

Contribution: To the best of our knowledge, we are the first to combine static and dynamic data-flow tracking for different levels of abstraction and *through multiple different applications*. Our solution improves the precision of OS level data flow tracking with minimal intra-process runtime tracking overhead.

2 Our Approach

We consider a setting with monitors at two levels of abstraction: a dynamic monitor at the OS level, based on system-call interposition [**?**], and one or more inline reference monitors at the application level. Our goal is to improve tracking precision at the OS level with minimal performance penalties. Although our approach is generic in nature and could be applied to any language or binary code, in this work we focus on an instantiation for the Java Bytecode (JBC) level.

We use standard terminology: a *source* is a method invoked by the application to get input data from the environment. A *sink* performs the dual output invocation. While in some contexts one can find detailed lists of source and sink methods [**?**], in general the choice is left to the analyst. In our work, a source (sink) is the invocation of a Java standard library method that overrides any overloaded version of InputStream.read (OutputStream.write) or Reader.read (Writer.write), or a method that indirectly invokes one of them, e.g., Properties.load(), which uses an input stream parameter to fill a properties table.

The idea is the following. If a source in an application is executed, the respective input's taint mark is stored. If a sink is executed, all sources (and therefore all taint marks) with potential flows to this sink are determined using a static mapping of potential flows between sources and sinks. There is hence a need to instrument sources and sinks, but *not all the instructions in-between them.*

Our approach consists of three phases:

??. Static analysis: An application X is analyzed for possible information flows between sources and sinks. During this phase we generate a report containing a list of all sources and sinks in the application and a mapping between each sink and every source it may depend on.

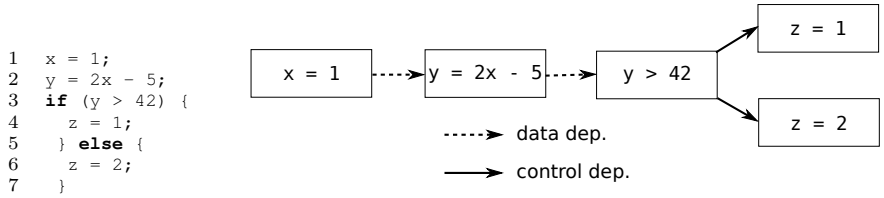


Fig. 1: A code snippet and its PDG

??. Instrumentation: All sinks and sources identified by the static analysis (and those instructions *only*) are instrumented in the bytecode of X, allowing us to monitor their execution.

??. Runtime: Every time a source or a sink is executed by the instrumented application, information about the data being read or written is exchanged with the OS-level monitor.

In the remainder of this section, we will describe these phases in details, using Zipper and Ftp-Client as examples. Notice, however, that in principle our work can be applied in a push-button fashion to any Java program.

2.1 Static Analysis

In this phase, we perform a static information flow analysis of the application and generate a list of all sources and sinks in the application and of their respective dependencies. To do so, we use JOANA [**?**,**?**], a static information flow analysis tool, but the choice is not binding because our approach is generic in nature and the techniques used by JOANA are also implemented by other tools, e.g. [**?**].

JOANA operates in two steps: first, it builds a *Program Dependence Graph* (PDG) [**?**] of the application; second, it applies slicing-based information flow analysis [**?**] on the PDG to find out which set of the sources influences which sinks. In order to reduce the number of false positives, JOANA leverages several program analysis techniques. In the following, we explain some fundamental concepts behind JOANA.

PDGs and Slicing: A PDG is a language-independent representation of a program. The nodes of a PDG represent statements and expressions, while edges model the syntactic dependencies between them. There exist many kinds of dependencies, among which the most important are *data dependencies*, (a statement using a value produced by another statement) and *control dependencies* (a statement or expression controlling whether another statement is executed or not). The PDG in Figure **??** contains a data dependency between the statements in line 1 and in line 2 because the latter uses the value of x produced by the former, and a control dependency between the **if**-statement in line 3 and the statements in lines 4 and 6 because whether line 4 or 6 is executed depends on the value of the expression in line 3.

PDG-based information flow analysis uses *context-sensitive slicing* [**?**], a special form of graph reachability: given a node n of the PDG, the *backwards slice*

of n contains all nodes from which n is reachable by a path in the PDG that respects calling-contexts. For sequential programs, it has been shown [**?**] that a node not contained in the backwards slice of n cannot influence n, hence PDG-based slicing on sequential programs guarantees *non-interference* [**?**]. It is also possible to construct and slice PDGs for concurrent programs [**?**]. However, in this context, additional kinds of information flows may exist, e.g. probabilistic channels [**?**]. So the mere slicing is not enough to cover *all* possible information flows between a source and a sink. A PDG- and slicing-based algorithm providing such guarantee has recently been developed and integrated into JOANA [**?**].

Analysis Options: JOANA is highly configurable and allows to configure different aspects of the analysis, e.g. to ignore all control flow dependencies caused by exceptions, or to specify different types of *points-to analysis* [**?**]. Points-to-analysis is an analysis technique which aims to answer the question which heap locations a given pointer variable may reference. JOANA uses points-to information during PDG construction to determine possible information flows through the heap and therefore depends heavily on the points-to analysis precision. JOANA supports several points-to analyses, including 0-1-CFA [**?**], k-CFA [**?**] and object-sensitive [**?**] points-to analysis.

The outcome of this phase is a list of the sources and sinks in the code of the application and a table that lists all the sources each sink depends on.

2.2 Instrumentation

In this phase, we take the report generated by the static analysis and instrument each identified source and sink. For each source or sink, the analysis reports the *signature* and the exact *location* (parent method and bytecode offset).

```
1  void zipIt(String file, String srcFolder) {
2   fos = new FileOutputStream(file);
3   zos = new ZipOutputStream(fos);
4   fileList = this.generateFileList(srcFolder);
5   byte[] buffer = new byte[1024];
6   for (String file : fileList) {
7    ze = new ZipEntry(file);
8    zos.putNextEntry(ze);
9    in = new FileInputStream(file);
10   int len;
11   while ((len = in.read(buffer)) > 0)
12    zos.write(buffer, 0, len);
13  in.close();}}
```

Listing 1.1: Java code fragment from Zipper

Consider the code snippet in Listing **??**, used in our Zipper application: static information flow analysis detects the flow from the source at line **??** (`Source1`), where the files are read, to the sink at line **??** (`Sink1`), where they are written into the archive. Listing **??** shows the corresponding analysis report: lines 1 - 9 specify that the return value of the `read` method invocation at bytecode offset `191` in method `zipIt` is identified as `Source1`. The same holds for `Sink1` (lines 12-20), but in this case the first parameter (line 19) is a sink, not a source. In the final part, the report also provides information about the dependency between `Sink1` and `Source1` (line 21 - 25), which is then used to model possible flows.

We use the *OW2-ASM* [**?**] instrumentation tool to wrap each reported source and sink with additional, injected bytecode instructions. We refer to the set of

these additional instructions as *inline reference monitor*. The outcome of this phase is an instrumented version of the original application, augmented with a minimal inline reference monitor that interacts with the OS-level monitor when a source or a sink is executed. This way incoming/outgoing flows of data from/to a resource, like files or network sockets, can be properly modeled.

```
1  <source><id>Source1</id>
2    <location>JZip.zipIt
3      (Ljava/lang/String;Ljava/lang/
           String;)V:191
4    </location>
5    <signature>
6      java.io.FileInputStream.read([B)I
7    </signature>
8    <return/>
9  </source><source><id>Source2</id>
10   ...
11 </source>
12 <sink><id>Sink1</id>
13   <location>JZip.zipIt
14     (Ljava/lang/String;Ljava/lang/
           String;)V:185
15   </location>
16   <signature>
17     java.util.zip.ZipOutputStream.
           write([BII)V
18   </signature>
19   <param index="1"/>
20 </sink>
21 <flows>
22   <sink id="Sink1">
23     <source id="Source1"/>
24   </sink>
25 </flows>
```

Listing 1.2: Static analysis report listing sinks, sources and their dependencies

2.3 Runtime

This phase represents the actual runtime data-flow tracking, where we execute the instrumented applications in a dynamically monitored OS. At runtime a single OS-level monitor exchanges information with multiple inlined bytecode-level reference monitors, one per application. We assume that the information to be tracked is initially stored somewhere in the system, e.g. in some files or coming from certain network sockets, and marked as sensitive. In our example in §**??** we assume data **E** and **F** to be already stored in `File 1` and `File 2`, respectively.

Once a source instruction is about to be executed, the instrumented code queries the OS-monitor to obtain information about the tainting of the input. It then associates this information to the source id (e.g. `ZipConfig` → `Source2` in our example). When a sink instruction is about to be executed, the instrumented code fetches tainting information from *all the sources the current sink depends on* according to the analysis report (`Source2` → `Sink2`). Such information denotes all the possible inputs the current output may depend on, but, most importantly, it denotes all the inputs the current output does *not* depend on: this is where we reduce false positives, mitigating the overapproximation of potential flows. The tainting information is then propagated to the output.

With this approach, even if the application reads additional data (like data **E**) before generating the output, the tainting associated with the sink (and, consequently, with the output) remains the same, as long as the input does not influence the output. In contrast, in a process treated as a black-box every output is as sensitive as the union of *all* the sources encountered till then. The information about the content being output by the current sink (`Sink2` → `ZipConfig`) is forwarded to the OS monitor, which will carry on the tracking

outside the boundaries of the application. Since the process described here applies to every instrumented application, this allows us to track the flows of data between any pair of applications, even through OS artifacts (like files), OS events (like copying a file) and non-instrumented processes (via black-box tracking).

3 Evaluation

Our goal is to improve system-wide, i.e. OS-level, data-flow tracking precision without the extreme overhead of process-level runtime data-flow tracking. We evaluated our work in terms of *precision* (false positives[3]) and *performance*, and addressed the following research questions by means of case studies:

RQ1 How much more precise is this approach with respect to the estimation provided by an OS-level monitor alone?

RQ2 How long does the static analysis phase take?

RQ3 How much slower is the instrumented application, and how do we compare with purely dynamic solutions?

We performed our experiments on the applications described in our running example (cf. §**??**), Zipper and Ftp-Client. Zipper was written by a student, while Ftp-Client was found online [**?**]. The code of these applications is intentionally minimal, in order to facilitate manual inspection of the results. Moreover, these applications stress-test our solution because our approach instruments only entry and exit points in the code (sources and sinks), but the vast majority of executed instructions are indeed sources or sinks; for comparison, we also run our solution on an application with little I/O and large amount of computation in-between: the Java Grande Forum Benchmark Suite[4], a benchmark for computationally expensive Java applications. We chose this framework, among others, to compare our results to those of related work [**?**].

3.1 Settings

Our evaluation was performed on a system with a 2.6 GHz Xeon-E5 CPU and 3GB of RAM. We ran our static analyser on all the applications using the different configurations described in §**??**. We report the median value for 30+ executions of each configuration, to weed out possible environmental noise. As OS monitor, we used an implementation from the literature [**?**]. All the runtime experiments use the objsens-D (§**??**) configuration for the static analysis phase. We decided for it because of its high precision in our tests; any other analysis, however, would generate statistically indistinguishable runtime performances.

3.2 Precision (RQ1) and Static Analysis Performance (RQ2)

First, by construction, our approach cannot be less precise than treating the processes as black boxes (= every output contains every input read so far), the

[3] We assume the static analysis to be sound: all actual flows are reported, i.e., there are no false negatives. Limitations of our approach are discussed in §**??**

[4] https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

typical conservative estimation made by OS-level monitors [**?**]. Second, while dynamic analyses usually rely on explicit flows only, static analyses consider additional kinds of dependencies between instructions (e.g. control-flow dependencies), generating more dependencies between sources and sinks. Third, even if we configure our static analyser to consider explicit-flows only, a static approach considers *every* possible execution at once, meaning that if at least one execution leads to a flow, then the sink statically depends on the source.

```
1  in=input();
2  if (cond) {
3    out=in;
4  }
5  output(out);
```

For instance, for the code on the left static analysis reports that the sink at line **??** depends on the source at line **??**, even considering explicit flows only. A runtime monitor would report the dependency only during those runs where condition `cond` at line **??** is satisfied. Replacing the runtime monitoring with a static dependency table introduces overapproximation by making the sink depending on the source during *every* execution, regardless of `cond`'s value.

We ran experiments on the scenario described in §**??**. We created the Zipper's configuration file, assigned to data **C**, and two files with random content (data **E** and **F**, respectively). In this scenario, we assumed that the only data read from the standard input is the password, marked as **P**. We then ran the scenario (i.e. we zipped the files using Zipper and sent them to the server using Ftp-Client) and looked at the sensitivity of the content that reached the sockets.

Table 1: Static analysis results for different configurations.

	Points-To	Time (s)	#Sources/ #Sinks	Precision (DI / D)
Ftp-Client	0-1-CFA	32	9 / 46	38% / 51%
	1-CFA	64	9 / 46	58% / 73%
	2-CFA	153	9 / 46	58% / 73%
	objsens	220	9 / 46	38% / 74%
Zipper	0-1-CFA	53	10 / 56	24% / 43%
	1-CFA	82	10 / 55	25% / 53%
	2-CFA	185	10 / 55	55% / 78%
	objsens	353	10 / 55	57% / 84%
JGFBS	0-1-CFA	211	8 / 84	56% / 59%
	1-CFA	580	8 / 81	71% / 75%
	2-CFA	626	8 / 81	71% / 77%
	objsens	360	8 / 81	73% / 79%

As expected, the execution using a black-box approach yielded a rather coarse-grained result (all data reached both sockets); in contrast, our solution provided the expected result (data **E** and **F** flowed only to the data socket, while **P** only to the control socket).

However, it is hard to *quantify* such an improvement in general. Considering that a black-box approach would always be as precise as our approach when every source is connected to every sink, a possible metric for precision improvement could be *the number of source-to-sink connections that we can safely discard*, thanks to static analysis. We let $\#flows$ denote the number of statically computed dependencies between sources and sinks, and measure precision as $1-(\#flows/(\#sources \times \#sinks))$, where 0 indicates that every source flows to every sink (like in the black-box approach) and 1 indicates that all sinks are independent from the sources, i.e. no data propagation. We are not aware of any better metric to measure precision of static analysis w.r.t dynamic monitoring.

As reported in Table **??**, we ran our analysis with various *points-to-analyses* (0-1-CFA [**?**], 1-CFA, 2-CFA, object-sensitive, cf. §**??**), considering only explicit (D), and additional implicit (DI), information flows. According to the formula

Table 2: Runtime analysis results. Underlined value taken from the literature, all others measured. Values in italic refer to results of comparable tests (cf. §**??**). $Zipper_{32}$ indicates the archiving of 261MB using internal buffers 32x bigger.

	Size (bytecode) orig.→instr.	Average overhead per sink/source		Overhead in total				
		Intra	Intra+OS	Intra	Intra+OS	[**?**]	[**?**]	[**?**]
Zipper	1611 → 2192	2.06x	22.92x	2.09x	34.28x	220.4x	-	-
Ftp-Client	9191 → 9785	0.16x	4.37x	0.28x	6.75x	25.7x	-	-
Java Grande	29003 → 30123	6.33x	144.65x	0.001x	0.07x	10.5x	0.25x - 1x	-
$Zipper_{32}$	1611 → 2192	0.24x	7.11x	0.33x	11.61x	19.7x	-	*15.2x - 28.1x*

above, the improved precision of the instrumented applications varies between 24% and 84% for Zipper, between 38% and 74% for Ftp-Client and between 56% and 79% for Java Grande F.B.S., depending on the configuration. Although some of these analyses are incomparable in theory, object-sensitivity tends to deliver more precision, as was already reported for various client analyses [**?**]. Note that these numbers are hard to relate to dynamic values, because they depend on the specific application under analysis and they do not take into account how many times a certain source/sink instruction is executed at runtime.

To answer **RQ2**, we also measured the time required to statically analyse our exemplary applications: between 30 and 626 seconds were needed to perform the static analysis (cf. Table **??**), of which 80-90% are invested in building the PDG, while the rest is spent on slicing. The choice of the points-to-analysis determines the size of the PDG and thus directly affects the total analysis time; our PDGs have between 10^4 and 10^6 nodes and between 10^5 and 10^7 edges.

3.3 Runtime performance (RQ3)

We tested our approach with multiple experiments based on our scenario (§**??**): transfer a 20K file to a remote server using Ftp-Client, and compress it using Zipper. We also ran our tool on the Java Grande F.B.S., the computationally expensive task with limited I/O used in the evaluation of [**?**]. We evaluated our approach (cf. Table **??**) in terms of the bytecode space overhead (column "Size (bytecode)"), the average execution time of a single instrumented source/sink (column "Average overhead per sink/source"), and the total execution time of the instrumented application (column "Overhead in total") compared to its native execution. We measured the execution runtime overhead with both monitors at the application and OS level (columns "Intra+OS"), and with just one inlined reference monitor at the application level observing only sources' and sinks' executions (columns "Intra"). In addition, we compared our work to other approaches, either by running our tool on the same scenario used to evaluate them [**?**] or, if possible, by running those tools on our tests. The latter is the case for *LibDFT* [**?**], an intra-process data-flow tracking framework for binaries.

Zipper and Ftp-Client applications are stress-testing our approach because they transfer data in blocks of 1KB at a time. This results in a huge number of read/write events: for comparison, creating a zip file from 261MB of data with

our Zipper generated ∼122K write and ∼256K read events, whereas *gzip*, an equivalent tool used in [**?**]'s evaluation, only generates 3781 writes and 7969 read system calls for the same input and the same output. Because [**?**] is a dynamic monitor that connects information flow tracking results for multiple applications across the system, we found a comparison to this work to be particularly relevant. To perform it, we ran a fourth experiment: archiving 261MB of linux source code with our Zipper application after increasing the size of the internal buffers by a factor of 32x; this way, for the same input, Zipper generates the same number of I/O events of the tool used in [**?**]. We are aware that comparing different applications is always tricky; however, since the number and type of generated events is almost identical, we believe the comparison to be informative and likely fair. Our results are presented in the last row of Table **??**. The overhead for archiving 261MB (11.61x) using our Zipper is smaller than the best value for gzip mentioned in [**?**] (15.2x-28.1x). Similarly, on the Java Grande test, we outperformed [**?**]'s analysis of one order of magnitude (0.07x vs 0.25x-0.5x).

Note that the static analysis and the instrumentation are executed only once per application. For this reason, we excluded the time to perform them from the computation of the relative runtime overhead (columns Intra and Intra+OS in Overhead, Table **??**). Also, the values in Table **??** do not include the time required to boot the Java Virtual Machine, which is independent of our instrumentation and thus irrelevant. It is worth noticing that we tried different configurations of LibDFT but we could only reproduce overheads more than ten times larger than those reported in the original paper [**?**].

4 Discussion

We now offer a general summary of our experimental results, elaborating on some of the technical and fundamental highlights and limitations of our approach.

By combining static and dynamic data flow technologies, we manage to track system-wide information flows between different programs and across different application layers. Our prototype implementation performs better than existing approaches although we are aware that this strongly depends on the application under analysis. While we have not "tuned" our approach to the examples in the case studies, we need to refrain from generalizing our findings. As we instrument only sources and sinks, on computationally intensive tasks with little I/O, like the Java Grande F.B.S., our tool exhibits a negligible overhead in practice (<0.07x). In more I/O intensive scenarios, our results are comparable or better than existing approaches. Note that while the tracking overhead per source/sink is stable (∼0.08ms "Intra", ∼2.2ms "Intra+OS"), the time to execute specific sources/sinks (e.g. >7ms for printing a certain string on standard output) can be longer than for others (e.g. ∼0.011ms for reading 1KB from a file), resulting in vastly different relative overhead.

We could improve the precision of our approach by leveraging additional information, e.g. the context in which a certain sink/source is executed [**?**]. However, this requires a) the use of a context-sensitive points-to-analysis, like 1-CFA,

usually more costly than a context-insensitve one (cf. §**??**), and b) additional instrumentation, which is the reason why we decided not to go for it. Other options to improve the precision of static analysis are ignoring certain kinds of flows, like those solely caused by exceptions, or manually adding declassification annotations to the code. While the first idea is acceptable, as long as one is fine with the respective change in the notion of soundness, we decided against manual annotations, envisioning the application of our tool in a scenario where static analysis is performed *automatically* on *unknown* code.

JOANA currently does not support dynamic language features like reflection and callbacks, challenging tasks for any static information flow analysis: dealing with reflection in a meaningful way requires approximating the possible values of strings which are passed as class or method names or to exploit runtime information [**?**], while callback-based applications (e.g. using Java Swing) require a model that captures the way the callback handlers are invoked. In other words, while JOANA can analyse multi-threaded programs (cf. §**??**), library-supported asynchronous communication between threads is still a limitation.

If we configured the static analysis to ignore all implicit flows (easy to circumvent [**?**]), the combination of our OS runtime monitor and the application reference monitors would guarantee a property similar to Volpano's *weak secrecy* [**?**]. On the other hand, a sound and precise system-wide non-interference analysis (including *all* information flows), would require to analyse all applications simultaneously, to also capture flows caused by the concurrent interactions on shared resources [**?**]. This is unfeasible even for a small number of applications and likely leads to prohibitively imprecise results. Our approach lies somewhere in-between: the static intra-process analysis guarantees non-interference between inputs and outputs of each application, while data flows across applications are captured at runtime. This property is stronger than weak secrecy, which completely ignores implicit flows, but still weaker than system-wide non-interference.

5 Related work

Approaches in the field of Information Flow Analysis can be roughly categorized in static, dynamic and hybrid solutions.

Static approaches analyze application code before it is executed and aim to detect all possible information flows [**?**,**?**]. A given program is certified as secure, if no information flow between sensitive sources and public sinks can be found. Such a static certification can for example be used to reduce the need for runtime checks [**?**]. Various approaches (apart from PDGs) can be found in the literature, usually based on type checking [**?**] or hoare logic [**?**]. Because of their nature, static approaches have problems with handling dynamic aspects of applications like callbacks or reflective code (§**??**), and are confined to the application under analysis, i.e. no system-wide analyses.

Dynamic approaches track data flows during execution and thus can also leverage additional information, like concrete user inputs, available only at runtime. *TaintDroid* [**?**] is a purely dynamic data flow tracking approach for system-

wide real-time privacy monitoring in Android. Despite its relatively small runtime overhead, TaintDroid focuses on explicit data flow tracking only. [**?**] proposes *ShadowReplica*, a highly optimized data flow tracker that leverages multiple threads to track data through binary files. While performance in general depends on the application under analysis, on I/O-intensive tasks ShadowReplica's runtime overhead is comparable to ours (cf. §**??**). [**?**] presents LibDFT, a binary-level solution to track data flows in-between registers and memory addresses. Although LibDFT's reported evaluation mentions little performance overhead, we could not reproduce these numbers: as shown in Table **??**, LibDFT imposed a bigger performance overhead than our approach; it is also unable to perform system-wide tracking because, in contrast to our approach, it cannot model flows towards OS resources (e.g. files) or in-between processes.

Whole-system tainting frameworks, on the other hand, can specifically track such kind of flows; among them we find *Panorama* [**?**], an approach at the hardware and OS levels to detect and identify privacy-breaching malware behaviour, *GARM* [**?**], a tool to track data provenance across multiple applications and machines, and *Neon*[**?**], a fine-grained system-wide tracking approach for derived data management. While the performance penalty they induce is comparable to ours, because of their dynamic nature, none of these tools can cope with implicit flows, nor exploit application-level semantics ("screenshot", "mail").

Hybrid approaches aim at combining static and dynamic information flow tracking approaches, usually to mitigate runtime-overhead. [**?**] presents a hybrid solution for fine-grained information flow analysis of Java applications; in this work, statically computed *security annotations* are used at runtime to track implicit information flows and to enforce security policies by denying the execution of specific method calls. In [**?**] the authors propose to augment a hybrid tracking approach with declassification rules to downgrade the security levels of specific flows and controlling information flows by allowing, inhibiting, or modifying events. Although both [**?**,**?**] show promising results, they do not take into account flows through OS-level abstractions, like files, nor between different applications or abstraction layers, as we do. We did not discuss so far the possibility of enforcing usage control requirements at the Java bytecode level in a *preventive* fashion [**?**] (i.e. execute a certain source/sink only if the tracker's response is affirmative), because, while requiring only minor changes in the instrumentation, denying method executions at this level may make the system unstable.

Other approaches model inter-application information flows by instrumenting sources and sinks in the monitored applications, relying on pure dynamic tracking [**?**] or on static analysis results [**?**] for the intra-application tracking. All of them, however, perform the inter-application flow tracking relying on the "simultaneous" execution of a sink in the sender application and a source in the receiver. None of them can model a flow towards an OS resource, like a file, nor towards a non-monitored application. In these scenarios, these approaches lose track of the data, while ours delegates the tracking to the OS level monitor.

6 Conclusions and Future Work

We described a new, generic approach to perform precise and fast system-wide data-flow tracking. We integrated static information flow analysis results with runtime technologies. In our case studies, our solution could track flows of data through and in-between different applications more precisely than the black-box approach does and faster than comparable dynamic approaches do. At present we cannot substantiate any claim of generalization of these results to other scenarios, but we are optimistic. While our proof-of-concept implementation connects executed Java code to an OS-level runtime monitor, other instantiations are possible. For instance, static approximations for flows in a database could be connected to dynamic measurements in a given application. Also, our general methodology is not restricted to specific programming languages or tools, so instantiations for languages other than Java are possible.

To the best of our knowledge, this is the first system-wide runtime analysis that replaces the internal behavior of applications by their static source/sink dependencies. Although hybrid approaches have already been proposed before, this kind of integration of static and dynamic results is the first of its kind.

Our experiments confirmed the intuition that the improvement in precision and performance depends on the type of information flows considered, and on the amount of I/O instructions executed (w.r.t the total number of instructions). Our solution is more suitable if this ratio is low, i.e. for applications that perform large computations on few inputs to produce a limited number of outputs.

We plan to apply our work to other programming languages, or the x86-binary level, although static analysis tools at this level exhibit bigger limitations. Additionally, we want to better understand the issues described in §**??**, in particular the exploitation of context-sensitive analysis information.

Acknowlegdements: This work was supported by the DFG Priority Programme 1496 "Reliably Secure Software Systems - RS^3" (grants PR-1266/1-2 and Sn11/12-1), and by the *Peer Energy Cloud* project, funded by the German Federal Ministry of Economic Affairs and Energy.