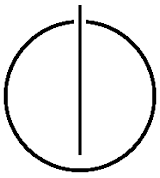
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Enhancing Data Flow Tracking for Data Usage Control

Alexander Fromm



FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Enhancing Data Flow Tracking for Data Usage Control

Alexander Fromm

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Florian Matthes
Prüfer der Dissertation:	
1.	Prof. Dr. Alexander Pretschner
2.	Prof. Dr. Stefan Tai, Technische Universität Berlin

Die Dissertation wurde am 16.12.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.03.2020 angenommen.

Acknowledgments

First and foremost, my grateful thanks goes to my supervisor Prof. Dr. Alexander Pretschner for his guidance and inexhaustible endurance during this work. In particular, many thanks goes to his sharpness and conciseness in all discussions that we had, and introducing me to the research world with all its facets: scientific writing, conferences, reviewing, research projects and presentations. Beyond that, I want to thank Prof. Dr. Alexander Pretscher for the possibility to participate in the *Software Campus* intiative and his grateful and outstanding support during that time. The Software Campus is one of my best experiences that I made during my Phd, because it gave me the opportunity to develop myself and to meet a lot of interesting and inspiring people from research and industry.

Furthermore, I want to gratefully thank Prof. Dr.-Ing. Stefan Tai for his kindness he showed me during our conversations. His valuable feedback, as well as the profound discussions with him, helped me to improve the present work in several inspiring directions and aspects.

I would like to thank to all my former colleagues at TU Munich for encouraging me when my moral was down. In particular, I would like to thank Florian, Enrico, Prachi, Tobias, Matthias, Dominik and Sebastian for the fruitful and cheering up conversations. Furthermore, many thanks to my co-authors for the fruitful and never-ending discussions and their direct, as well as indirect, contribution to this work.

Thanks to the German Federal Ministry for Economic Affairs and Energy (BMWi) and the German Federal Ministry for Education and Research (BMBF) which supported this thesis.

Thanks to all the Bachelor and Master students who conducted their theses under my supervision and whose results directly and indirectly influenced the contribution of this work.

Finally, I want to thank my family, and my mother and my father in particular, for their unconditional support I received over these years and for constantly reminding me that there is also a life outside of work. With your unwavering support this work would not have been possible.

Zusammenfassung

Access Control (AC) ist die erste Wahl von Dateneignern um ihre Daten vor unautorisierten Zugriffen zu schützen, und so den Zugriff nur auf autorisierte Datenkonsumenten zu limitieren. Sobald jedoch der initiale Zugriff einmal gewährt wurde sind Dateneigner darüber besorgt wie deren Daten in Zukunft durch den Datenkonsumenten genutzt werden. Insbesondere mit dem Aufkommen neuer Datenverarbeitungstechnologien, wie beispielsweise Cloud Computing, werden diese Bedenken verstärkt.

Das Forschungsfeld um verteilte Datennutzungskontrolle (*Data Usage Control (DUC)*) erweitert konzeptionell das Konzept von AC und liefert Mittel, Techniken und Mechanismen um Restriktionen hinsichtlich dessen wie Daten genutzt oder auch nicht genutzt werden dürfen sobald der initiale Zugriff einmal gewährt wurde zu erzwingen. Datennutzungsrestriktionen können temporale, propositionale, räumliche und kardinale Einschränkungen umfassen. Moderne DUC Systeme bedienen sich *Data Flow Tracking (DFT)* Techniken um Datennutzungsrestriktionen auf eine *daten*-zentrierte Art und Weise auf allen Kopien eines zu schützenden Datums zu erzwingen.

Obwohl bereits eine Vielzahl an verschiedenen DFT Ansätzen in der Forschung existieren sind deren Lösungen hinsichtlich Portabilität und Performanz limitiert. Ein Hauptgrund dafür ist, dass all diese Lösungen es erfordern ihre Trackinglogik über die gesamte Anwendung, einschließlich deren System- (z.B. Java Systemklassen für Java basierte Anwendungen) und 3^{rd}-party Bibliotheken, zu injizieren und zu verteilen. Ein solches Vorgehen führt dazu das der gesamte Code, welcher für die Ausführung der Applikation notwendig ist, miteinander verzahnt wird. Zum Beispiel wird dadurch die Ausführung einer überwachten Java Anwendung auf einer handelsüblichen Java Laufzeitumgebung ohne passend modifizierte Java Systemklassen nicht möglich. Ein praktikabler Weg um diesen Vorbehalt zu mildern wäre es einen DFT Monitor auf einer unteren Software Abstraktionsschicht zu platzieren, wie beispielsweise auf der Betriebssystemebene, und so alle auf ihr betriebenen Anwendungen zu überwachen. Dadurch wäre eine Trackinglogik innerhalb der Anwendung nicht notwendig. Forschungsergebnisse zeigen jedoch, dass ein solcher Ansatz zu *Überapproximation* führen kann, bei der der Fluss von Daten konservativ zwischen allen Ausgaben und Eingaben einer Anwendung propagiert wird. Im schlimmsten Fall kann es zu einem Systemzustand führen wo alles überall hin fließt. Diese Bedenken und daraus resultierende Probleme werden in Kapitel 1 detaillierter diskutiert und beschrieben.

Um die zuvor beschriebenen Bedenken zu adressieren, stellt diese Thesis einen neuartigen *hybriden* Ansatz vor um den Fluss von Daten innerhalb einer Anwendung zu überwachen.

Der vorliegende Ansatz kombiniert statische Informationsflussanalyse und dynamische Datenflussüberwachung um selektiv nur diese Programmstellen zu überwachen, welche auch tatsächlich relevant für den Fluss eines Datums sind. Auf diese Weise, und wie die vorliegenden Evaluationsergebnisse zeigen, sind wir in der Lage dynamische DFT-Tracker aus der Literatur Performanz mäßig zu übertreffen. Des Weiteren, stellt der vorliegende Ansatz die Portabilität einer zu überwachenden Anwendung sicher, da Trackinglogik nur an den Programmstellen injiziert wird wo es den Fluss eines Datums auch tatsächlich zu überwachen gilt. DUC Systeme profitieren vom vorliegenden Ansatz vor allem dadurch, dass Überapproximation bei der Datenflussüberwachung reduziert wird und somit ein präziseres Datenfluss Trackingergebnis für die Durchsetzung von Datennutzungsrestriktionen vorliegt.

Abstract

Access Control (AC) is one of the first choices for data owners to protect their data from unauthorized access and to restrict and limit data disclosure only to a selected set of authorized data consumers. However, once initial access has been granted data owners are concerned about how their data may be used further on in future by the data consumers. In particular, with the advent of new data processing technologies (e.g. Cloud Computing) this concern increases.

The research field of distributed *Data Usage Control* (DUC) extends the concept of AC and provides means, techniques, and mechanisms to enforce restrictions on how data may or may not be used once initial access has been granted. Such data usage restrictions encompass temporal, propositional, spatial, and cardinal constraints. Modern DUC systems employ DFT techniques to enforce data usage restrictions in a *data*-centric manner on all copies of a protected data item.

Although a plethora of different DFT approaches already exist in research, those solutions fall short in terms of portability and performance aspects. The main reason is, that those solutions require to inject and to spread their tracking logic over the entire application, including 3^{rd}-party libraries and code from the surrounding run-time environment (e.g. Java system-classes for Java-based applications). However, such an approach stick together the entire code which is required to run the application. For instance, running a monitored Java application on an off-the-shelf Java Run-time Environment would not be possible without properly modified Java system-classes. A practical way to address and to mitigate those caveats is to place a data flow tracker at a lower software abstraction layer, e.g. the operating system, and to monitor all applications which run on top of that layer. This way, no tracking logic is required at higher abstraction layers. However, as research results have shown such an approach can result in *overapproximation*, where data flows are conservatively propagated between all inputs and outputs of an application. In the worst case this might result in a system state where everything flows to everywhere. Note, we discuss those concerns and problems more thoroughly in Chapter 1.

To address the previously described concerns, this thesis presents a novel *hybrid* approach to track the flow of data inside applications. The present approach combines static information flow analysis with dynamic data flow tracking to track selectively only those program locations that are actually relevant for a flow of data. This way, and as the present evaluation results show, we are able to outperform performance-wise related dynamic DFT-trackers from the literature. Further, by design the present solution preserves the portability of the monitored applications,

as tracking logic is only injected at those code regions where the flow of data needs actually to be monitored. Beyond that, DUC systems benefit from the present approach as it reduces overapproximation in data flow tracking, and thus, provides a precise data flow tracking result to enforce data usage restrictions only on those data items that are actually affected.

Contents

1. Introduction **1**

1.1. Motivation 1
1.2. Running Example 3
1.3. Gap Analysis and Problem Statement 5
1.4. Assumptions 9
1.5. Solution and Contribution 10
1.6. Thesis outline 11
1.7. Relevant Publications 12

2. Foundations on Usage Control **13**

2.1. Usage Control Elementaries 13
2.2. Usage Control Policies and Mechanisms 15
2.2.1. Specification Level Policy 15
2.2.2. Implementation Level Policy 17
2.2.3. Usage Control Mechanisms 19
2.3. Generic Data Flow Model 21
2.4. Usage Control Infrastructure 24
2.5. Policies by Examples 32

3. Hybrid Data Flow Tracking **33**

3.1. State of the Art Information Flow Analysis 34
3.2. Overview of the hybrid approach 35
3.3. Static Information Flow Analysis 38
3.4. Run-time Data Flow Tracking 43
3.5. Data Flow Tracking Model for Java 52
3.6. Evaluation 61
3.6.1. Precision 62
3.6.2. SHRIFT versus HDFT++ 66
3.6.3. Performance 78

3.6.4. Threats to Validity 82
3.7. Strengths and Limitations 84
3.8. Summary and Conclusion 88

4. Related work **91**
4.1. Information Flow Tracking 91
4.2. Usage Control 97

5. Conclusion and Future Work **99**
5.1. Conclusion 99
5.2. Future Work 100

A. Appendix **103**
A.1. Analysis reports 103

Index of Acronyms **119**

1. Introduction

1.1. Motivation

Within the last decade, data processing systems have undergone an impressive transformation: from heavyweight, monolithic applications towards dynamic distributed software systems. Additionally, the rise of mobile-, ubiquitous-, pervasive-, and cloud-computing augments data processing systems to capture and process data more efficiently and in larger quantities than ever before. For instance, WhatsApp [123], a popular mobile messaging service, processes several terabytes on data every day from its more than one billion users. Frequently, the processed data $\mathcal{D}$ is sensitive, and hence, needs to be protected from a data security perspective.

A prevalent approach for data protection is *Access Control* (AC) [18, 48, 104–106]. Conceptually, AC systems restrict and limit data disclosure only to a selected set of authorized users, henceforth termed *data-consumer*. Access restrictions are specified and imposed by a *data-provider* in form of past or present conditions, henceforth termed *provisions*, that must be fulfilled by the data consumer at the moment in time when an access request is made, e.g. t_0 in Figure 1.1. However, once initial access has been granted AC protection for the released and disclosed data $\mathcal{D}$ is lost, and especially, data providers are concerned about how their data is used further on in the future $t_n \geq t_0$ by the data consumer.

The research field of distributed *Data Usage Control* (DUC) [89] tackles such types of concerns and provides mechanisms to specify and to enforce data usage *obligations* on data [83, 90]. Obligations, henceforth also termed *policies*, concern the future usage of data at timestep $t_n \geq t_0$ (cf. Figure 1.1) and encompass temporal, propositional, spatial, and cardinal restrictions on how data may or may not be used once initial access have been granted at timestep t_0. Adherence to data usage control policies, like *"Do not disseminate my data"*, must be continuously monitored inside a data processing system before, during, and/or after the usage of data [83]. To do so, installed reference monitors continuously observe and evaluate a system execution against data usage control policies. Policy enforcement may happen either in a *preventive* or *detective* mode [91]. Conceptually, the former one actively tries to prevent a policy violation by either allow, modify, delay, or inhibit the current execution. Whereas, *detective enforcement*, simply

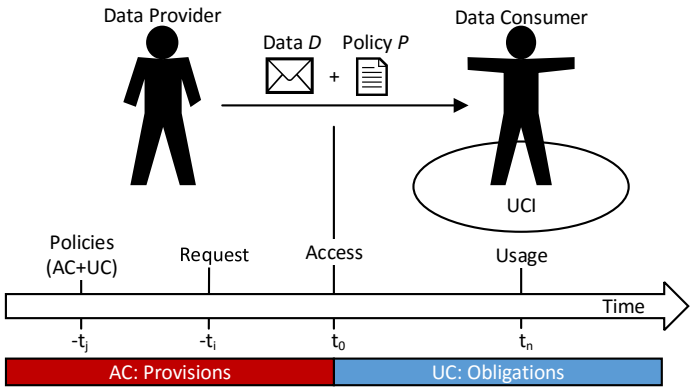


Figure 1.1.: DUC is an extension of AC and addresses the problem of how data may or may not be used at time point t_n once initial access has been granted at t_0.

said, works by observing, detecting, and logging policy violations for an a posteriori analysis.

Data usage control policies are expressed either in a *container*- or *data*-centric fashion. Intuitively, the former one specifies policies in terms of a data item's concrete technical *representation*. A technical representation serves as a kind of storage *container* for a data item at one particular *abstraction layer*, and may be for instance a file at the file-system layer or a Java-object at the Java-layer respectively. For example, to prevent a picture (stored in a file `pic1.jpg`) from dissemination a container-centric policy specification would be *"Do not disseminate pic1.jpg"* (Policy P_1) [92]. Note, the container identifier, which is the file name in this case, is part of the policy specification. As data mostly moves around in a data processing system and therefore may take various representations and containers in particular, e.g. `pic1.jpg` may be copied or edited and saved to `pic2.jpg`, it is necessary to write a modified version of Policy P_1 that addresses `pic2.jpg` as well. Otherwise, the content of `pic1.jpg` can be easily unlinked from its policy by just copying the file.

However, as one assumes with such an approach the number of policies easily explodes, and hence, makes the policy maintenance process quite cumbersome, e.g. policy updates due to changing obligations. Because of that, the notion of *representation-independent* data usage control was introduced [92], which distinguishes between abstract data (i.e. the content that the user actually wants to protect) and its concrete representation. This concept allows the specification of *data-centric* policies where data usage restrictions are tailored around the abstract notion of a data item, and hence, are independent from a data's concrete representation. For instance, Policy P_1 can be specified in a data-centric manner as *"Do not disseminate my picture"*(Policy P_2), and thus, talks about all possible representations of a data item. However, to detect all possible representations and copies of a protected data item, DUC has been extended

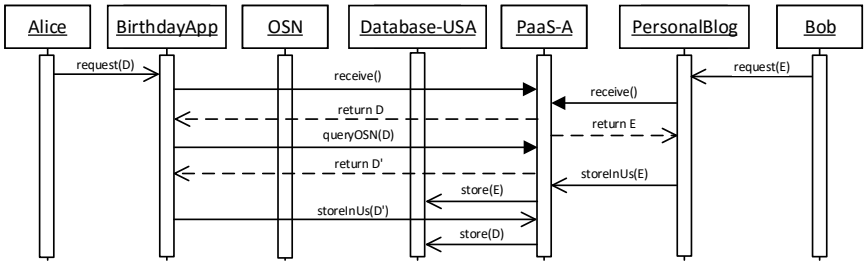


Figure 1.2.: BirthdayApp and PersonalBlog are using the functions `receive()` and `storeInUs` to receive and store data. Alice protects her transmitted data D by a policy that requires to track the flow of D between those functions. As BirthdayApp and PersonalBlog share the same Tomcat web-server instance inside PaaS-A, a flow of data is also mistakenly reported for PersonalBlog.

and combined with concepts and mechanisms from the research field of DFT [39, 95] in order to track the flow of data in a data processing system. That way, DUC systems are provided with valuable information about where protected data items reside, and hence, are affected by a DUC policy in a representation-independent manner.

This thesis investigates the DFT pillar of DUC systems. Based on a thorough and rigorous analysis of current state of the art data flow tracking approaches and the illumination of their drawbacks in the field of DUC, this work provides a new data flow tracking concept. Taking the identified caveats into account, this thesis contributes with novel DFT approaches that make improvements regarding precision, performance, and portability of current state of the art DFT solutions.

1.2. Running Example

To illustrate and emphasize the core problems that this thesis builds upon, this section describes a figurative sample use case scenario from the cloud domain (cf. Figure 1.3): *BirthdayApp* and *PersonalBlog*, both Java-based web-services, are operated on a single Tomcat web-server instance running inside *PaaS-A*. To serve their functionality both services rely on external data storage. For the sake of simplicity, we assume in Figure 1.3 a database instance for that function. Although the following scenario description is based on Java web-services, our proposed approach and solution is generic and can also be applied for other kinds of programs, like Java standalone applications.

For an *Online Social Network (OSN)*, like Facebook[1], it is quite common to open its platform

[1] `https://www.facebook.com`

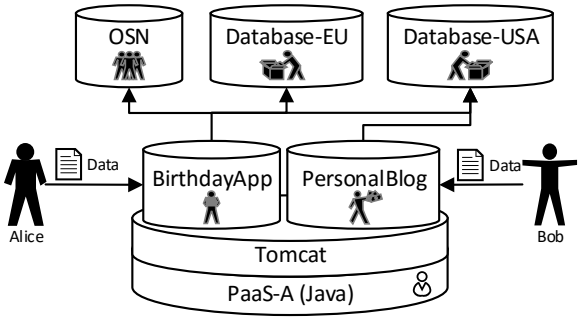


Figure 1.3.: PaaS provider *PaaS-A* provides a shared run-time environment for two cloud services: BirthdayApp, an external third-party birthday calendar service for an OSN, and PersonalBlog, a web application to manage and create a personal blog. Further, each service uses external databases to store their generated data.

for third-party services. In Figure 1.3, Alice uses BirthdayApp to display her friend's birthday dates in a calendar. To do so, BirthdayApp needs to query the OSN network on behalf of Alice to retrieve all her friends including their birthday dates (function `F1`). Furthermore, BirthdayApp uses external databases to persistently store their accruing data, like friend-relationships or birthday dates (function `F2`). As one may notice, those are located in the EU and the USA which have different data protection regulations. Technically, PaaS-A provides an SDK-library with the build-in functions `queryOSN(user)` to perform `F1`, `storeInUs(data)` to perform `F2`, and `receive()` to receive data from a user's request to its running services. As Figure 1.2 illustrates BirthdayApp uses those build-in functions to implement its functionality.

Next to BirthdayApp the Tomcat web-server also runs PersonalBlog, a web-service to manage and to create personal blog channels. PersonalBlog is used by Bob (Figure 1.3) to post articles about his latest experiences in cloud programming on his public blog channel. In contrast to Alice, Bob does not set a great value upon privacy and actually does not care that PersonalBlog is using persistent database storage in the USA. Hence, there is no need to track the flow of data between the build-in functions `receive()` and `storeInUs(data)` inside PersonalBlog.

Note, in theory cloud computing postulates to provide an unlimited amount of computational resources, in practice however, computational resources are still limited. Therefore cloud providers have to share their computing platforms among their customers in order not to waste computational resources. Because of that, BirthdayApp and PersonalBlog are operated by PaaS-A on the same identical run-time execution platform, and thus, technically share the same *Java Runtime Environment (JRE)* and Tomcat web-server instance.

As Alice is an extremely privacy-aware person, she would like to prevent BirthdayApp from storing her data in the USA by enforcing the following DUC policies:

Policy P_1 "Do not disseminate my personal data"

Policy P_2 "Store my data only on servers in the EU"

Policy P_3 "Encrypt my data before transmitting them to servers in the USA"

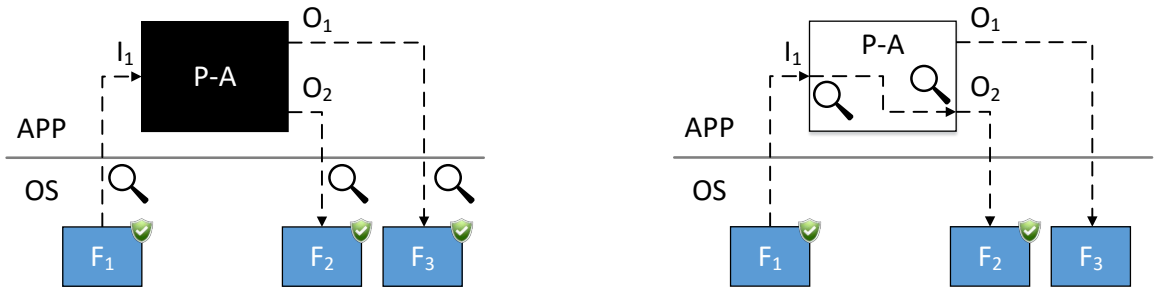
Policy P_4 "Delete my data within 30 days after receipt"

To enforce Policy P_1– Policy P_4 with a DFT based DUC system, it is necessary to track the flow of data inside BirthdayApp, in particular, the data flow between the functions `receive()` and `storeInUs(data)`. However, as BirthdayApp runs on a shared JRE, current data flow trackers are not able to properly track the flow of data inside BirthdayApp separately from PersonalBlog, and therefore, would mistakenly also report a flow of data inside PersonalBlog. The following Section 1.3 illuminates those concerns and caveats.

1.3. Gap Analysis and Problem Statement

To track the flow of data, research in the field of distributed *Data Usage Control* (DUC) proposes a plethora of different *Data Flow Tracking* approaches and techniques which address different abstraction layers, like Android [26], X11 [95], the Internet Protocol [53], MS-Office [108], MS-Windows [124], or Thunderbird [69]. Conceptually, all those solutions have in common to propagate a *taint-label* along the lines of executed program instructions. The value of a taint-label is considered as an alias or identifier which represents the actual protected data item.

Depending on the abstraction layer, a DFT tracker may take into account layer-agnostic semantics within its taint propagation mechanism and tracking result. For instance, TBUCE [69], a data flow tracker for the THUNDERBIRD mail-client, is designed and implemented as a THUNDERBIRD-extension, and thus, is able to take into account domain knowledge from the inside of the mail-client, like the notion of an e-mail. This way, TBUCE is able, for instance, to track that an e-mail was printed or that e-mail content flows from one e-mail to another or across different THUNDERBIRD instances. On a figurative software abstraction layer stack, as illustrated in Figure 1.5, TBUCE would reside at the top as it is particularly tailored and designed for THUNDERBIRD. However, as one may notice, although TBUCE provides domain-specific tracking results its field of application is limited to THUNDERBIRD due to its tailored implementation. Tracking the flow of e-mails inside other mail-clients, like MS-Outlook, requires another, dedicated implementation. This means in corollary, transferring this methodological approach to our running example (cf. Section 1.2), we would need to implement a dedicated monitor for BirthdayApp in order to enforce Policy P_1. Enforcing Policy P_1 inside another web-



(a) Overapproximation

(b) Precise tracking from inputs to outputs

Figure 1.4.: As in Figure 1.4a no information is available about the data flow dependencies between input- and output-channels ($\mathtt{I}_x$, $\mathtt{O}_x$), data flow tracker at the OS layer would conservatively propagate the taint-label from file $\mathtt{F}_1$ to $\mathtt{F}_2$ and $\mathtt{F}_3$. However, in Figure 1.4b data flow tracking monitors are placed at the application level, which analyzes data flow dependencies between input- and output-channels, and hence, precisely propagates the taint-label only from file $\mathtt{F}_1$ to $\mathtt{F}_2$.

service, like PersonalBlog, would again require a dedicated monitor specifically designed for PersonalBlog.

In contrast, Wüchner et al. [124] provide a DFT tracker at the MS-Windows operating system layer. Using system call interposition, their solution observes and inspects low-level system calls from all processes running on top of the operating system, and thus, their solution resides at the bottom in our software abstraction layer stack in Figure 1.5. On one hand, such an approach makes it quite hard for a process (or application) to hide from being monitored. But on the other hand, as this approach operates on a low abstraction layer (cf. Figure 1.5), higher-level semantics and processes' internals (like input-ouput dependencies) are not available, and hence, may lead to imprecise tracking results. Figure 1.4 illustrates this caveat: as the monitor 🔍 at the OS level has no insights about how outputs $\mathtt{O}_1$ and $\mathtt{O}_2$ depend on the input $\mathtt{I}_1$, Wüchner et al. consider the *Application under Scrutiny (AuS)* as a *black-box* and conservatively propagate the taint-label (🛡) to both outputs, although $\mathtt{I}_1$ only flows into $\mathtt{O}_2$, as Figure 1.4b reveals. This way, it is guaranteed that no data flows get lost and missed. But on the downside, it overapproximates and mistakenly propagates taint-labels to outputs that are not dependent or affected by any sensitive input, and hence, reports data flows that do not even exist, e.g. the flow from $\mathtt{I}_1$ to $\mathtt{O}_1$ in Figure 1.4a. Henceforth, we term this type of concern *overapproximation*. Due to this imprecision, the entire system may get tainted after some time [124]. In the worst case, this approach would lead to a scenario where a taint-label is spread over the entire AuS system, and hence, a policy like Policy P_1 would be falsely enforced on data items that are actually not even affected by Policy

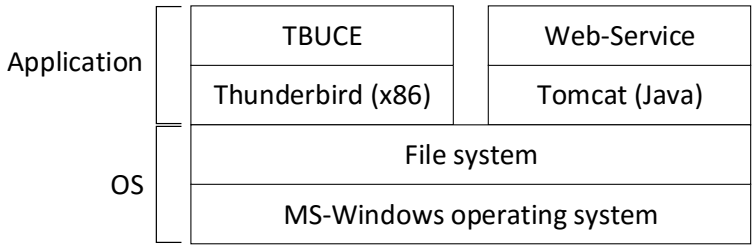


Figure 1.5.: Sample software abstraction layer stack with four abstraction layers. The two layers at the top are used by the application to perform its domain-specific function. Whereas the two layers at the bottom provide the fundamental and rundimentary functions to operate and run the applications.

P_1. Moreover, as BirthdayApp and PersonalBlog are running on the same JRE in our running example (cf. Section 1.2) and therefore in the same process, an OS-level monitor is not able to differentiate data flows between those two web-services. Thus, enforcing Policy P_1 would affect both web-services.

Dynamic Taint Analysis (DTA) has been proposed as a technique to track the flow of data inside applications, and thus, to detect dependencies between output- and input-channels. *Input-channels* (henceforth termed *sources*) are those instructions in a program that transfer data from the outside into an application, like reading data from the network socket. Whereas, *output-channels* (henceforth termed *sinks*) are exactly the opposite and transfer data from the inside to the outside, like writing data to a network-socket. *Processing-Instructions* reside on a path of instructions that in sum transfer data from input- to output-channels (like arithmetic operations). In contrast to the approaches from Lörscher [69] and Wüchner et al. [124], DTA operates at the application's code-layer, e.g. at the x86-binary- (like LIBDFT [54]) or Java-bytecode- (like PHOSPHOR [10]) layer. DTA relies on full instrumentation of the entire application including system- and third-party-libraries and works conceptually this way: initially, each program variable gets a taint-label assigned. On each executed input-channel, the taint-label value of the receiver variable inside the application, i.e. the variable which points to the data read from the input-channel, is set to a unique identifier. At run-time, this taint-label value is propagated along the lines of executed commands. Once an output-channel is reached, the taint-label values of the output parameters reveal on which input-channels they depend.

However, to do so DTA requires to modify and to inject its tracking logic into the entire application code. This includes, cf. Figure 1.6, the *program*, which contains the code that implements the actual program logic, the *Runtime Environment (RTE)*, which contains the required execution code, and possibly 3^{rd}*-party-libraries*. Such modifications encompass not only adding

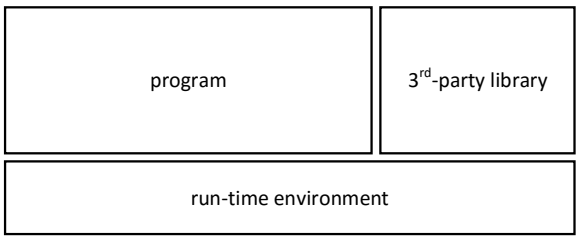


Figure 1.6.: Conceptually, an application is composed of code that either directly belongs to the program, to a 3rd-party library, or to the RTE.

a taint-label for each variable, but also, for instance in case of PHOSPHOR (a popular DFT tracker for Java-based applications), modifying the Java method- and class-signatures in order to propagate taint-labels properly. Taint propagation happens not only at the program-code level but also through 3rd-party-libraries and all Java system-classes which belongs to RTE. As one may notice, such deep modifications are quite invasive and tightly bind program-, 3rd-party-library-, and RTE-system-classes together. This thesis advocates that such an approach affects the portability of the AuS and makes it quite difficult to port and run the adapted and modified AuS inside other, off-the-shelf run-time execution environments. Derived from that, this thesis addresses the research questions

RQ1: How can we improve the tracking precision, and thus, reduce overapproximation of the tracking results compared to a pure black-box approach?

RQ2: How can we track efficiently and effectively data flows inside an AuS from its input- to its output-channels, and simultaneously, preserve and maintain the portability of the AuS?

A common trait of existing DFT systems, like PHOSPHOR [10] or LIBDFT [54], is it, to treat an AuS as if it owns an entire RTE where no other applications are executed and operated. However, for particular domains where computational resources are valuable goods, and therefore are shared among different applications, such a requirement might be too costly. For instance in our running scenario in Section 1.2, BirthdayApp and PersonalBlog share the same identical Tomcat web-server instance, and thus, the same JRE installation. A preliminary task to enforce Policy P_1 on Alice's transmitted data to BirthdayApp, is the detection of all data flow dependencies between PaaS-A's build-in instructions `receive()` and `storeInUs(data)`, which are used to receive data from a user request and to store data in the USA respectively. To do so, a conventional

DFT tracker, like PHOSPHOR [10], has to add its tracking logic into the code of BirthdayApp, its used 3rd-party-library-classes, into the Tomcat web-server, and into all Java system-classes, as it relies on a full instrumentation of the entire application code including system- and third-party-libraries. Moreover, tracking logic must also be added into PersonalBlog, as otherwise we would need a separated, unmodified Tomcat web-server instance only for PersonalBlog.

One may notice, as PersonalBlog uses the same RTE, i.e. the same JRE (especially the same Java system classes which are stored within the Java system-library `rt.jar` of each JRE installation) and the same Tomcat web-server instance, and therefore the same identical methods to read and store data, PHOSPHOR [10] (and other data flow tracking solutions [81, 96]) reports a flow of data for PersonalBlog as well. The reason is that PHOSPHOR [10] introduces a taint-label value at the end of an input-instruction. For instance, in case of method `receive()`, PHOSPHOR puts a taint-label value at the end of `receive()`'s implementation. In reverse, it means that with every execution the same taint mark is introduced independently from the calling AuS. Exactly this point leads to the issue that a critical flow of data is also mistakenly reported for PersonalBlog, although it is not of particular interest as there is no need for DUC policy enforcement according to our running scenario. However, to enforce Policy P_1 in a representation-independent manner, i.e. on all copies in a shared RTE, the corresponding DUC system, and its DFT system in particular, must be able to differentiate between those web-services and track the flow of data separately. In more generic terms and under the assumption that multiple AuS are running on the identical RTE, this thesis addresses the research question:

RQ3: How can we track the flow of data separately inside different AuS that share the same RTE?

In sum, this work advocates that current DFT solutions for DUC systems are insufficient (cf. Chapter 4), as they either (i) suffer from *overapproximation* (**Caveat 1**) [39, 53, 124], and therefore provide imprecise tracking results, (ii) or they impose too much performance overhead (**Caveat 2**) [54], (iii) or they affect the portability of the AuS (**Caveat 3**)[10]. This thesis proposes a new approach to track the flow of data, addressing the previous caveats.

1.4. Assumptions

The term "Runtime Environment" is widely used in different domains, and thus, may have different notions. For instance, in our running example in Section 1.2, one may consider the

platform PaaS-A as the run-time environment which may include, apart from the Tomcat web-server, also all other components like load-balancer, containerization, virtualization etc. Another notion would be to consider the Tomcat web-server as an RTE. So depending on where a line is drawn, the term RTE may have different notions and may encompass different components. RTEs have been build for different programming languages, like Java [45], Ruby [101], or PHP [86]. Moreover, they have been applied to different domains and pervade plenty of different disciplines. As one of the most popular RTE, the JRE has been adapted for Android (to run mobile applications), or for the *Amazon's Elastic Beanstalk (AEB)* [2] (to run cloud services).

However, in order to avoid ambiguity, this thesis considers the immediately enclosing unit that interprets and executes AuS's commands and instructions as an RTE. Further, we assume that RTEs abstract away from lower technical details and provide, via a programming interface, a uniform view on universal, commonly used basic functions to applications. Such functions, for instance, may encompass reading/writing from/to files or network-sockets. Concretely, as the present work has been prototyped in Java we consider the Java Runtime Environment, their system-classes `rt.jar` in particular, as the RTE to execute the AuS. We do not consider a Tomcat web-server as an RTE (cf. our running example in Section 1.2), because simply speaking the Tomcat web-server is a collection of libraries to support the execution of Java web-services inside a running JRE (e.g. they provide the implementation of *Java Specification Requirements (JSR)* 109). Thus, we consider them as a kind of third-party-libraries in our software-stack (cf. Figure 1.5). Moreover, we also do not consider an entire PaaS-platform as an RTE because an entire PaaS-plaform encompasses much more components, like load-balancer, containerization, virtualization, etc, and thus, are out of scope in this work.

Further, we assume a *lazy-but-benign* AuS which lacks mechanisms to track the flow of data inside for subsequent enforcement of data usage restrictions. This means we do not assume a malicious AuS that actively tries to circumvent the proposed solution.

1.5. Solution and Contribution

To address the research questions from Section 1.3, this thesis proposes a *hybrid* approach which combines *Static Information Flow Analysis (SIFA)* and *Dynamic Data Flow Tracking (DDFT)* techniques to improve the data flow tracking pillar in DUC systems, concerning precision and performance of the data flow tracker, as well as the portability of the AuS.

SIFA analyzes the flow of data statically at the programming level (or at one of its intermediate representation) without executing the AuS. In contrast, DDFT tracks the flow of data at AuS's run-time by injecting a kind of reference monitor into AuS; Section 3.1 provides a more thorough

and deeper discussion on the distinction between those two approaches. Combining both techniques, as we do, enable us to statically precompute data flow dependencies and to track selectively only those program locations that are actually relevant for a flow of data inside the AuS; any other locations are omitted. That way, our approach provides valuable information about how data propagates through and where multiple data copies reside within an AuS (web-services in our use case scenario, cf. Section 1.2). With our approach, we aim to support data usage control policy enforcement more precisely in a representation independent manner.

To the best of the author's knowledge, this thesis is the first work that investigates and provides a hybrid data flow tracking solution for data usage control enforcement. As a proof of concept, Java-based prototypes (viz. SHRIFT and HDFT++, cf. Chapter 3) have been implemented to show the underlying concepts. In summary, this thesis contributes with:

1. a novel hybrid data flow tracker that preserves the portability of an AuS while reducing overapproximation and performance overhead.

2. a hybrid data flow tracker to support DUC systems in their policy enforcement. This approach is novel as the run-time tracking logic tracks data flows only at the application level and does not rely on tracking logic within the RTE. That way, data flow tracking results for each application are isolated from each other and do not intervene.

3. the first instantiation of a generic data flow model [39] for Java bytecode. Backed by this model, usage control policies are specified and enforced in a representation-independent manner on all copies of a protected data item (cf. Chapter 2).

1.6. Thesis outline

This thesis is structured as follows. Chapter 2 provides fundamental foundations in DUC. It describes the DUC's underlying system model, its policy specification language, and provides a high-level logical view on the elementary components of a DUC infrastructure. Furthermore, it provides a generic data flow model that is instantiated in Section 3.5 for Java bytecode. The core contribution of that thesis, viz. *hybrid data flow tracking* approach, is described in Chapter 3. Furthermore, this chapter provides our evaluation results regarding tracking precision and run-time performance overhead, and discusses the strengths and limitations of our approach. Chapter 4 provides related work from the field of *Information Flow Tracking (IFT)* and DUC. Finally, Chapter 5 concludes this thesis and describes potential future works and directions regarding the field of IFT.

1.7. Relevant Publications

To address the research questions RQ1 - RQ3, the research results presented in this thesis are built upon various research publications which had been published during the conduction of this dissertation in [51], [71], and[28]. This section gives an overview of those publications and, wherever it is appropriate, those research publications will be referred to in this thesis.

First and foremost, our motivation's running sample in Section 1.2 was mainly driven by the published work in [51]. In that paper we present a compliance monitoring solution in the field of online social networks (OSN) which prevents third-party applications to use OSN-user data in a non-compliant way.

Research results on combining static and dynamic data flow techniques, forming the foundation of this thesis, have been published in [71], a joint work between Lovat et al. and the author of this dissertation. The work in [71] presents SHRIFT and addresses in the first place the over-approximation problem (RQ1). Further, this paper also provides the SHRIFT implementation for Java applications and shows its evaluation results in terms of performance and precision. The present dissertation contributes to SHRIFT [71] with regard to the overall approach of combining static and dynamic data flow techniques, its Java prototype implementation to track the flow of data at the application software layer, as well as, with its performance and precision evaluation results. We want to explicitly mention that this dissertation does not contribute to the second part of SHRIFT [71] which combines multiple data flow tracking monitors from different software abstraction layers in order to track the flow of data in a system-wide manner between and across different system processes and software abstraction layers.

We have published HDFT++, an extension of SHRIFT, in [28]. HDFT++ tracks in addition to sources and sinks also intermediate instructions residing on a data flow dependency path between sinks and sources. This way, HDFT++ is able to detect if a statically detected data flow dependency actually happens at run-time or not. At that point, we would like to refer to Chapter 3 where HDFT++ is described and also compared to SHRIFT.

2. Foundations on Usage Control

Usage Control (UC) is an extension of *Access Control* (AC) [63, 64] and concerns not only who is authorized to access which data, but also how data may or may not be used in future once initial access has been granted [39, 42, 83, 90, 92, 94]. To do so, a *data provider* (an entity who discloses and gives data away) imposes data usage restrictions, formally specified in data usage control policies $\mathcal{P}$ with the *Obligation Specification Language (OSL)* (cf. Section 2.2), on their data $\mathcal{D}$. Once access has been granted to a *data consumer* (entity who request and receive data), data $\mathcal{D}$ is transferred together with $\mathcal{P}$ in a sticky manner to the data consumer (cf. Figure 1.1). This, however, only applies if a proper usage control infrastructure UCI (cf. Section 2.4) is installed and capable to enforce $\mathcal{P}$ on the data consumer's side. An important distinction needs to be mentioned: AC takes decisions based on *provisions*, i.e. authorization constraints that refer to past conditions upon whose fulfillment access gets granted. On the contrary, *Usage Control (UC) obligations* specify constraints that refer to the future usage of data. The following sections describe the fundamental UC concepts upon which this thesis builds upon.

2.1. Usage Control Elementaries

This section describes the fundamental concepts upon which usage control policies are built. Conceptually, DUC policies are expressed as propositional, temporal, and cardinal constraints over system runs. A system run is modeled as a set of timed *Traces* $\mathcal{T} : \mathbb{N} \rightarrow \mathbb{P}(\mathcal{E})$, that maps abstract time points $t \in \mathbb{N}$ to a set of *events* $\mathcal{E}$. Each $t \in \mathbb{N}$ represents the continuous *timeframe* between $(t-1, t]$, in which an event may happen. An event consists of a name $\mathcal{N}_\mathcal{E}$ and a set of parameters $\mathcal{J}$, which in turn are defined by a name $\mathcal{N}_\mathcal{P}$ and a value $\mathcal{V}_\mathcal{P}$. For any event $e \in \mathcal{E}$, $e.n$ denotes the event's name and $e.p$ its set of parameters (including parameter names and values).

$$\mathcal{J} \subseteq \mathcal{N}_\mathcal{P} \times \mathcal{V}_\mathcal{P}$$
$$\mathcal{E} \subseteq \mathcal{N}_\mathcal{E} \times \mathcal{C}_\mathcal{E} \times \mathbb{P}(\mathcal{J})$$

We assume that every event contains two special, reserved parameters by default, viz. $obj \in \mathcal{N}_\mathcal{P}$ and $isActual \in \mathcal{N}_\mathcal{P}$. The former one denotes the target primary object of an event, such as a

file or network request data. Whereas, the latter one is boolean-typed and denotes if an event is *actual*, i.e. $isActual = true$, and has already happened or if it is *desired*, i.e. $isActual = false$, and attempted to happen; syntactically $\mathcal{A}(\cdot)$ and $\mathcal{D}(\cdot)$ captures this distinction. Moreover, this distinction is necessary to differentiate between events that trigger usage control mechanisms, and hence needs to be approved by usage control policies, and events that are already approved or even do not need approval, and therefore are actually allowed to be executed. In that syntax an example event declaration is

$$e_\tau = (send, \{(obj, movie), (isActual, true), (destIp, 131.154.160.1), (destPort, 8080)\}) \tag{Equation 2.1}$$

Event e_τ specifies that data *movie* has been sent over the network to the destination IP 131.154.160.1 and destination port 8080. As one may notice, this is an actual event as $\mathcal{A}(e_\tau) = true$ and therefore this event has already happened.

Further, we categorize events into specific event-classes $\mathcal{C}_\mathcal{E}$ which we divide into *data-usage* and *container-usage* . Intuitively, data-usage events encompass and address every representation of a data item, and therefore all containers where a protected data item resides. Contrary, container-usage events address a single container. Furthermore, we define a function $getClass : \mathcal{E} \rightarrow EventClass$ that determines the event-class for an event $e \in \mathcal{E}$.

Beyond that, DUC provides the concept of *event refinement* which enables universal quantification over the unmentioned parameters. To capture that, $refinesEv \subseteq \mathcal{E} \times \mathcal{E}$ defines a refinement relation on events. An event $e_1 \in \mathcal{E}$ refines an event $e_2 \in \mathcal{E}$, if e_1 and e_2 have identical names and if all parameters (names and their values) of e_2 are a subset of e_1

$$\forall e_1, e_2 \in \mathcal{E} : \; e_1 \; refinesEv \; e_2 \iff e_1.n = e_2.n \; \wedge \; e_1.p \supseteq e_2.p$$

Event refinement is a helpful concept during policy specification as it allows to quantify over all unmentioned parameters. For instance, Equation 2.2 shows an event specification (derived from Equation 2.1) that covers all possible ports, thus $e_\tau \; refinesEv \; e'_\tau$.

$$e'_\tau = (send, (obj, movie), (isTry, true), (destIp, 131.154.160.1)) \tag{Equation 2.2}$$

However, at system's run-time all actually executed events are "maximally refined", i.e. all parameters are determined and have a value; *maxRefEv* captures that case:

$$maxRefEv = \{e \in \mathcal{E} \;\; | \;\; \nexists e' \in \mathcal{E} \; : \; e' \neq e \; \wedge \; e' \; refinesEv \; e\}$$

In consideration of the example in Equation 2.1, to declare the same event for any arbitrary port, one would need to replicate this declaration for all possible ports, ranging from 0 to 65535.

However, as this would be a cumbersome task *variable events* $\mathcal{VE}$ are introduced that enables the specification of variables $Var : \mathcal{N}_\mathcal{V} \rightarrow \mathcal{D}_\mathcal{V}$ in an event declaration. Mapping function Var maps a variable name $\mathcal{N}_\mathcal{V}$ to a domain of possible variable values $\mathcal{D}_\mathcal{V} = \mathbb{P}(\mathcal{N}_\mathcal{E} \cup \mathcal{V}_\mathcal{P})$

$$\mathcal{VE} \subseteq (\mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{V}) \times \mathcal{C}_\mathcal{E} \times \mathbb{P}(\mathcal{N}_\mathcal{P} \times (\mathcal{V}_\mathcal{P} \cup \mathcal{N}_\mathcal{V}))$$

In that context, function $Inst_\mathcal{E} : \mathcal{VE} \rightarrow \mathbb{P}(\mathcal{E})$ instantiates a variable event $e \in \mathcal{VE}$ by replacing all variable names $n_v \in \mathcal{N}_\mathcal{V}$ with a concrete value v_v from its domain $d_v \in \mathcal{D}_\mathcal{V}$; let $e[n_v \mapsto v_v]$ denote such a replacement (note, more than one variable n_v may be specified).

$$\begin{aligned}
&\forall e \in \mathcal{VE},\ n_{v_1}, \ldots, n_{v_k} \in \mathcal{N}_\mathcal{V},\ d_{v_1}, \ldots, d_{v_k} \in \mathcal{D}_\mathcal{V} : \\
&VarIn(e) = \{n_{v_1} \mapsto d_{v_1}, \ldots, n_{v_k} \mapsto d_{v_k}\} \\
&\quad \Rightarrow Inst_\mathcal{E}(e) = \{e[n_{v_1} \mapsto v_{v_1}, \ldots, n_{v_k} \mapsto v_{v_k}] \mid \wedge_{i=1}^{k} v_{v_i} \in d_{v_i}\}
\end{aligned}$$

Equipped with the concept of variable events, Equation 2.3 shows the the extended event declaration from Equation 2.1 with $Var = \{v_1 \mapsto \{0, \ldots, 65535\}\}$

$$e''_\tau = (send, \{(obj, movie), (isTry, true), (destIp, 131.154.160.1), (destPort, v_1)\}) \qquad \text{(Equation 2.3)}$$

2.2. Usage Control Policies and Mechanisms

The semantic of UC policies are defined over timed traces (cf. Section 2.1). Data usage restrictions and requirements are expressed as UC policies at two levels: *specification-level* and the *implementation-level*. Specification-level policies express *what* the constraints are, whereas implementation-level policies specify *how* those constraints should be enforced. In a nutshell, the latter one serves as a configuration for a usage control mechanism.

2.2.1. Specification Level Policy

A *Specification Level Policy (SLP)* specifies constraints upon the future usage of data. Formally, it is based on the *Obligation Specification Language (OSL)*, a first-order-temporal language to express propositional (Ψ^+), temporal (Γ^+), cardinal (Ω^+), and spatial constraints on the usage of data. A UC policy $UcPolicy ::= \mathbb{P}(\mathcal{VE}) \times \mathbb{P}(\Phi^+)$ consists of a set of event declarations $\mathcal{VE}$ (as described in Section 2.1) and a set of obligation formulas Φ^+. Equation 2.4 shows the supported

operators of Φ^+ in *Extended Backus-Naur Form (EBNF)*.

$$
\begin{aligned}
\Theta^+ &::= \mathcal{VE} \mid \mathbb{N} \mid \mathit{String} \mid \ldots \\
\Phi^+ &::= (\Phi^+) \mid \Psi^+ \mid \Gamma^+ \mid \Omega^+ \mid \mathcal{A}(\mathcal{VE}) \mid \mathcal{D}(\mathcal{VE}) \mid \mathit{forall}\, \mathcal{N}_\mathcal{V}\ \mathit{in}\ \mathcal{D}_\mathcal{V} : \Phi^+ \\
\Psi^+ &::= \mathit{true} \mid \mathit{false} \mid \mathit{not}(\Phi^+) \mid \Phi^+\ \mathit{and}\ \Phi^+ \mid \Phi^+\ \mathit{or}\ \Phi^+ \mid \Phi^+\ \mathit{implies}\ \Phi^+ \mid \mathit{eval}(\Theta^+) \\
\Gamma^+ &::= \Phi^+\ \mathit{until}\ \Phi^+ \mid \mathit{always}(\Phi^+) \mid \Phi^+\ \mathit{after}\ \mathbb{N} \mid \Phi^+\ \mathit{within}\ \mathbb{N} \mid \Phi^+\ \mathit{during}\ \mathbb{N} \\
\Omega^+ &::= \mathit{repmax}(\mathbb{N}, \Psi^+) \mid \mathit{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi^+) \mid \mathit{repuntil}(\mathbb{N}, \Psi^+, \Phi^+)
\end{aligned}
\qquad \text{(Equation 2.4)}
$$

$\mathcal{A}(\cdot)$ and $\mathcal{D}(\cdot)$ specify if the passed event declaration $\mathcal{VE}$ is an actual or desired event. The $\mathit{forall}\,[\cdot] : \varphi$ operator specifies that formula $\varphi \in \Phi^+$ has to hold for all variables $\mathcal{N}_\mathcal{V}$, substituted by their values $\mathcal{V}_\mathcal{V}$. The $\mathit{eval}(\Theta^+)$ is used to specify conditions that are not covered by Ψ^+, Γ^+, and Ω^+, and hence, for instance enables the specification of physical time- and location-based constraints:

$$
\begin{aligned}
\forall n_1 \in \mathit{TIME} : \mathcal{A}((\mathit{print}, \{(\mathit{obj}, \mathit{secret.doc}), (\mathit{time}, n_1)\})) \\
\rightarrow \big(\neg \mathit{eval}(8 \leq n_1 \leq 18) \rightarrow e(\mathit{log}, \{(\mathit{obj}, \mathit{secret.doc}), (\mathit{time}, n_1)\})\big)
\end{aligned}
$$

As $\mathit{eval}(\Theta^+)$ takes as parameter any possible expression, its semantic is left unspecified and referred to as $[\![\mathit{eval}(\theta)]\!]_{\mathit{eval}}$ for $\theta \in \Theta^+$. Operators in Ψ^+ have standard semantics from propositional logic. The *until* operator has the weak-until semantic from *Linear Temporal Logic (LTL)* because *liveness*-properties are not considered to be relevant in the context of data usage protection [43]. φ_1 *until* φ_2 is true if $\varphi_1 \in \Phi^+$ is true until $\varphi_2 \in \Phi^+$ eventually becomes true, or $\varphi_1 \in \Phi^+$ is always *true*. $\mathit{always}(\varphi)$ specifies that $\varphi \in \Phi^+$ must be true at each moment in time in the future; this operator can be also expressed by φ *until false*. φ *after* n is true if $\varphi \in \Phi^+$ becomes true after $n \in \mathbb{N}$ timesteps. φ *within* n is true if $\varphi \in \Phi^+$ was at least once true within the last $n \in \mathbb{N}$ timesteps. Whereas, φ *during* n specifies that $\varphi \in \Phi^+$ has to be true at each timestep during the last $n \in \mathbb{N}$ timesteps. $\mathit{repmax}(n, \varphi)$ specifies that $\varphi \in \Psi^+$ must happen at most $n \in \mathbb{N}$ times in future. $\mathit{replim}(n, \mathit{low}, \mathit{up}, \varphi)$ is true if $\varphi \in \Psi^+$ is true between $\mathit{low} \in \mathbb{N}$ and $\mathit{up} \in \mathbb{N}$ times within the next $n \in \mathbb{N}$ timesteps. $\mathit{repuntil}(n, \varphi_1, \varphi_2)$ is true if $\varphi_1 \in \Psi^+$ is true at most $n \in \mathbb{N}$ times until $\varphi_2 \in \Phi^+$ eventually becomes true. As one may note, SLP are expressed in future-time. The rationale for that is: at the moment in time data usage obligations are specified, their actual intention is, to impose constraints on the future usage of data. Given Ψ^+, the semantic of events is defined by $\models_e \subseteq \mathcal{S} \times \Psi^+$, whereas $\mathcal{S} \subseteq \mathit{maxRefEv} \times \{\mathit{desired}, \mathit{actual}\}$ specifies all maximal refined actual and

desired events at system's run-time:

$$\forall e' \in maxRefEv,\ e \in \mathcal{VE}$$
$$(e', actual) \models_e \mathcal{A}(e) \Leftrightarrow \exists e'' \in \mathcal{E} :\ e'\ refinesEv\ e'' \wedge e'' \in Inst_{\mathcal{E}}(e) \wedge$$
$$(e', desired) \models_e \mathcal{D}(e) \Leftrightarrow \exists e'' \in \mathcal{E} :\ e'\ refinesEv\ e'' \wedge e'' \in Inst_{\mathcal{E}}(e)$$

For a trace $t \in \mathcal{T}$, timestep $i \in \mathbb{N}$, and a formula $\varphi \in \Phi^+$, the relation $(t,i) \models_f \varphi$ denotes that trace t satisfies φ at time i. The formal semantic of Φ^+ is defined by the relation $\models_f \subseteq (\mathcal{T} \times \mathbb{N}) \times \Phi^+$:

$$\forall t \in \mathcal{T}; i \in \mathbb{N}; \varphi \in \Phi^+ \bullet (t,i) \models_f \varphi \Leftrightarrow \varphi \neq false \wedge$$
$$\big(\exists e \in \mathcal{VE} \bullet (\varphi = \mathcal{A}(e) \vee \varphi = \mathcal{D}(e)) \wedge \exists e' \in t(i) : e' \models_e \varphi$$
$$\vee\ \exists \varphi_1, \varphi_2 \in \Phi^+ \bullet \varphi = \varphi_1\ implies\ \varphi_2 \wedge \big(\neg((t,i) \models_f \varphi_1) \vee (t,i) \models_f \varphi_2\big)$$
$$\vee\ \exists \theta \in \Theta^+ \bullet \varphi = eval(\theta) \wedge [\![\varphi]\!]_{eval} = true$$
$$\vee\ \exists n_v \in \mathcal{N}_{\mathcal{V}};\ d_v \in \mathcal{D}_{\mathcal{V}};\ \varphi_1 \in \Phi^+ \bullet \varphi = (forall\ n_v \in d_v : \varphi_1) \wedge \forall v_v \in d_v : (t,i) \models_f \varphi_1[n_v \mapsto v_v])$$
$$\vee\ \exists \varphi_1, \varphi_2 \in \Phi^+ \bullet \varphi = \varphi_1\ until\ \varphi_2 \wedge \big(\forall n \in \mathbb{N} : i \leq n \Rightarrow (t,n) \models_f \varphi_1$$
$$\vee\ (\exists u \in \mathbb{N} : i < u \wedge (t,u) \models_f \varphi_2 \wedge \forall n \in \mathbb{N} : i \leq n < u \Rightarrow (t,n) \models_f \varphi_1)\big)$$
$$\vee\ \exists n \in \mathbb{N};\ \varphi_1 \in \Phi^+ \bullet \varphi = \varphi_1\ after\ n \wedge (t, i+n) \models_f \varphi_1$$
$$\vee\ \exists n \in \mathbb{N}_1;\ l, r \in \mathbb{N};\ \varphi_1 \in \Psi^+ \bullet \varphi = replim(n,l,r,\varphi_1) \wedge l \leq \#\{j \in \mathbb{N}_1 | j \leq n \wedge (t, i+j) \models_f \varphi_1\} \leq r$$
$$\vee\ \exists n \in \mathbb{N};\ \varphi_1 \in \Psi^+;\ \varphi_2 \in \Phi^+ \bullet \varphi = repuntil(n, \varphi_1, \varphi_2)$$
$$\wedge\ \big((\exists u \in \mathbb{N}_1 : (t, i+u) \models_f \varphi_2 \wedge (\forall v \in \mathbb{N}_1 :\ v < u \Rightarrow \neg((t, i+v) \models_f \varphi_2))$$
$$\wedge\ (\#\{j \in \mathbb{N}_1 | j < n \wedge t(i+j) \models_e \varphi_1\}) \leq n)$$
$$\vee\ (\#\{j \in \mathbb{N}_1 | t(i+j) \models_e \varphi_1\}) \leq n\big)$$
$$\vee\ \exists n \in \mathbb{N};\ \varphi_1 \in \Psi^+ \bullet \varphi = repmax(n, \varphi_1) \wedge (t,i) \models_f repuntil(n, \varphi_1, false))$$
$$\vee\ \exists n \in \mathbb{N};\ \varphi_1 \in \Psi^+ \bullet \varphi = \varphi_1\ within\ n \wedge (t,i) \models_f replim(n, 1, n, \varphi_1)$$
$$\vee\ \exists n \in \mathbb{N};\ \varphi_1 \in \Psi^+ \bullet \varphi = \varphi_1\ during\ n \wedge (t,i) \models_f replim(n, n, n, \varphi_1)\big)$$

2.2.2. Implementation Level Policy

In contrast to SLPs, which describe a particular data usage requirement (i.e. what has to be enforced), an *Implementation Level Policy (ILP)* $\mathcal{P}_{\mathcal{I}}$ specifies how data usage obligations must be enforced within a target system. To do so, ILPs are structured as *Event-Condition-Action (ECA)* rules [1, 61, 70]. An ECA rule has the following semantic: **On** *Event* **If** *Condition* **Do** *Action*. This means, when a trigger-*Event* E emerges and the trigger-*Condition* C evaluates to true then execute *Action* A. Thereby, E specifies the desired event $\mathcal{D}(E)$ that triggers $\mathcal{P}_{\mathcal{I}}$'s verification; C

specifies a condition in $\varphi \in \Phi^-$, the past version of OSL (cf. Equation 2.5); and finally, A specifies the action (inhibition, modification, execution [90]) that has to be taken, if C verifies to true at the moment in time E occurs. Note, in the following $\mathcal{P}_\mathcal{I}.e$ denotes accessing the trigger event e of an ILP policy; $\mathcal{P}_\mathcal{I}.\varphi$ denotes accessing the condition part φ of an ILP policy; and finally, $\mathcal{P}_\mathcal{I}.a$ denotes accessing the action part $a \in A$ of an ILP policy.

As suggested, ILP uses a past dialect of OSL for describing conditions. The reason is, that at run-time data usage decisions have to be taken on system traces, i.e. on events that have already happened. Equation 2.5 shows the past version of OSL. In a nutshell, all operators from Φ^+ are translated into their corresponding past versions. Kumari et al. [60, 61] provide a structural approach and methodology to translate SLP into ILP policies, i.e. to translate Φ^+ into Φ^-.

$$
\begin{aligned}
\Theta^- &::= \mathcal{VE} \mid \mathbb{N} \mid \mathit{String} \mid \ldots \\
\Phi^- &::= (\Phi^-) \mid \Psi^- \mid \Gamma^- \mid \Omega^- \mid \mathcal{A}(\mathcal{VE}) \mid \mathcal{D}(\mathcal{VE}) \mid \mathit{forall}\, \mathcal{N}_\mathcal{V}\ \mathit{in}\ \mathcal{D}_\mathcal{V} : \Phi^- \\
\Psi^- &::= \mathit{true} \mid \mathit{false} \mid \mathit{not}(\Psi^-) \mid \Psi^-\ \mathit{and}\ \Psi^- \mid \Psi^-\ \mathit{or}\ \Psi^- \mid \Psi^-\ \mathit{implies}\ \Psi^- \mid \mathit{eval}(\Theta^-) \\
\Gamma^- &::= \Phi^-\ \mathit{since}\ \Phi^- \mid \boxdot\, \Phi^- \mid \Phi^-\ \mathit{before}\ \mathbb{N} \mid \Phi^-\ \mathit{within}\ \mathbb{N} \mid \Phi^-\ \mathit{during}\ \mathbb{N} \\
\Omega^- &::= \mathit{repmax}(\mathbb{N}, \Psi^-) \mid \mathit{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi^-) \mid \mathit{repsince}(\mathbb{N}, \Psi^-, \Phi^-)
\end{aligned}
\qquad \text{(Equation 2.5)}
$$

Almost all operators in Φ^- have a similar semantic as in Φ^+, except for some exceptions. φ_1 *since* φ_2 specifies that $\varphi_1 \in \Phi^-$ has to be true ever since $\varphi_2 \in \Phi^-$ happened. $\boxdot\, \varphi$ specifies that $\varphi \in \Phi^-$ is true in all timesteps before, including the current one. φ *before* n is true if $\varphi \in \Phi^-$ was true exactly in the $n^{th} \in \mathbb{N}$ timestep ago. *within* and *during* have similar semantics as their corresponding future version, and specify that an event $\varphi \in \Phi^-$ has to be true at least once in the past $n \in \mathbb{N}$ timesteps, or in each past timesteps respectively. $\mathit{repmax}(\varphi, n)$ specifies that $\varphi \in \Psi^-$ was true not more than $n \in \mathbb{N}$ times in the past. $\mathit{repsince}(n, \varphi_1, \varphi_2)$ specifies that $\varphi_1 \in \Psi^-$ must happen at most $n \in \mathbb{N}$ times ever since $\varphi_2 \in \Phi^-$ happened. $\mathit{replim}(n, l, m, \varphi_1)$ specifies that $\varphi_1 \in \Phi^-$ was true between $l \in \mathbb{N}$ and $m \in \mathbb{N}$ times within the last $n \in \mathbb{N}$ timesteps. The formal semantics of Φ^- is defined by $\models_{f^-} (\mathcal{T} \times \mathbb{N}) \times \Phi^-$:

$$\forall t \in \mathcal{T};\ i \in \mathbb{N};\ \varphi \in \Phi^{-} : (t,i) \models_{f^-} \varphi \iff (\varphi \neq \mathit{false}) \wedge$$
$$\big(\exists e \in \mathcal{VE} : (\varphi = \mathcal{A}(e) \vee \varphi = \mathcal{D}(e)) \wedge \exists e' \in t(i) : e' \models_e \varphi$$
$$\vee\ \exists \varphi_1 \in \Psi^{-} \bullet \varphi = \mathit{not}(\varphi_1) \wedge \neg((t,i) \models_{f^-} \varphi_1)$$
$$\vee\ \exists \varphi_1,\ \varphi_2 \in \Psi^{-} \bullet \varphi = \varphi_1\ \mathit{or}\ \varphi_2 \wedge ((t,i) \models_{f^-} \varphi_1 \vee (t,i) \models_{f^-} \varphi_2)$$
$$\vee\ \exists \varphi_1,\ \varphi_2 \in \Psi^{-} \bullet \varphi = \varphi_1\ \mathit{and}\ \varphi_2 \wedge (t,i) \models_{f^-} \mathit{not}(\mathit{not}(\varphi_1)\ \mathit{or}\ \mathit{not}(\varphi_2))$$
$$\vee\ \exists \varphi_1, \varphi_2 \in \Psi^{-} \bullet \varphi = \varphi_1\ \mathit{implies}\ \varphi_2 \wedge \big(\neg((t,i) \models_{f^-} \varphi_1 \vee (t,i) \models_{f^-} \varphi_2)\big)$$
$$\vee\ \exists \theta \in \Theta^{-} \bullet \varphi = \mathit{eval}(\theta) \wedge [\![\varphi]\!]_{\mathit{eval}} = \mathit{true}$$
$$\vee\ \exists n_v \in \mathcal{N}_{\mathcal{V}};\ d_v \in \mathcal{D}_{\mathcal{V}};\ \varphi_1 \in \Phi^{-} \bullet \varphi = (\mathit{forall}\ n_v\ \in\ d_v : \varphi_1) \wedge \forall v_v \in d_v\ :\ (t,i) \models_{f^-} \varphi_1[n_v \mapsto v_v])$$
$$\vee\ \exists \varphi_1,\ \varphi_2 \in \Phi^{-} \bullet \varphi = \varphi_1\ \mathit{since}\ \varphi_2 \wedge \big((\forall n \in \mathbb{N} : n \leq i\ \Rightarrow\ (t,n) \models_{f^-} \varphi_1)$$
$$\quad \vee\ (\exists u \in \mathbb{N} :\ u \leq i\ \wedge (t,u) \models_{f^-} \varphi_2 \wedge \forall v \in \mathbb{N} : u < v \leq i\ \Rightarrow\ (t,v) \models_{f^-} \varphi_1)\big)$$
$$\vee\ \exists n \in \mathbb{N},\ \varphi_1 \in \Phi^{-} \bullet\ \varphi = \varphi_1\ \mathit{before}\ n\ \wedge\ n \leq i\ \wedge\ (t, i-n) \models_{f^-} \varphi_1$$
$$\vee\ \exists n, l, r \in \mathbb{N},\ \varphi_1 \in \Psi^{-} \bullet \varphi = \mathit{replim}(n, l, r, \varphi_1)$$
$$\quad \wedge\ l \leq (\#\{j \in \mathbb{N} \mid j \leq \mathit{min}(n,i)\ \wedge\ t(i-j) \models_{f^-} \varphi_1\}) \leq r$$
$$\vee\ \exists n \in \mathbb{N},\ \varphi_1 \in \Psi^{-},\ \varphi_2 \in \Phi^{-},\ e \in \mathcal{E}\ \bullet \varphi = \mathit{repsince}(n, \varphi_1, \varphi_2)$$
$$\quad \wedge\ \big((\exists u \in \mathbb{N}_1 :\ u \leq i\ \wedge (t, i-u) \models_{f^-} \varphi_2 \wedge (\forall v \in \mathbb{N} :\ v < u \Rightarrow \neg((t, i-v) \models_{f^-} \varphi_2))$$
$$\qquad \wedge\ (\#\{j \in \mathbb{N} \mid j \leq u\ \wedge\ t(i-j) \models_{f^-} \varphi_1\} \leq n))$$
$$\quad \vee\ (\#\{j \in \mathbb{N} \mid j \leq i\ \wedge\ t(i-j) \models_{f^-} \varphi_1\} \leq n)\big)$$
$$\vee\ \exists n \in \mathbb{N},\ \varphi_1 \in \Psi^{-} \bullet \varphi = \mathit{repmax}(n, \varphi_1) \wedge (t,i) \models_{f^-} \mathit{repuntil}(n, \varphi_1, \mathit{false})$$
$$\vee\ \exists n \in \mathbb{N},\ \varphi_1 \in \Psi^{-} \bullet \varphi = \varphi_1\ \mathit{within}\ n\ \wedge (t,i) \models_{f^-} \mathit{replim}(n, 1, n, \varphi_1)$$
$$\vee\ \exists n \in \mathbb{N},\ \varphi_1 \in \Psi^{-} \bullet \varphi = \varphi_1\ \mathit{during}\ n\ \wedge (t,i) \models_{f^-} \mathit{replim}(n, n, n, \varphi_1)\big)$$

2.2.3. Usage Control Mechanisms

Usage Control mechanisms are installed at the data consumer's side and are means by which data providers control the usage of their data. Mechanisms are configured by a list of ILPs, that specify when a mechanism is applicable, i.e. what are the trigger events $ve \in \mathcal{VE}$ and conditions $\varphi \in \Phi^{-}$ that must match, and what are the compensating actions that need to be taken once a mechanism applies. Mechanisms may compensate the trigger event ve by

Inhibition $m_{inh} \subseteq \mathcal{VE} \times \Phi^{-}$: execution of ve is prohibited and not converted into an actual event, i.e. $\neg\mathcal{A}(ve)$.

Execution $m_{exc} \subseteq \mathcal{VE} \times \Phi^{-} \times \mathbb{P}(\mathcal{VE})$: *ve* is executed if no other ILP prohibits it. Furthermore, a set of n events may be executed $\wedge_{i=1}^{n} \mathcal{D}(ve_i)$ in addition. Note, those additional events are desired as their execution may be subject to other ILPs.

Modification $m_{mod} \subseteq \mathcal{VE} \times \Phi^{-} \times \mathbb{P}(\mathcal{VE})$: *ve* is executed in a modified form, e.g. some of the event's parameter values are changed. In reverse this means, the execution of *ve* is prohibited and a modified version of *ve*' is executed. Therefore, a modifier is modeled as a composition of inhibitors and executors.

Let $fire \subseteq \mathcal{VE} \times \Phi^{-}$ specify the situation when an ILP policy $\mathcal{P}_{\mathcal{I}}$ is supposed to fire

$$fire(ve, \mathcal{P}_{\mathcal{I}}.\varphi) \Leftrightarrow \big(\mathcal{D}(ve) \wedge (\mathcal{A}(ve) \rightarrow \mathcal{P}_{\mathcal{I}}.\varphi)\big) \qquad \text{(Equation 2.6)}$$

Then the semantics of m_{exc}, m_{mod}, m_{inh} are defined as

$$m_{exc}(ve, \varphi, Exc) \Leftrightarrow \forall VarIn(ve) : fire(ve, \varphi) \rightarrow \wedge_{x_i \in Exc} \mathcal{D}(x_i)$$
$$m_{mod}(ve, \varphi, Mod) \Leftrightarrow \forall VarIn(ve) : fire(ve, \varphi) \rightarrow (\neg \mathcal{A}(ve) \wedge m_{exc}(ve, \varphi, Mod))$$
$$m_{inh}(ve, \varphi) \Leftrightarrow \forall VarIn(ve) : fire(ve, \varphi) \rightarrow \neg \mathcal{A}(ve)$$

For a set of n_1 executing, n_2 modifying, and n_3 inhibiting ILPs in a computing system, their union computes to

$$M \longleftrightarrow \bigwedge_{i=1}^{n_1} m_{exc}(ve_i^{exc}, \varphi_i^{exc}, Exc_i) \wedge \bigwedge_{i=1}^{n_2} m_{mod}(ve_i^{mod}, \varphi_i^{mod}, Mod_i) \wedge \bigwedge_{i=1}^{n_3} m_{inh}(ve_i^{inh}, \varphi_i^{inh})$$

Once, the desired event $\mathcal{D}(e)$ is allowed to be executed it is converted into an actual event $\mathcal{A}(e)$, as long as, its execution does not trigger any modifying or inhibiting ILPs. An ILP $\mathcal{P}_{\mathcal{I}}$ triggers if e refines $\mathcal{P}_{\mathcal{I}}$'s trigger event $\mathcal{P}_{\mathcal{I}}.e$ and its condition $\mathcal{P}_{\mathcal{I}}.\varphi$ evaluates to true. In summary, a maximally refined desired event either transforms into an actual event, or a modifying or inhibiting ILP gets triggered by e, and hence, would prevent e's execution:

$$M_{default} \longleftrightarrow \bigwedge_{e \in maxRefEv} \mathcal{D}(e) \rightarrow \Big(\mathcal{A}(e) \vee \bigvee_{\substack{(ve,\varphi) : M \rightarrow m_{inh}(ve,\varphi) \\ \vee\ M \rightarrow m_{mod}(ve,\varphi,Mod)}} \exists VarIn(ve) : e\ refinesEv\ ve\ \wedge \varphi\Big)$$

Finally, $M_{complete}$ defines the semantics of all combined ILPs in a system:

$$M_{complete} \longleftrightarrow M \wedge M_{default} \qquad \text{(Equation 2.7)}$$

It may be the case that the combination of different ILPs leads to inconsistent data usage specifications. For instance, a modifying m_{mod_a} ILP transforms an event a into b and another

ILP m_{mod_b} transforms an event b into a. This thesis does not address the problem of circular dependencies, and formal inconsistencies in particular, between ILPs, and therefore refer to the work in [94], where model checking techniques are used to compare and to identify discrepancies among ILP policies.

Put it simply, once an ILP is activated, it serves as a configuration for a mechanism installed at the data consumers' side. This thesis assumes that once a policy is violated it remains violated forever and the violation is reported at each moment in time. However, depending on the context another strategy would be more applicable. For instance, to report the violation the first time and to deactivate the ILP afterwards, and hence, to reset the mechanism respectively. Because a general rule does not exist that determines which strategy is the best, a policy themselves must specify what must happen once a violation occurs. Technically, mechanisms can be implemented using run-time monitoring and verification techniques [40, 41] or *Complex Event Processing (CEP)* technologies [15, 55].

2.3. Generic Data Flow Model

As described in Section 2.2.2, data usage control policies and data obligations are mostly specified and expressed in terms of events. However, policy issuers intend to impose data usage obligations on their actual *data* rather than on single events. Moreover, as data might evolve at system's run-time from one copy to another, policy issuers desire the protection of all copies (henceforth also termed *representations*) of their data, and not only the first initial data item. For instance, a picture F at the filesystem might be modified or converted from one format F_A into another F_B. These are different representations of the same content, and hence, a policy like *"Do not disseminate my picture" (P_2)* must affect F_A and F_B. It is noteworthy, that the same content may also exist on different *abstraction layers*. Related to the previous example, picture F downloaded from the Internet may exist at least in form of an HTML DOM-object, a file within the browser cache, and as network packets at the network layer.

As a consequence, enforcement of data obligations requires knowledge of the different representations. To do so, the usage control model is extended with data flow tracking features that enable to track different representations, and hence, to track how data evolves and disseminates in a system [39]. Separating data from its concrete representations enables the expression and enforcement of data usage requirements in terms of data, like P_2, rather than layer-specific representations, like P_1. The enforcement mechanism would then need to take into account every representation of a protected data item. This section describes a generic data flow model that enables to model and to describe how data flows among different representations within a

system. Chapter 3 instantiates this model for HDFT++, and hence, forms the foundation of the proposed hybrid data flow tracking approach.

The generic data flow model [39, 95] is a state transition system and provides formal and operational concepts to model data flows in a system. A *state* σ captures which data is stored in which representation, whereas a *state transition* models how data flows from one representation to another. Formally, the data flow tracking model is described by the tuple

$$(\mathcal{D}, \mathcal{C}, \mathcal{E}, \mathcal{I}, \mathcal{F}, \sigma, \sigma_i, \mathcal{R})$$

Set $\mathcal{D}$ represents all to be protected *data* items in a system. Whereas, set $\mathcal{C}$ denotes all possible representations (henceforth also termed *containers*) in a system where potentially data may be stored, e.g. variables or files. All relevant system events, e.g. method invocations or system calls, that potentially cause a flow of data, and hence change the system state, are denoted by set $\mathcal{E}$; notably, at system's run-time only actual events cause data flows. Principals $\mathcal{I} \subseteq \mathcal{C}$ are all active entities in a system, e.g. a process or a thread that can initiate events. $\mathcal{F}$ is the set of all naming identifiers that are used to identify containers in a system, e.g. process- or object-id. Finally, all possible system states are specified by

$$\sigma := s \times l \times f$$

A state is compound by three mappings:

- A *storage function* $s : (\mathcal{C} \rightarrow 2^{\mathcal{D}})$ that maps a container to a set of data items
- An *alias function* $l : (\mathcal{C} \rightarrow 2^{\mathcal{C}})$ that models an alias relation between different containers. An alias relation captures the fact that some containers get implicitly updated whenever other containers do. If $c_2 \in l(c_1)$ for $c_1, c_2 \in \mathcal{C}$, then any data written to c_1 is immediately propagated to c_2.
- A *naming function* $f : (\mathcal{I} \times \mathcal{F} \rightarrow \mathcal{C})$ that maps uniquely a tuple of principle $\mathcal{I}$ and naming identifier $\mathcal{F}$ to a container $\mathcal{C}$

$\sigma_i \in \sigma$ denotes a system's initial state. The set of all possible state transitions in a system is defined by relation $\mathcal{R} \subset \sigma \times \mathcal{E} \rightarrow \sigma$. For a given moment in time $n \in \mathbb{N}$, $states : (Trace \times \mathbb{N}) \rightarrow \sigma$ computes the dissemination of data after executing trace $t \in \mathcal{T}$ until timestep $n-1$.

$$states(t, n) \begin{cases} \sigma_i & if\, n = 0 \\ \mathcal{R}(states(t, n-1), t(n-1)) & if\, n > 0 \end{cases}$$

Additional notations for modelling state transitions are defined. Let $x \in \{s, l, f\}$ denote the access of a mapping function in state $\sigma \in \sigma$. For any mapping $\sigma.x : \mathcal{J} \rightarrow \mathcal{K}$ and a variable

$v \in V \subseteq \mathcal{J}$, define $\sigma.x' = \sigma.x[v \leftarrow expr]_{v \in V}$ with $\sigma.x' : \mathcal{J} \rightarrow \mathcal{K}$ such that $\sigma.x'(y) = expr$ if $y \in V$ and $\sigma.x'(y) = \sigma.x(y)$ otherwise. Multiple state changes for disjoint sets $\bigcap_{i \leq n \in \mathbb{N}} V_i = \emptyset$ are defined by a function composition $\circ$; the replacement is done atomically and simultaneously.

$$\sigma.x[v_1 \leftarrow expr_{v_1}; \ldots; x_n \leftarrow expr_{v_n}]_{v_1 \in V_1, \ldots, v_n \in V_n} =$$
$$\sigma.x[v_n \leftarrow expr_{v_n}]_{v_n \in V_n} \circ \ldots \circ \sigma.x[v_1 \leftarrow expr_{v_1}]_{v_1 \in V_1}$$

The main driver to combine data usage control concepts with data flow tracking is, to specify data usage control policies in a data-centric manner, i.e. policy issuers specify data obligations in terms of data rather than layer-specific containers/representations. To complement that approach, the concept of event refinement from Section 2.1 needs to be redefined and adapted in the presence of states. The rationale is, that at run-time system events $e \subseteq maxRefEv$ refer to concrete containers, while usage control policies might be specified in terms of data or containers. Because of that, the decision-taking process must also take into account the system's current state $\sigma \in \Sigma$, as a state provides information about which data is stored within a container. Let *getClass* be defined as in Section 2.1, then the system must satisfy the conditions:

$$\forall e \in \mathcal{E} : \ getClass(e) = dataUsage \Leftrightarrow \exists par \in \mathcal{D} : (obj \mapsto par) \in e.p$$
$$\forall e \in \mathcal{E} : \ getClass(e) = containerUsage \Leftrightarrow \exists par \in \mathcal{C} : (obj \mapsto par) \in e.p$$

In this respect, an event e_1 refines an event e_2 in the presence of states,

- if both has the same class $getClass(e_1) == getClass(e_2)$ and e_1 $refinesEv_\Sigma$ e_2, or
- if both has the same name, and $getClass(e_1) == containerUsage$, and there exist a data item d stored in a container c such that $(obj \mapsto d) \in e_1.p$ and $(obj \mapsto c) \in e_2.p$, and all parameters in e_1 have the same value as in e_2 except for parameter *obj*

Formally $refinesEv_\Sigma \subseteq (\mathcal{E} \times \Sigma) \times \mathcal{E}$ is defined

$$\forall e_1, e_2 \in \mathcal{E}, \forall \sigma \in \Sigma : (e_1, \sigma) \ refinesEv_\Sigma \ e_2 \ \Leftrightarrow$$
$$(getClass(e_1) == getClass(e_2) \ \wedge \ e_1 \ refinesEv \ e_2 \ )$$
$$\vee \ \big(getClass(e_1) == containerUsage \ \wedge \ getClass(e_2) == dataUsage$$
$$\wedge \ \ e_1.n == e_2.n \ \wedge \exists d \in \mathcal{D}, c \in \mathcal{C} : d \in \sigma.s(c)$$
$$\wedge \ (obj \mapsto c) \in e_1.p \ \wedge (obj \mapsto d) \in e_2.p \ \wedge (e_2.p \setminus \{obj \mapsto d\}) \subseteq (e_1.p \setminus \{obj \mapsto c\})\big)$$

To specify data obligations in the presence of states, the OSL language Φ^+ is extended with state-based operators Π^+, that enables to specify conditions on the system's data flow state σ.

$$\Pi ::= isNotIn(\mathcal{D}, \mathbb{P}(\mathcal{C})) \mid isMaxIn(\mathcal{D}, \mathbb{N}, \mathbb{P}(\mathcal{C})) \mid isCombinedWith(\mathcal{D}, \mathcal{D}, \mathbb{P}(\mathcal{C}))$$
$$\Phi^+ ::= \Phi^+ \mid \Pi$$

Operator $isNotIn(d, \mathbb{C})$ is true if $d \in \mathcal{D}$ is not in any of the specified containers $\mathbb{C} \subseteq \mathcal{C}$, $isMaxIn(d, n, \mathbb{C})$ is true if data $d \in \mathcal{D}$ is contained in at most $n \in \mathbb{N}$ containers in $\mathbb{C} \subseteq \mathcal{C}$, $isCombinedWith(d_1, d_2, \mathbb{C})$ evaluates to true if there exists at least one container in $\mathbb{C} \subseteq \mathcal{C}$ that contains both $d_1, d_2 \in \mathcal{D}$. Formally, the semantics of Π are specified by $\models_S \subseteq (Trace \times \mathbb{N}) \times \Pi$

$$\forall t \in Trace; n \in \mathbb{N}; \varphi \in \Pi; \sigma \in \Sigma : (t, n) \models_S \varphi \Leftrightarrow \sigma = states(t, n) \wedge$$
$$\big(\exists d \in \mathcal{D}, \mathcal{C}_s \subseteq \mathcal{C} \bullet \varphi = isNotIn(d, \mathcal{C}_s) \wedge \forall c' \in \mathcal{C} : \; d \in \sigma.s(c') \Rightarrow c' \notin \mathcal{C}_s$$
$$\vee \; \exists d_1, d_2 \in \mathcal{D}, \; \mathbb{C} \subseteq \mathcal{C} \bullet \varphi = isCombinedWith(d_1, d_2, \mathbb{C}) \wedge \exists c' \in \mathbb{C} : \; d_1 \in \sigma.s(c') \wedge \; d_2 \in \sigma.s(c')$$
$$\vee \; \exists d \in \mathcal{D}, m \in \mathbb{N}, \mathbb{C} \subseteq \mathcal{C} \bullet \varphi = isMaxIn(d, m, \mathbb{C}) \wedge \; | \#\{c \in \mathbb{C} \mid d \in \sigma.s(c)\} | \leq m \big)$$

Dually, $\Phi^- ::= \Phi^- \mid \Pi$ extends the past version of OSL Φ^- with the state-based operators Π.

2.4. Usage Control Infrastructure

As illustrated in Figure 1.1, a dedicated *Usage Control Infrastructure (UCI)* is required on the data consumer's side and must perform the following tasks to enforce usage control policies:

- Manage and deploy ILP usage control policies
- Monitor events within the system that are relevant for data-usage and verify if their execution does not violate any deployed ILP usage control policies
- Enforce usage control decisions, once they have been made, by inhibiting, modifying, or delaying system events
- Monitor events that lead to a flow of data, and hence, might change the system's data flow state $\sigma \in \Sigma$
- Record and maintain a system's data flow state $\sigma \in \Sigma$ at each moment in time $n \in \mathbb{N}$

To address these requirements, the foundations on usage control, as described in Section 2.1 - Section 2.3, are operationalized in a layer-agnostic generic architecture (cf. Figure 2.1): *Policy Enforcement Point (PEP)*, *Policy Decision Point (PDP)*, *Policy Information Point (PIP)*, and *Policy Management Point (PMP)*. As one may recognize, this architecture design follows the *separation of concerns* principle [65] and is leaned on the established XACML [79] and COPS [22, 57] reference architecture. However, this thesis underlying UCI infrastructure components have slightly different semantics and assigned functions than those reference architectures. In a nutshell, the UCI exhibits three components that build and form the carthorse of this thesis

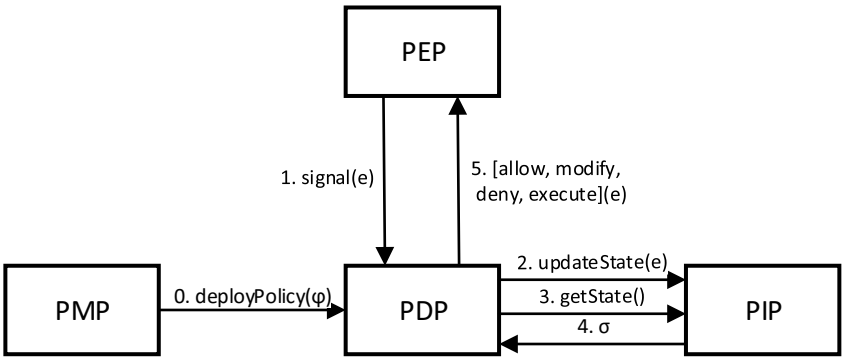


Figure 2.1.: Usage Control Infrastructure

underlying infrastructure: PEP, PDP, PIP. The PEP is in charge to extract and to notify context information in the form of system events $e \in \mathcal{E}$ to the PDP (cf. step 1 in Figure 2.1). Based on the signaled system events, the PDP decides if their executions do not violate deployed policies and replies its decision back to the corresponding PEP for enforcement (cf. step 4 in Figure 2.1). In case data usage events are signaled, the PDP also queries the PIP (cf. step 2 and 3 in Figure 2.1) to take into account the current data flow state $\sigma \in \Sigma$ of the system. Note, this architecture provides a logical view on its components and their interplay, and by design is able to support different abstraction layers, including the Java abstraction layer. Their technical implementation, however, may differ (e.g. using simple taint analysis to implement the PIP).

The **Policy Enforcement Point** implements the right-hand side of Equation 2.7. In the literature PEPs have been implemented for different abstraction layers such as Android [26, 97], Java [29, 30], JavaScript [85], ChromiumOS [122], MS Office [108] and Windows [124], Mozilla Thunderbird [69] and Firefox [62], MySQL [68], OpenBSD [39], OpenNebula [66], and X11[95]. All those implementations have in common, that they are tailored for one specific abstraction layer to monitor and intercept layer-specific system events properly. For instance, a Java-agnostic PEP [29] may intercept and interpret Java method calls as events, whereas PEPs at the operating system layer [39, 124] interpret system calls as events. However, because the PEPs are unaware of any deployed policies and do not have semantics about monitored events, the PEP must signal each intercepted event to the PDP to get its execution approved. Depending on the PDP's taken decision the PEP allows, denies, or modifies the execution of the currently intercepted event. Note, each generated and signaled event $e \in \mathcal{E}$ is maximally refined, i.e. $e \in maxRefEv$. Beyond that, the PEP also intercepts and signals all data flow relevant events to the PIP to notify and to keep track of how data flows.

The **Policy Decision Point** implements the function in Equation 2.6 and continuously evaluates if the execution of a particular signaled system event could potentially violate a deployed

policy. To do so, the PDP is configured with a set of ILP policies and works from a high-level perspective as follows:

1. First, an intercepted event $e \in \mathit{maxRefEv}$ is matched against deployed policies $\mathcal{P}_\mathcal{I}$. An event e matches a policy event $\mathcal{P}_\mathcal{I}.e$ if e *refinesEv* $\mathcal{P}_\mathcal{I}.e$ holds.
2. Second, the PDP evaluates the policy's condition part $\mathcal{P}_\mathcal{I}.\varphi$. In case the condition includes state-based operators $\pi \in \Pi$, the PDP also queries the PIP to get additional information about the current data flow state of the system.
3. Finally, if the condition part of a policy evaluates to true, the specified action is applied.

Moreover, for temporal constraints where it is not trivial and probably also not practical to determine the exact moment in time, the PDP supports the concept of a *timestep* interval. A timestep interval may have different sizes and allows to cluster a set of events that occur within a certain timeslot (e.g. 2 seconds or 1 hour). This concept is quite helpful to estimate approximately when a refining event actually happened. For instance, $\mathcal{P}_\mathcal{I}$ (cf. Equation 2.8) specifies that a manager must be notified ($\mathcal{P}_\mathcal{I}.a$) in case a clerk process a service request ($\mathcal{P}_\mathcal{I}.e$) not within the prescribed time of 10 days ($\mathcal{P}_\mathcal{I}.\varphi$).

$$\mathcal{P}_\mathcal{I} = \begin{cases} \mathcal{P}_\mathcal{I}.e = & (\mathit{processReq}, \{(\mathit{obj}, d), (\mathit{role}, \mathit{clerk}), (\mathit{type}, \mathit{service})\}) \\ \mathcal{P}_\mathcal{I}.\varphi = & \mathit{not}((\mathit{reqService}, \{(\mathit{obj}, d)\})\ \mathit{within}\ 10)\ \mathit{and} \\ & \mathit{repsince}(1, (\mathit{reqService}, \{(\mathit{obj}, d)\}), \mathit{false}) \\ \mathcal{P}_\mathcal{I}.a = & (\mathit{notify}, \{(\mathit{obj}, d), (\mathit{role}, \mathit{manager})\}) \end{cases} \quad \text{(Equation 2.8)}$$

Whenever $\mathcal{P}_\mathcal{I}$'s evaluation is triggered by $\mathcal{P}_\mathcal{I}.e$, it is quite unlikely that exactly at the same point in time the condition event e_1 *refinesEv* $(\mathit{reqService}, \{(\mathit{obj}, d)\})$ happened exactly 10 days (864000 seconds) before. It is more likely, that an event e_1 happened approximately 10 days ± 12 hours ago. Related to Equation 2.8, this means, that once $\mathcal{P}_\mathcal{I}$ gets triggered, a PDP instance would evaluate the condition part for a timeframe between 9.5 days ± 12 hours.

Once, a policy $\mathcal{P}_\mathcal{I}$ gets deployed on the PDP, the PDP instantiates a new evaluation mechanism $\mathcal{M}_\mathcal{I}$ and configures its ECA parts according to the information provided by $\mathcal{P}_\mathcal{I}$, i.e. $\mathcal{M}_\mathcal{I}.e = \mathcal{P}_\mathcal{I}.e$, $\mathcal{M}_\mathcal{I}.\varphi = \mathcal{M}_\mathcal{I}.\varphi$, and $\mathcal{M}_\mathcal{I}.a = \mathcal{P}_\mathcal{I}.a$. While ILPs specify how obligations must be fulfilled, mechanisms $\mathcal{M}_\mathcal{I}$ are concrete instances that are able to verify if such obligations actually hold at run-time. As a policy condition $\mathcal{P}_\mathcal{I}.\varphi$ might exhibit complex and nested structures, the PDP represents the condition part $\mathcal{M}_\mathcal{I}.\varphi$ as an *expression tree*. Figure 2.2 shows the expression tree for condition $\mathcal{P}_\mathcal{I}.\varphi$ from Equation 2.8. Thereby, an expression tree's leave nodes represent either events $\mathcal{E}$, state-based operators Π, or boolean constants *true* and *false*. Whereas, its

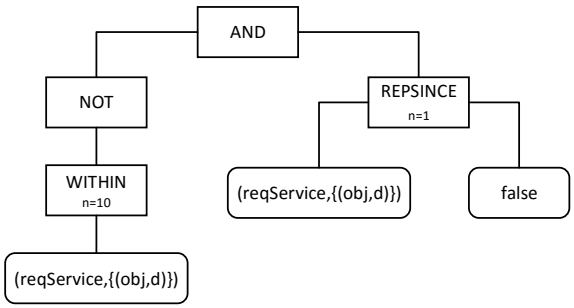


Figure 2.2.: Sample expression tree that shows the condition part $\mathcal{P}_{\mathcal{I}}.\varphi$ from Equation 2.8.

internal nodes represent all the other operators, including temporal, propositional, and cardinal operators. Moreover, all nodes of an expression tree are stateful and store how often their state changes during the current (and in case of temporal operators, in previous) timesteps.

Figure 2.3 illustrates the interplay between the UCI components. Once a policy gets instantiated and deployed on the PDP (cf. steps 1-3), the PEP continuously monitors and signals intercepted events $e \in \mathcal{E}$ to the PDP (cf. steps 4 and 5). To evaluate a received event e on an up-to-date data flow state, the PDP first redirects e to the PIP, which updates its data flow state according to e's semantic (cf. step 6 and 7). If e is of type *dataUsage*, i.e. $getClass(e) =$ *dataUsage*, the PDP updates the internal state of all leaf nodes of an expression tree. For leave nodes of type state-based operator, the PDP delegates the update process to the PIP. Boolean typed leave nodes *true* and *false* are invariant and do not need an update. The internal state of event-typed leave nodes $e \in \mathcal{E}$ counts the number of refining events that happened within the current timestep, i.e. (e', σ) $refinesEv_{\Sigma}$ $e =$*true*. The update process for leave nodes happens for all instantiated mechanism, independent of their trigger event. That way, the state of all leave nodes of all expression trees are consistent with the happened event.

Once all leave nodes of all expression trees are updated, a policy is triggered for two reasons: either a signaled event occurs or a timestep has elapsed and a policy mechanism gets triggered. In both cases the entire condition $\mathcal{M}_{\mathcal{I}}.\varphi$ needs to be evaluated, starting from the root node (which represents the outermost operator of a $\mathcal{M}_{\mathcal{I}}.\varphi$ condition) to its leave nodes. Let $\texttt{assess}(\mathcal{M}_{\mathcal{I}}.\varphi, EoT, n)$ denote this recursive evaluation process for a timestep $n \in \mathbb{N}$ and a root condition $\mathcal{M}_{\mathcal{I}}.\varphi$. $EoT =$ *true* holds in case the evaluation process gets triggered by an elapsed timeframe. Formally, $\texttt{assess}(\mathcal{M}_{\mathcal{I}}.\varphi, EoT, n)$ is reflected by $(t,n) \models_{f^-} \mathcal{M}_{\mathcal{I}}.\varphi$ and $(t,n) \models_{S} \mathcal{M}_{\mathcal{I}}.\varphi$ with $t \in \mathcal{T}$, but technically, however, this evaluation process might be more or less complex depending on the internal node's operator type. Mechanism's actions $\mathcal{M}_{\mathcal{I}}.a$ are applied in case the evaluation process results in $\texttt{assess}(\mathcal{M}_{\mathcal{I}}.\varphi, EoT, n)$=`true`. Table 2.1 provides a technical view in pseudo-code on how $\texttt{assess}(\mathcal{M}_{\mathcal{I}}.\varphi, EoT, n)$ works for the different

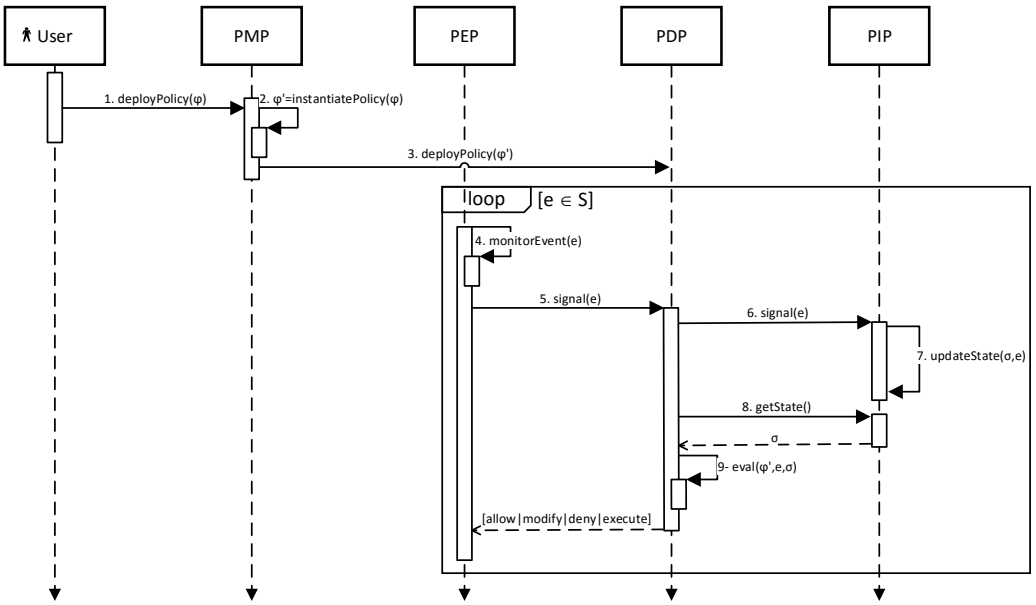


Figure 2.3.: Usage Control Infrastructure

operators in Φ^{-}.

The **Policy Information Point** keeps track of how data flows within a data processing system. To do so, it implements the generic data flow model (cf. Section 2.3) and provides the possibility to instantiate the state transition function R for a concrete system. Based on the intercepted system events, the PIP updates its internal data flow state $\sigma \in \Sigma$ according to the system events semantic. Thus, the PIP knows at each moment in time which data $d \in \mathcal{D}$ is stored in which container $c \in \mathcal{C}$ in a system. Chapter 3 provides a detailed view on the implementation and instantiation of that model for the present work.

The **Policy Management Point** is responsible for the management of usage control policies. This includes deployment and revocation of policies, as well as, editing and translating policies between different formats [59].

Table 2.1.: Technical evaluation of operators $\mathcal{P}_{\mathcal{I}}.\varphi \in \Phi^{-}$ by the PDP [52]

$\mathcal{P}_{\mathcal{I}}.\varphi \in \Phi^{-}$	Description
not(φ) `return !assess(`φ`,`*EoT*`,`n`)`	Returns the negated result of the recursive evaluation of $\varphi \in \Psi^{-}$ at a timestep $n \in \mathbb{N}$.
φ_1 *and* φ_2 `return assess(`φ_1`,`*EoT*`,`n`) && assess(`φ_2`,`*EoT*`,`n`)`	Evaluates $\varphi_1 \in \Psi^{-}$ and $\varphi_2 \in \Psi^{-}$ recursively for timestep $n \in \mathbb{N}$, and returns the AND conjunction of both.
φ_1 *or* φ_2 `return assess(`φ_1`,`*EoT*`,`n`) || assess(`φ_2`,`*EoT*`,`n`)`	Evaluates $\varphi_1 \in \Psi^{-}$ and $\varphi_2 \in \Psi^{-}$ recursively for timestep $n \in \mathbb{N}$, and returns the OR conjunction of both.
φ_1 *implies* φ_2 `return` *not*(φ_1) `|| assess(`φ_2`,`*EoT*`,`n`)`	Evaluates $\varphi_1 \in \Psi^{-}$ and $\varphi_2 \in \Psi^{-}$ separatetly and returns true if either φ_1 is false or φ_2 is true.

$\boxdot \varphi$
`if (!s.always)`
`  return false`
`s.always &= assess(`φ`,`*EoT*`,`n`)`
`return s.always`
	Returns true if $\varphi \in \Phi^{-}$ was true at each moment in time in the past. To do so, state variable always stores the evaluation result of φ for the current timestep n.

replim(n, l, m, φ)
`boolean statePhi =`
`assess(`φ`,`*EoT*`,`n`)`
`if (s.prev[s.next]){`
`  s.counter -= 1`
`}`
`if (statePhi){`
`  s.counter += 1`
`}`
`s.prev[s.next] = statePhi`
`result = s.counter >= l`
`result &= s.counter <= m`
`return result`
	An internal buffer s.prev of size n stores if $\varphi \in \Psi^{-}$ was true at timestep s.next, which is initialized with zero. An internal state counter s.counter stores if φ was true at timestep next. Finally, true is returned if φ happened between $l \in \mathbb{N}$ and $m \in \mathbb{N}$ times within the last $n \in \mathbb{N}$ timesteps.

φ_1 *since* φ_2

```
boolean statePhi1 =
assess(φ1,EoT,n)
boolean statePhi2 =
assess(φ2,EoT,n)

s.alwaysPhi1 &= statePhi1
if (s.alwaysPhi1)
  return true
if (s.statePhi2)
  s.alwaysPhi1SincePhi2 =
  true
else
  s.alwaysPhi1SincePhi2 &=
  statePhi1

return s.alwaysPhi1SincePhi2
```

Variable `s.statePhi1` and `s.statePhi2` temporarily store the evaluation result of φ_1 and φ_2 respectively. `s.alwaysPhi1SincePhi2` and `s.alwaysPhi1` are internal boolean [s]tate variables of since-node and store if φ_1 has been true since ever φ_2 happened, or φ_1 was true since ever. Those state variables are initialized to `s.alwaysPhi1=true` and `s.alwaysPhi1SincePhi2=false`.

φ *during* n

```
boolean statePhi =
assess(φ,EoT,n)
if (statePhi){
  s.counter -= 1
}
else{
  s.counter = n
}
return (s.counter==0)
```

Returns true if $\varphi \in \Phi^{-}$ happened in all the last $n \in \mathbb{N}$ timesteps. To do so, `s.counter` is set initially to `n`. If φ is true, every evaluation cyclus decrements `s.counter` by one for the respective timestep. The total evaluation results in true if `s.counter` reaches zero.

φ *within* n

```
boolean statePhi =
assess(φ,EoT,n)
if(statePhi){
  s.counter = n
} else {
  s.counter -= 1
}
return (s.counter > 0)
```

Returns true if $\varphi \in \Phi^{-}$ happened at least once within the last $n \in \mathbb{N}$ timesteps. To do so, an internal state counter `s.count` is set to `n`, once φ verifies to true. With every evaluation cycle `s.count` is reduced by one. True is returned in case `s.count` is bigger zero.

φ before n

```
boolean result = s.prev[n %
i]
if (EoT)
  s.prev[n % i] =
  assess(φ,EoT,n)
return result
```

State variable `s.prev` is a boolean array of size `i`, and stores the evaluation results of φ for the last `i` timesteps. All entries in `prev` are initialized with `false`. In case the end of timeframe is reached, i.e. $EoT = true$, the evaluation result for the last `n` timestep is updated.

repmax(φ, n)

```
boolean statePhi =
assess(φ,EoT,n)
if (statePhi)
  s.counter += 1;
return (s.counter < n)
```

On each successful evaluation of φ an internal state counter s.counter is incremented by one. True is returned, as long as the upper limit n is not reached.

repsince$(n, \varphi_1, \varphi_2)$

```
boolean statePhi1 =
assess(φ1,EoT,n)
boolean statePhi2 =
assess(φ2,EoT,n)
if (!(s.counter < n))
  return false
s.alwaysPhi1 &= statePhi1
if (s.alwaysPhi1){
  s.counter += 1
  return (s.counter < n)
}
if (s.statePhi2){
  s.alwaysPhi1SincePhi2 =
  true
  s.counter = 0
}
else{
  s.alwaysPhi1SincePhi2 &=
  statePhi1
  if(s.alwaysPhi1SincePhi2)
    s.counter += 1
}
return (s.counter < n)
```

An internal state variable `s.counter` stores the number of times $\varphi_1 \in \Psi^{\frown}$ was true, either since ever or since φ_2 happened. Once, `s.counter` exceeds the upper limit `n`, false is returned. State variables are initialized as follows: `s.alwaysPhi1 = true`, `s.alwaysPhi1SincePhi2 = false`, and `s.counter = 0`.

2.5. Policies by Examples

Based on the previously described foundations in DUC, Table 2.2 illustrates the formal representation of our example policies Policy P_1 – Policy P_4, from our running scenario (cf. Section 1.2), as ECA rules. To enforce Policy P_1 we assume, for the sake of simplicity, that dissemination of data happens only through network communications. Because of that, Policy P_1 inhibits a *send* event if the network container $\mathcal{C}_{net}$ contains protected *data* from the file "secret.txt". Under the assumption of the variable mapping function $Var = \{r \mapsto \{EU, US, RUS, CHN, JPN, \ldots\}\}$, Policy P_2 specifies to inhibit any attempts to store *data* on non-EU servers, i.e. condition $not(eval(r == "EU"))$ is satisfied, whereas Policy P_3 permits the storage of encrypted data on US servers, i.e. condition $eval(r == "USA")$ holds; note, Policy P_3 uses a helper function *enc* to encrypt data. In both policies, the event parameter *receiver* specifies the destination server on which data is stored. In contrast to the previous policies, Policy P_4 is evaluated on every trigger event $< any >$ inside the target system and specifies to delete data after 30 days of reception.

Table 2.2.: Policy P_1 - Policy P_4 from our running scenario in Section 1.2 as formal ECA-rules.

Policy P_1	*"Do not disseminate my data from secret.txt"*
$\mathcal{P}_{\mathcal{I}}.e$	$(send, \emptyset)$
$\mathcal{P}_{\mathcal{I}}.\varphi$	$not(isNotIn(data, \mathcal{C}_{net}))$
$\mathcal{P}_{\mathcal{I}}.a$	*inhibit*
Policy P_2	"Store my data only on servers in the EU"
$\mathcal{P}_{\mathcal{I}}.e$	$(store, \{(obj, data), (receiver, r)\})$
$\mathcal{P}_{\mathcal{I}}.\varphi$	$not(eval(r == "EU"))$
$\mathcal{P}_{\mathcal{I}}.a$	*inhibit*
Policy P_3	"Encrypt my data before transmitting it to servers in the USA"
$\mathcal{P}_{\mathcal{I}}.e$	$(store, \{(obj, data), (receiver, r)\})$
$\mathcal{P}_{\mathcal{I}}.\varphi$	$eval(r == "USA")$
$\mathcal{P}_{\mathcal{I}}.a$	$(store, \{(obj, enc(data)), (receiver, r)\})$
Policy P_4	"Delete my data within 30 days after receipt"
$\mathcal{P}_{\mathcal{I}}.e$	$< any >$
$\mathcal{P}_{\mathcal{I}}.\varphi$	$(receiving, \{(obj, data)\})$ *before 30 and* $replim(30, 0, 0, (receiving, \{(obj, data)\}))$
$\mathcal{P}_{\mathcal{I}}.a$	$(delete, \{(obj, data)\})$

3. Hybrid Data Flow Tracking

This chapter describes a hybrid data flow tracking approach to support distributed Data Usage Control policy enforcement. Contents of this chapter have been published in [28, 71].

As Section 2.3 describes, DFT trackers are an integral component of modern DUC systems to specify and to enforce data usage restrictions in a representation independent manner. Although the research field of DFT has been and is being researched, existing solutions fall short in terms of Caveat 1, Caveat 2, and Caveat 3 (cf. Section 1.3). In particular, in a domain where computational resources are shared between different applications, as it is the case within our running scenario (cf. Section 1.2), Caveat 3 might affect AuS's portability too much and tightly binds the data flow tracking mechanism with the web-service's run-time environment [10, 81, 96].

To counteract those limitations, this chapter provides and describes a *hybrid* approach in two different flavors, namely SHRIFT and its extended version HDFT++, which both leverage statically precomputed information flow analysis results to track the flow of data through applications at run-time. In particular, SHRIFT and HDFT++ aim to enhance the portability of AuS and to increase the tracking precision compared to a pure black-box approach (cf. Section 3.1). As a proof-of-concept, SHRIFT and HDFT++ are implemented in Java and evaluated on a set of different applications.

This chapter is structured as follows: Section 3.1 recaps state-of-the-art data flow tracking approaches and techniques and unveils the main caveats and drawbacks of those approaches. Section 3.2 provides a high-level overview of the methodology that underpins SHRIFT and HDFT++. Section 3.3 describes the static analysis part of the proposed hybrid approach, and in Section 3.4 the integration and deployment of SHRIFT and HDFT++ is provided. Section 3.5 provides the formal data flow model which HDFT++ builds upon, to track not only data flow dependencies but also their taken execution paths; in a nutshell, this model is an instantiation of the generic data flow model from Section 2.3 and can be considered as a configuration for

the PIP component inside our UC architecture (cf. Section 2.4). Evaluation results and the strengths and limitations of both hybrid approaches are discussed in Section 3.6 and Section 3.7 respectively.

3.1. State of the Art Information Flow Analysis

Research literature proposes two fundamentally different approaches to analyze the flow of data: *Static Information Flow Analysis (SIFA)* and *Dynamic Data Flow Tracking (DDFT)* [102].

SIFA analyzes the AuS at the program level (or at one of its intermediate representation). They are popular as they detect data flow (henceforth termed *explicit* flows) and control flow (henceforth termed *implicit* flows) dependencies without executing the AuS, and aim to detect all possible information flows [20, 120]. In particular, for implicit flows they also provide certainty about information flows for non-executed branches. A given program is certified as secure, if its (critical) sources do not interfere with its critical sinks, i.e. if no information flows between them. Such a static certification can be used, for example, to reduce the need for run-time checks [21]. Various static approaches (apart from *Program Dependence Graph (PDG)* [98]) are proposed in the literature, which are usually based on type checking [76, 88, 120] or dataflow analysis [3, 4, 9, 20]. To this day, SIFA based approaches are instantiated for different programming languages (like FRAMA-C for C code [27, 56], or JOANA for Java-bytecode [49]) and for different domains (like Andromeda for Java-based web-services [114, 115], or FLOWDROID for ANDROID applications [8]). Although SIFA based tools are quite powerful and widely used, for instance within an app store vetting process to assess if an application adheres to predefined properties, they suffer from scalability issues: even for small and mid-size applications, static analysis demands huge computational resources. In addition, due to implicit dependencies and missing run-time values, SIFA reports information flows which are hard to assess if they may get "really problematic" at run-time. Moreover, handling dynamic aspects in applications, like dynamic callbacks or reflective code, are confined to the AuS. Further, declassification, i.e. relaxing the non-interference property of data flow dependencies, needs to be performed at programming language level without the support of higher-level semantic or run-time values.

In contrast, DDFT-approaches implement a kind of *reference monitor* and analyze the flow of data at run-time for a specific abstraction layer (e.g. the operating system layer [39, 124] or the application layer [10, 54, 69]). Taking into account applications' run-time information, and possibly higher-level semantics, DDFT approaches are popular to be very precise, specifically in the presence of callbacks and reflection. To this day, DDFT based approaches are implemented, for different programming languages (e.g. PHOSPHOR [10] for Java-bytecode, or Trishul [77, 116]

for the Java VM, or LIBDFT [54] for x86-binaries) and different domains (like TAINTDROID [23] for ANDROID, or Maalej [72] for iOS). A prevalent technique used and implemented by those tools is *taint analysis*, which attaches meta-information in form of a *taint-label* to each program variable. By default, the value of a taint label is empty. New taint values are introduced once data flows into an application via a critical source. For instance, a file read-operation would set the taint value of the receiving variable to a specific value. At run-time, those taint values are propagated with every executed instruction according to their semantic. To this end, DDFT-approaches need to monitor each command on the execution path, and therefore, on the downside incur a non-negligible performance overhead; that is where Caveat 2 originates from, cf. Section 1.3. Moreover, taint-labels are not only propagated at the application level but also through the whole RTE. To this end, for instance, the approach in TRISHUL [77, 116] requires to modify the JRE, whereas PHOSPHOR [10] requires modifications of all Java system classes and its used 3^{rd}-party libraries. Moreover, TAINTDROID relies on tracking logic inside the operating system itself, and therefore, requires a modified version of ANDROID, which needs to be installed on the device beforehand; especially, the latter one is a tedious and cumbersome task. However, this thesis advocates that such approaches make it quite difficult to port and run the adapted and modified AuS inside other, off-the-shelf run-time execution environments (that is where Caveat 3 originates from). Moreover, for particular use case domains, like within our running scenario from Section 1.2, where run-time environments usually run multiple SaaS services at the same time, e.g. multiple Java web services are served by a single Tomcat [112] web-server instance, such deep modifications may also lead to intersections of data flows between different SaaS services running simultaneously on the same run-time environment.

3.2. Overview of the hybrid approach

This section provides a high-level overview of our *hybrid* approach which builds the foundation for SHRIFT (cf. Figure 3.2a), and its extension HDFT++ (cf. Figure 3.2b). The basic idea is to use statically pre-computed information flow analysis results to employ a minimal run-time monitor that monitors the actual executed data flows inside the AuS. Whereas SHRIFT tracks the flow of data from inputs I_x to outputs O_x based on a statically pre-computed mapping, HDFT++ extends this approach and also tracks the executed program locations C_x that are taken on the dependency path from I_x to O_x. The advantage of HDFT++ is a potential increase of the data flow tracking precision at run-time, as it reports only those data flows which are actually executed; we provide a tangible example in section 3.4 and compare SHRIFT and HDFT++ in section 3.6.2 to illustrate this advantage. Although the Java programming language

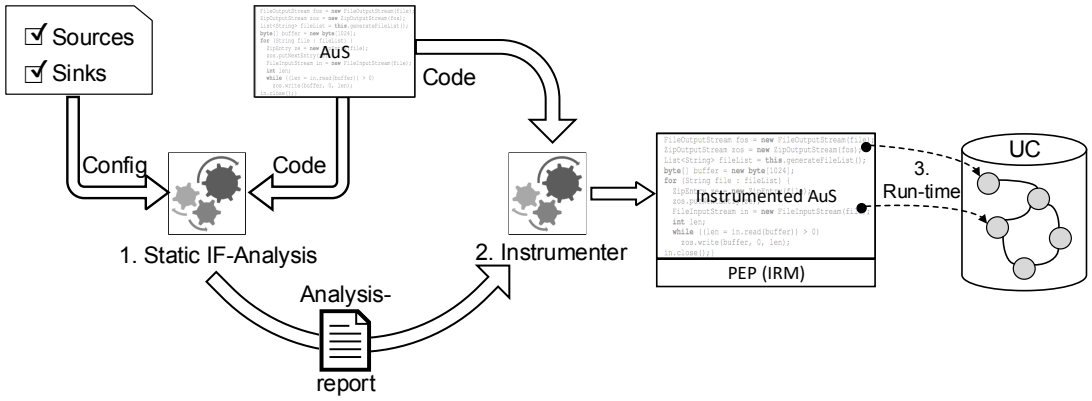


Figure 3.1.: First, our approaches statically analyze the AuS for possible sources, sinks, and sink-source-dependencies. The result is stored into an analysis-report. Second, based on the result, we instrument the code of AuS and inject a minimal run-time monitor (IRM). At run-time, the IRM signals the actually executed sources and sinks to the UC- PIP. In addition, HDFT++'s IRM also signals the executed instructions on a sink-source-dependency path. This way, the PIP keeps track of how data flows and disseminates inside the AuS.

is used to illustrate the idea, the overall approach is generic and can be instantiated for other run-time environments as well, like *PHP* or *Ruby*. The SHRIFT and HDFT++ approach is built upon three parts.

Static Information Flow Analysis (cf. Section 3.3): This step statically analyzes the AuS to detect all input (*source*) and output (*sink*) channels within the application (cf. Figure 3.1). Input channels are those instructions that transfer data from the outside to the inside, like e.g. reading data from a file or a Http-request, whereas output channels are exactly the opposite, like e.g. writing data into a file or a Http-response. Beyond that, this phase also computes dependencies between the set of sources and sinks, as well as the instructions that reside on those dependencies. A dependency exists if data, originating from a source, flow either explicitly or implicitly into a sink [37]. As SHRIFT and HDFT++ are prototypically instantiated for Java, an invocation of a Java standard library method, that reads or writes data from and to the file- and network-I/O, is considered as a source or a sink. Finally, all identified sources, sinks, as well as dependencies between them are reported in an *analysis-report* at the end of this step.

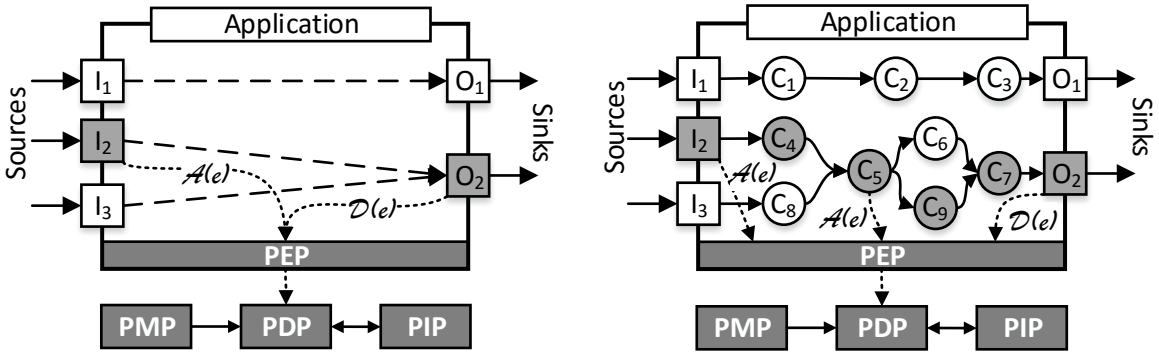
Run-time Data Flow Tracking (cf. Section 3.4): Based on the analysis-report, SHRIFT and HDFT++ inject additional instructions into the AuS (cf. Figure 3.2). The SHRIFT approach instruments only those commands which correspond to a source- or a sink-instruction, and propagates data flows based on the reported data flow dependencies. At run-time, SHRIFT

signals for each executed sink the list of sources it depends on to the PIP. Based on the internal PIP-state and the list of sources, the PIP propagates only those sources to the sink which have been executed and therefore have already a mapping inside the PIP-state. For instance in Figure 3.2a, once sink O_2 is going to be executed SHRIFT signals that it depends on the sources I_2 and I_3. However, at that moment in time only source I_2 was executed before, and thus, the PIP propagates only the data item from I_2 into O_2. The PIP ignores source I_3 as no mapping exists for it.

HDFT++ instruments, in addition to sources and sinks, also all instructions C_x that reside on a sink-source dependency. This way, HDFT++ is able to propagate the flow of data along the taken execution path, and thus, to distinguish if a reported data flow actually happens or not at run-time. To make this advantage more tangible, assume that the instruction C_5 in Figure 3.2b is an `if`-condition, as illustrated for instance in the code example in Listing 3.4. Depending on the input data, which is read by the `input()`-method in this code snippet, the instruction in Listing 3.4 line 4 is executed (which corresponds to C_9 in Figure 3.2) or not (which corresponds to C_6 in Figure 3.2). HDFT++ is able to distinguish which path is taken, and thus compared to SHRIFT, only reports a flow of data if the instruction in Listing 3.4 line 4 is executed, otherwise no flow is reported by HDFT++.

Anyway, at run-time SHRIFT and HDFT++ serve as an IRM and extract information about the running application. For instance, for sources and sinks which read and write data from and to files, SHRIFT and HDFT++ extract the filename and file-descriptor of the respective file and signal that information to the UC infrastructure. Based on that, the PIP updates its internal state and creates a mapping between the filename and the container. Note, the filename serves as a naming identifier in our data flow model, cf. section 3.5. Extracted information is signaled in form of events to a UC infrastructure, which is conceptually designed as described in Section 2.4. Events are classified in desired events $\mathcal{D}(e)$ whose execution admissibility needs to be approved by the PDP, and actual event $\mathcal{A}(e)$, which are directly sent to the PIP to keep track of data flows. The latter one instantiates and implements the data flow tracking model (cf. Section 3.5), and thus, based on the signaled events $\mathcal{A}(e)$ keeps track of how data flows and disseminates inside an application. That way, the PIP knows at each moment in time which data originates from which input-channel, is stored at which program location, and flows into which output-channel inside the AuS.

Data Flow Tracking Model (cf. Section 3.5): Depending on a sink-source-dependency, data might flow from a source to a sink across different program locations at run-time. To track those locations and the dissemination of data inside an application, the generic data flow tracking model from Section 2.3 is instantiated for the Java domain, taking into account the different



(a) SHRIFT (b) HDFT++

Figure 3.2.: HDFT++ in Figure 3.2b is an extension of SHRIFT and tracks not only the execution of one particular dependency $\{I_x, O_x\}$-tuple, but also all executed instructions C_x residing on a sink-source dependency. This way, HDFT++ is able to distinguish if a statically reported data flow dependency actually happens or not at run-time (Section 3.4 elaborates in detail on that aspect).

operations and their semantics that might occur on a dependency path. In a nutshell, the instantiated data flow model implements a state transition system and captures the flow of data at each moment in time as a specific state of the AuS. A state represents which sources flow to which sinks inside the AuS at run-time. State transitions are triggered by the actually executed instructions on a sink-source-dependency. This thesis argues, that representing the flow of data as a sequence of state transitions, and each state as a snapshot of flowed data, makes the enforcement of data usage restrictions in a representation independent manner more convenient, than on a sequence or list of all execution traces that an application has taken. Note, this model is used by HDFT++ to handle cases as illustrated in Listing 3.4 (cf. Section 3.5).

3.3. Static Information Flow Analysis

To identify program locations that might correspond to a data flow dependency, this phase statically analyzes the AuS's Java-bytecode and detect all locations that might correspond to a source, a sink, or an instruction on a dependency path. To do so, this phase uses JOANA [49], a state-of-the-art static information flow analysis tool for Java. First, JOANA transforms the AuS's bytecode into a *Static-Single-Assignment (SSA)* [16] form which is a language-independent representation of the AuS. SSA demands that each variable in a program must be defined before it is used and that a variable is assigned exactly once. If a variable is assigned more than once, each assignment generates a new variable at the SSA level.

Second, based on the SSA representation, JOANA builds the *System Dependence Graph (SDG)* of the application which is a special graph representation of the AuS. An SDG is a direct graph $G = (V, E)$, whose vertices V represent program instructions and statements in SSA form, whereas its edges E represent control- and data-dependencies between its vertices [37]. SDGs incorporate *inter-* and *intra-*procedural dependencies, i.e. dependencies that are not only within a single procedure but also cross procedure's boundaries. A *control-dependency* exists between vertices x and y if the evaluation of y influences the execution of x; whereas, a *data-dependency* exists if y may use a value that is computed at x.

Third, using a program's SDG and a high-level description of sources and sinks, JOANA applies *slicing* techniques to compute if a set of vertices (identified as sources) might affect – either directly or transitively via control- and data-dependencies – the execution or value of another set of vertices (identified as sinks); Listing 3.2 illustrates a sample source-/sink-specification. To do so, Joana uses *context-sensitive* slicing [99], a special form of graph reachability analysis: given a node n of the SDG, identified as a sink, Joana computes the *backward-slice* for n, which is the set B of all those nodes from where n is reachable through a path in the SDG and that respect the calling context. For sequential programs, it has been shown that a node which is not contained in the backward slice $m \notin B$ can not influence the execution of n [44, 121], and hence, SDG-based slicing on sequential programs guarantees in a certain sense *non-interference* [33, 34], which stipulates that low outputs are independent from high/sensitive inputs. However, for concurrent programs this approach was extended by Giffhorn [31, 32] to take additional kinds of information flows into account, like e.g. probabilistic channels [103].

The result of the slicing-process is a subset-graph $G' \subset G$ of the original SDG G that represents dependencies between sinks and sources. Henceforth, we term with *chop* a single dependency path from a source to a sink inside G'. A chop includes not only the vertices for source- and sink-instructions, but also all vertices that correspond to instructions that lead to a flow of data from a source to sink. Henceforth, we term all vertices residing on a single data flow dependency as *chopCMD* and the union of all chopCMDs over all chops as *Points of Interest (PoI)*. Figure 3.3 illustrates an example chop for an explicit data flow dependency path: sink in line 12 has an information flow dependency on line 1; nodes 4 and 6 are the corresponding chopCMD on that dependency. Put simply, a chop is considered as the intersection between a source's *Forward-Slice*, i.e. all instructions that are either explicitly or indirectly affected by and reachable from a source, and a sink's *Backward-Slice*, i.e. all instructions that affect and influence a sink [37]. Beyond that, JOANA also supports different *points-to* analysis-techniques [5, 111] (like 0-1-CFA[36], k-CFA [109], object-sensitiveness [74]), to take into account not only the static data type but also possible dynamic data types of a variable.

```
1   a = source();
2   while (n > 0){
3     x = v();
4     d = a;
5     if (x > 0)
6       c = d + 25;
7     else
8       c = 25;
9     n--;
10  }
11  y = a;
12  sink(c);
```

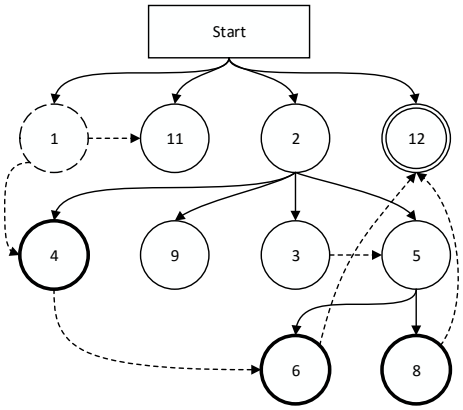


Figure 3.3.: Example code and its chop between a source (dashed circle) and a sink (doubly circled). Dashed arrows symbolize data flow dependencies; solid arrows symbolize control dependencies; bold circles are chopCMD vertices.

The results of the static analysis are written by JOANA into an *analysis-report*. For each identified sink, JOANA provides detailed information about the sources and chopCMDs it depends on, and where their corresponding instructions are located within the bytecode. This includes (cf. example in Listing 3.1): the parent method (`ParMethod`), the bytecode-offset (`offset`) where a PoI is located; the Java signature of invoked sinks or sources (`signature`); and the executed operation in SSA-representation for chopCMD instruction (`Label`). Based on the analysis-report, additional instructions are injected into the application to extract context information about each PoI and to track the flow of data from sources to sinks across chopCMD instructions.

Remarks on the source and sink specification (cf. Listing 3.2): JOANA uniquely identifies a source or a sink (in Java bytecode notation) by a triple, composed of the fully qualified class name, the method name, and the parameter that is either passed to or returned from the method invocation. As parameter names are not available in Java bytecode, the position of a parameter inside the method signature is used instead. For instance, "`param: 1`" in Listing 3.2 specifies the first parameter of the method `write` as a sink and the first parameter of the method `read` as a source. Whereas the reserved term `ret` specifies the return value of `getParameter` as a source. It is important to note that usually the list of sources and sinks is provided to the analysis by a security expert, who has specific domain knowledge (e.g. does the application use only JNI to call

Listing 3.1: Example static analysis-report generated by JOANA. Multiple sources (cf. line 1) and sinks (cf. line 1) are listed, as well as, dependencies between them (cf. line 1).

```
1  <sources>
2    <source>
3      <id>Source1</id>
4      <location>JZip.zipIt(Ljava/lang/String;Ljava/lang/String;)V:191</location>
5      <signature>java.io.FileInputStream.read([B)I</signature>
6      <param index="1"/>
7    </source>
8    <source>
9      <id>Source2</id>
10     ...
11   </source>
12 </sources>
13 <sinks>
14   <sink>
15     <id>Sink1</id>
16     <location>JZip.zipIt(Ljava/lang/String;Ljava/lang/String;)V:185</location>
17     <signature>java.util.zip.ZipOutputStream.write([BII)V</signature>
18     <param index="1"/>
19   </sink>
20   <sink>
21     <id>Sink2</id>
22     ...
23   </sink>
24 </sinks>
25 <flows>
26   <sink id="Sink1">
27     <source id="Source1"/>
28     <chop>
29     <chopNode bci="39" lab="v11.start()" om="jzip.JZip.main([Ljava/lang/String;)V"/>
30     <chopNode bci="41" lab="v13.load(v9)" om= "jzip.JZip.loadConfig(Ljava/lang/String;)V"/>
31     ...
32     </chop>
33   </sink>
34 </flows>
```

its own native libraries) about critical dependencies between sources and sinks inside the AuS. To a certain extent, the list of sources and sinks has to be chosen manually, e.g. by reading the API documentation and then deciding which methods and parameters are relevant. For example, one may consider `FileOutputStream.write()` or `HttpServletRequest.getParameter()` together with appropriate parameters as sinks.

Listing 3.2: Example sink- source-specification that is used by JOANA to determine which method invocations might correspond to a sink or a source

```
1  Source:
2    class:              Ljava/servlet/http/HttpServletRequest
3    method:             getParameter(Ljava/lang/String;)Ljava/lang/String;
4    param:              ret
5    includeSubclasses:  true
6    indirectCalls:      true
7  Source:
8    class:              Ljava/io/InputStream
9    method:             read([B)I
10   param:              1
11   includeSubclasses:  true
12   indirectCalls:      true
13 ...
14 Sink:
15   class:              Ljava/io/Writer
16   method:             write(Ljava/lang/String;)V
17   param:              1
18   includeSubclasses:  false
19   indirectCalls:      false
20 ...
```

By default, JOANA detects sources and sinks within the AuS by simply matching the full method signature (including their class-membership) within the SDG against the source- and sink-specification. However, in the presence of inheritance and polymorphism, such a strategy would miss some sources and sinks, especially those that are overwritten within their child-classes. For instance, consider line 17 in Listing 3.3 that reads data from a file into a buffer via a `FileInputStream` and the source specification of `InputStream` in Listing 3.2: although `FileInputStream` inherits from `InputStream`, and therefore has the same semantic for the `read`-method as its parent-class, JOANA by default does not detect the invocation in line 17 as a source because this method invocation belongs to another class. A possible approach to solve that obstacle would be to list every method manually and explicitly that could potentially be a source or a sink. However, listing all possible sources and sinks in all their variations along the class hierarchy might be a tedious and error-prone task because some methods may be missed, especially when multiple child-classes overwrite methods which are actually specified

as sources or sinks within their parent-classes. To make the source- and sink-detection process more convenient we chose the following approach: we list only the most general source- and sink-declarations, e.g. `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer` for Java I/O classes. After that, these specifications are extended automatically by Joana using the following rule: "If s is a source/sink and s' overrides s, because s' is a child-method, then also s' is a source/sink". JOANA implements this rule by inspecting and analyzing the hierarchy class of the given program; this rule is activated with `includeSubclasses = true` in Listing 3.2 for each source or sink.

Another issue that we have faced is the case when applications do not invoke a source or a sink directly from the application-code, but indirectly, e.g. via a nested library call. For example, listing 3.3 contains a call to the method `Properties.load()` in line 5 which takes an input stream as a parameter and uses it to fill a properties table. This method is not included by the previous rule because `Properties.load()` itself is not a source. For that reason, the source- and sink-specification is again extended automatically by the following rule: "If s is a source/sink, s' may call s and p' is a parameter of s', then s' is also considered as source/sink". JOANA implements this rule by using a call graph of the application, which is also built and used during SDG construction, so it can be reused here. This feature is activated with `indirectCalls = true` in Listing 3.2 for each source or sink.

3.4. Run-time Data Flow Tracking

Consider the code snippet in Listing 3.3 that is used in our test-suite application JZip to zip files (Section 3.6): based on the sink-source-specification (cf. Listing 3.2) static information flow analysis detects the flow from the `read`-method in line 17, where the to-be-zipped files are read, to the `write`-method in line 18, where those files are written into the archive. Listing 3.1 shows the corresponding analysis report: line 2 – line 7 specify that the first parameter (line 6) of the `read` method-invocation (line 5) at bytecode offset `191` (line 4) in method `zipIt` (line 4) is identified as `Source1` (line 3). The same holds for `Sink1` (line 14 – line 19) but in this case the first method parameter (line 18) is identified as a sink. The analysis-report also provides information about the dependency between `Sink1` and `Source1` (line 26 – line 33); if a sink would depend on multiple sources, i.e a 1-to-n dependency, then a `sink`-tag (line 26 – line 33) would wrap multiple sources.

Based on the analysis-report, this phase injects a piece of code for each reported PoI into the AuS. At run-time, this code serves as a minimal IRM and extracts and signals information about each executed PoI to the PIP and PDP (cf. Figure 2.1). Such information could be, for instance,

the filename from where data is read. To do that, this thesis proposes two different approaches: SHRIFT and HDFT++. Although both approaches share the same static analysis phase (cf. Section 3.3) the injected tracking and propagation logic behave differently at run-time. SHRIFT uses the pre-computed static dependency mapping (e.g. cf. line 26 - line 33 in Listing 3.1) to propagate the flow of data between sources and sinks, whereas HDFT++ also takes into account all chopCMDs residing on a sink-source dependency. Beyond that, both approaches verify on every executed source or sink instruction if the execution trace of the AuS adheres to deployed DUC policies. Thus, both IRM approaches take the role of the PEP in our UCI infrastructure (cf. Figure 2.1); from henceforth, we use the terms IRM and PEP interchangeably. The outcome of this phase is an instrumented version of the original AuS augmented with a PEP that interacts with the DUC's PDP and PIP on each executed PoI. We use the tool *ObjectWeb ASM* [80], a bytecode manipulation and engineering framework for Java bytecode, to instrument and to inject the PEP into the AuS. The remaining section describes both approaches in detail and illuminates and emphasizes the differences between them.

Listing 3.3: Example Java code fragment for zipping files inside an application

```
1  void zipIt(String file, String srcFolder) {
2    byte[] buffer = new byte[1024];
3    Properties prop = new Properties();
4    InputStream is = this.getClass().getResourceAsStream("jzip.properties");
5    prop.load(in);
6
7    is.read(buffer);
8
9    FileOutputStream fos = new FileOutputStream(file);
10   ZipOutputStream zos = new ZipOutputStream(fos);
11   List<String> fileList = this.generateFileList(srcFolder);
12   for (String file : fileList) {
13     ZipEntry ze = new ZipEntry(file);
14     zos.putNextEntry(ze);
15     FileInputStream in = new FileInputStream(file);
16     int len;
17     while ((len = in.read(buffer)) > 0)
18       zos.write(buffer, 0, len);
19     in.close();
20   }
21 }
```

SHRIFT instruments only source- and sink-instructions inside the AuS that have a data flow dependency between each other (cf. Figure 3.2a). For instance, according to the analysis-report in Listing 3.1 SHRIFT instruments `Sink1` and `Source1` inside the AuS and leaves out `Source2`, as it does not have any data flow dependencies to any sinks. Furthermore, as one may notice, all information provided by the analysis-report are inherently static by nature, i.e. they are elicited

from the AuS's bytecode by inspection without executing the AuS. For instance, `Source1` is specified as a Java `FileInputStream.read` invocation which is located at offset `191` inside the method `JZip.zipIt`. However, the analysis-report does not provide dynamic run-time information, like the filename that was read at `Source1` or which data was written into `Sink1`. Such information are extremely valuable as they provide further details about a source or a sink, and reveal in particular, if data emanated from a source is already tainted, i.e. is linked to and affected by a DUC policy or not. To catch such information, our PEP injects for each source- and sink-instruction an *extractor*, which filters and extracts additional information about the respective sources and sinks at run-time. As those extractors are quite specific for one particular type of source or sink, and may also differ from one type to the other, the set of extractors are extendable and configurable inside the PEP to extract also information which have not been considered before, like e.g. extracting REST-Urls from a REST-interface. This thesis underlying prototype implementation contains extractors for

1. file-I/O: extracts the file-descriptor ($fd \in \mathcal{F}$) and the file-name ($fn \in \mathcal{F}$) on each file-system read or write operation
2. network-I/O: extracts the IP-($ip \in \mathcal{F}$) and port-information ($port \in \mathcal{F}$) from incoming and outgoing network accesses
3. database-I/O: extracts the database-($dbName \in \mathcal{F}$), table-($tabName \in \mathcal{F}$), and table-field-name ($tabFieldName \in \mathcal{F}$), on each database read or write access

For instance, as `Source1` in Listing 3.1 reads data from a file our injected file-I/O extractor extracts (if available) the filename $fn_I \in \mathcal{F}$ and file-descriptor $fd_I \in \mathcal{F}$ from where the data was read; for the sake of illustration only the filename is used in the following. The extracted filename fn_I is signaled along with the source identifier (cf. `id`-tag inside the enclosing `source`-tag in Listing 3.1) as an actual event $\mathcal{A}(e)$ to the DUC's PIP. Based on this information, the PIP updates its internal data flow storage function $\sigma'.s[\sigma.f(\texttt{Source1}) \leftarrow \sigma.s(\sigma.f(fn_I)) \cup \sigma.s(\sigma.f(\texttt{Source1}))]$ and thus keeps track of which data was read by the AuS.

Processing and handling of sinks works similar (note, sinks are only signaled if at least one of its depending sources are executed): as `Sink1` writes zipped data into a file our injected file-I/O extractor extracts and signals the filename $fn_O \in \mathcal{F}$ along with the sink identifier (cf. `id`-tag inside the enclosing `sink`-tag in Listing 3.1) to the PIP. Based on the sink identifier and the precomputed analysis-report, the PIP determines all sources a sink depends on and updates its internal data flow storage function accordingly, i.e. in this sample the storage mapping for `Sink1` is updated as follows $\sigma'.s[\sigma.f(\texttt{Sink1}) \leftarrow \sigma.s(\sigma.f(\texttt{Sink1})) \cup \sigma.s(\sigma.f(\texttt{Source1}))]$. Furthermore,

the PIP also maintains an alias relation between the sink identifier and the filename fn_O, as on every sink execution the entire data a sink points to (according to the current PIP's state) may flow into the output-file fn_O, i.e. $\sigma'.l[\sigma.f(\texttt{Sink1}) \leftarrow \sigma.f(\texttt{Sink1}) \cup \sigma.f(fn_O)]$. This way, the PIP traces that the output data, which is written into the file fn_O at `Sink1`, may potentially contain derived data from file fn_I. Thus, the PIP knows at each moment in time which data flowed from which source into which sink through an application.

However, as one may notice SHRIFT detects the flow of data based on a statically precomputed mapping. It does not differentiate if a particular data flow path is still critical when a sink is reached. Regardless of the instructions on a data flow dependency, SHRIFT always reports that a flow of data happened. For instance, consider the code in Listing 3.4: the actual crucial instruction that leads to a flow of data is wrapped in an `if`-branch in line 4, whose execution actually depends on the concrete value of the variable `condition`. However, for SIFA tools it is almost impossible to compute statically the concrete value for variable `condition`, especially when its computation also depends on user input. Therefore, SIFA tools conservatively overapproximate such cases and report the flow of data from line 1 to line 8 in Listing 3.4.

Listing 3.4: Example of an explicit flow dependency.

```
1 byte[] in=input(); //source
2 byte[] out;
3 if (condition) {
4   out=in;
5 } else {
6   out = new byte[10];
7 }
8 output(out); //sink
```

HDFT++ extends SHRIFT (cf. Figure 3.2b) and also monitors all chopCMDs (cf. Section 3.3), i.e. all those instructions that reside on a sink-source dependency path. In detail, HDFT++ monitors all executed PoI, i.e. the source-, sink-, and chopCMD-instructions, and extracts and signals information about each of them to the PIP at run-time. To do that, HDFT++ also uses the same set of extractors for source- and sink-instructions as SHRIFT and extracts additional information about each executed source and sink, like e.g. the filename of a source. Further, HDFT++ leverages the `chopNode`-tag from the analysis-report and injects its tracking logic in such a way, that all statically raised information about chopCMDs are signaled to the PIP at run-time. For instance, from the `chopNode`-tag in Listing 3.1 line 30 HDFT++ derives that at the bytecode offset `41` (cf. `bci`-attribute) the method `load` is called on an object identified by `v13` at JOANA's SSA representation layer (cf. `lab`-attribute in Listing 3.1 line 30). Moreover, a parameter, identified with `v9`, is passed to that method call. At run-time, HDFT++ signals that information to the PIP together with the following extracted from the involved operands in a chopCMD:

1. Java Caller-ObjectId, -Class, -Method
2. Java Callee-ObjectId, -Class, -Method
3. Java Parameter-ObjectId and -Names
4. Memory-Addresses of involved Java objects
5. Process-, Thread-Id

Based on that information, HDFT++'s PIP continuously updates its internal data flow state on every executed chopCMD. However, depending on actual run-time values only a subset of all chopCMDs on a dependency path may be executed and thus may lead to different tracking results. For instance, the value of variable `condition` in Listing 3.4 controls if data either flows from the variable `in` to `out` or not. This example illustrates, that depending on the actually executed chopCMDs and run-time values, data may flow differently inside the AuS (involving different program variables) for the same statically reported data flow dependency, and thus, may result in different PIP's data flow states. In case the variable `in` is assigned to `out` in Listing 3.4 line 4, the PIP would end up in a state where both variables point to the same data item, otherwise not. As HDFT++ monitors the execution of each chopCMD on a dependency path, HDFT++ is able to cope with situations where run-time values influence the actual tracking result, and thus, is able to differentiate if a flow of data really happens or not at run-time. To reflect and to model how data flows across chopCMDs inside an AuS, HDFT++ instantiates in Section 3.5 the formal data flow model from Section 2.3 for Java bytecode. Simply speaking, this instantiation serves as a configuration for the DUC's PIP component. Notified by the PEP about the executed PoI, the PIP updates continuously its current state of flowed data, and hence, keeps track of how data propagates through the AuS from sources, along chops, to the sinks. Thus, the PIP knows at each moment in time which data originates from which source and is flowed into which sink inside the AuS.

Fundamentally HDFT++ does not propagate a taint-label along the lines of executed bytecode commands, as it happens in a pure dynamic approach, but instead, it updates the PIP's internal data flow state on every executed chopCMD. This way, HDFT++ does not need to inject complex, performance-affecting tracking logic at each chopCMD, as the complete taint propagation logic (cf. Section 3.5) happens inside the PIP. Depending on the actual taken data flow path at run-time, PIP's data flow state may result in different storage-, alias-, and naming-mappings, which either comply with a data usage control policy or not. This means, that once a sink is reached the PIP state (i.e. the current storage-, alias-, and naming-mapping, cf. Section 3.5) reveals which data flow path was taken and if one of the sink's parameter may contain data that is affected and

protected by a data usage control policy. For instance, Policy P_1 is violated if the storage function $s(srcnet)$ returns for a sink network container $srcnet = f((ip \times port))$ a data item $d \in \mathcal{D}$ that is covered by Policy P_1, i.e. $d \in s(f((ip \times port)))$. As we do not know beforehand which of those chopCMDs are the actual crucial and relevant ones, HDFT++ has to monitor all chopCMDs on a data flow dependency from a source to a sink.

Listing 3.5: Example application which reads data from a file and transmits it to a remote server.

```
1  public class SampleReadSend {
2    public static void main(String[] args) {
3      new SampleReadSend().sample();
4    }
5    public void sample() {
6      int rand = new Random().nextInt(10);
7      String[][] ips = {{"172.16.33.1","8080","EU"},
8                        {"110.22.55.1","9090","US"}};
9      String[] dest = rand%2 == 0 ? ips[0] : ips[1];
10     String fileContent = "";
11     try {
12       fileContent = this.readFile("secret.txt");//Read secret file
13       if(rand > 5){
14         fileContent = this.readFile("public.txt");//Read public file
15       }
16       Socket destSocket = new Socket(dest[0],Integer.parseInt(dest[1]));
17       System.out.println("Sending data to "+dest[1]+" "+rand);
18       PrintWriter pw = new PrintWriter(destSocket.getOutputStream(),true);
19       pw.write(fileContent);
20       pw.close();
21       destSocket.close();
22     } catch (IOException e) { e.printStackTrace(); }
23   }
24   private String readFile(String fileName) throws IOException{
25     String _return = "", line = "";
26     FileReader fr = new FileReader(fileName);
27     BufferedReader bf = new BufferedReader(fr);
28     while ((line = bf.readLine()) != null)
29       _return += line;
30     return _return;
31   }
32 }
```

At run-time, however, depending on the taken control flow path only a subset of those monitored chopCMDs are actually executed, and thus, may produce a slightly different data flow state inside the PIP once the sink at the end of a dependency is reached.

Listing 3.6: Static analysis-report excerpt for the example code in Listing 3.5.

```
<source>
  <id>Source0</id>
  <location>SampleReadSend.readFile(Ljava/lang/String;)...
    ...Ljava/lang/String;:51</location>
  <signature>java.io.BufferedReader.readLine()Ljava/lang/String;</signature>
  <return/>
</source>
...
<sink>
  <id>Sink0</id>
  <location>SampleReadSend.sample()V:195</location>
  <signature>java.io.PrintWriter.write(Ljava/lang/String;)V</signature>
  <param index="1"/>
</sink>
...
<flows>
  <sink id="Sink0">
    <source id="Source0"/>
    <chop>
      <chopNode bci="-8" lab="PHI v20 = #(), v19"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="6" lab="v5 = new java.io.FileReader"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="35" lab="v14 = valueOf(v20)"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="42" lab="v17 = v12.append(v10)"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="45" lab="v19 = v17.toString()"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="51" lab="v10 = v7.readLine()"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="60" lab="return v20"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="107" lab="v36 = this.readFile(#(public.txt))"
        om="SampleReadSend.sample()V"/>
      <chopNode bci="120" lab="v40 = this.readFile(#(secret.txt))"
        om="SampleReadSend.sample()V"/>
      <chopNode bci="176" lab="v63 = new java.io.PrintWriter"
        om="SampleReadSend.sample()V"/>
      <chopNode bci="195" lab="v63.write(v41)"
        om="SampleReadSend.sample()V"/>
    </chop>
  </sink>
</flows>
```

The following illustrates HDFT++'s behavior along the Java code example in Listing 3.5. Note, here we use a simplified example to illustrate the core idea behind HDFT++. But conceptually, this piece of code reflects how web-applications, e.g. our BirthdayApp from Section 1.2 may

transmit data to a remote server. Typically, a web-application would read input data from an `HttpRequest` instead from a file and transmit it to a remote service via a socket connection.

Listing 3.5 reads data in line 12 from a secret file into the variable `fileContent` and transmits it to a remote server in line 19. Depending on the randomly generated value `rand` (cf. Listing 3.5 line 6) the content of variable `fileContent` may be overwritten by some public data from the file `public.txt` in Listing 3.5 line 14. Further, we assume that the content of `secret.txt` is protected by Policy P_1. For the sake of simplicity, we suppose that Policy P_1 is technically enforced by prohibiting network access if the payload contains data from `secret.txt`. Thus, depending on the actual run-time value of the variable `rand` the sink in Listing 3.5 line 19 may be executed, because variable `fileContent` contains data from the file `public.txt` or its execution is prohibited because variable `fileContent` contains data from the file `secret.txt`.

Listing 3.6 shows the analysis-report for our code sample in Listing 3.5 (note, Listing 3.6 is an excerpt from the full analysis-report in Listing A.1). As expected, the data flow dependency between the source `BufferedReader.readline` (cf. Listing 3.5 line 28) and the sink `PrintWriter.write` (cf. Listing 3.5 line 19) is reported, including all related chopCMDs. When Policy P_1 is deployed in the PDP, an initial state mapping is created inside the PIP between the filename `secret.txt` and the data identifier `D1` $\in \mathcal{D}$ (cf. Listing 3.7 line 1); for the sake of simplicity, we use `D1` as a data identifier in this example. Once the source in Listing 3.5 line 28 is executed, HDFT++ extracts and signals to the PIP from which file the content was read. Based on this information, the PIP determines the corresponding data identifier and propagates it along the lines of executed chopCMDs towards the sink; for the file `secret.txt`, the PIP determines `D1` from its state as the data identifier. HDFT++ also deduces from the analysis-report (cf. Listing 3.6 line 30) that the source's return value is assigned to a variable that is identified by `v10` at JOANA's SSA representation. Because of that, the PIP updates its state with Listing 3.7 line 4. Moreover, HDFT++ derives from the chop node label `'v17 = v12.append(v10)'` in Listing 3.6 line 26 that `v10`'s data identifier also may potentially flow into `v17` and `v12`. HDFT++ tracks that at run-time with line 6 and line 5 in Listing 3.7. All the other, remaining chopCMDs on a dependency path are processed the same way, i.e. the PIP updates its current state accordingly to the executed chopCMD at run-time. As a chopCMD may be any possible Java bytecode command, and thus, may update the PIP state differently depending on its semantic, Section 3.5 instantiates the formal data flow model from Section 2.3 for the Java bytecode level.

Once the sink in Listing 3.5 line 19 is reached, HDFT++ determines if the `PrintWriter` object `pw` is writing variable `fileContent` into a network-socket outputstream or not. In case of a network-socket outputstream, HDFT++ also extracts the destination IP and port to which data is sent. However, at that point in time the PIP state in Listing 3.7 reveals to the PDP that the sink

parameter, which is identified by v41 at JOANA's SSA representation (cf. Listing 3.6 line 40), contains data from the file `secret.txt`. Therefore, as executing the sink would violate Policy P_1 the PDP rejects the execution of the sink in Listing 3.5 line 19.

Listing 3.7: PIP state only reading `secret.txt`.

```
1                                                          secret.txt ---> D1
2                                                  Proc4222|Source0 ---> D1
3      3775970392|readFile(Ljava/lang/String;)Ljava/lang/String;|ret ---> D1
4      3775970392|readFile(Ljava/lang/String;)Ljava/lang/String;|v10 ---> D1
5      3775970392|readFile(Ljava/lang/String;)Ljava/lang/String;|v12 ---> D1
6      3775970392|readFile(Ljava/lang/String;)Ljava/lang/String;|v17 ---> D1
7      3775970392|readFile(Ljava/lang/String;)Ljava/lang/String;|v19 ---> D1
8      3775970392|readFile(Ljava/lang/String;)Ljava/lang/String;|v20 ---> D1
9                                          3775970392|sample()V|v36 ---> D1
10                                         3775970392|sample()V|v40 ---> D1
11                                         3775970392|sample()V|v41 ---> D1
12     3783646496|readFile(Ljava/lang/String;)Ljava/lang/String;|v20 ---> D1
13 3783817496|append(Ljava/lang/String;)Ljava/lang/StringBuilder;|p1 ---> D1
```

Listing 3.8: PIP state reading `secret.txt` and subsequently `public.txt`.

```
1                                                  NET:172.16.33.1:8080 ---> D2
2                                                           public.txt ---> D2
3                                                           secret.txt ---> D1
4                                                      Proc3068|Source0 ---> D2
5                                                Proc3068|Thread1|Sink0 ---> D2
6      3775970248|readFile(Ljava/lang/String;)Ljava/lang/String;|ret ---> D2
7      3775970248|readFile(Ljava/lang/String;)Ljava/lang/String;|v10 ---> D2
8      3775970248|readFile(Ljava/lang/String;)Ljava/lang/String;|v12 ---> D2
9      3775970248|readFile(Ljava/lang/String;)Ljava/lang/String;|v17 ---> D2
10     3775970248|readFile(Ljava/lang/String;)Ljava/lang/String;|v19 ---> D2
11     3775970248|readFile(Ljava/lang/String;)Ljava/lang/String;|v20 ---> D2
12                                         3775970248|sample()V|v36 ---> D2
13                                         3775970248|sample()V|v40 ---> D2
14                                         3775970248|sample()V|v41 ---> D2
15     3783646520|readFile(Ljava/lang/String;)Ljava/lang/String;|v20 ---> D2
16     3783860112|readFile(Ljava/lang/String;)Ljava/lang/String;|v20 ---> D2
17 3783817520|append(Ljava/lang/String;)Ljava/lang/StringBuilder;|p1 ---> D2
```

Listing 3.8 shows the PIP state when variable `rand` in Listing 3.5 is larger than 5. In that case, the value of variable `fileContent` is overwritten with the content from `public.txt` (identified with $D2 \in \mathcal{D}$ inside the PIP state) in Listing 3.5 line 14. This means, that in this case executing the sink is no longer prohibited by Policy P_1. Thus, the PIP updates its state with line 1 in Listing 3.8 once the sink is executed; we use the extracted target IP and port information inside the naming identifier to track to which remote machine data was transmitted. In contrast, Listing 3.7 has no "`NET:[ip]:[port]`" mapping entry in its state as the sink execution was

prohibited by Policy P_1. Note, we show in Listing 3.7 and Listing 3.8 all PIP state entries which have been generated during our runs. Although only some of them are relevant to decide if Policy P_1 is violated or not, all other entries are needed to follow the dependency chain from a source to a sink. As the PIP state may grow quite fast after several runs a proper extension mechanism, which sanitizes the PIP state from unneeded mappings, could be implemented in future work.

3.5. Data Flow Tracking Model for Java

In contrast to SHRIFT, which tracks the flow of data from sources to sinks based on a statically pre-computed mapping, HDFT++ leverages chopCMDs in order to follow the actually taken data flow path through an application. We argue, that depending on run-time values and the executed instructions only a subset of the reported data flow dependencies between sources and sinks are actually triggered. This section instantiates the generic data flow model (cf. Section 2.3) for the Java programming language, to reflect the actual propagation semantic and logic for a single executed chopCMD. Moreover, this instantiation implements a state transition system where each state captures which data is stored at which program location inside the application. State transitions are triggered at run-time by the executed Java bytecode-instruction at each chopCMD. HDFT++ leverages not only the Java bytecode semantic, but also SSA-information provided by JOANA (cf. e.g. `Label` in Listing 3.1) about each sink, source, or chopCMD. Put it simply, this instantiation is used as a configuration for the PIP to reflect the propagation semantic at the chopCMD level.

In contrast to a pure dynamic run-time tracker that has to monitor any single bytecode instruction (like PHOSPHOR [10]), HDFT++ monitors point-wise only those commands that are actually relevant for a flow of data. Furthermore, HDFT++ does not propagate a taint-label via shadow variables inside the AuS, but instead it updates the internal PIP's mapping on every executed chopCMD according to the state transition update rules, which are described in the following. This way, the PIP state reveals at each moment in time which data has flowed from which sources to which sinks, and thus, provides valuable information to enforce data usage policies. For instance, consider a temporal data usage restriction like *Delete my data after 5 days*. As personal data might move from one to the other container within 5 days, it is necessary to track and record its dissemination inside the AuS. Otherwise, policy enforcement would miss some of the protected personal data.

Containers, $\mathcal{C}$. Java variables are used to read and write data in memory regions (like *Java Stack* or *Java Heap*). Apart from the variable's data type, Java distinguishes between *primitive-*

(like numeric or boolean types) and *reference-typed* (like class or interface types) variables. They differ in their assignment behavior: primitive-typed variables are assigned by copying the value of the variable. As opposed to this, reference-typed variables are assigned by copying the memory address of the stored data. Therefore, after a reference-typed assignment both variables point to the same data item, so that, data modifications are immediately propagated to both variables. HDFT++ defines set $\mathcal{C} = \mathcal{C}_{\mathcal{P}} \cup \mathcal{C}_{\mathcal{R}}$ as the union between primitive- ($\mathcal{C}_{\mathcal{P}}$) and reference-typed ($\mathcal{C}_{\mathcal{R}}$) containers.

Principals, $\mathcal{I}$. Each command in a Java application is executed by a Java-Thread and might lead to a flow of data. Therefore, set $\mathcal{I}$ is instantiated as all threads inside a Java program. A thread always runs within a process, thus a Java-Thread is identified by the tuple $\mathcal{I} = \mathit{ProcessID} \times \mathit{ThreadID}$. In this context set $\mathcal{D}$ defines all data items that can be processed by $\mathcal{I}$.

Naming identifiers, $\mathcal{F}$. A Java variable is identified by its *name* inside a *scope*. A scope is a code region where a variable is visible and accessible. Depending on the variable's lifetime, HDFT++ distinguishes between *class-*, *instance-*, and *method-level* scopes. Class-level scopes are labeled with the *fully qualified name (FQN)* of the class (*ClassFQN*), whereas instance-level scope-labels take also into account the memory address *Address* of an object. Method-level scope-labels extends the previous one and identify local variables that only life during the execution of method *MethodName*. We distinguish between identifiers for objects $\mathcal{F}_{\mathcal{I}}$, arrays $\mathcal{F}_{\mathcal{A}}$, array elements $\mathcal{F}_{\mathcal{AE}}$, static fields $\mathcal{F}_{\mathcal{SF}}$, instance fields $\mathcal{F}_{\mathcal{IF}}$, local variables in static $\mathcal{F}_{\mathcal{SV}}$ and instance $\mathcal{F}_{\mathcal{IV}}$ methods, as well as, return values of static $\mathcal{F}_{\mathcal{SR}}$ and instance $\mathcal{F}_{\mathcal{IR}}$ methods. Hence, set $\mathcal{F} = \mathcal{F}_{\mathcal{I}} \cup \mathcal{F}_{\mathcal{A}} \cup \mathcal{F}_{\mathcal{AE}} \cup \mathcal{F}_{\mathcal{SF}} \cup \mathcal{F}_{\mathcal{IF}} \cup \mathcal{F}_{\mathcal{SV}} \cup \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SR}} \cup \mathcal{F}_{\mathcal{IR}}$.

$$
\begin{aligned}
\mathcal{F}_{\mathcal{I}} &\subseteq (\mathit{ClassFQN} \times \mathit{Address}) \\
\mathcal{F}_{\mathcal{A}} &\subseteq (\mathit{ClassFQN} \times \mathit{Address}) \\
\mathcal{F}_{\mathcal{AE}} &\subseteq (\mathcal{F}_{\mathcal{A}} \times \mathbb{N}) \\
\mathcal{F}_{\mathcal{SF}} &\subseteq (\mathit{ClassFQN} \times \mathit{FieldName}) \\
\mathcal{F}_{\mathcal{IF}} &\subseteq (\mathcal{F}_{\mathcal{I}} \times \mathit{FieldName}) \\
\mathcal{F}_{\mathcal{SV}} &\subseteq (\mathit{ClassFQN} \times \mathit{MethodName} \times \mathit{VarName}) \\
\mathcal{F}_{\mathcal{IV}} &\subseteq (\mathcal{F}_{\mathcal{I}} \times \mathit{MethodName} \times \mathit{VarName}) \\
\mathcal{F}_{\mathcal{SR}} &\subseteq (\mathit{ClassFQN} \times \mathit{MethodName}) \\
\mathcal{F}_{\mathcal{IR}} &\subseteq (\mathcal{F}_{\mathcal{I}} \times \mathit{MethodName})
\end{aligned}
$$

Events, $\mathcal{E}$. Depending on the kind of chopCMD instruction, HDFT++ distinguishes between different groups of operations that lead to a flow of data: assignments, arithmetic operations, and method invocations. HDFT++'s event definition does not only rely on the semantic of pure Java bytecode (like `iload` bytecode instruction) but also takes into account SSA-information provided by JOANA's intermediate program representation (cf. for instance `Label` in Listing 3.1).

Therefore, any pure Java bytecode instruction that leads to a data flow can be classified into one of those groups. Note, arithmetic operations and method invocations result only in a flow of data if their computed result values are used in another, subsequent instruction. Moreover, we intentionally exclude Java exceptions in our model because they may negatively affect the precision of our tracking results. This has to do with the fact that every operation, in particular I/O operations, may cause an exception in Java, and thus, may result in unprecise tracking results where everything flows to everything. However, SHRIFT and HDFT++ can be extended in future work to track also those kinds of flows, but at the charge of less precise tracking result.

We define a helper function *isOpaque*(c) to determine if a specific Java-class $c \in$ *ClassFQN*, or their instantiation $c \in \mathcal{F}_{\mathcal{I}}$ respectively, shall be equipped with our HDFT++ tracker (i.e. *isOpaque*(c) == *false*) or not (i.e. *isOpaque*(c) == *true*). Henceforth, we term c an *opaque-class* if it is not equipped with HDFT++. Furthermore, function *isOpaque*(c) builds the foundation to implement, realize, and increase portability within the HDFT++ approach, as it allows us to keep the tracking logic only at the program level, and e.g. independent from tracking code inside Java system classes (c.f. Figure 1.6). For instance, for the Java system class $c =$ *java.lang.Object*, *isOpaque*(c) returns `true`. The remaining of this section provides the formal description of all events that were identified to lead to a flow of data. Note, we do not provide an event definition for each single Java bytecode command, but instead, an event definition which covers and includes multiple Java bytecode commands. For instance, `BinaryAssign` covers all arithmetic bytecode commands which include, addition, subtraction, division, and multiplication. Note, the present model does not cover Java exceptions or threads.

`BinaryAssign` is triggered whenever the result of an arithmetic operation between two local variables *arg1* and *arg2* is assigned to a result variable *res*. Hence, data flows from *arg1* and *arg2* to *res*, and replaces the previous content in *res*. We model this event only for primitive-typed local variables because an arithmetic operation can not be performed on reference-typed variables.

$$\forall s \in [\mathcal{C} \rightarrow 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \rightarrow 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \{\mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}\} \rightarrow \mathcal{C}_{\mathcal{P}}]; \forall p \in \mathcal{I}; \forall arg1, arg2, res \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}} :$$
$$(\sigma, p, BinaryAssign(arg1, arg2, res), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \Longrightarrow \quad & \sigma'.s[\sigma.f(p, res) \leftarrow \sigma.s(\sigma.f(p, arg1)) \cup \sigma.s(\sigma.f(p, arg2))] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f = \sigma.f \end{aligned}$$

`UnaryAssign` is triggered whenever a unary operation is executed. In case of primitive-typed variables, this event performs either a type conversion (like the Java bytecode command `i2b` which converts an integer into a byte value), or negation (like the Java bytecode command `ineg` which negates an integer value) of the operator *arg*. Furthermore, as this operation modifies the

original value of the *arg*, the result of this operation defines, according to the SSA specification, a new variable *res* which contains the result of a unary operation.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall arg, res \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}} :$$
$$\forall \sigma.f(p, arg), \sigma.f(p, res) \in \mathcal{C}_{\mathcal{P}} : (\sigma, p, UnaryAssign(arg1, res), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s[\sigma.f(p, res) \leftarrow \sigma.s(\sigma.f(p, arg))] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f = \sigma.f \end{aligned}$$

In case of reference-typed variables, a unary operation corresponds to a typecast on variable *arg*, and therefore, variable *res* still points to the same object as *arg* after a typecast.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \ forallf \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall arg, res \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}} :$$
$$\sigma.f(p, arg), \sigma.f(p, res) \in \mathcal{C}_{\mathcal{R}} : (\sigma, p, UnaryAssign(arg1, res), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s = \sigma.s \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p, res) \leftarrow \sigma.f(p, arg)] \end{aligned}$$

`ReadArray` is triggered whenever an element at index i is read from an array *arr* and is assigned to a local variable *var*. For primitive-typed values the content of the array element (arr, i) is copied to *var*

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall arr \in \mathcal{F}_{\mathcal{A}}; \forall i \in \mathbb{N}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$$
$$\forall \sigma.f(p,(arr, i)), \sigma.f(p, var) \in \mathcal{C}_{\mathcal{P}} : (\sigma, p, ReadArray(arr, i, var), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s[\sigma.f(p, var) \leftarrow \sigma.s(\sigma.f(p,(arr, i)))] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f = \sigma.f \end{aligned}$$

In case of reference-typed variables variable *var* points to the same object as (arr, i)

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall arr \in \mathcal{F}_{\mathcal{A}}; \forall i \in \mathbb{N}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$$
$$\forall \sigma.f(p,(arr, i)), \sigma.f(p, var) \in \mathcal{C}_{\mathcal{R}} : (\sigma, p, ReadArray(arr, i, var), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s = \sigma.s \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p, var) \leftarrow \sigma.f(p,(arr, i))] \end{aligned}$$

`WriteArray` is triggered whenever a local variable *var* is assigned to an array *arr* at index i. HDFT++ models *arr* and each array element (arr, i) as separated containers and link them via

an alias mapping. For a primitive-typed variable *var* this mapping is fixed, and thus, the content of an array element (*arr*, *i*) is overwritten with the content of *var*.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall arr \in \mathcal{F}_{\mathcal{A}}; \forall i \in \mathbb{N}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$$
$$\forall \sigma.f(p,(arr,i)), \sigma.f(p,var) \in \mathcal{C}_{\mathcal{P}} : (\sigma, p, WriteArray(arr,i,var), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s[\sigma.\mathcal{F}p,(arr,i) \leftarrow \sigma.\mathcal{F}\sigma.f(p,var)] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f = \sigma.f \end{aligned}$$

In case of reference-typed variables, HDFT++ models each array element as an alias relation to *var*. For doing so, HDFT++ first deletes all possibly existing alias mappings for (*arr*, *i*), except the mapping with *arr*, and afterwards, create a new alias mapping between (*arr*, *i*) and *var*.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall arr \in \mathcal{F}_{\mathcal{A}}; \forall i \in \mathbb{N}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$$
$$\forall \sigma.f(p,(arr,i)), \sigma.f(p,var) \in \mathcal{C}_{\mathcal{R}} : (\sigma, p, WriteArray(arr,i,var), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s = \sigma.s \\ \wedge \quad & \sigma'.l[\sigma.f(p,arr) \leftarrow \sigma.f(p,var) \cup \sigma.l(\sigma.f(p,arr)) \setminus \sigma.f(p,(arr,i))] \\ \wedge \quad & \sigma'.f[(p,(arr,i)) \leftarrow \sigma'.f(p,var)] \end{aligned}$$

`ReadField` is triggered whenever a static or instance field *fld* is read and their content is assigned to a local variable *var*. After the assignment variable *var* either contains a copy of the field data

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall fld \in \mathcal{F}_{\mathcal{IF}} \cup \mathcal{F}_{\mathcal{SF}}; \forall p \in \mathcal{I}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$$
$$\forall \sigma.f(p,fld), \sigma.f(p,var) \in \mathcal{C}_{\mathcal{P}} : (\sigma, p, ReadField(fld,var), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s[\sigma.f(p,var) \leftarrow \sigma.s(\sigma.f(p,fld))] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f = \sigma.f \end{aligned}$$

or variable *var* points to the same container as *fld*

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall fld \in \mathcal{F}_{\mathcal{IF}} \cup \mathcal{F}_{\mathcal{SF}}; \forall p \in \mathcal{I}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$$
$$\forall \sigma.f(p,fld), \sigma.f(p,var) \in \mathcal{C}_{\mathcal{R}} : (\sigma, p, ReadField(fld,var), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s = \sigma.s \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p,var) \leftarrow \sigma.f(p,fld)] \end{aligned}$$

`WriteField` is triggered whenever the content of a local variable *var* is assigned to a static or instance field *fld*. Here we also have to distinguish between primitive- and reference-typed

variables, as depending on its type *fld* either contains the same value or points to the same data as *var*. For primitive-typed variables, HDFT++ assumes that the alias relationship between *o* and *fld* is already established as the field container stays the same.

$\forall s \in [\mathcal{C} \rightarrow 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \rightarrow 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \rightarrow \mathcal{C}]; \forall p \in \mathcal{I}; \forall fld \in \mathcal{F}_{\mathcal{IF}} \cup \mathcal{F}_{\mathcal{SF}}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$
$\forall \sigma.f(p, fld), \sigma.f(p, var) \in \mathcal{C}_{\mathcal{P}} : (\sigma, p, WriteField(fld, var), \sigma') \in \mathcal{R}$

$$\begin{aligned} \Longrightarrow \quad & \sigma'.s[\sigma.f(p, f\,l\,d) \leftarrow \sigma.s(\sigma.f(p, var))] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f = \sigma.f \end{aligned}$$

Furthermore, if *fld* belongs to an instance of an opaque class *o*, i.e. $isOpaque(o) == true$, then HDFT++ adds an alias relation between *o* and *fld*; this makes it easier to get all data from the opaque object.

$\forall s \in [\mathcal{C} \rightarrow 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \rightarrow 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \rightarrow \mathcal{C}]; \forall p \in \mathcal{I}; \forall fld \in \mathcal{F}_{\mathcal{IF}} \cup \mathcal{F}_{\mathcal{SF}}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}};$
$\forall \sigma.f(p, fld), \sigma.f(p, var) \in \mathcal{C}_{\mathcal{R}} : (\sigma, p, WriteField(fld, var), \sigma') \in \mathcal{R}$

$$\begin{aligned} \Longrightarrow \quad & \sigma'.s = \sigma.s \\ \wedge \quad & \sigma'.l[\sigma.f(p, fld) \leftarrow \sigma.l(\sigma.f(p, fld)) \cup \sigma.f(p, o)]_{o \in \{\mathcal{F}_{\mathcal{I}} | isOpaque(o)\}} \\ \wedge \quad & \sigma'.f[(p, fld) \leftarrow \sigma.f(p, var)] \end{aligned}$$

`CallInstanceMethod` is triggered whenever an instance method is invoked. Method parameters are passed via a stack frame on the operand stack into the method. HDFT++ distinguishes between actual variable identifiers $as \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}$ and the formal identifiers $ps \in \mathcal{F}_{\mathcal{IV}}$. The former ones are used inside the caller method to pass values into the method, whereas the latter ones are identifiers which are used inside the called method. When a method returns its corresponding stack frame is removed from the stack. What possibly remains on the stack is the return value. In a broader sense, HDFT++ considers passing variables into a method as a kind of variable assignment between formal and actual parameters of a method invocation, and thus, also distinguishes between primitive- and reference-typed variables: *ps* either takes a copy of its corresponding *as* or points to the same data as *as*. If the class of the callee object *o* is opaque HDFT++ adds an alias relation from parameter *ps* to *o* because HDFT++ does not have any information about how data is further distributed inside the callee object.

$\forall s \in [\mathcal{C} \rightarrow 2^{\mathcal{D}}], \forall l \in [\mathcal{C} \rightarrow 2^{\mathcal{C}}], \forall f \in [\mathcal{I} \times \mathcal{F} \rightarrow \mathcal{C}], \forall p \in \mathcal{I}, \forall o \in \mathcal{F}_{\mathcal{I}}, \forall ps \in \mathcal{F}_{\mathcal{IV}}, \forall as \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}} :$
$(\sigma, p, CallInstanceMethod(o, as, ps), \sigma') \in \mathcal{R}$

$$\begin{aligned} \Longrightarrow \quad & \sigma'.s[\sigma.f(p, ps) \leftarrow \sigma.f(\sigma.f(p, as))]_{\sigma.f(p,as),\ \sigma.f(p,ps) \in \mathcal{C}_{\mathcal{P}}} \\ \wedge \quad & \sigma'.l[\sigma.f(p, ps) \leftarrow \sigma.l(\sigma.f(p, ps)) \cup \sigma.f(p, o)]_{isOpaque(o)} \\ \wedge \quad & \sigma'.f[(p, ps) \leftarrow \sigma.f(p, as)]_{\sigma.f(p,as),\ \sigma.f(p,ps) \in \mathcal{C}_{\mathcal{R}}} \end{aligned}$$

`CallStaticMethod` is triggered whenever a static method is invoked. Formally, it is similar to `CallInstanceMethod` except that there is no callee object o. Thus, adding aliases from the object to the parameters is not needed.

$$\begin{aligned}
&\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall ps \in \mathcal{F}_{\mathcal{SV}}; \forall as \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}} : \\
&(\sigma, p, \mathit{CallStaticMethod}(as, ps), \sigma') \in \mathcal{R} \\
&\quad \sigma'.s[\sigma.f(p,ps) \leftarrow \sigma.s(\sigma.f(p,as))]_{\sigma.f(p,as),\sigma.f(p,ps) \in C_P} \\
\implies &\quad \sigma'.l = \sigma.l \\
&\quad \sigma'.f[(p,ps) \leftarrow \sigma.f(p,as)]_{\sigma.f(p,as),\sigma.f(p,ps) \in C_R}
\end{aligned}$$

Passing return values back from a callee method to a caller method are modeled in HDFT++ with two separated events: `PrepareMethodReturn` and `ReturnInstanceMethod`, or respectively `ReturnStaticMethod` for static method invocations. For the sake of simplicity, we consider only the case where return values are assigned to local variables inside the caller method. Assignments to object fields are modeled with two consecutive events: `ReturnInstanceMethod` followed by a `WriteField`.

`PrepareMethodReturn` is triggered at the end of a method just before it returns. At this point, the value of the method's local variable *var* is temporarily stored into an intermediate return-variable *ret* and provided as a copy or a reference to the caller-method, where it is possibly assigned to another variable.

$$\begin{aligned}
&\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}; \forall ret \in \mathcal{F}_{\mathcal{IR}} \cup \mathcal{F}_{\mathcal{SR}}; \\
&\forall \sigma.f(p,var), \sigma.f(p,ret) \in \mathcal{C}_{\mathcal{P}} : (\sigma, p, \mathit{PrepareMethodReturn}(var, ret), \sigma') \in \mathcal{R} \\
\implies &\quad \sigma'.s[\sigma.f(p,ret) \leftarrow \sigma.s(\sigma.f(p,var))] \\
&\wedge \quad \sigma'.l = \sigma.l \\
&\wedge \quad \sigma'.f = \sigma.f
\end{aligned}$$

In case of reference-typed return values, the formal definition is as follows

$$\begin{aligned}
&\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}; \forall ret \in \mathcal{F}_{\mathcal{IR}} \cup \mathcal{F}_{\mathcal{SR}}; \\
&\forall \sigma.f(p,var), \sigma.f(p,ret) \in \mathcal{C}_{\mathcal{R}} : (\sigma, p, \mathit{PrepareMethodReturn}(var, ret), \sigma') \in \mathcal{R} \\
\implies &\quad \sigma'.s = \sigma.s \\
&\wedge \quad \sigma'.l = \sigma.l \\
&\wedge \quad \sigma'.f[(p,ret) \leftarrow \sigma.f(p,var)]
\end{aligned}$$

`ReturnInstanceMethod` is triggered after a method execution m on instance o returns to its caller-method and the method's stack frame is popped from the operand stack. HDFT++

models that by emptying all local variables $lvar \in \mathcal{F}_{\mathcal{IV}}$, including primitive- and reference-typed, that were instantiated inside method m. Additionally, a possible return value *ret* is assigned to a local variable *var* inside the caller method. In case $isOpaque(o) == \texttt{false}$ this event is modeled as

$$
\begin{aligned}
&\forall s \in [\mathcal{C} \rightarrow 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \rightarrow 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \rightarrow \mathcal{C}]; \forall ret \in \mathcal{F}_{\mathcal{IR}}; \forall m \in MethodName; \forall lvar \in \mathcal{F}_{\mathcal{IV}}; \\
&\forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}; \forall p \in \mathcal{I}, \forall o \in \mathcal{F}_{\mathcal{I}} : (\sigma, p, ReturnInstanceMethod(o, m), \sigma') \in \mathcal{R} \\
\Longrightarrow \quad & \sigma'.s[\sigma.f(p, var) \leftarrow \sigma.s(\sigma.f(p, ret)); f(p, lvar) \leftarrow \emptyset]_{\sigma.f(p,var), \sigma.f(p,lvar), \sigma.f(p,ret) \in \mathcal{C}_{\mathcal{P}}} \\
\wedge \quad & \sigma'.l = \sigma.l \\
\wedge \quad & \sigma'.f[(p, var) \leftarrow \sigma.f(p, ret); (p, lvar) \leftarrow \emptyset]_{\sigma.f(p,var), \sigma.f(p,lvar), \sigma.f(p,ret) \in \mathcal{C}_{\mathcal{R}}}
\end{aligned}
$$

Return values from opaque objects o are modeled differently, i.e. $isOpaque(o) == \texttt{true}$: as HDFT++ do not track data flows inside o, HDFT++ can not determine which data is flowed into *ret*, either passed via method parameters *ps* or instance fields *fld*. Therefore, HDFT++ models the connection between *ps*, *fld*, and o as an alias relationship (cf. `WriteField` and `CallInstanceMethod`) and assign all data items of o, as well as from its aliases, to *var*.

$$
\begin{aligned}
&\forall s \in [\mathcal{C} \rightarrow 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \rightarrow 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \rightarrow \mathcal{C}]; \forall ret \in \mathcal{F}_{\mathcal{IR}}; \forall m \in MethodName; \\
&\forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}; \forall p \in \mathcal{I}; \forall o \in \mathcal{F}_{\mathcal{I}} : (\sigma, p, ReturnInstanceMethod(o, m), \sigma') \in \mathcal{R} \\
\Longrightarrow \quad & \sigma'.s[\sigma.f(p, var) \leftarrow \sigma.s(o) \cup \sigma.s(c)]_{c \in \{\mathcal{C} | o \in \sigma.l(c)\}} \\
\wedge \quad & \sigma'.l = \sigma.l \\
\wedge \quad & \sigma'.f = \sigma.f
\end{aligned}
$$

`ReturnStaticMethod` is triggered after the invocation of a static method m returns to its caller method. It slightly differs to the `ReturnInstanceMethod`, as there is a callee class $cl \in$ *ClassFQN* instead of a callee object o. Furthermore, as we do not model data flows within opaque classes, i.e. $isOpaque(cl) == true$, we directly propagate data from a method's actual parameters *prm* to its return value $ret = (cl, m) \in \mathcal{F}_{\mathcal{SR}}$, and hence, possibly to a local variable *var* that receives the return value inside the caller method. *lvar* are all those variables that are created and used during the execution of m.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}; \forall ret \in \mathcal{F}_{\mathcal{SR}};$$
$$\forall prm, lvar \in \mathcal{F}_{\mathcal{SV}}; \forall m \in MethodName; \forall cl \in ClassFQN : (\sigma, p, ReturnStaticMethod(cl, m), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s[\sigma.f(p, var) \leftarrow \sigma.s(\sigma.f(p, prm))]_{\sigma.f(p,prm) \in \mathcal{C}_{\mathcal{P}}} \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p, var) \leftarrow \sigma.f(p, prm)]_{\sigma.f(p,prm) \in \mathcal{C}_{\mathcal{R}}} \end{aligned}$$

In case $isOpaque(cl) == false$, we know that a precedent `PrepareMethodReturn` has happened, and therefore, either copy the content of variable *ret* into the container of *var* (for primitive-typed values) or remap the *var* to the same container as *ret* (for reference-typed values).

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}]; \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}]; \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}]; \forall p \in \mathcal{I}; \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}; \forall ret \in \mathcal{F}_{\mathcal{SR}};$$
$$\forall prm, lvar \in \mathcal{F}_{\mathcal{SV}}; \forall m \in MethodName; \forall cl \in ClassFQN : (\sigma, p, ReturnStaticMethod(cl, m), \sigma') \in \mathcal{R}$$
$$\begin{aligned} \implies \quad & \sigma'.s[\sigma.f(p, var) \leftarrow \sigma.s(\sigma.f(p, ret)); \sigma.f(p, lvar) \leftarrow \emptyset]_{\sigma.f(p,var), \sigma.f(p,lvar), \sigma.f(p,ret) \in \mathcal{C}_{\mathcal{P}}} \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p, var) \leftarrow \sigma.f(p, ret); (p, lvar) \leftarrow \emptyset]_{\sigma.f(p,lvar), \sigma.f(p,ret) \in \mathcal{C}_{\mathcal{R}}} \end{aligned}$$

`Source` is triggered whenever a Java application reads data from the outside into its process memory via a file (*IO*), network (*NET*), or database (*DB*) read-operation. Thus, HDFT++ defines the set $\texttt{TYPE} = \{\texttt{IO}, \texttt{NET}, \texttt{DB}\}$. Typically, a source reads data from different locations *LOC* that are uniquely identified by *locid* (e.g. an absolute file-path/-name, or a remote system's ip and port). To use data *dat* from a source, an application must assign *dat* to a reference variable *var*. HDFT++ uses a helper function $getInputCont : TYPE \times LOC \mapsto \mathcal{C}_{\mathcal{R}}$ that provides for a *TYPE* and *LOC* a container $\mathcal{C}_{\mathcal{R}}$ for subsequent processing.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}], \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}], \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}], \forall p \in \mathcal{I}, \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}},$$
$$\forall f(p, var) \in \mathcal{C}_{\mathcal{R}}, \forall t \in TYPE :$$
$$\begin{aligned} (\sigma, p, Source(t, locid, var), \sigma') \in R \implies \quad & \sigma'.s[\sigma.f(p, var) \leftarrow dat id] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p, var) \leftarrow getInputCont(t, locid)] \end{aligned}$$

`Sink` is triggered whenever data *d* is transmitted from a variable *var* to a location *locid* outside of a process memory. This happens via a write-operation of type $t \in TYPE$. To protocol where

data flows to the outside, HDFT++ creates a container c, identified by $(t, locid)$, and map it to d.

$$\forall s \in [\mathcal{C} \to 2^{\mathcal{D}}], \forall l \in [\mathcal{C} \to 2^{\mathcal{C}}], \forall f \in [\mathcal{I} \times \mathcal{F} \to \mathcal{C}], \forall p \in \mathcal{I}, \forall var \in \mathcal{F}_{\mathcal{IV}} \cup \mathcal{F}_{\mathcal{SV}}, \forall f(p, var) \in \mathcal{C}_{\mathcal{R}} :$$

$$\begin{aligned}(\sigma, p, Sink(var, t, locid), \sigma') \in R \implies \quad & \sigma'.s[\sigma.f(p,(t, locid)) \leftarrow \sigma.s(\sigma.f(p, var))] \\ \wedge \quad & \sigma'.l = \sigma.l \\ \wedge \quad & \sigma'.f[(p,(t, locid)) \leftarrow c]\end{aligned}$$

3.6. Evaluation

This section illustrates and describes our conducted experiments and provides its evaluation results in two dimensions: *precision* and *performance*. We have used a test-suite of different Java-based applications for our experiments, standalone- and web-applications in particular: *JZip* and *JFTP* are standalone-applications to zip and to transmit files via FTP respectively. *BirthdayApp* is a Facebook-based web-application that fetches and displays birthday data from a Facebook user and its friends in a calendar. For doing so, it authenticates a user to and queries its birthday data from Facebook. *SnipSnap* and *PersonalBlog* are web-applications for blogger to post and to publish blog-entries on a personal website, like an electronic diary. To save blog entries, those web-applications are additionally backed up with a database storage. JZip and BirthdayApp are applications that have been implemented by a student during his bachelor thesis, whereas the others are taken from the *Stanford SecuriBench* [1] repository. Note, BirthdayApp and PersonalBlog are pretty similar to those from our running example in section 1.2, except that they are using a local database instead of a cloud storage service to store their data.

Based on our test-suite we address in Section 3.6.1 Caveat 1 and evaluate how much more precise the present approaches are compared to a pure *black-box* approach (cf. Figure 1.4a). Moreover, we provide a case study in Section 3.6.2 and show, along the lines of Java code samples, the benefits of HDFT++ compared to SHRIFT. Note, we do not provide a general metric to characterize and quantify HDFT++'s strengths, as this highly depends on actual run-time values, and thus, on the actually taken control flow path through an application.

Finally, as our hybrid approach and its prototype implementation injects additional instructions into the AuS, Section 3.6.3 addresses Caveat 2 and evaluates and reveals the *performance* run-time overhead, that our approaches impose compared to the AuS's native execution. Moreover, performance results are benchmarked and compared with PHOSPHOR [10] and LIBDFT [54], two pure dynamic data flow tracker. Besides that, to show the applicability of our solution, we also conducted experiments on pure Java standalone-applications.

[1] `https://suif.stanford.edu/~livshits/securibench/`

3.6.1. Precision

This section addresses Caveat 1 and conducts experiments to answer the research question RQ1

> How can we improve the tracking precision, and thus, reduce overapproximation of the tracking results compared to a pure black-box approach?

In first place, we are interested in the precision gain of SHRIFT and HDFT++ compared to a pure black-box approach. By construction, SHRIFT and its extension HDFT++ cannot be less precise than treating the AuS as a black-box where every output contains every input read so far (cf. Figure 1.4a). Compared to a pure dynamic data flow tracker which considers only a single execution trace at a time, static information flow analysis takes into account *all* possible execution traces at once, and thus, is able to detect and analyze explicit and implicit dependencies between sources and sinks. If at least one execution trace leads to a flow of data then the sink statically depends on the source. For instance, static analysis reports for the code in Listing 3.4, that the sink at line 8 depends on the source at line 1, even considering explicit flows only.

In general, however, it is hard to quantify such precision gain. Considering that a black-box approach would always be as precise as our approach, where every source is connected to every sink, a possible metric for precision improvement could be *the number of source-to-sink connections that can be safely discarded*, thanks to static analysis. Let *#Flows* denote the number of statically computed dependencies between sinks and sources, we formally define precision gain compared to a black-box tracking as [71]

$$Precision := 1 - (\#flows / (\#sources \times \#sinks)) \qquad \text{(Equation 3.1)}$$

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; at the moment of writing, the author is not aware of any better metric to measure precision of static analysis with reference to dynamic monitoring. Note, our precision metric has a reduction notion and must not be misunderstood with a prediction accuracy.

As reported in Table 3.1, we statically analyzed each AuS in our test-suite with various *points-to-analysis* (0-1-CFA [36], 1-CFA, object-sensitivity, cf. Section 3.3), considering only *explicit (D)* and additional *implicit (DI)* information flows; multi-column "(D)/(DI)" reports our results, whereas the left-hand side of the slash-symbol illustrates the measured values considering explicit flows

Table 3.1.: Static analysis results with different points-to configurations. The number of detected sources, sinks, and flows are listed in the columns "#Src./#Snk." and "#Flows", whereas "Precision" shows the achieved precision in % according to Equation 3.1. All values within the multi-column "(D/DI)" are reported considering only explicit (D) and additional implicit (DI) information flows.

AuS	Points-To	#Src./#Snk.	(D/DI)	
			#Flows	Precision
JZip	0-1-CFA	33 / 56	192 / 666	89.6 / 64.0
	1-CFA	10 / 55	135 / 213	75.5 / 61.3
	obj.-sens.	10 / 55	115 / 186	79.1 / 66.2
JFTP	0-1-CFA	7 / 4	6 / 22	78.6 / 21.4
	1-CFA	7 / 4	4 / 21	85.7 / 25.0
	obj.-sens.	7 / 4	6 / 19	78.6 / 32.1
BirthdayApp	0-1-CFA	13 / 99	98 / 646	92.3 / 51.8
	1-CFA	13 / 45	20 / 229	96.6 / 60.9
	obj.-sens.	13 / 45	18/229	96.9 / 60.9
PersonalBlog	0-1-CFA	8 / 37	162 / 258	45.3 / 12.8
	1-CFA	8 / 32	136 / 183	46.9 / 28.5
	obj.-sens.	8 / 32	56 / 167	78.1 / 34.8
SnipSnap	0-1-CFA	108 / 101	2554 / 8060	76.6 / 26.1
	1-CFA	108 / 95	848 / 6229	91.7 / 39.3
	obj.-sens.	108 / 95	291 / 5964	97.2 / 41.9

(D) only, and the right-hand side additional implicit (DI) flows. According to Equation 3.1, the achieved precision gain compared to a pure black-box approach (cf. column "precision") varies between 58% and 89.61% for JZip, between 66.47% and 77.01% for JFTP, between 60.85% and 96.92% for BirthdayApp, between 34.76% and 46.87% for PersonalBlog, and between 41.87% and 71% for SnipSnap, depending on the configuration. These numbers also confirm that taking into account additional implicit flows leads to more sink-source dependencies, and thus according to our metric in Equation 3.1, to a result that is less precise than considering only explicit dependencies. Because of that, we use analysis-reports with a focus on explicit flows only for our experiments in Section 3.6.3 and Section 3.6.2. Although some of these analyses are incomparable in theory, object-sensitivity tends to deliver more precision, as already reported for various client analyses [74]. This effect is also partially observable here, at least for the configurations in which indirect flows are ignored (D). The reason is that the points-to analysis result is mainly used to compute data dependencies and has only a limited effect on the control dependencies. For instance, for JFTP object-sensitivity even results in worse precision than 1- and 0−1-CFA. Note, these numbers are hard to relate to dynamic values because they depend on the specific AuS and do not take into account how many times a certain source- or sink-instruction is executed at run-time. Furthermore, our proposed metric in Equation 3.1 is only based on parameters that are statically collected from the static analysis phase and does not take into account run-time data and values. Therefore, we argue that our reported precision values build the lowest threshold for precision improvement compared to a black-box approach.

Although these experiments are primarily devoted to investigate the precision gain compared to a pure black-box approach, where sensitive inputs are conservatively propagated to all outputs, we also measured other dimensions of our static information flow analysis experiments in order to get an intuition about the computational-power and -resources that are required to run the experiments. Concretely, we report in Table 3.2 the required time in minutes to build the SDG (column "TSDG") and to run the analysis (column "TANA", which also includes the time to build the SDG), as well as, the occupied memory size in GB (column "Memory"). As Table 3.2 illustrates, the analysis time and required memory size highly varies between the different points-to configurations and also depends on considering implicit flows or not. On closer inspection, we can observe that generating the SDG graph (column "TSDG"), which is the base to run the static information flow analysis, takes the bulk of the total analysis time "TANA" for the majority of experiments. In all experiments, building the SDG requires between $\sim 75\%$ – $\sim 99\%$ of the total analysis time. For instance, even for small applications, like JZip, the analysis time varies between 0.43 min. – 6.11 min., whereof 5 out of 6 experiments require more than $> 75\%$ of the total analysis time for the SDG generation while the rest is spent on slicing and running the

Table 3.2.: Static analysis results for different points-to configurations. All values within the multi-column “(D/DI)” are reported considering only explicit (D) and additional implicit (DI) information flows. The columns “TANA” and “Memory" show the required time in minutes (which also includes the time to build the SDG, cf. column “TSDG”) and the required memory consumption in GB to run the analysis. Columns “TSDG”, “# SDG-edges” and “# SDG-nodes” illustrate the required time in minutes to build the SDG, the number of generated SDG-edges and SDG-nodes respectively.

AuS	Points-To	(D/DI)				# SDG-nodes
		TANA	Memory	TSDG	# SDG-edges	
JZip	0-1-CFA	0.43 / 0.48	0.80 / 0.78	0.32 / 0.26	755285 / 828403	61417
	1-CFA	1.67 / 1.39	3.02 / 3.04	1.64 / 1.33	1420140 / 1510753	95687
	obj.-sens.	6.07 / 6.11	1.35 / 0.99	6.01 / 6.05	1381996 / 1467541	142396
JFTP	0-1-CFA	15.76 / 11.74	13.07 / 13.07	15.52 / 11.63	15847564 / 17927207	418264
	1-CFA	348.29 / 214.04	61.43 / 60.99	348.16 / 212.6	115121370 / 129436286	2180511
	obj.-sens.	7.61 / 6.30	10.98 / 10.97	7.08 / 5.79	7637990 / 10647881	711901
BirthdayApp	0-1-CFA	756.46 / 408.10	83.09 / 92.87	754.79 / 404.17	755285 / 828403	873757
	1-CFA	63.60 / 40.13	26.31 / 26.65	63.42 / 39.57	42772213 / 46438973	650231
	obj.-sens.	124.17 / 104.00	73.66 / 79.07	123.89 / 102.59	176754894 / 180293272	802674
PersonalBlog	0-1-CFA	7819.04 / 3900.29	191.95 / 163.58	7777.39 / 3871.10	266702264 / 281122765	1471277
	1-CFA	151.73 / 90.20	30.88 / 31.58	150.82 / 89.10	41411923 / 44749831	744892
	obj.-sens.	907.63 / 553.74	105.58 / 88.87	905.14 / 549.32	116561672 / 138294911	1987404
SnipSnap	0-1-CFA	4698.94 / 3322.28	179.68 / 166.43	2997.90 / 1506.51	277905089 / 295487804	1750316
	1-CFA	224.32 / 165.79	55.48 / 80.21	219.76 / 154.56	75797664 / 91331507	1562335
	obj.-sens.	2333.33 / 2595.56	232.42 / 256.36	2328.78 / 2528.05	304736893 / 426257053	4322479

information flow analysis algorithm. Depending on the chosen points-to analysis, the SDG size determined by the number of nodes and edges (column # SDG-nodes and # SDG-edges) varies tremendously: for instance, for JZip the graph size exhibits between $60 * 10^3$ and 10^5 nodes, as well as, between $75.5 * 10^5$ and $1.3 * 10^6$ edges, and thus directly affects the total analysis time. As expected, all experiments taking into account implicit flows (DI) generate more SDG-edges than considering explicit flows only (D), and hence, results in more dependencies between sources and sinks. In the (D) configuration, SDGs have between 11% and 34% fewer edges than in the respective (DI) configuration. In sum, we have observed that the chosen points-to analysis tremendously influences the SDG size, and thus directly affects the total analysis time, and SDG's build time in particular. Moreover, even for small applications, like JZip, the required memory resources are non-negligible: calling a simple Java system class from the application code involves an avalanche of other related Java classes which have to be taken into account for the analysis.

3.6.2. SHRIFT versus HDFT++

As SHRIFT and HDFT++ reuse a statically pre-computed analysis-report (cf. Section 3.3) to track only those data flow dependencies that actually lead to a flow of data, both approaches exhibit at least a better tracking precision than a pure black-box approach (cf. Section 3.6.1). However, their run-time behavior is different. SHRIFT monitors only those sink- and source-instructions which have a data flow dependency between each other (cf. Section 3.4), whereas HDFT++ additionally tracks intermediate instructions which reside on a dependency path. This way, HDFT++ is able to differentiate more accurately if a data flow actually happens or not at run-time. For instance, consider the flow of data from line 1 to line 8 in listing 3.4: irrespective of the `condition` value in line 3, SHRIFT always reports the flow of data even if the assignment in Line 4 does not happen. In contrast, HDFT++ reports the flow of data only if the `condition` value is `true` at run-time, because then the assignment in line 4 is executed; otherwise, no flow of data is reported by HDFT++.

However, assessing and quantifying the precision gain of HDFT++ compared to SHRIFT in general is non trivial. Depending on actual run-time values only a subset of chop instructions may be executed on a data flow dependency, and thus, would make a dependency path critical, so that data can flow from a source to a critical sink. In this section, we examine the benefits of HDFT++ compared to SHRIFT along the lines of different Java source-code samples which take different randomly generated input data. We show further, how the PIP-state evolves with HDFT++ and SHRIFT and illustrate when HDFT++ provides a more accurate tracking result. For the sake of illustration, we always show the whole PIP state with all mapping entries which

are created at run-time. Although someone may argue that not all PIP state entries are actually required to take a policy decision, we retort that all the other entries are needed to follow the dependency chain from a source to a sink. As the PIP state may grow quite fast after several runs, a proper extension mechanism which sanitizes the PIP state from unneeded mappings could be implemented in future work.

For the sake of simplicity, we use dedicated code samples to make HDFT++'s precision gain more tangible. We do not claim that our chosen code samples are complete, but to the best of our knowledge they reflect common code structures, like branch- and loop-commands or inheritance relations, which are typically used inside applications. All code samples in this section have been statically analyzed with object-sensitiveness enabled and targeting explicit flows only (cf. Section 3.6.1 and Section 3.6.3 for the reasoning).

Sample 1: Branch instruction

Our first code sample in Listing 3.9 shows a data flow dependency between the `sink`- (line 11) and the `source`-method (line 3). The assignment instruction '`out=in`' in line 6 is the main reason for the existence of this data flow dependency. Depending on the actual run-time size of the byte-array `out` (which is randomly determined in line 2) the assignment instruction is executed or not, and thus, either transfers the data item in a copy-by-reference manner from the `source` to the `sink` or not.

Listing 3.9: Data flow dependency crosses a `IF`-command

```
1  public static void main(String[] args) {
2    int size  = new Random().nextInt(10);
3    byte[] in = source("Test").getBytes();
4    byte[] out = new byte[size];
5    if(in.length < out.length){
6      out = in;
7    }
8    else {
9      out = "DoNothing".getBytes();
10   }
11   sink(new String(out));
12 }
13
14 public static String source(String s){return s;}
15 public static void sink(String s){System.out.println(s);}
```

Listing 3.10 shows the corresponding analysis-report which is generated during the static analysis phase for Listing 3.9 (cf. Section 3.3). As expected, it lists the data flow dependency between the `source`- and the `sink`-method (sink-source-pairs in line 16 and line 17), as well

as the corresponding chop nodes (chopNode-tags, line 18 – line 35) which belongs to that dependency path. Each chop node provides information about its unique program location inside the AuS, which is identified by the bytecode offset bci and the fully-qualified method name om (because of limited space reason, we use a shortened version of the fully-qualified method name and omit the Java package information).

Listing 3.10: Static analysis-report for the IF-example in Listing 3.9.

```
1  <source>
2    <id>Source0</id>
3    <location>..main(.)V:38</location>
4    <signature>..source(Ljava/lang/String;)Ljava/lang/String;</signature>
5    <return/>
6  </source>
7  ...
8  <sink>
9    <id>Sink0</id>
10   <location>..main(.)V:76</location>
11   <signature>..sink(Ljava/lang/String;)V</signature>
12   <param index="1"/>
13 </sink>
14 ...
15 <flows>
16   <sink id="Sink0">
17     <source id="Source0"/>
18     <chop>
19       <chopNode bci="38" lab="v20 = source(#(Test))" om="..main(.)V"/>
20       <chopNode bci="41" lab="v22 = v20.getBytes()" om="..main(.)V"/>
21       <chopNode bci="44" lab="v23 = v22" om="..main(.)V"/>
22       <chopNode bci="46" lab="v24 = new []" om="..main(.)V"/>
23       <chopNode bci="48" lab="v25 = v24" om="..main(.)V"/>
24       <chopNode bci="50" lab="v26 = v22.length" om="..main(.)V"/>
25       <chopNode bci="52" lab="v27 = v24.length" om="..main(.)V"/>
26       <chopNode bci="53" lab="if (v26 >;= v27) goto 63" om="..main(.)V"/>
27       <chopNode bci="57" lab="v32 = v22" om="..main(.)V"/>
28       <chopNode bci="58" lab="goto 68" om="..main(.)V"/>
29       <chopNode bci="63" lab="v30 = #(DoNothing).getBytes()" om="..main(.)V"/>
30       <chopNode bci="66" lab="v31 = v30" om="..main(.)V"/>
31       <chopNode bci="68" lab="v34 = new java.lang.String" om="..main(.)V"/>
32       <chopNode bci="73" lab="v34.<init>;(v33)" om="..main(.)V"/>
33       <chopNode bci="76" lab="sink(v34)" om="..main(.)V"/>
34       <chopNode bci="-8" lab="PHI v33 = v22, v30" om="..main(.)V"/>
35     </chop>
36   </sink>
37 </flows>
```

Moreover, it also provides the Java command statement in JOANA's intermediate SSA representation language (cf. lab-attribute). Based on the analysis-report in Listing 3.10, we monitor

the code in Listing 3.9 with HDFT++ and SHRIFT to investigate how the PIP-state evolves when the `sink`-method in line 6 is reached. In particular, we are interested in the PIP-states when the assignment instruction in Listing 3.9 line 6 is executed and also when it is not executed.

Listing 3.11 shows the excerpt from HDFT++'s PIP-state if line 6 in Listing 3.9 is executed, i.e. in this case the byte-array size of variable `out` is greater than the byte-array size of variable `in` in Line 3, and therefore the assignment instruction in Listing 3.9 line 6 is executed. Each line shows a simplified mapping between the naming identifier $\mathcal{F}$ (left-hand side) and the data identifier $\mathcal{D}$ (right-hand side). For the sake of simplicity, this mapping representation omits the container identifier $\mathcal{C}$ and instead directly shows which naming identifiers point to which data items inside the PIP-state. Because of limited space reason, we only show the last tail of the naming identifiers and omit the fully qualified naming terms, as described in Section 3.5.

Listing 3.11: HDFT++'s PIP-state with executed assignment instruction in Listing 3.9 Line 6

```
1  ..Source0                          ---> Source0
2  ..3863861312|main(.)V|v20          ---> Source0
3  ..3863861312|main(.)V|v22          ---> Source0
4  ..3863861312|main(.)V|v33          ---> Source0
5  ..3863861312|main(.)V|v34          ---> Source0
6  ..Sink0                            ---> Source0
```

HDFT++ derives from the analysis-report that the `source` method is called inside the method `..main(..)V` at bytecode-offset 38 (cf. XML-attribute `bci` in Listing 3.10 line 19) and that its return value is assigned to `v20` which is an identifier at JOANA's SSA representation language. This fact is tracked by HDFT++ in Listing 3.11 line 2; note, `Source0` is a unique identifier inside the PIP to identify data that originates from the `source` method call. Next, when the `source`-method's return value is transformed into a byte-array by the `getBytes()` method call in Listing 3.9 line 3, HDFT++ deduces from the corresponding chop node in Listing 3.10 line 20 that this method is called on an object identified by `v20` and that the return value is assigned to `v22`. At that point, the PIP state reveals that the identifier `v20` is pointing to `Source0`, and thus, the method call `getBytes()` may transfer `Source0` to `v20` as well. Because of that, HDFT++ updates the PIP-state with line 3 in Listing 3.11. Line 4 and line 5 in Listing 3.11 are the most crucial mapping-entries in the PIP-state. According to Listing 3.10 line 33, identifier `v34` represents the passed parameter to the `sink`-method at JOANA's intermediate language, and thus, corresponds to the `String`-object in Listing 3.9 line 11. Whereas, `v33` represents the parameter which is passed to the `String`-constructor in Listing 3.9 line 11 (cf. Listing 3.10 line 32). As it is statically not possible to determine if line 6 or line 9 is executed in Listing 3.9, Listing 3.10 line 34 reports that identifier `v33` either points to `v22` or `v30`, which themselves either points to the source method invocation (cf. Listing 3.10 line 20) or to the constant string

'DoNothing' (cf. Listing 3.10 line 29). Note, Listing 3.10 line 34 does not report v32 from line 27 because the assignment in Listing 3.9 line 6 happens in a copy-by-reference manner and v32 is not used afterwards in any Java statements. JOANA resolves and detects this fact statically and therefore only reports v22. However, based on Listing 3.10 line 34 HDFT++ knows that v33 may potentially point to v22 or v30 at run-time. Therefore, whenever v22 or v30 are overwritten, i.e. line 20 or line 29 in Listing 3.10 is executed, HDFT++ also updates the mapping for v33 to point to the same data identifier as v22 or v30 do (cf. Listing 3.11 line 4). As in this sample only line 20 from Listing 3.10 is executed, Listing 3.11 shows in line 4 that v33 points to Source0; in Listing 3.12 we will discuss the case if variable out is overwritten with a constant string (cf. Listing 3.9 line 9). Based on that information and the PIP-state, HDFT++ reports in Listing 3.11 line 6 that data from the source has flowed into the sink. UC infrastructures benefit from such tracking results and are able, depending on the deployed policies inside the PDP, either to preventively prohibit or detectively log the violation of such flows.

In contrast to Listing 3.11, Listing 3.12 shows the PIP-state if the assignment instruction in Listing 3.9 line 6 is not executed. In that case, the byte-array size of variable out is smaller than the byte-array size of variable in, and therefore, variable out is overwritten with a constant string in line 9. Because of that, the corresponding chop node in Listing 3.10 line 29 is executed at run-time, and thus, v33 maps to the constant string and not to Source0. Therefore, the data flow dependency between line 3 and line 11 in Listing 3.9 gets disrupted at run-time and data can not flow from the source to the sink. Thus, HDFT++ does not report a data flow in Listing 3.12. Note, v20 and v22 in Listing 3.12 line 2 and line 3 are identifiers from JOANA's intermediate language representation, and are pointing according to line 19 and line 20 in Listing 3.10 to the source-method's return value.

Listing 3.12: HDFT++'s PIP-state without executed assignment instruction in Listing 3.9 Line 6

```
1  ..Source0                          ---> Source0
2  ..3863861312|main(.)V|v20          ---> Source0
3  ..3863861312|main(.)V|v22          ---> Source0
```

To make HDFT++'s achieved precision gain (cf. Listing 3.11 and Listing 3.12) more tangible, we also monitored the code in Listing 3.9 with SHRIFT. In contrast to HDFT++, SHRIFT only leverages source- and sink-information from the analysis-report (cf. Listing 3.10 line 1, line 8, line 17, and line 16) and does not take chop node information into account. Because of that, SHRIFT does not track the individual instructions on a dependency path, and thus, is not able to differentiate if the assignment in Listing 3.9 line 6 is executed or not. Once a sink is reached, SHRIFT signals to the PIP on which sources the sink depends on. Therefore SHRIFT always reports a flow of data from the source- to the sink-method in Listing 3.13 line 2, even if the

assignment in Listing 3.9 line 6 is not executed.

Listing 3.13: SHRIFT's PIP-state after executing the code in Listing 3.9.

```
1  ..Source0                    ---> Source0
2  ..Sink0                      ---> Source0
```

Sample 2: Loop instruction

Listing 3.14 shows our second sample where a data flow dependency between the `source`- and `sink`-method passes a `WHILE`-loop-statement. Similar to Listing 3.9, the size of the output byte-array `out`, i.e. the value of variable `size`, is randomly determined in line 2. If the size of byte-array `out` is large enough, input data from variable `in` is transferred in a copy-by-value manner to byte-array `out` in line 7.

Listing 3.14: Example of a data flow dependency crossing a WHILE-command

```
1  public static void main(String[] args) {
2    int size = new Random().nextInt(10);
3    byte[] in = source("Test").getBytes();
4    byte[] out = new byte[size];
5    int idx = 0;
6    while(idx < in.length && out.length>in.length){
7      out[idx] = in[idx];
8      idx++;
9    }
10   sink(new String(out));
11 }
12
13 public static String source(String s){return s;}
14 public static void sink(String s){System.out.println(s);}
```

Listing 3.15 shows the corresponding analysis-report. As expected, JOANA detects the flow from the `source` to the `sink` and also reports all chop nodes that are part of a dependency path. The most crucial chop nodes are in line 26 and line 27: both chop nodes together forge the assignment in Listing 3.14 line 7. Thereby, line 26 represents the right-hand side of this assignment, i.e. reading and loading the content from the byte-array `in` at index `idx` on the Java stack, and line 27 represents the left-hand side of the assignment, i.e. writing and storing the top stack entry (which correspond to the previously read value from byte-array `in`) into the byte-array `out` at index `idx`; v22 and v24 are identifiers at JOANA's intermediate representation layer for variable `in` and `out` respectively. Note, the analysis-report reports two chop nodes for a Java assignment-statement because assignment-statements are compiled into a `load` and `store` bytecode command by the Java compiler and each of them represents an own chop node inside the SDG.

Listing 3.15: Analysis-report for the WHILE-example in Listing 3.14.

```
1  <source>
2    <id>Source0</id>
3    <location>..main(.)V:38</location>
4    <signature>..source(Ljava/lang/String;)Ljava/lang/String;</signature>
5    <return/>
6  </source>
7  ...
8  <sink>
9    <id>Sink0</id>
10   <location>..main(.)V:89</location>
11   <signature>..sink(Ljava/lang/String;)V</signature>
12   <param index="1"/>
13 </sink>
14 ...
15 <flows>
16   <sink id="Sink0">
17     <source id="Source0"/>
18     <chop>
19       <chopNode bci="38" lab="v20 = source(#(Test))" om="..main(.)V"/>
20       <chopNode bci="41" lab="v22 = v20.getBytes()" om="..main(.)V"/>
21       <chopNode bci="44" lab="v23 = v22" om="..main(.)V"/>
22       <chopNode bci="46" lab="v24 = new []" om="..main(.)V"/>
23       <chopNode bci="48" lab="v25 = v24" om="..main(.)V"/>
24       <chopNode bci="50" lab="v27 = #(0)" om="..main(.)V"/>
25       <chopNode bci="52" lab="goto 69" om="..main(.)V"/>
26       <chopNode bci="61" lab="v31 = v22[v35]" om="..main(.)V"/>
27       <chopNode bci="62" lab="v24[v35] = v31" om="..main(.)V"/>
28       <chopNode bci="63" lab="v33 = v35 + #(1)" om="..main(.)V"/>
29       <chopNode bci="69" lab="v28 = v22.length" om="..main(.)V"/>
30       <chopNode bci="70" lab="if (v35 >;= v28) goto 81" om="..main(.)V"/>
31       <chopNode bci="74" lab="v29 = v24.length" om="..main(.)V"/>
32       <chopNode bci="76" lab="v30 = v22.length" om="..main(.)V"/>
33       <chopNode bci="77" lab="if (v29 >; v30) goto 61" om="..main(.)V"/>
34       <chopNode bci="81" lab="v36 = new java.lang.String" om="..main(.)V"/>
35       <chopNode bci="86" lab="v36.<init>;(v24)" om="..main(.)V"/>
36       <chopNode bci="89" lab="sink(v36)" om="..main(.)V"/>
37       <chopNode bci="-8" lab="PHI v35 = #(0), v33" om="..main(.)V"/>
38     </chop>
39   </sink>
40 </flows>
```

Listing 3.16 shows the corresponding HDFT++'s PIP-state with the executed assignment-statement in Listing 3.14 line 7. Line 2 and line 3 in Listing 3.16 have the same reasoning as in sample 1 (cf. Listing 3.11 line 2 and line 3). Line 4 and line 5 in Listing 3.16 are the mappings which points to the byte-array `out` and its elements. Thereby, identifier v31 is a helper identifier at the SSA level: it first takes a value from byte-array `in` (identified by v22, cf. Listing 3.15 line 20) at index `idx` (identified by v35, cf. Listing 3.15 line 26), and second, assigns its value to the

Listing 3.16: HDFT++'s PIP-state with executed assignment-statement in Listing 3.14 line 7

```
1  ..Source0                        ---> Source0
2  ..3863862904|main(.)V|v20        ---> Source0
3  ..3863862904|main(.)V|v22        ---> Source0
4  ..3863862904|main(.)V|v24        ---> Source0
5  ..3863862904|main(.)V|v31        ---> Source0
6  ..3863862904|main(.)V|v36        ---> Source0
7  ..Sink0                          ---> Source0
```

byte-array `out` (identified by `v24`, cf. Listing 3.15 line 27) at index `idx` (identified by `v35`, cf. Listing 3.15 line 27). According to Listing 3.15 line 36, `v36` is the intermediate variable identifier that is passed as a parameter to the `sink`-method. At run-time, HDFT++ detects that `v24` which points to `Source0` (cf. Listing 3.16 line 4) flows into `v36` (cf. Listing 3.15 line 35). Therefore, HDFT++ reports that `Source0` has also flowed into the sink `Sink0` (cf. Listing 3.16 line 7).

Listing 3.17 shows the PIP-state without the executed assignment-statement from Listing 3.14 line 7. As expected HDFT++ does not report the flow of data because the copy-by-value assignment is not executed. Similar as in Listing 3.12, the intermediate identifiers `v20` and `v22` are pointing to the return value from the `source`-method invocation, cf. Listing 3.15 line 19 and line 20.

Listing 3.17: HDFT++'s PIP-state without executed assignment-statement in Listing 3.14 line 7

```
1  ..Source0                        ---> Source0
2  ..3863861336|main(.)V|v20        ---> Source0
3  ..3863861336|main(.)V|v22        ---> Source0
```

To compare HDFT++'s tracking results with SHRIFT, we also run and analyzed Listing 3.15 with SHRIFT. Listing 3.18 shows the corresponding tracking result. As both code samples, Listing 3.9 and Listing 3.14, are structure-wise pretty similar (except that data is transferred in a copy-by-value manner in Listing 3.14) SHRIFT provides the same tracking results for both code samples, regardless of the assignment instruction in Listing 3.14 line 7.

Listing 3.18: SHRIFT's PIP-state after executing the code in Listing 3.14.

```
1  ..Source0                        ---> Source0
2  ..Sink0                          ---> Source0
```

Sample 3: Inheritance relation

Listing 3.19 shows our third sample where the data flow dependency between the `source`- and `sink`-method crosses a randomly chosen Java object (cf. Listing 3.19 line 6).

Listing 3.19: Example of a data flow dependency passing through an inheritance relation

```
1  public static void main(String[] args) {
2    int size = new Random().nextInt(10);
3    byte[] in = source("Test").getBytes();
4    byte[] out;
5
6    Copy[] copies = new Copy[]{new Copy(), new CopyByRef(), new CopyByValue()};
7    int i = new Random().nextInt(3);
8    out = copies[i].copy(in);
9
10   sink(new String(out));
11 }
12 public static String source(String s){return s;}
13 public static void sink(String s){System.out.println(s);}
14
15 // Root class for all Copy-classes. Returns an empty byte array
16 public static class Copy{
17   public byte[] copy(byte[] a){
18     System.out.println("Copy");
19     return new byte[10];
20   }
21 }
22
23 /* Returns a new reference which points to the byte-array parameter */
24 public static class CopyByRef extends Copy{
25   public byte[] copy(byte[] a){
26     System.out.println("CopyByRef");
27     byte[] _return=a;
28     return _return;
29   }
30 }
31
32 /* Copies every element from the byte-array parameter to the return array */
33 public static class CopyByValue extends Copy{
34   public byte[] copy(byte[] a){
35     System.out.println("CopyByValue");
36     byte[] _return = new byte[a.length];
37     for(int i = 0; i < a.length; i++){
38       _return[i] = a[i];
39     }
40     return _return;
41   }
42 }
```

Listing 3.20: Analysis-report for the INHERITANCE-example in Listing 3.19.

```
<source>
  <id>Source0</id>
  <location>..main(.)V:38</location>
  <signature>..source(Ljava/lang/String;)Ljava/lang/String;</signature>
  <return/>
</source>
<sink>
  <id>Sink0</id>
  <location>..main(.)V:113</location>
  <signature>..sink(Ljava/lang/String;)V</signature>
  <param index="1"/>
</sink>
<flows>
<sink id="Sink0">
  <source id="Source0"/>
  <chop>
    <chopNode bci="1" lab="<init>;()" om="Copy.<init>;()V"/>
    <chopNode bci="10" lab="v8 = new []" om="Copy.copy([B)[B"/>
    <chopNode bci="12" lab="return v8" om="Copy.copy([B)[B"/>
    <chopNode bci="1" lab="this.<init>;()" om="CopyByRef.<init>;()V"/>
    <chopNode bci="9" lab="v7 = p1" om="CopyByRef.copy([B)[B"/>
    <chopNode bci="11" lab="return p1" om="CopyByRef.copy([B)[B"/>
    <chopNode bci="1" lab="this.<init>;()" om="CopyByValue.<init>;()V"/>
    <chopNode bci="-8" lab="PHI v17 = #(0), v15" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="9" lab="v7 = p1.length" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="10" lab="v8 = new []" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="14" lab="v11 = #(0)" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="22" lab="v13 = p1[v17]" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="23" lab="v8[v17] = v13" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="24" lab="v15 = v17 + #(1)" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="34" lab="return v8" om="CopyByValue.copy([B)[B"/>
    <chopNode bci="38" lab="v20 = source(#(Test))" om="..main(.)V"/>
    <chopNode bci="41" lab="v22 = v20.getBytes()" om="..main(.)V"/>
    <chopNode bci="46" lab="v25 = new Copy[]" om="..main(.)V"/>
    <chopNode bci="51" lab="v27 = new Copy" om="..main(.)V"/>
    <chopNode bci="58" lab="v25[#(0)] = v27" om="..main(.)V"/>
    <chopNode bci="61" lab="v30 = new CopyByRef" om="..main(.)V"/>
    <chopNode bci="68" lab="v25[#(1)] = v30" om="..main(.)V"/>
    <chopNode bci="71" lab="v33 = new CopyByValue" om="..main(.)V"/>
    <chopNode bci="78" lab="v25[#(2)] = v33" om="..main(.)V"/>
    <chopNode bci="98" lab="v41 = v25[v39]" om="..main(.)V"/>
    <chopNode bci="100" lab="v43 = v41.copy(v22)" om="..main(.)V"/>
    <chopNode bci="105" lab="v45 = new java.lang.String" om="..main(.)V"/>
    <chopNode bci="110" lab="v45.<init>;(v43)" om="..main(.)V"/>
    <chopNode bci="113" lab="sink(v45)" om="..main(.)V"/>
  </chop>
</sink>
</flows >
```

Those Java objects are derived from the Java class Copy and overwrite its method copy(byte[] a) with an own individual copying behavior. The super-class method Copy.copy(byte[] a) always returns a new byte-array, irrespective of parameter a's value, whereas the sub-class methods either return a new reference (cf. CopyByRef.copy(byte[] a)) or a copy-by-value copy (cf. CopyByValue.copy(byte[] a)) of the passed byte-array parameter "a". The main difference between those copy methods is, that the latter one allocates new memory for the copied byte-array, whereas the former one provides a new reference to the same memory region.

Depending on the randomly chosen Java object, one of those copy-method implementations is invoked in Listing 3.19 line 8 and transfers data from its input parameter to its return value. Listing 3.20 shows an excerpt from the static analysis-report for Listing 3.19 (the complete analysis-report is provided in Listing A.2). As JOANA performs a whole-program analysis the flow of data between the source- and sink-method, as well as its corresponding chop-nodes, are reported, taking into account the different copy(byte[] a) method implementations, which potentially could be invoked in Listing 3.19 line 8. Listing 3.20 line 17 – line 19 list all chop nodes which belongs to the method Copy.copy(byte[] a); line 20 – line 22 list all chop nodes which belongs to the method CopyByRef.copy(byte[] a); line 23 – line 31 list all chop nodes which belongs to the method CopyByValue.copy(byte[] a). Line 42 in Listing 3.20 is the most crucial one: this chop node represents the copy-method invocation on a randomly chosen Java object from the array copies (cf. line 8 and line 6 in Listing 3.19); the identifier v41 in Listing 3.20 line 42 represents the randomly chosen Java object on which this method is invoked. Note, line 34 – line 40 in Listing 3.20 provides all the chop nodes which create and fill up the copies array with the corresponding Copy objects (v25 is the corresponding identifier from JOANA's SSA representation).

Listing 3.21: HDFT++'s PIP state with executed Copy.copy(byte[] a) method.

```
1  ..Source0                          ---> Source0
2  ..3863889976|sample()V|v20         ---> Source0
3  ..3863889976|sample()V|v22         ---> Source0
```

As every copy(byte[] a) method-implementation has a different behavior regarding the way how data is transferred from the input parameter "a" to the return value, we run HDFT++ several times on the code in Listing 3.19 to examine how the PIP-state evolves when each of those different method-implementations is executed in line 8. Listing 3.21 shows the PIP-state when the method Copy.copy(byte[] a) is executed. As this method always returns a new, empty byte-array, the PIP-state does not report a flow of data from the source to the sink. Similar as in sample 1 and 2, the intermediate identifiers v20 and v22 are pointing to the return value from the source-method (cf. Listing 3.20 line 32 and line 33).

Listing 3.22 shows the PIP-state when the method `CopyByRef.copy(byte[] a)` is executed. As this method returns a new reference to the byte-array parameter "a", the PIP-state reports that data has flowed from the `source` to the `sink` (cf. Listing 3.22 line 7). The intermediate identifier v43 points to the return value from the `copy(byte[] a)` method invocation (cf. Listing 3.20 line 42), whereas v45 points to the method parameter which is passed to the `sink`-method (cf. Listing 3.20 line 45).

Listing 3.22: HDFT++'s PIP state with executed `CopyByRef.copy(byte[] a)` method.

```
1  ..Source0                                ---> Source0
2  ..CopyByRef|3871454440|copy([B)[B|p1     ---> Source0
3  ..3863890152|main(.)V|v20                ---> Source0
4  ..3863890152|main(.)V|v22                ---> Source0
5  ..3863890152|main(.)V|v43                ---> Source0
6  ..3863890152|main(.)V|v45                ---> Source0
7  ..Sink0                                  ---> Source0
```

Listing 3.23 shows the PIP state when the `CopyByValue.copy(byte[] a)` is executed. As this method returns an identical copy of the byte-array "a" the PIP-state reports that data has flowed from the `source` to the `sink` (cf. Listing 3.23 line 7). Note, as copying of data happens in a copy-by-value manner inside the method `CopyByValue.copy(byte[] a)` HDFT++ ++ also reports v8 in Listing 3.23 line 4 which basically corresponds to the variable `_return` in Listing 3.19 line 38.

Listing 3.23: HDFT++'s PIP state with executed `CopyByValue.copy(byte[] a)` method.

```
1  ..Source0                                  ---> Source0
2  ..3863890024|main(.)V|v20                  ---> Source0
3  ..3863890024|main(.)V|v22                  ---> Source0
4  ..CopyByValue|3871626072|copy([B)[B|v8     ---> Source0
5  ..3863890024|main(.)V|v43                  ---> Source0
6  ..3863890024|main(.)V|v45                  ---> Source0
7  ..Sink0                                    ---> Source0
```

We also analyzed the code in Listing 3.19 with SHRIFT to investigate how the randomly chosen Java object in Listing 3.19 line 8 may influence the tracking result. However, here we could observe the same behavior as in the previous samples: as SHRIFT does not take into account any run-time information, the tracking result in Listing 3.24 stays always the same. Even if the method `Copy.copy(byte[] a)` is executed in Listing 3.19 line 8, which even does not transfer any data, SHRIFT still reports a flow of data.

Listing 3.24: SHRIFT's PIP-state after executing the code in Listing 3.20.

```
1  ..Source0                    ---> Source0
2  ..Sink0                      ---> Source0
```

3.6.3. Performance

To track the flow of data from a source to a sink, SHRIFT and HDFT++ inject additional instructions into the AuS. As any additional instruction needs to be processed at run-time, it may affect and influence the native execution time of the AuS, i.e. the time the AuS would need without being monitored. This sub-section addresses Caveat 2 and describes the experiments, including their evaluation results, to investigate the performance overhead SHRIFT and HDFT++ impose on the AuS.

We set up the following experimental setting: before running each performance experiment each AuS has been statically analyzed concerning possible information flows (cf. Section 3.6.1). As JOANA is highly configurable, like with or without *object-sensitiveness* [74], the static analysis phase needs more or less resources depending on the configuration setting (cf. Table 3.1). However, because of its good trade-off between (high) precision and resource consumption all AuSs have been statically analyzed with object-sensitiveness and explicit flows only (cf. Section 3.6.1); any other points-to analysis would generate statistically indistinguishable run-time performance [71]. Prior to the experiment's execution and to factor out needed instrumentation-time, we statically instrumented the AuS according to the analysis report. All performance experiments have been executed on a virtual system with a 8-core CPU (2.6 GHz Xeon-E5) and 10GB of RAM. To weed out possible environmental noise, median values for 30 runs are reported in Table 3.3.

To evaluate the performance overhead of our approaches, a test-suite of different Java-based standalone- (JZip and JFTP) and web-applications (BirthdayApp, SnipSnap, WebGoat, and PersonalBlog) have been used, whose computational loads range from low to high. While the former ones are directly executed in the *Java Virtual Machine (JVM)*, web-applications need a run-time-container to operate; *Apache Tomcat 8.0.9* has been used for that. Further, all required SHRIFT- or HDFT++-classes have been added to the Java classpath on startup to be available at run-time for the injected PEP inside the AuS.

Table 3.3 shows the performance evaluation of our test-suite. Column "native" yields the execution time without, and columns "SHRIFT" and "HDFT++" provide the execution time with the corresponding tracking logic in place. To trigger different dependencies inside the AuS, we run experiments with different inputs (like using different file sizes) or we executed different tasks inside the AuS (like LI or LO for BirthdayApp).

For the JZip and JFTP experiments, we vary the file-size (1MB and 10MB) and their internal buffer for file- and network-I/O operations. By default, both applications use a 1-kByte internal buffer for reading and writing data. We increased this buffer to 32-kByte for the experiments JZip_{32} and JFTP_{32} to investigate how a larger buffer affects the run-time performance overhead.

Table 3.3.: Performance measurement results in milliseconds. Median values for 30 runs are reported. Column “native” lists the native execution time of the AuS, whereas the columns “SHRIFT” and “HDFT++” show the execution time with SHRIFT and HDFT++ enabled. JZip and JFTP were evaluated with an internal buffer of 1-kByte. whereas JZip $_{32}$ and JFTP $_{32}$ reports the same experiments with an 32-kByte internal-buffer size. The prototype implementation is benchmarked with LIBDFT [54] and. if possible. with PHOSPHOR [10].

AuS	Input	native	SHRIFT	SHRIFT/native	HDFT++	HDFT++/native	[54]	[54]/native	[10]	[10]/native
JZip	1MB	38.53	51.76	1.34	99.78	2.58	581.82	15.10	83.49	2.16
	10MB	87.64	164.04	1.87	818.44	9.33	1814.54	20.70	708.91	8.10
JZip$_{32}$	1MB	31.91	33.34	1.04	37.00	1.15	320	10.02	71.72	2.24
	10MB	68	72 .49	1.06	116.21	1.70	1392.57	20.47	691.51	10.16
JFTP	10MB	693.88	725.73	1.045	1727.52	2.48	12781.96	18.42	1873.82	2.7
JFTP$_{32}$	10MB	283.94	295.94	1.042	659.54	2.32	9831.86	34.62	1104.85	3.89
BirthdayApp	SP	0.31	0.66	2.12	1,03	3.33	18.81	60.68	–	
	LI	510.91	543.41	1.06	562.11	1.10	1984.59	3.88		
	LO	0.32	0.51	1.59	24.86	77.68	12,54	39.21		
PersonalBlog	UC	14.04	14.77	1.05	19.97	1.42	271.28	19.32	–	
SnipSnap	UC	12.79	14.79	1.15	15.32	1.19	247.35	19.33	–	

In case of the *BirthdayApp* application, we executed different tasks (cf. BirthdayApp-block in Table 3.3): display the start page (row SP), login to (row LI) and logout from (row LO) Facebook. Finally, for the *SnipSnap* and *PersonalBlog* web-application we executed a use-case scenario: display the start- and login-page, editing and creating a new post, and logout from the web-application (row UC).

These evaluation results reveal that tasks with a higher native execution time exhibit a lower relative run-time overhead (column "SHRIFT/native" and "HDFT++/native" in Table 3.3). For instance, as the BirthdayApp's facebook-login task LI needs to query Facebook's API to authenticate a user its relative run-time overhead is lower than rendering the start page after logout SPLO. However, this observation does not hold for all experiments: as zipping a 10MB file with a 1-kByte internal buffer takes natively longer than a 1MB file, the relative overhead increases with the file size. In case of BirthdayApp though, the time expensive Facebook authentication happens only once at the beginning, and thus, absorbs the imposed run-time overhead.

Moreover, we discovered that the program structure, that is passed by a dependency inside the AuS, has also an influence on the performance overhead. For instance, in case of JZip the dependency (i.e. chop) crosses a loop structure that encompasses chopCMDs for reading and writing source-files to the zipped-file. Because of that, a lot of notification events are exchanged between the PEP and the other UC components. To confirm this statement, we repeat the JZip experiments but with a 32-kByte internal buffer (row JZip $_{32}$ in Table 3.3) for reading and writing data. Our evaluation results show, that with a larger internal buffer the total performance overhead gets smaller. The same also holds for JFTP.

To benchmark our hybrid approaches with competitive data flow trackers, the same experiments have been conducted with LIBDFT [54] and (partially) with PHOSPHOR [10]. LIBDFT is a dynamic data flow tracker for x86-binaries and instruments every single instruction inside the binary. Our proposed hybrid approaches outperform performance-wise LIBDFT in all experiments and achieve better results because data flow tracking happens only selectively at those program locations that actually lead to a flow of data. Such a comparison may be considered as unfair because LIBDFT injects its tracking logic into the JVM itself to monitor executed bytecode commands, and therefore, its total performance overhead would be also influenced by the JVM's bootstrapping- and shutdown-time. However, we were very careful the way how we conducted the comparison: we measured and compared the performance overhead between the start and the end of each executed use-case in Table 3.3. For instance, for JZip we measured the performance overhead between the start and the end of the Java `main`-method; in case of web-applications we measured the time between entering and exiting the HTTP(-GET) method. This way we were able to factor out bootstrapping- and shutdown-time required by the JVM.

As LIBDFT is specially designed for x86-binaries, we also benchmarked our approaches against PHOSPHOR, a pure dynamic data flow tracker for Java, and Java-bytecode in particular. PHOSPHOR requires that the AuS, as well as the complete run-time environment, has to be instrumented to propagate data flows. This includes not only Java system classes but also any middleware or third-party library used by the AuS. Because of that, the same experiments could not be evaluated on the whole test-suite with PHOSPHOR, as some of the Java classes belonging to the Tomcat-web-server and/or third-party framework (like Spring [110] or *Java Cryptography Extension*) did not pass the JRE signature verification, and hence, could not be loaded after instrumentation. The results achieved in the experiments JFTP, JFTP $_{32}$, and JZip $_{32}$, outperform PHOSPHOR, whereas for JZip the proposed approach slows down the execution time about ∼1.0x compared to PHOSPHOR. The reason is, that the latter experiments stress-test the hybrid approach: zipping files in chunks of 1 kByte leads to a lot notifications between PEP and the other UC components. JZip$_{32}$ underpins this statement as here the larger internal buffer (32-kByte) leads to fewer notifications, and hence, to a reduced performance-overhead.

Remarks on *performance*: SHRIFT and HDFT++ inject a minimal IRM into the AuS, that tracks point-wise only those program locations that actually lead to a flow of data. As both approaches monitor fewer program locations than a pure dynamic data flow tracker does (with full instrumentation), we could achieve performance-wise better results (cf. Section 3.6). Moreover, instead of signaling extracted context information on every PoI's execution which actually impose unnecessary performance overhead and does not provide better tracking results, SHRIFT and HDFT++ transmit collected information only the first time when a PoI is executed. Moreover, the PEP signals executed chopCMDs and sinks only when their respective source was executed before.

Remarks on *portability*: SHRIFT and HDFT++ are able to increase the portability of the monitored AuS, as run-time tracking logic is only injected in the application's program code (cf. Figure 1.6), excluding 3^{rd}-party libraries and Java's JRE system-classes. As the static analysis phase in SHRIFT and HDFT++ performs a *whole-program* data flow analysis, taking into account the entire application's code, data flow dependencies which start and end at the application program code and pervade the entire JRE are detected. Because of that, the present work argues that SHRIFT's and HDFT++'s run-time tracking logic is not necessarily required for the excluded code parts. Thus, the monitored AuS can be executed on any off-the-shelf JRE. Concerning tracking precision, we have to rely on the static analysis results for the excluded code parts. At run-time, however, this might lead to imprecision as we are not able to handle properly situations inside Java system classes or 3^{rd}-party libraries (cf. Section 3.4 Listing 3.4). However, at the end the tradeoff between precision versus portability and performance remains:

a more precise tracking result requires to spread tracking logic over the entire AuS (i.e. application's program code, 3^{rd}-party and system libraries) at the expense of less portability and higher performance overhead (cf. section 3.6.3), and vice versa.

3.6.4. Threats to Validity

Although we evaluated our proposed approaches carefully there are several threats to validity which are discussed in the following.

Evaluation Environment. We used a virtual machine with a pre-installed Tomcat web-server instance to run our performance experiments in Section 3.6.3. One may argue that such an environmental setting does not properly reflect a PaaS cloud scenario as we have motivated in Section 1.2. Though this is a valid argument, we advocate that our employed environmental setting is pretty similar to the way how PaaS cloud platforms are providing their execution environments. For instance, Amazon's Elastic Beanstalk [2] provides a Java-Tomcat software stack at its central core to run Java-based web-applications. Thus, we advocate that running our experiments on a full-blown PaaS cloud platform would result in the same, comparable relative performance overhead. Moreover, we would like to emphasize that the present work considers the immediately enclosing unit which runs the AuS's code as a "run-time environment". For instance, in case of a Java program the Java Runtime Environment is the "run-time environment" which runs the AuS's code, or the Python interpreter in case of a Python application. We do not subsume under the term "run-time environment" all other components inside a PaaS cloud platform, like load balancers, containers, virtualization, etc.

Generalisability to arbitrary applications. How well do SHRIFT and HDFT++ generalize to arbitrary Java applications? As SIFA is a central element in the present work our approaches also inherit their limitations. This means in reverse, our approaches generalize to arbitrary applications as good as SIFA tools are able to do it. For instance, Java reflection and system callbacks are fundamental limitations in the field of SIFA for which no general solution is available (cf. Section 3.7 for a thorough discussion on that). As some applications in our test-suite are using such Java features, our test-results do not cover those source code regions inside the AuS which are called via system callbacks or reflection.

Further, the required memory resources to run static analysis also affect the generalisability of our approaches. SIFA tools, like JOANA or FLOWDROID, rely on SDGs which are an abstract, whole-program representation of the complete AuS, including all required system- and third-party-libraries. Depending on the application size, generating and storing an SDG may require different large memory space. As our results show in Section 3.6.1 analyzing even small applications, like JZip (which has approximately 330 lines of program code and one external

library), needs already non-negligible memory resources. With larger applications, like SnipSnap (which has approximately 26000 lines of code and 36 external libraries), those requirements get larger (cf. Table 3.2). A common technique to approach that issue are *flow models* [7]. These are pre-computed summaries about data flows inside specific code regions of the AuS. During the actual static analysis those information are reused, instead of analyzing those code regions represented by the flow models again. Applying flow models, for instance, on Java system- and third-party-libraries can reduce the memory consumption for static analysis as a full analysis of those libraries is not needed. However, in the end SIFA techniques are able to scale and to generalize to a certain degree, and not in general to arbitrary Java applications. Potential future research work in that direction could be to run static information flow analysis on a distributed cluster of interconnected machines, instead on a single machine, as current approaches in the field of SIFA are doing it.

Test-suite. We chose a set of different Java-based standalone- and web-applications for our evaluation. Those applications have different sizes regarding lines of code and number of used third-party libraries. For instance, JZip is our smallest application with approximately 330 lines of code and uses one external library to parse command line parameters. In contrast, SnipSnap is the largest application in our test-suite with approximately 26000 lines of code and with 36 external libraries. The remaining applications are somewhere in between: PersonalBlog has 3606 lines of code and 33 external libraries, BirthdayApp has 1931 lines of code and 7 external libraries, and JFTP has 3010 lines of code and no external libraries. Further, some applications were developed by us, like JZip, and some were downloaded from the internet, like PersonalBlog. With this collection of applications, we aime to achieve a diversified test-suite which exhibit different sizes and different levels of complexity, in order to investigate how SHRIFT and HDFT++ behaves not only on small and less complex programs, like JZip, but also on larger and more complex applications, like Snipsnap or BirthdayApp (which, for example, are using XML-parser- or REST-Client-libraries). We do not claim for completeness of our test-suite, more-complex and larger applications can be added to our test-suite, keeping limitations of static analysis in mind.

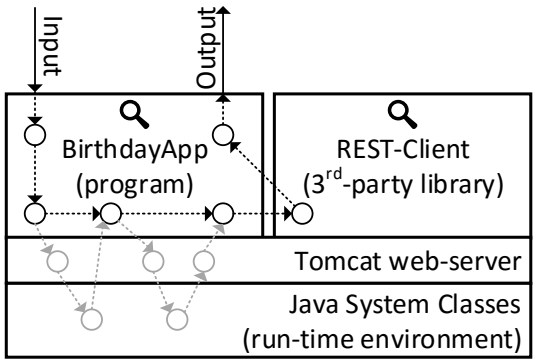
Precision. Our primary driver to improve data flow tracking precision is the reduction of overapproximation in a black-box tracking approach (cf. Section 1.3) where every source flows to every sink, i.e. the number of potential data flow dependencies is $\#flows_b = \#sources \times \#sinks$. We quantified our precision improvement in Section 3.6.1 as the number of data flow dependencies which can be safely discarded from $\#flows_b$, i.e. the number of data flows which would mistakenly be reported in a black-box approach. Depending on the static analysis configuration, the number of discarded dependencies differ, and thus, provides different precision results according to our

metric (cf. Section 3.6.1). Thus, as SHRIFT and HDFT++ are reusing those results at run-time both provide by design at least a better tracking precision than a pure black-box approach (cf. Section 3.6.1). HDFT++ is an extension of SHRIFT and additionally tracks intermediate instructions on a sink-source data flow dependency, aiming to weed out mistakenly reported dependencies, which may not occur at run-time due to different run-time values. Because SHRIFT does not take into account instructions on a data flow dependency path, SHRIFT is not able to distinguish if a data flow dependency is still critical or not when its sink is reached. Quantifying HDFT++'s precision gain over SHRIFT with a general metric is non-trivial, as it highly depends on the actual run-time values, and thus, on the taken data flow path inside the application (cf. Section 3.4 Listing 3.4). To the best of our knowledge, we illustrated in Section 3.6.2 HDFT++'s precision gain compared to SHRIFT along the lines of a use case study and point-out under which conditions HDFT++ provides a more precise tracking result than SHRIFT. However, it may happen that a crucial instruction which leads to a data flow, like line 4 Listing 3.4, may also be located inside a RTE's system- or a 3rd-party-library-class which is not monitored by HDFT++ for the sake of portability. In this case, HDFT++ would miss it and also mistakenly report a flow of data although a flow may not happen. But in the end, the tradeoff between precision and portability remains: more precise tracking results require data flow tracking inside the entire AuS at the cost of less portability, and vice versa.

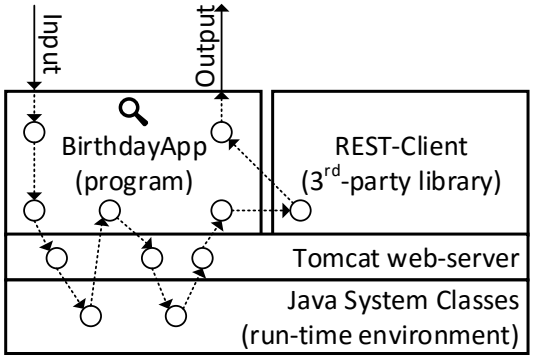
3.7. Strengths and Limitations

Compared to pure dynamic taint tracking, like PHOSPHOR [10] or LIBDFT [54], SHRIFT and HDFT++ do not require to instrument each command within the AuS. By utilizing statically pre-computed analysis results a "minimal" IRM is injected which monitors only those instructions that lead to a flow of data. This way, the run-time performance overhead of our hybrid approach exhibits less or comparable performance overhead than a pure dynamic tracker, while collecting valuable information about executed and passed data flow dependencies.

To address RQ2 and RQ3, SHRIFT and HDFT++ are able to inject their tracking logic only inside a selected code region, and thus, are able to preserve application's portability. Figure 3.4 illustrates this aspect along the lines of our BirthdayApp application from our running example (cf. Section 1.2). Considering Figure 1.6, we divide BirthdayApp into the code blocks: BirthdayApp-program (which contains the main application logic), REST-Client (which enables to call OSN's REST-API), a Tomcat-Webserver (which provides all libraries for Java web-service support, i.e. those implement the JSR 109), and Java system classes (which are provided by the standard JRE installation); note, for the sake of simplicity we illustrate only one external library in Fig-



(a) Tracking logic 🔍 is injected inside the BirthdayApp program block and the REST-Client. Java system-classes and Tomcat web-server are excluded.



(b) Tracking logic 🔍 is distributed over all code blocks, including BirthdayApp program, REST-Client, Tomcat web-server, and Java system-classes.

Figure 3.4.: BirthdayApp is modularised into a program block, which contains the actual application logic, a REST-Client, to call OSN's REST-API, a Tomcat web-server, which provides libraries to run Java web-services, and Java system-classes. Figure 3.4a illustrates the SHRIFT and HDFT++ approach: tracking logic 🔍 is injected selectively at the BirthdayApp program block and into the REST-Client. Contrary, dynamic data flow trackers, like PHOSPHOR, also injects its tracking logic inside the Tomcat web-server and Java system-classes, in order to propagate the flow of data across every single command ○ (cf. Figure 3.4b). Thus, all components are tightly interwoven.

ure 3.4, viz. REST-Client, although BirthdayApp has in total 7 libraries. A pure dynamic data flow tracker, like PHOSPHOR, disperses and injects its tracking logic 🔍 into all those code blocks (cf. Figure 3.4b) in order to track the flow of data from inputs to outputs. Thus, all code blocks get tightly coupled and interwoven so that they need each other to operate at run-time. In particular, BirthdayApp-program, which contains the main application logic, gets dependent on appropriately instrumented Java system-classes, REST-Client- and Tomcat web-server-library.

In contrast, as SHRIFT and HDFT++ are based on a static analysis phase, which takes into account the entire code base, they are able to track data flow dependencies only inside selected code regions and exclude designated code blocks at run-time. For instance, based on the static analysis-report SHRIFT and HDFT++ monitor only those program instructions inside the BirthdayApp-program code block that are relevant for a flow of data, and do not rely on any components or modifications inside the run-time environment, i.e. the JVM or the Java system classes in particular (cf. Figure 3.4b). Because of that, the monitored BirthdayApp-program can be easily transferred and executed, together with the injected tracking logic, on different

off-the-shelf JRE run-time environments without requiring specially instrumented Java system-classes or third-party-libraries. We argue, that the portability benefits of our proposed approach come into full effect inside domains where computational resources are costly and therefore shared among different applications; for instance, inside the cloud domain where a common run-time execution platform is shared among multiple cloud services, i.e. technically those cloud services are executed on the same JRE (cf. Section 1.2). Moreover, as our experiments show, instrumenting the entire environment may also lead to non-operational applications (cf. Section 3.6.3). However, in the end the portability benefits of our proposed approach highly depends on which code blocks are instrumented and equipped with the tracking logic inside the AuS (cf. the schematic illustration in Figure 1.6). Although it is technically possible to place tracking logic inside all code blocks of an application and to reuse them inside other applications, we do not recommend this method as it can lead to falsified run-time data flow tracking results. For instance, reusing the instrumented REST-Client library from BirthdayApp inside another application, like PersonalBlog, would lead to the case that excuted chopCMDs, which belong to data flow dependencies from BirthdayApp, are also reported under the execution of PersonalBlog. Because of that, this thesis advocates to place tracking logic only inside those code blocks which are not shared between applications at run-time. As Java system classes are usually shared between all Java applications which run on the JRE, we do not inject our tracking logic inside those system classes and rely, with regard to data flow dependencies, on the static analysis-report for the excluded code blocks. As third-party libraries, like BirthdayApp's REST-Client, are usually bundled and deployed together with an application we also inject our tracking logic into them (cf. Figure 3.4a) – we assume that third-party libraries are integral components of applications, and thus, are not shared betwen applications at run-time. But, however, if third-party libraries are also shared between applications at run-time (the same case as with Java system classes) the present approaches also support to exclude tracking logic inside those third-party libraries (because of the same, previously described reasons). In the end, it is a matter between tracking precision and run-time overhead. A more precise tracking result demands to inject tracking logic into all code blocks at the expense of higher performance run-time overhead and vice versa.

As SHRIFT and HDFT++ are based on static information flow analysis techniques they inherit and suffer from their limitations. By its nature, static information flow analysis is not able to distinguish every possible execution and therefore introduces overapproximation which results in imprecision in the information flow analysis. One possibility to improve precision is the use of a more precise points-to analysis. But this usually comes at the price of considerably longer analysis times and higher memory consumption, meaning scalability problems [71]. The

scalability problems are worsened by the fact that even small Java applications use large parts of the Java standard library – sometimes just referencing a prominent class name makes the structures which JOANA constructs (callgraph and SDG) very large. Currently, JOANA performs a *whole-program analysis* which means that all libraries used by the AuS need to be analyzed every time. There exists an approach to make the PDG construction more modular by pre-computing appropriate approximations of library PDGs and re-using them when calls of library methods are encountered [7, 35], but this approach has not been fully integrated yet, so it is unclear whether it brings considerable performance gains in practice. In the end, however, the problem remains a trade-off between precision and performance.

Another limitation of JOANA is the inability to properly analyze applications that do not have a main entry point, but are rather used through callback handlers which originate from system calls/interrupts (e.g. UI-interactions, like mouse-clicks, in Swing). Callback handlers and reflective code are critical programming constructs that impose further challenges for static whole-program analysis. Analyzing callback-based applications requires a model that captures the way callback handlers are used (e.g. which simulates the user). Such a model, for example, could be obtained by running the application, by specification in a dedicated language, or by simulating all possible callback connections.

Furthermore, like callbacks, dealing with reflection in a sound but not overly imprecise way is not a JOANA-specific issue but rather a fundamental challenge in static information flow analysis, for which a general precise solution is impossible. Additional analyses, like string analysis or run-time information [11], may help to resolve reflection (e.g. find out the name of a dynamically loaded class), but in general either very coarse assumptions have to be made or unresolvable reflective code has to be ignored. Further, the contribution of this work does not focus on side-channel attacks, like timing or power analysis attacks, which could be used to infer information or features about a specific data item. In a worst-case scenario, such an attack may expose the original data item, and thus, would allow an attacker to use the data item in an unrestricted manner.

The notion of soundness depends on the notion of information flow. In the present approach, information flows solely caused by exceptions are intentionally ignored. This has to do with the fact that every I/O operation may cause an exception, making the execution of every source influencing every following sink by possible failing. JOANA can handle exceptional control-flow, but during our evaluation this feature was disabled.

If we run the static analysis phase to detect explicit flows only then the inlined reference monitor inside the application guarantees a property similar to Volpano's *weak secrecy* [119]. However, it would be easy to circumvent the analysis by transforming each direct assignment within the

application into an "indirect" assignment (i.e. a loop that leaks the value of a variable one bit at a time via a control-flow dependency). This way, the analysis would report no dependencies between sources and sinks. However, sound and precise system-wide non-interference assessments (including implicit flows) require a static analysis of all applications together at once. This is, because independent analyses for single applications are inherently non-compositional, they cannot model dependencies generated by the concurrent interactions on shared resources [103]. Due to its exponential nature, a global all-at-once analysis would be unfeasible even for a small number of applications of a reasonable size and would also likely lead to results that are too conservative to be useful (i.e. too many false positives). The proposed approach resides somehow in-between these two extremes: by considering all possible flows during the intra-process analysis, non-interference between inputs and outputs is guaranteed for each application if they do not appear in the report, while data flows through and across applications are captured at run-time. This property is stronger than weak secrecy, which ignores intra-process implicit flows, but still weaker than system-wide non-interference, due to the aforementioned general lack of compositionality of the analyses for different applications.

3.8. Summary and Conclusion

In summary, this chapter presents SHRIFT and its extension HDFT++, two hybrid data flow tracking solutions. Both approaches are specifically designed to provide a reasonable balance between performance and portability of the AuS. Furthermore, these solutions are innovative as they minimize the number of and track only those program instructions at the application's program code level (cf. Figure 1.6) that actually contribute to a flow of data from sources to sinks. At run-time SHRIFT and HDFT++ collect valuable information about the executed data flows in order to support UC policy enforcement, like Policy P_1. Conceptually, SHRIFT and HDFT++ share the same methodological steps (cf. section 3.2):

(i) statically analyze the AuS for sinks, sources, and possible dependencies between them.

(ii) instrument program instructions inside the AuS based on the analysis result from (i). SHRIFT instruments only instructions that correspond to a source or sink, whereas HDFT++ also instruments all instructions in between sources and sinks.

(iii) extract and signal context information, like read files, to the UC PIP component that keeps track of flowed data inside the AuS.

We instantiated SHRIFT and HDFT++ prototypically for Java and evaluated them on a set of different Java applications. Our results (cf. Section 3.6) show that the run-time performance

overhead highly depends on the program structure of the analyzed application as well as on the computational load of its native execution. Nevertheless, we observed that the relative performance overhead gets smaller for computational-intensive than for non-computational-intensive applications. Furthermore, compared to LIBDFT [54] and PHOSPHOR [10], SHRIFT and HDFT++ impose less or approximately similar run-time slow-down. By design, both approaches ensure the portability of the AuS as they do not rely on tracking logic inside the run-time environment (cf. Figure 3.4). That way, once a AuS is equipped with such a solution it can be executed on any run-time environment. Furthermore, our design decision also provides for both approaches at least a better tracking precision than a pure black-box approach. Further, as SHRIFT and HDFT++ exhibit a different run-time behavior, which results in different tracking precisions, we also conducted in Section 3.6.2 a comparison between SHRIFT and HDFT++. We illustrate along the lines of sample source codes in which situations HDFT++ provides a more precise tracking result than SHRIFT.

4. Related work

This chapter relates the present work to related and similar work from the literature. In particular, it discusses the substantial distinctions in terms of Information Flow Tracking (cf. Section 4.1) and distributed Data Usage Control (cf. Section 4.1), the two major pillars this thesis builds upon.

4.1. Information Flow Tracking

The research field of *Information Flow Tracking* (IFT) tackles the question of how data flows inside a data processing system from its inputs to its outputs. A clear understanding of how data flows through a system may contribute to solving different problems, e.g. malware detection [125], privacy protection [23] or Usage Control policy enforcement [92], as it is envisaged in this thesis. Beyond that, IFT solutions have also been successfully applied within different domains to secure the confidentiality and integrity of data (like Android [8], Web-Applications [114], Java-Application [10]). Those solutions mostly follow a *static* or *dynamic* approach [102]. *Hybrid* approaches, where statically computed results are combined with dynamic run-time information are rare but become increasingly attractive. The following shortly recaps static IFT approaches and broadly discusses dynamic and hybrid IFT approaches, as the present work is closely related to the latter; cf. Section 3.3 for a more thorough discussion of static information flow analysis.

To detect possible information flows, static approaches analyze the complete code of the AuS without executing it and consider all possible information flow traces at once [20, 120]. A given program is certified as secure if no flow of information between sensitive sources and public sinks can be found. Such a static certification can be used, for example, to reduce the need for runtime checks [21]. Various approaches (apart from PDGs) can be found in the literature, usually based on type checking [76, 88, 120], hoare logic [9], or taint analysis [3, 4]. Because of their nature, static approaches have problems with handling dynamic aspects of applications like callbacks or reflective code (Section 3.7) and are confined to the application under analysis.

In contrast, dynamic approaches mark sensitive data items with a dedicated label and propagate this label along the lines of executed commands at run-time. Moreover, they are able to

leverage additional run-time information, like concrete user input or the file that was read by a source. Strictly speaking, and in a more abstract sense, SHRIFT and HDFT++ are also a kind of dynamic data flow tracker. But in contrast to pure dynamic approaches from the literature, SHRIFT and HDFT++ optimize the number of program locations that need to be monitored at run-time as they rely on statically pre-computed results. Therefore, the remaining section focuses on pure dynamic approaches and discusses their demarcation to the work in this thesis; a discourse on static approaches for data flow analysis is provided in Section 3.1.

LIBDFT [54] provides a pure dynamic data flow tracker for x86 binaries by using *shadow tag maps* to store taint marks for every single register and memory address. At run-time, those taint marks are properly propagated along the lines of executed binary instructions. Although LIBDFT's reported evaluation results show little performance overhead [54] those numbers were not reproducible during our evaluation (cf. Section 3.6.3): on most of the use cases in Table 3.3 LIBDFT imposes a larger performance overhead than the presented approach.

SHADOWREPLICA [46, 47], which builds upon LIBDFT, is a dynamic data flow tracker for x86 binaries. Instead of inlining, SHADOWREPLICA decouples and offloads its data flow tracking logic in a separated *analyzer*-thread. Only a small, optimized piece of code is injected into the AuS which collects and transmits run-time data, like involved memory addresses of an instruction, via a ring-buffer to the analyzer. This way, the AuS run-time performance is only affected by the piece of code which collects run-time data. Compared to our approach, SHADOWREPLICA relies on a whole run-time instrumentation and does not consider an AuS as a composition of different code blocks. Because of that, this thesis advocates that SHADOWREPLICA affects the portability of AuS, and especially, is not well applicable within domains where run-time environments are simultaneously shared between different applications, like in our running sample in Section 1.2.

Nair et al. propose TRISHUL [77], a policy-based information flow control framework for Java-based applications. For one thing, TRISHUL injects its tracking logic into all parts (cf. Figure 1.6) of an AuS, and for another thing, TRISHUL requires a dedicated JVM which is equipped with a modified JavaCC compiler in order to rearrange the method code layout. According to the authors, the latter implementation is required to retrieve the object of a polymorphic method invocation. Anyway, as one may notice the TRISHUL approach is quite invasive and definitely handicaps the portability of a monitored AuS, as a properly modified JVM is always required to run the application.

In contrast to TRISHUL, Bell et al. provide PHOSPHOR [10], a pure dynamic data flow tracker for Java-based applications, which tracks the flow of data at the Java bytecode level and does not require a modified JVM. To do so, PHOSPHOR requires to instrument all Java classes which

belong to the AuS including the program, third-party libraries, and Java system-classes (cf. Figure 1.6). Moreover, PHOSPHOR also requires to modify the method signatures of all Java classes in order to properly propagate taint-labels at run-time. We argue that such an approach negatively affects the portability of the AuS as every execution of the program, which contains the actual AuS's logic, requires properly instrumented Java system-classes and third-party libraries. Moreover, PHOSPHOR considers methods with return values as possible sources and therefore introduces a taint-label value at the end of a method implementation. Introducing taint-label values this way makes the portability of the AuS more worse, especially when potential source methods belong to Java system-classes. Beyond that, for computational environments where resources (e.g. Java system-classes) are shared among different applications, e.g. in the cloud where a single JRE runs multiple cloud services (cf. Section 1.2), the PHOSPHOR approach would require dedicated instrumented Java system-classes for any single monitored service, which practically jacks up the costs. In contrast, our approach instruments and tracks only dedicated program locations at the code level that lead to a flow of data. Because of that, our approach additionally ensures by design the portability of the AuS once it is equipped with our tracker.

Apart from the literature which investigates IFT for particular programming languages or one of its intermediate representation, a major body of work is devoted to research IFT problems within different domains. In what follows, we exemplarily describe related work from the cloud and Android domain.

Pappas et al. propose CLOUDFENCE [81], a data flow tracking framework for the cloud domain. The authors intend that CLOUDFENCE is offered by cloud hosting providers, like IaaS- or PaaS-providers, to their tenants, i.e. service providers which run their services on an *Infrastructure as a Service (IaaS)*-/ PaaS-infrastructure. Via a dedicated API, service providers are able to integrate CLOUDFENCE into their services and mark sensitive user input, that needs to be protected, with a dedicated label. However, integration of CLOUDFENCE happens in a discretionary manner where service providers are not obliged to use CLOUDFENCE or may even put API-calls deliberately wrong. Contrarily, our approach tracks the data flow in a mandatory manner and focuses on collecting detailed run-time information about the executed program instructions. Beyond that, CLOUDFENCE uses PIN [87], an analysis-tool which performs whole process instrumentation, i.e. data flow tracking logic is dispersed over the entire codes base. Contrary to our approach, such a solution negatively affects the portability of the AuS.

To detect data leakages between multiple PaaS tenants (which for example may happen by a misconfigured cloud software stack), Priebe et al. propose CLOUDSAFETYNET [96], a data flow tracker at the network layer. To do so, CLOUDSAFETYNET provides a Javascript-based

client-library which enables a cloud service consumer to tag sensitive user input, e.g. an HTML form field. A tag is encrypted with the tenant's public key. Furthermore, CLOUDSAFETYNET equips a tenant's cloud service with a dedicated socket-level monitor which analyzes and parses all incoming and outgoing data traffic, with the aim of detecting tagged user input. A data leakage exists if such a monitor is not able to decrypt a submitted tag and therefore assumes that it must be a foreign tag label. Thus, the quintessence is that a data leakage is detected once a service client request or response contains tags from different PaaS-tenants. However, CLOUDSAFETYNET does not track the flow of data inside a PaaS-tenant's service, and therefore, exhibits conceptually similarities to the black-box approach and also suffers from the overapproximation problem (cf. Figure 1.4a).

TAINTDROID [23] provides a purely dynamic data flow tracking approach at the program-variables-, method-, file-, and message-level for system-wide real-time privacy monitoring within Android. To do so, TAINTDROID places dedicated monitors not only at the program's code level but also inside the Android operating system. Although the results show a relatively small runtime overhead, TAINTDROID has to place data flow monitors not only at the application level but also into the operating system. Thus, TAINTDROID limits the overall portability of the Android application. For instance, to monitor a specific application with TAINTDROID the entire Android device needs to be flashed.

Yin et al. [125] propose PANORAMA to detect malware in Windows systems. To do that, PANORAMA relies on *taint graphs* whose vertices represent operating system resources and edges represent the flow of data from one resource to another. Based on taint graphs and a set of policies, which describe the characteristic of malign behavior, PANORAMA detects patterns inside the taint graph that matches the malicious behavior. A further key element in PANORAMA's data flow tracking solution is a so-called *shadow memory* which stores the taint status of each physical memory, CPU's registers, the hard disk, and network interface buffer. Once a data source gets tainted, PANORAMA performs a fine-grained hardware-level taint tracking and monitors every CPU instruction or *Direct Memory Access (DMA)* operation which affects the taint status of the shadow memory. This way PANORAMA is able to track data flows system-wide in a Windows operating system. Although PANORAMA detects all malware samples from their test set with a zero false-negative rate, its data flow tracking approach slows down the AuS's run-time performance by an average factor of 20. However, as SHRIFT and HDFT++ reuse statically precomputed information flow results at run-time, both solutions impose much less performance overhead than PANORAMA.

Demsky [19] presents GARM, a tool to track data provenance information across multiple applications and machines. GARM instruments the application binary to track and to store the

flow of data within and across applications, and beyond that, to monitor interactions with the OS. Similar to our approaches, GARM leverages static analysis to generate optimized dynamic instrumentation by eliminating redundant provenance computations which may happen when the same value or variable is repeatedly combined with other values. However, GARM's static analysis focuses on optimizing dynamic instrumentation for a single *basic block*, which is defined as a sequence of instructions with one entry and possibly multiple exit points. In contrast, the present approach relies on static analysis which performs a whole-program information flow analysis. This way, the present results are able to optimize dynamic intra-application data flow tracking by monitoring all source-, sink-, and chop-node-instructions which are relevant for a data flow dependency.

Zhang et al. [127] present NEON, a fine-grained data flow tracking approach for derived data management. NEON is implemented as an extension for the XEN virtual machine monitor and tracks data flows on the granularity of individual bytes by tainting each memory address with an *n*-bit taint mark. Taint marks are propagated on each memory write or read access through and across systems. Although NEON presents a sophisticated fine-grained tracking approach it suffers from a high false-negative rate, where derived data from a tainted source does not acquire a taint, and false positives, where data becomes tainted through an unintended dependency. Additionally, the performance overhead that is imposed by NEON is too high for daily-used applications.

Hybrid approaches aim at combining static and dynamic data flow tracking techniques to optimize certain aspects of data flow trackers. For instance, one aspect could be to mitigate run-time overhead by reducing the number of tracking points inside the AuS or to cope with implicit flows at run-time. Usually, the AuS first undergoes a static analysis phase and subsequently a dynamic analysis phase, based on the previous results.

As a step toward this direction, Chandra et al. [13] present a fine-grained information flow analysis approach for Java-based applications. Under the assumption to have only the Java bytecode available, their approach composes static and dynamic techniques with the aim to make dynamic analysis more intelligent about implicit flows. During the static phase their approach computes *security annotations* that are used to inject compensating commands into the AuS for taken and non-taken control flow paths. At run-time, those annotations reveal which branch is executed and how its execution affects the variable taint-labels within the executed and non-executed branches. However, their approach requires to instrument all instructions that might lead to a flow of data including even those that do not contribute to a data flow. In sum, Chandra et al. [13] provides primarily a dynamic data flow tracker which leverages static analysis to also track implicit flows at run-time; however, taking implicit flows into account might result

in less tracking precision, cf. Section 3.3. Contrarily, the present approach uses static analysis to instrument only those program locations that actually lead to a flow of information between a source and a sink. This way, our approach optimizes and reduces run-time performance overhead and improves the precision of tracking results.

JADAL [75] is probably the most related approach to ours. To detect data leaks inside Java-based applications, the authors propose a two-stage approach to detect data flows from high-level inputs to low-level outputs. In the first stage, the AuS is statically analyzed to compute all those program instructions that may contribute to, and thus, are relevant for data leakage. In a second stage, JADAL injects its tracking logic into the AuS only at those program locations that are reported from the previous stage. This way, JADAL monitors only those program instructions at run-time that are relevant for a flow of data, excluding all other program instructions. However, to do that JADAL requires to inject and to spread its tracking logic over the entire AuS's codebase. We argue, that compared to our approach, such an approach negatively affects the portability of the AuS as all three application code parts, i.e. the program, 3rd-party libraries, and Java system-classes (cf. Figure 1.6) are tightly weld together by the tracking logic. Furthermore, our approach provides a semantic model that shows how the flow of data is propagated at run-time, at every single instruction, residing on a chop. Unfortunately, we were not able to benchmark HDFT++ and SHRIFT against JADAL, as we could not get access to JADAL's implementation (even after contacting JADAL's authors several times).

As pure non-interference is too strict to be practical, Rocha et al. [100] propose a hybrid data flow tracking approach in combination with declassification rules in order to allow data flows if they satisfy certain conditions. During the static analysis phase, their approach computes possible dependencies between sinks and sources as well as, based on declassification rules, possible program locations that might downgrade the security level of a dependency path from high to low. This way, a tainted data item may be safely released to a sink if it passes a downgrading instruction at run-time. Although their approach shows promising results, their primary goal is to relax non-interference: the security label of a data flow is only downgraded if certain declassification criteria are satisfied at run-time on a data flow dependency, and thus, would permit that data flows into the corresponding sink. The present work, in contrast, aims to improve the tracking precision with a hybrid data flow tracking approach and monitors only those program instructions that reside on a dependency path. This way, we are able to distinguish if a data flow really happens or not, whereas Rocha et al. assume that a reported data flow will always happen, irrespective of the executed instructions in between of a data flow dependency path. From our point of view the present work and the results from Rocha et al. are complementary to each other; combining both results is a potential candidate for future work.

To use the (statically computed) information flow dependencies at run-time our hybrid approach uses a technique called *Inline Reference Monitor* (IRM) [25]. Different IRM approaches have been proposed in the literature for different abstraction layers [118, 126], including the Java bytecode layer [24]. To enforce security properties, IRMs usually inject code into a target application to intercept and process sensitive events. Typically, this happens inside the monitored process. Our approach, in contrast, offloads and outsources such event processing to an external tracker. This allows us to aggregate data flow tracking results across and from multiple distributed systems, and to have the flexibility of changing policies for data and for applications at runtime, without requiring restarting nor re-instrumenting the running applications.

4.2. Usage Control

As described in Chapter 2, *Usage Control* (UC) is an extension of access control and tackles the question of how data may or may not be used once initial access has been granted. Originally, UC was introduced and coined by Park et al. in [82, 84], and further extended to UCON_{ABC} [83, 107] by the same authors. In particular, the latter one provides a comprehensive formal model which imposes to evaluate obligations before, during, and after the usage of an object. However, their model has a *container*-centric characteristic, i.e. obligation constraints are specified on one particular object-item and do not take into account multiple copies of a protected object.

Hilty et al. investigate the problem of a clear and precise understanding of how data protection requirements have to be specified and enforced in a system [43]. The authors propose to differentiate between *observable*- and *non-observable*-obligations which forms the foundations to distinguish between preventive and detective enforcements [93]. Furthermore, this framework was refined and extended in [42] to *Obligation Specification Language* (OSL), a formal language to specify data usage obligations in the form of propositional-, temporal-, and cardinal-constraints. In [60, 61] Kumari et al. provide a framework to translate high-level OSL-based data usage requirements into low-level, machine-readable policies. Apart from OSL, other usage control frameworks and policy languages have been proposed, like UCON_{ABC} [83], XACML [79], Ponder [17, 117], or PrimeLife Policy Language [6, 12, 113]. Howbeit, a comparison of those policies is provided in [38, 58] and not further discussed at this point.

As OSL's main purpose focuses on the specification of data usage restrictions by the data owner, Pretschner et al. provide in [90] a formal model to characterize enforcement mechanisms in terms of traces on top of OSL. According to the authors, these mechanisms are categorized into *executors* (which allows a particular data usage), *modifiers* (which modify some parts of a data usage), *inhibitors* (which inhibit a particular data usage), and *delayers* (which delay a

particular data usage). In its original form, UC policies are expressed and enforced on data in a *container*-centric manner, i.e. data usage restrictions are specified in terms of the container name where data was stored. For instance, to protect a secret that is stored in a file “private.txt” from dissemination a possible policy would be “Do not disseminate private.txt”. However, in a data processing system where data may be copied, e.g. file “private.txt" gets copied to “public.txt", a container-centric approach requires to write a UC policy also for “public.txt" and for all other copies as well. As this might become a tedious and cumbersome task if the number of copies grows, Harvan et al. [39] extend the UC model and integrates a generic data flow model which provides a holistic solution to enforce data usage restrictions on all possible copies of a data item. Furthermore, this way it has become possible to specify data usage restrictions in a *data*-centric manner where data usage restrictions are centralized around a concrete data item instead of its representation.

In further works, UC infrastructures have been designed and implemented for different ecosystems, like Android [26], MS Windows [124], MS Office [108], X11 [95], the IP Internet Protocol [53], Thunderbird [69], SIP [50], Grid Computing [14, 73], or Online Social Networks [51, 62]. As those solutions are tailored for one particular abstraction layer they are able to take into account layer-specific domain knowledge in order to track the flow of data and to enforce UC-policies. Thus, depending on the collected domain knowledge tracking and enforcement results are more or less fine-grained. Howbeit, a comprehensive survey on UC enforcement mechanisms and usage control is provided in [67, 78].

5. Conclusion and Future Work

This chapter summarises and concludes the present work in Section 5.1 and discusses and points out future work in Section 5.2.

5.1. Conclusion

The research field of DUC tackles the fundamental question of how data may or may not be used once initial access has been granted by the data owner. In order to implement such functionality, modern DUC systems heavily rely on IFT systems to capture the distinction between data and its concrete representation. This way, it is possible to track and to enforce DUC policies on all copies of a policy affected data item. In recent years, a body of work has been published in the literature that addresses this aspect and proposes different approaches and solutions to track the flow of data. However, as pointed out in Section 1.3 those solutions (i) suffer from *overapproximation* (Caveat 1) [39, 53, 124], (ii) or they impose too much performance overhead (Caveat 2) [54], (iii) or they affect the portability of the AuS (Caveat 3)[10].

To address those caveats, this thesis contributes with a novel data flow tracking approach for DUC systems, and addresses the research questions (cf. Section 1.3) *How can we improve the tracking precision, and thus, reduce overapproximation of the tracking results compared to a pure black-box approach?* **(RQ1)**, *How can we track efficiently and effectively data flows inside an AuS from its input- to its output-channels, and simultaneously, preserve and maintain the portability of the AuS?* **(RQ2)**, *How can we track the flow of data separately inside different AuS that share the same RTE?* **(R3)**.

To address these research questions, this work contributes with a two-staged hybrid data flow tracking solution which combines static and dynamic data flow analysis techniques. The first stage statically analyzes the AuS on possible data flow dependencies and dumps the result into an analysis-report. This stage detects all instructions that correspond either to a source (i.e. data flows into the AuS), a sink (i.e. data flows out of the AuS), or an instruction that resides on a data flow dependency path. The second stage uses those results and injects a minimal run-time monitor into the AuS which monitors only those program locations that are reported in the analysis-report, as only those program locations are actually relevant for a flow of data.

We propose two variants of instrumentation: SHRIFT and its extension HDFT++ (cf. Chapter 3). While SHRIFT monitors source and sink instructions only, and propagates the flow of data based on a static mapping, HDFT++ takes also into account all instructions that reside on a data flow dependency path from a source to a sink. This way, HDFT++ is able to take a more accurate decision if a data flow path is still critical once the program flow reaches a sink instruction (cf. Listing 3.4). However, this surplus is implemented on the costs of a slightly worse run-time performance overhead compared to SHRIFT, as our evaluation results reveal in Section 3.6.3. By design SHRIFT and HDFT++ are able to inject their tracking logic only at the program-code level (cf. Figure 1.6) and abstract away shared third-party- or Java system-classes. This way, they preserve the portability of the AuS as they rely only on tracking logic inside the program-code, and thus, are able to execute the monitored AuS on any off-the-shelf JVM. In contrast, compared solutions from the literature, like PHOSPHOR [10], also require to inject tracking logic inside the Java system-classes or the JVM, and therefore, negatively affect the AuS's portability.

In sum, our solution contributes to modern DUC systems by mitigating the overapproximation concerns (cf. Figure 1.4) and increasing the portability of the monitored AuS. Of course, someone may argue that existing data flow trackers, like PHOSPHOR [10], may be sufficient to track the flow of data for DUC policy enforcement and there is no need to keep the AuS portable. But in the end, however, we learned that this point of view highly depends on the setting and environment where the flow of data has to be tracked for DUC policy enforcement. For instance, we would not benefit that much from the portability advantages if applying our proposed approach on a single Java-based application, which runs in a single Java process. In that case, we could also simply use e.g. PHOSPHOR which disseminates its tracking logic over the entire process. However, for a setting where multiple applications run within the same Java process, as it happens for web-applications in our running scenario (Section 1.2), our approach definitely provides great benefits. With our approach we are able to track the flow of data selectively, modularized within different applications, running within the same process.

5.2. Future Work

The results presented in this thesis provide a basement in several directions for future work.

Static information flow analysis is a fundamental element in our overall approach as our run-time data flow tracker is based on the generated analysis-report. Therefore, the more powerful static information flow analysis performs, the more precise we can track the flow of data at run-time. However, current state-of-the-art static information flow analyzers are not able to handle properly *reflection* during the analysis. Although some solutions have been proposed in

the literature [11], a general approach for reflection-handling has not been found. *Callbacks* are another type of programming language constructs that challenge static information flow analyzer. The main problem is, that callbacks are triggered and called at any point in time by components that are outside of the process, e.g. a mouse click event is passed via the operating system into the application where a callback-method handles the event properly. Therefore, callbacks are pretty hard to statically analyze with respect to information flow.

Extension for distributed applications: This thesis focuses on applications that run at most in one process. Applications whose logic is spread across different processes or across multiple machines are not covered by our solution. Kelbert [52] provides already a data flow model to track data flows in a distributed setting. Integrating his solution with ours would provide a solution where data flows are tracked comprehensively in a distributed setting.

Evaluation: This thesis's evaluation shows that the precision improvement compared to a pure black-box approach highly depends on the chosen analysis configuration. Considering explicit flows only provides a better precision result in all our experiments, than taking into account additional implicit flows. However, HDFT++ monitors not only source and sink instructions, as SHRIFT do, but also all instructions in between which contribute to the flow of data. This way, HDFT++ is able to assess if a data flow is still critical or not once a sink is reached (cf. Section 3.4). In order to understand and to provide a formal description about the actual precision gain of HDFT++ compared to SHRIFT further investigations and experiments are required. As one may notice, quantifying such an improvement in general is a non-trivial task as the precision gain of HDFT++ highly depends on actual run-time values, which influence if a particular code may be executed or not at run-time (cf. variable `condition` in Listing 3.4).

Declassification: The research field of *declassification* provides means and mechanisms to specify and enforce declassification policies, and thus, to relax non-interference [33], i.e. reducing the sensitivity level of a data flow dependency. For instance, consider that a data item gets encrypted on a data flow dependency path at run-time. Under the assumption, that only the data owner knows the private key for decryption, a declassification mechanism may allow to release encrypted data into a sink as only the private-key owner is able to decrypt and recover the original data. However, extending HDFT++ and SHRIFT with the notion of declassification will definitely contribute reducing the overapproximation problem, and thus, to provide a more precise PIP-state. As already suggested in Section 4.1, combining and complementing the present results with the work from Rocha et al. [100] will be definitely a potential candidate for future work. In the end, such an extension contributes to DUC systems as it allows to take data usage policy decisions more precisely only on policy-affected data items.

A. Appendix

A.1. Analysis reports

This section shows the complete analysis-reports for our examples in Listing 3.5 (cf. Listing A.1) and Listing 3.19 (cf. Listing 3.20). Those analysis-reports show all sources, sinks, and chopCMDs (cf. chop-tags) of all detected data flow dependencies.

Listing A.1: The complete static analysis-report from the example in Listing 3.5, where a secret value is read from a file and transmitted to a remote server. Depending on a random value, the secret value is overwritten with a public value.

```
<source>
  <id>Source0</id>
  <location>SampleReadSend.readFile(Ljava/lang/String;)...
    ...Ljava/lang/String;:51</location>
  <signature>java.io.BufferedReader.readLine()Ljava/lang/String;</signature>
  <return/>
</source>
<sink>
  <id>Sink0</id>
  <location>SampleReadSend.sample()V:195</location>
  <signature>java.io.PrintWriter.write(Ljava/lang/String;)V</signature>
  <param index="1"/>
</sink>
<flows>
  <sink id="Sink0">
    <source id="Source0"/>
    <chop>
      <chopNode bci="-8" lab="PHI v20 = #(), v19"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="6" lab="v5 = new java.io.FileReader"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="11" lab="v5.&lt;init&gt;(p1)"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="16" lab="v7 = new java.io.BufferedReader"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="22" lab="v7.&lt;init&gt;(v5)"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
      <chopNode bci="27" lab="goto 51"
        om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
```

```
30        <chopNode bci="30" lab="v12 = new java.lang.StringBuilder"
31          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
32        <chopNode bci="35" lab="v14 = valueOf(v20)"
33          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
34        <chopNode bci="38" lab="v12.&lt;init&gt;(v14)"
35          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
36        <chopNode bci="42" lab="v17 = v12.append(v10)"
37          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
38        <chopNode bci="45" lab="v19 = v17.toString()"
39          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
40        <chopNode bci="51" lab="v10 = v7.readLine()"
41          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
42        <chopNode bci="56" lab="if (v10 != #(null)) goto 30"
43          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
44        <chopNode bci="60" lab="return v20"
45          om="SampleReadSend.readFile(Ljava/lang/String;)Ljava/lang/String;"/>
46        <chopNode bci="-8" lab="PHI v41 = v36, v40"
47          operation="assign" om="SampleReadSend.sample()V"/>
48        <chopNode bci="107" lab="v36 = this.readFile(#(public.txt))"
49          om="SampleReadSend.sample()V"/>
50        <chopNode bci="114" lab="if (v7 &lt;= #(5)) goto 125"
51          om="SampleReadSend.sample()V"/>
52        <chopNode bci="120" lab="v40 = this.readFile(#(secret.txt))"
53          om="SampleReadSend.sample()V"/>
54        <chopNode bci="125" lab="v42 = new java.net.Socket"
55          om="SampleReadSend.sample()V"/>
56        <chopNode bci="131" lab="v43 = v32[#(0)]"
57          om="SampleReadSend.sample()V"/>
58        <chopNode bci="134" lab="v44 = v32[#(1)]"
59          om="SampleReadSend.sample()V"/>
60        <chopNode bci="135" lab="v46 = parseInt(v44)"
61          om="SampleReadSend.sample()V"/>
62        <chopNode bci="138" lab="v42.&lt;init&gt;(v43, v46)"
63          om="SampleReadSend.sample()V"/>
64        <chopNode bci="143" lab="v48 = java.lang.System.out"
65          om="SampleReadSend.sample()V"/>
66        <chopNode bci="146" lab="v49 = new java.lang.StringBuilder"
67          om="SampleReadSend.sample()V"/>
68        <chopNode bci="152" lab="v49.&lt;init&gt;(#(Sending data to ))"
69          om="SampleReadSend.sample()V"/>
70        <chopNode bci="157" lab="v52 = v32[#(1)]"
71          om="SampleReadSend.sample()V"/>
72        <chopNode bci="158" lab="v54 = v49.append(v52)"
73          om="SampleReadSend.sample()V"/>
74        <chopNode bci="163" lab="v57 = v54.append(#( ))"
75          om="SampleReadSend.sample()V"/>
76        <chopNode bci="167" lab="v59 = v57.append(v7)"
77          om="SampleReadSend.sample()V"/>
78        <chopNode bci="170" lab="v61 = v59.toString()"
79          om="SampleReadSend.sample()V"/>
```

```
80        <chopNode bci="173" lab="v48.println(v61)"
81          om="SampleReadSend.sample()V"/>
82        <chopNode bci="176" lab="v63 = new java.io.PrintWriter"
83          om="SampleReadSend.sample()V"/>
84        <chopNode bci="182" lab="v65 = v42.getOutputStream()"
85          om="SampleReadSend.sample()V"/>
86        <chopNode bci="186" lab="v63.&lt;init&gt;(v65, #(1))"
87          om="SampleReadSend.sample()V"/>
88        <chopNode bci="195" lab="v63.write(v41)"
89          om="SampleReadSend.sample()V"/>
90      </chop>
91    </sink>
92  </flows>
```

Listing A.2: The full analysis-report for Listing 3.19. It shows a data flow dependency which contains an inheritance relation on its path from a source to a sink.

```
1  <source>
2    <id>Source0</id>
3    <location>..main(.)V:38</location>
4    <signature>..source(Ljava/lang/String;)Ljava/lang/String;</signature>
5    <return/>
6  </source>
7  <sink>
8    <id>Sink0</id>
9    <location>..main(.)V:113</location>
10   <signature>..sink(Ljava/lang/String;)V</signature>
11   <param index="1"/>
12 </sink>
13 <flows>
14 <sink id="Sink0">
15   <source id="Source0"/>
16   <chop>
17     <chopNode bci="1" lab="<init>;()" om="Copy.<init>;()V"/>
18     <chopNode bci="0" lab="v4 = java.lang.System.out" om="Copy.copy([B)[B"/>
19     <chopNode bci="5" lab="v4.println(#(Copy))" om="Copy.copy([B)[B"/>
20     <chopNode bci="10" lab="v8 = new []" om="Copy.copy([B)[B"/>
21     <chopNode bci="12" lab="return v8" om="Copy.copy([B)[B"/>
22
23     <chopNode bci="1" lab="this.<init>;()" om="CopyByRef.<init>;()V"/>
24     <chopNode bci="0" lab="v4 = java.lang.System.out" om="CopyByRef.copy([B)[B"/>
25     <chopNode bci="5" lab="v4.println(#(CopyByRef))" om="CopyByRef.copy([B)[B"/>
26     <chopNode bci="9" lab="v7 = p1" om="CopyByRef.copy([B)[B"/>
27     <chopNode bci="11" lab="return p1" om="CopyByRef.copy([B)[B"/>
28
29     <chopNode bci="1" lab="this.<init>;()" om="CopyByValue.<init>;()V"/>
30     <chopNode bci="-8" lab="PHI v17 = #(0), v15" om="CopyByValue.copy([B)[B"/>
31     <chopNode bci="0" lab="v4 = java.lang.System.out" om="CopyByValue.copy([B)[B"/>
32     <chopNode bci="5" lab="v4.println(#(CopyByValue))" om="CopyByValue.copy([B)[B"/>
33     <chopNode bci="9" lab="v7 = p1.length" om="CopyByValue.copy([B)[B"/>
```

```
34      <chopNode bci="10" lab="v8 = new []" om="CopyByValue.copy([B)[B"/>
35      <chopNode bci="12" lab="v9 = v8" om="CopyByValue.copy([B)[B"/>
36      <chopNode bci="14" lab="v11 = #(0)" om="CopyByValue.copy([B)[B"/>
37      <chopNode bci="15" lab="goto 29" om="CopyByValue.copy([B)[B"/>
38      <chopNode bci="22" lab="v13 = p1[v17]" om="CopyByValue.copy([B)[B"/>
39      <chopNode bci="23" lab="v8[v17] = v13" om="CopyByValue.copy([B)[B"/>
40      <chopNode bci="24" lab="v15 = v17 + #(1)" om="CopyByValue.copy([B)[B"/>
41      <chopNode bci="29" lab="v12 = p1.length" om="CopyByValue.copy([B)[B"/>
42      <chopNode bci="30" lab="if (v17 < v12) goto 22" om="CopyByValue.copy([B)[B"/>
43      <chopNode bci="34" lab="return v8" om="CopyByValue.copy([B)[B"/>
44
45      <chopNode bci="38" lab="v20 = source(#(Test))" om="..main(.)V"/>
46      <chopNode bci="41" lab="v22 = v20.getBytes()" om="..main(.)V"/>
47      <chopNode bci="44" lab="v23 = v22" om="..main(.)V"/>
48      <chopNode bci="46" lab="v25 = new Copy[]" om="..main(.)V"/>
49      <chopNode bci="51" lab="v27 = new Copy" om="..main(.)V"/>
50      <chopNode bci="55" lab="v27.<init>;()" om="..main(.)V"/>
51      <chopNode bci="58" lab="v25[#(0)] = v27" om="..main(.)V"/>
52      <chopNode bci="61" lab="v30 = new CopyByRef" om="..main(.)V"/>
53      <chopNode bci="65" lab="v30.<init>;()" om="..main(.)V"/>
54      <chopNode bci="68" lab="v25[#(1)] = v30" om="..main(.)V"/>
55      <chopNode bci="71" lab="v33 = new CopyByValue" om="..main(.)V"/>
56      <chopNode bci="75" lab="v33.<init>;()" om="..main(.)V"/>
57      <chopNode bci="78" lab="v25[#(2)] = v33" om="..main(.)V"/>
58      <chopNode bci="79" lab="v35 = v25" om="..main(.)V"/>
59      <chopNode bci="81" lab="v36 = new java.util.Random" om="..main(.)V"/>
60      <chopNode bci="85" lab="v36.<init>;()" om="..main(.)V"/>
61      <chopNode bci="89" lab="v39 = v36.nextInt(#(3))" om="..main(.)V"/>
62      <chopNode bci="92" lab="v40 = v39" om="..main(.)V"/>
63      <chopNode bci="98" lab="v41 = v25[v39]" om="..main(.)V"/>
64      <chopNode bci="100" lab="v43 = v41.copy(v22)" om="..main(.)V"/>
65      <chopNode bci="103" lab="v44 = v43" om="..main(.)V"/>
66      <chopNode bci="105" lab="v45 = new java.lang.String" om="..main(.)V"/>
67      <chopNode bci="110" lab="v45.<init>;(v43)" om="..main(.)V"/>
68      <chopNode bci="113" lab="sink(v45)" om="..main(.)V"/>
69    </chop>
70  </sink>
71  </flows >
```

Bibliography

[1] J. J. Alferes, F. Banti, and A. Brogi. “An Event-Condition-Action Logic Programming Language”. In: *Logics in Artificial Intelligence: 10th European Conference, JELIA 2006 Liverpool, UK, September 13-15, 2006 Proceedings*. Ed. by M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 29–42. ISBN: 978-3-540-39627-7. DOI: 10.1007/11853886_5.

[2] *Amazon's Elastic Beanstalk*. https://docs.aws.amazon.com/de_de/elasticbeanstalk/latest/dg/concepts.platforms.html.

[3] T. Amtoft and A. Banerjee. “Information Flow Analysis in Logical Form”. In: *Static Analysis*. Springer Berlin Heidelberg, 2004.

[4] T. Amtoft, S. Bandhakavi, and A. Banerjee. “A Logic for Information Flow in Object-oriented Programs”. In: vol. 41. 1. New York, NY, USA: ACM, Jan. 2006, pp. 91–102. DOI: 10.1145/1111320.1111046.

[5] L. O. Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. University of Cophenhagen, 1994.

[6] C. A. Ardagna et al. “PrimeLife Policy Language”. In: *W3C Workshop on Access Control Application Scenarios*. Nov. 2009, pp. 1–6. ISBN: 978-88-97253-00-6.

[7] S. Arzt and E. Bodden. “StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 725–735. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884816.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps”. In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 259–269.

[9] J. Banatre, C. Bryce, and D. L. Métayer. *Compile-Time Detection of Information Flow in Sequential Programs*. 1994.

[10] J. Bell and G. Kaiser. “Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. Portland, Oregon, USA: ACM, 2014, pp. 83–101. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660212.

[11] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. “Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders”. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011.

[12] L. Bussard, G. Neven, and F.-S. Preiss. “Downstream Usage Control”. In: *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*. July 2010, pp. 22–29. DOI: 10.1109/POLICY.2010.17.

[13] D. Chandra and M. Franz. “Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine”. In: *23rd Annual Computer Security Applications Conference*. Dec. 2007, pp. 463–475. DOI: 10.1109/ACSAC.2007.37.

[14] M. Colombo, F. Martinelli, P. Mori, and A. Lazouski. “On Usage Control for GRID Services”. In: *International Joint Conference on Computational Sciences and Optimization*. Vol. 1. Apr. 2009, pp. 47–51. DOI: 10.1109/CSO.2009.479.

[15] G. Cugola and A. Margara. “Processing Flows of Information: From Data Stream to Complex Event Processing”. In: *ACM Comput. Surv.* 44.3 (June 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320.

[17] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. “The Ponder Policy Specification Language”. English. In: *Policies for Distributed Systems and Networks*. Ed. by M. Sloman, E. Lupu, and J. Lobo. Vol. 1995. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 18–38. ISBN: 978-3-540-41610-4. DOI: 10.1007/3-540-44569-2_2.

[18] S. De Capitani di Vimercati. “Access Control Policies, Models, and Mechanisms”. In: *Encyclopedia of Cryptography and Security*. Ed. by H. C. A. van Tilborg and S. Jajodia. Boston, MA: Springer US, 2011, pp. 13–14. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_806.

[19] B. Demsky. “Cross-application Data Provenance and Policy Enforcement”. In: *ACM Trans. Inf. Syst. Secur.* (2011).

[20] D. E. Denning. “A Lattice Model of Secure Information Flow”. In: *Comm. ACM* (1976).

[21] D. E. Denning and P. Denning. “Certification of Programs for Secure Information Flow”. In: *Comm. ACM* (1977).

[22] D. Durham, J. Boyle, R. Cohen, R. Rajan, S. Herzog, and A. Sastry. *RFC 2748: The COPS (Common Open Policy Service) Protocol*. 2000.

[23] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 393–407.

[24] Ú. Erlingsson. “The Inlined Reference Monitor Approach to Security Policy Enforcement”. AAI3114521. PhD thesis. Ithaca, NY, USA, 2004.

[25] Ú. Erlingsson and F. B. Schneider. “IRM Enforcement of Java Stack Inspection”. In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. 2000.

[26] D. Feth and A. Pretschner. “Flexible Data-Driven Security for Android”. In: *2012 IEEE Sixth International Conference on Software Security and Reliability*. June 2012, pp. 41–50. DOI: 10.1109/SERE.2012.14.

[27] *Frama-C*. http://frama-c.com/.

[28] A. Fromm and V. Stepa. “HDFT++ Hybrid Data Flow Tracking for SaaS Cloud Services”. In: *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*. June 2017, pp. 12–19. DOI: 10.1109/CSCloud.2017.17.

[29] A. Fromm, F. Kelbert, and A. Pretschner. “Data Protection in a Cloud-Enabled Smart Grid”. In: *Smart Grid Security*. Ed. by J. Cuellar. Vol. 7823. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 96–107. ISBN: 978-3-642-38029-7. DOI: 10.1007/978-3-642-38030-3_7.

[30] R. Gay, J. Hu, and H. Mantel. “CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement”. In: *Information Systems Security: 10th International Conference, ICISS 2014, Hyderabad, India, December 16-20, 2014, Proceedings*. Ed. by A. Prakash and R. Shyamasundar. Cham: Springer International Publishing, 2014, pp. 378–398. ISBN: 978-3-319-13841-1. DOI: 10.1007/978-3-319-13841-1_21.

[31] D. Giffhorn. “Slicing of Concurrent Programs and its Application to Information Flow Control”. PhD thesis. Karlsruher Institut für Technologie, 2012.

[32] D. Giffhorn and G. Snelting. “A new algorithm for low-deterministic security”. In: *International Journal of Information Security* 14.3 (2015), pp. 263–287. ISSN: 1615-5270. DOI: 10.1007/s10207-014-0257-6.

[33] J. A. Goguen and J. Meseguer. “Security Policies and Security Models”. In: *1982 IEEE Symposium on Security and Privacy*. Apr. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[34] J. A. Goguen and J. Meseguer. “Unwinding and Inference Control”. In: *1984 IEEE Symposium on Security and Privacy*. Apr. 1984, pp. 75–75. DOI: 10.1109/SP.1984.10019.

[35] J. Graf. “Information Flow Control with SDGs — Improving Modularity, Scalability and Precision for Object Oriented Languages”. Forthcoming. PhD thesis. Karlsruhe Institute of Technology, Department of Informatics, 2014.

[36] D. Grove and C. Chambers. “A Framework for Call Graph Construction Algorithms”. In: *ACM Trans. Program. Lang. Syst.* (2001).

[37] C. Hammer and G. Snelting. “Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs”. In: *International Journal of Information Security* 8.6 (Dec. 2009), pp. 399–422. DOI: 10.1007/s10207-009-0086-1.

[38] W. Han and C. Lei. “A Survey on Policy Languages in Network and Security Management”. In: *Computer Networks* 56.1 (2012), pp. 477–489. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.09.014.

[39] M. Harvan and A. Pretschner. “State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition”. In: *Third International Conference on Network and System Security*. Oct. 2009, pp. 373–380. DOI: 10.1109/NSS.2009.51.

[40] K. Havelund and A. Goldberg. “Verified Software: Theories, Tools, Experiments”. In: ed. by B. Meyer and J. Woodcock. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Verify Your Runs, pp. 374–383. ISBN: 978-3-540-69147-1. DOI: 10.1007/978-3-540-69149-5_40.

[41] K. Havelund and G. Rosu. *Monitoring Java Programs with Java PathExplorer*. Tech. rep. 2001.

[42] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. “A Policy Language for Distributed Usage Control”. In: *Proceedings of the 12th European Conference on Research in Computer Security*. ESORICS’07. Dresden, Germany: Springer-Verlag, 2007, pp. 531–546. ISBN: 3-540-74834-2, 978-3-540-74834-2.

[43] M. Hilty, D. Basin, and A. Pretschner. “On Obligations”. In: *Proceedings of the 10th European Conference on Research in Computer Security*. ESORICS’05. Milan, Italy: Springer-Verlag, 2005, pp. 98–117. ISBN: 3-540-28963-1, 978-3-540-28963-0. DOI: 10.1007/11555827_7.

[44] S. Horwitz, T. Reps, and D. Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: ACM, 1988, pp. 35–46. ISBN: 0-89791-269-1. DOI: 10.1145/53990.53994.

[45] *Java*. https://java.com/de/.

[46] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. “ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 235–246. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516704.

[47] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. “A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware”. In: *In Proc. of the 19 th NDSS*. 2012.

[48] X. Jin, R. Krishnan, and R. Sandhu. “A Unified Attribute-based Access Control Model Covering DAC, MAC and RBAC”. In: *Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy*. DBSec’12. Paris, France: Springer-Verlag, 2012, pp. 41–55. ISBN: 978-3-642-31539-8. DOI: 10.1007/978-3-642-31540-4_4.

[49] *JOANA*. http://joana.ipd.kit.edu.

[50] G. Karopoulos, P. Mori, and F. Martinelli. “Usage Control in SIP-based Multimedia Delivery”. In: *Comput. Secur.* 39 (Nov. 2013), pp. 406–418. ISSN: 0167-4048. DOI: 10.1016/j.cose.2013.09.005.

[51] F. Kelbert and A. Fromm. “Compliance Monitoring of Third-Party Applications in Online Social Networks”. In: *2016 IEEE Security and Privacy Workshops (SPW)*. May 2016, pp. 9–16. DOI: 10.1109/SPW.2016.13.

[52] F. M. Kelbert. “Data Usage Control for Distributed Systems”. Dissertation. München: Technische Universität München, Mar. 2016.

[53] F. Kelbert and A. Pretschner. “Data usage control enforcement in distributed systems”. In: *Proceedings of the third ACM conference on Data and application security and privacy*. CODASPY ’13. San Antonio, Texas, USA: ACM, 2013, pp. 71–82. ISBN: 978-1-4503-1890-7. DOI: 10.1145/2435349.2435358.

[54] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. "Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems". In: *SIGPLAN Not.* 47.7 (Mar. 2012), pp. 121–132. ISSN: 0362-1340. DOI: 10.1145/2365864.2151042.

[55] R. Keskisärkkä and E. Blomqvist. "Semantic complex event processing for social media monitoring-a survey". In: *Proceedings of Social Media and Linked Data for Emergency Response (SMILE) Co-located with the 10th Extended Semantic Web Conference, Montpellier, France. CEUR workshop proceedings (May 2013)*. 2013.

[56] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27.3 (2015), pp. 573–609. ISSN: 1433-299X. DOI: 10.1007/s00165-014-0326-7.

[57] A. Kulkarni and J. Walker. *RFC 4261: Common Open Policy Service (COPS) Over Transport Layer Security (TLS)*. 2005.

[58] P. Kumaraguru, J. Lobo, L. F. Cranor, and S. B. Calo. "A Survey of Privacy Policy Languages". In: *Proceedings of the 3rd Symposium on Usable Privacy and Security / Workshop on Usable IT Security Management*. ACM, 2007.

[59] P. Kumari. "Model-Based Policy Derivation for Usage Control". PhD thesis. Technische Universität München, Garching b. München, Germany, 2015.

[60] P. Kumari and A. Pretschner. "Deriving implementation-level policies for usage control enforcement". In: *Proc. 2nd ACM Conference on Data and Application Security and Privacy*. San Antonio, Texas, USA, 2012, pp. 83–94. ISBN: 978-1-4503-1091-8. DOI: 10.1145/2133601.2133612.

[61] P. Kumari and A. Pretschner. "Model-Based Usage Control Policy Derivation". In: *Engineering Secure Software and Systems*. Ed. by J. Jürjens, B. Livshits, and R. Scandariato. Vol. 7781. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 58–74. ISBN: 978-3-642-36562-1. DOI: 10.1007/978-3-642-36563-8_5.

[62] P. Kumari, A. Pretschner, J. Peschla, and J.-M. Kuhn. "Distributed Data Usage Control for Web Applications: A Social Network Implementation". In: *Proceedings of the First ACM Conference on Data and Application Security and Privacy*. CODASPY '11. San Antonio, TX, USA: ACM, 2011, pp. 85–96. ISBN: 978-1-4503-0466-5. DOI: 10.1145/1943513.1943526.

[63] B. W. Lampson. "Protection". In: *SIGOPS Oper. Syst. Rev.* 8.1 (Jan. 1974), pp. 18–24. ISSN: 0163-5980. DOI: 10.1145/775265.775268.

[64] C. E. Landwehr. *Protection (Security) Models and Policy*. 1997.

[65] P. A. Laplante. *What Every Engineer Should Know About Software Engineering (What Every Engineer Should Know)*. Boca Raton, FL, USA: CRC Press, Inc., 2007. ISBN: 0849372283.

[66] A. Lazouski, G. Mancini, F. Martinelli, and P. Mori. "Architecture, Workflows, and Prototype for Stateful Data Usage Control in Cloud". In: *2014 IEEE Security and Privacy Workshops*. May 2014, pp. 23–30. DOI: 10.1109/SPW.2014.13.

[67] A. Lazouski, F. Martinelli, and P. Mori. "Usage Control in Computer Security: A Survey". In: *Computer Science Review* 4.2 (2010), pp. 81–99. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2010.02.002.

[68] D. Lienert. "Distributed Usage Control for the MySQL Database Server". Diploma Thesis. Karlsruhe Institute of Technology, Germany, 2012.

[69] M. Lörscher. "Data Usage Control for the Thunderbird Mail Client". MA thesis. University of Kaiserslautern, Germany, 2012.

[70] E. Lovat. "Cross-layer Data-centric Usage Control". PhD thesis. Technische Universität München, Garching b. München, Germany, 2015.

[71] E. Lovat, A. Fromm, M. Mohr, and A. Pretschner. "SHRIFT System-Wide HybRid Information Flow Tracking". In: *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 371–385.

[72] M. Maalej. "Usage Control for Apple iOS". Master Thesis. Technische Universität München, Germany, 2012.

[73] F. Martinelli, P. Mori, and A. Vaccarelli. "Towards Continuous Usage Control on Grid Computational Services". In: *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services*. Oct. 2005, pp. 82–82. DOI: 10.1109/ICAS-ICNS.2005.93.

[74] A. Milanova, A. Rountev, and B. G. Ryder. "Parameterized Object Sensitivity for Points-to Analysis for Java". In: *ACM Trans. Softw. Eng. Methodol.* (2005).

[75] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. "Combining Static and Dynamic Data Flow Analysis: A Hybrid Approach for Detecting Data Leaks in Java Applications". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: ACM, 2015, pp. 1573–1579. ISBN: 978-1-4503-3196-8. DOI: 10.1145/2695664.2695887.

[76] A. C. Myers. “JFlow: Practical Mostly-static Information Flow Control”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’99. San Antonio, Texas, USA: ACM, 1999, pp. 228–241. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292561.

[77] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. *Trishul: A Policy Enforcement Architecture for Java Virtual Machines*. 2008.

[78] Å. A. Nyre. “Usage Control Enforcement - A Survey”. In: *Availability, Reliability and Security for Business, Enterprise and Health Information Systems*. Ed. by A. Tjoa, G. Quirchmayr, I. You, and L. Xu. Vol. 6908. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 38–49. ISBN: 978-3-642-23299-2. DOI: 10.1007/978-3-642-23300-5_4.

[79] Organization for the Advancement of Structured Information Standards (OASIS). “eXtensible Access Control Markup Language (XACML) Version 3.0”. In: *OASIS Standard* (Jan. 2013), pp. 1–154.

[80] *OW2-ASM instrumentation framework*. http://asm.ow2.org/.

[81] V. Pappas, V. Kemerlis, A. Zavou, M. Polychronakis, and A. Keromytis. “CloudFence: Data Flow Tracking as a Cloud Service”. English. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by S. Stolfo, A. Stavrou, and C. Wright. Vol. 8145. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 411–431. ISBN: 978-3-642-41283-7. DOI: 10.1007/978-3-642-41284-4_21.

[82] J. Park and R. Sandhu. “Originator control in usage control”. In: *Proceedings Third International Workshop on Policies for Distributed Systems and Networks*. 2002, pp. 60–66. DOI: 10.1109/POLICY.2002.1011294.

[83] J. Park and R. Sandhu. “The UCONABC Usage Control Model”. In: *ACM Trans. Inf. Syst. Secur.* 7.1 (Feb. 2004), pp. 128–174. ISSN: 1094-9224. DOI: 10.1145/984334.984339.

[84] J. Park and R. Sandhu. “Towards usage control models: beyond traditional access control”. In: *Proc. 7th ACM Symposium on Access Control Models and Technologies*. Monterey, California, USA, 2002, pp. 57–64. ISBN: 1-58113-496-7. DOI: 10.1145/507711.507722.

[85] J. Peschla. “Information Flow Tracking for JavaScript in Chromium”. Master’s Thesis. University of Kaiserslautern, Germany, 2012.

[86] *PHP*. http://php.net/.

[87] *PIN*. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[88] F. Pottier and V. Simonet. "Information Flow Inference for ML". In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 319–330. ISSN: 0362-1340. DOI: 10.1145/565816.503302.

[89] A. Pretschner. "An Overview of Distributed Usage Control". In: *Proc. International Conference on Knowledge Engineering. Knowledge Engineering: Principles and Techniques*. Romania, July 2009.

[90] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. "Mechanisms for Usage Control". In: *Proc. 2008 ACM Symposium on Information, Computer and Communications Security*. Mar. 2008, pp. 240–244. ISBN: 978-1-59593-979-1. DOI: 10.1145/1368310.1368344.

[91] A. Pretschner, M. Hilty, F. Schutz, C. Schaefer, and T. Walter. "Usage Control Enforcement: Present and Future". In: *Security Privacy, IEEE* 6.4 (July 2008), pp. 44–53. ISSN: 1540-7993. DOI: 10.1109/MSP.2008.101.

[92] A. Pretschner, E. Lovat, and M. Büchler. "Representation-Independent Data Usage Control". In: *Data Privacy Management and Autonomous Spontaneus Security*. Vol. 7122. Springer Berlin Heidelberg, 2012, pp. 122–140. ISBN: 978-3-642-28878-4.

[93] A. Pretschner, F. Massacci, and M. Hilty. "Usage Control in Service-Oriented Architectures". In: *Trust, Privacy and Security in Digital Business: 4th International Conference, TrustBus 2007, Regensburg, Germany, September 3-7, 2007. Proceedings*. Ed. by C. Lambrinoudakis, G. Pernul, and A. M. Tjoa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 83–93. ISBN: 978-3-540-74409-2. DOI: 10.1007/978-3-540-74409-2_11.

[94] A. Pretschner, J. Rüesch, C. Schaefer, and T. Walter. "Formal Analyses of Usage Control Policies". In: *International Conference on Availability, Reliability and Security*. ARES. Mar. 2009, pp. 98–105. DOI: 10.1109/ARES.2009.100.

[95] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, T. Walter, et al. "Usage control enforcement with data flow tracking for x11". In: *Proc. 5th Intl. Workshop on Security and Trust Management*. 2009, pp. 124–137.

[96] C. Priebe, D. Muthukumaran, D. O' Keeffe, D. Eyers, B. Shand, R. Kapitza, and P. Pietzuch. "CloudSafetyNet: Detecting Data Leakage Between Cloud Tenants". In: *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*. CCSW '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 117–128. ISBN: 978-1-4503-3239-2. DOI: 10.1145/2664168.2664174.

[97] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. “DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android”. In: *2014 Ninth International Conference on Availability, Reliability and Security*. Sept. 2014, pp. 40–49. DOI: 10.1109/ARES.2014.13.

[98] T. Reps, S. Horwitz, and M. Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. San Francisco, California, USA: ACM, 1995, pp. 49–61. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199462.

[99] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. “Speeding Up Slicing”. In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT ’94. New Orleans, Louisiana, USA: ACM, 1994, pp. 11–20. ISBN: 0-89791-691-3. DOI: 10.1145/193173.195287.

[100] B. Rocha, M. Conti, S. Etalle, and B. Crispo. “Hybrid Static-Runtime Information Flow and Declassification Enforcement”. In: *Information Forensics and Security, IEEE Transactions on* 8.8 (Aug. 2013), pp. 1294–1305. ISSN: 1556-6013. DOI: 10.1109/TIFS.2013.2267798.

[101] *Ruby*. https://www.ruby-lang.org/de/.

[102] A. Russo and A. Sabelfeld. “Dynamic vs. Static Flow-Sensitive Security Analysis”. In: *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*. CSF ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 186–199. ISBN: 978-0-7695-4082-5. DOI: 10.1109/CSF.2010.20.

[103] A. Sabelfeld and A. Myers. “Language-based information-flow security”. In: *Selected Areas in Communications, IEEE Journal on* (2003).

[104] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. “Role-based access control models”. In: *Computer* 29.2 (Feb. 1996), pp. 38–47. ISSN: 0018-9162. DOI: 10.1109/2.485845.

[105] R. S. Sandhu and P. Samarati. “Access Control: Principle and Practice”. In: *Comm. Mag.* 32.9 (Sept. 1994), pp. 40–48. ISSN: 0163-6804. DOI: 10.1109/35.312842.

[106] R. S. Sandhu. “Lattice-Based Access Control Models”. In: *Computer* 26.11 (Nov. 1993), pp. 9–19. ISSN: 0018-9162. DOI: 10.1109/2.241422.

[107] R. Sandhu and J. Park. “Usage Control: A Vision for Next Generation Access Control”. In: *Computer Network Security: Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2003, St. Petersburg, Russia, September 21-23, 2003. Proceedings*. Ed. by V. Gorodetsky, L. Popyack, and V. Skormin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 17–31. ISBN: 978-3-540-45215-7. DOI: 10.1007/978-3-540-45215-7_2.

[108] S. Saxena. “Data Usage Control In Office Application”. Master’s Thesis. Technische Universität München, Germany, 2014.

[109] O. Shivers. “Control Flow Analysis in Scheme”. In: *Proc. PLDI*. 1988.

[110] *Spring*. https://spring.io/.

[111] B. Steensgaard. “Points-to Analysis in Almost Linear Time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 32–41. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237727.

[112] *Tomcat*. http://tomcat.apache.org.

[113] S. Trabelsi, A. Njeh, L. Bussard, and G. Neven. “PPL Engine: A Symmetric Architecture for Privacy Policy Handling”. In: *W3C Workshop on Privacy and data usage control*. W3C, 2010, pp. 1–5. ISBN: 978-88-97253-01-3.

[114] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. “Andromeda: Accurate and Scalable Security Analysis of Web Applications”. English. In: *Fundamental Approaches to Software Engineering*. Ed. by V. Cortellessa and D. Varró. Vol. 7793. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 210–225. ISBN: 978-3-642-37056-4. DOI: 10.1007/978-3-642-37057-1_15.

[115] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. “TAJ: Effective Taint Analysis of Web Applications”. In: *SIGPLAN Not.* 44.6 (June 2009), pp. 87–97. ISSN: 0362-1340. DOI: 10.1145/1543135.1542486.

[116] *Trishul*. http://srijith.net/vu/trishul/publications.php.

[117] K. Twidle, N. Dulay, E. Lupu, and M. Sloman. “Ponder2: A Policy System for Autonomous Pervasive Environments”. In: *Autonomic and Autonomous Systems, 2009. ICAS ’09. Fifth International Conference on*. Apr. 2009, pp. 330–335. DOI: 10.1109/ICAS.2009.42.

[118] D. Vanoverberghe and F. Piessens. “A Caller-Side Inline Reference Monitor for an Object-Oriented Intermediate Language”. In: *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*. 2008.

[119] D. M. Volpano. “Safety Versus Secrecy”. In: *Proceedings of the 6th International Symposium on Static Analysis*. 1999.

[120] D. Volpano, C. Irvine, and G. Smith. “A Sound Type System for Secure Flow Analysis”. In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), pp. 167–187. ISSN: 0926-227X.

[121] D. Wasserrab and D. Lohner. “Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing”. In: *6th International Verification Workshop - VERIFY-2010*. 2010.

[122] P. Wenz. “Data Usage Control for ChromiumOS”. Diploma Thesis. Karlsruhe Institute of Technology, Germany, 2012.

[123] *WhatsApp*. `https://www.whatsapp.com/`.

[124] T. Wüchner and A. Pretschner. “Data Loss Prevention Based on Data-Driven Usage Control”. In: *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. 2012, pp. 151–160. DOI: 10.1109/ISSRE.2012.10.

[125] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. “Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. Alexandria, Virginia, USA: ACM, 2007, pp. 116–127. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315261.

[126] B. Zeng, G. Tan, and Ú. Erlingsson. “Strato: A Retargetable Framework for Low-level Inlined-reference Monitors”. In: *Proceedings of the 22Nd USENIX Conference on Security*. 2013.

[127] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. “Neon: System Support for Derived Data Management”. In: *SIGPLAN Not.* 45.7 (Mar. 2010), pp. 63–74. ISSN: 0362-1340. DOI: 10.1145/1837854.1736008.

Index of Acronyms

AC Access Control

AuS Application under Scrutiny

CEP Complex Event Processing

DDFT Dynamic Data Flow Tracking

DFT Data Flow Tracking

DMA Direct Memory Access

DTA Dynamic Taint Analysis

DUC Data Usage Control

EBNF Extended Backus-Naur Form

ECA Event-Condition-Action

AEB Amazon's Elastic Beanstalk

IaaS Infrastructure as a Service

ILP Implementation Level Policy

IFT Information Flow Tracking

IRM Inline Reference Monitor

JRE Java Runtime Environment

JVM Java Virtual Machine

LTL Linear Temporal Logic

OSL Obligation Specification Language

OSN Online Social Network

PaaS Platform as a Service

PDP Policy Decision Point

PEP Policy Enforcement Point

PDG Program Dependence Graph

PIP Policy Information Point

PMP Policy Management Point

PoI Points of Interest

RTE Runtime Environment

SDG System Dependence Graph

SIFA Static Information Flow Analysis

SLP Specification Level Policy

SSA Static-Single-Assignment

UC Usage Control

UCI Usage Control Infrastructure

JSR Java Specification Requirements

List of Tables

2.1. Technical evaluation of DUC policy operators 29
2.2. Sample DUC policies as ECA-rule representation 32

3.1. Static analysis results concerning precision 63
3.2. Static analysis results concerning SDG size and required computational resources 65
3.3. Run-time performance results 79

List of Figures

1.1. DUC is an extension of AC 2
1.2. Alice protects her transmitted data *D* by a policy 3
1.3. PaaS provider *PaaS-A* provides a shared run-time environment 4
1.4. The overapproximation problem 6
1.5. Software abstraction layer stack 7
1.6. The three code blocks of an application 8

2.1. Usage Control Infrastructure 25
2.2. Sample expression tree 27
2.3. Usage Control Infrastructure 28

3.1. The SHRIFT and HDFT++ approach overview 36
3.2. SHRIFT versus HDFT++ 38
3.3. Sample SDG chop and its corresponding Java source code 40
3.4. BirthdayApp and its modularization into code blocks 85

List of Listings

3.1. Example static analysis-report generated by JOANA 41
3.2. Example sink- source-specification that is used by JOANA 42
3.3. Example Java code fragment for zipping files inside an application 44
3.4. Example of an explicit flow dependency. 46
3.5. Example application which reads data from a file and transmits it to a remote server. 48
3.6. Static analysis-report excerpt for the example code in Listing 3.5. 49
3.7. PIP state only reading `secret.txt`. 51
3.8. PIP state reading `secret.txt` and subsequently `public.txt`. 51
3.9. Data flow dependency crosses a `IF`-command 67
3.10. Static analysis-report for the `IF`-example in Listing 3.9. 68
3.11. HDFT++'s PIP-state withexecuted assignment instruction in Listing 3.9 Line 6 . . 69
3.12. HDFT++'s PIP-state withoutexecuted assignment instruction in Listing 3.9 Line 6 70
3.13. SHRIFT's PIP-state after executing the code in Listing 3.9. 71
3.14. Example of a data flow dependency crossing a WHILE-command 71
3.15. Analysis-report for the `WHILE`-example in Listing 3.14. 72
3.16. HDFT++'s PIP-state withexecuted assignment-statement in Listing 3.14 line 7 . . 73
3.17. HDFT++'s PIP-state withoutexecuted assignment-statement in Listing 3.14 line 7 73
3.18. SHRIFT's PIP-state after executing the code in Listing 3.14. 73
3.19. Example of a data flow dependency passing through an inheritance relation 74
3.20. Analysis-report for the `INHERITANCE`-example in Listing 3.19. 75
3.21. HDFT++'s PIP state with executed `Copy.copy(byte[] a)` method. 76
3.22. HDFT++'s PIP state with executed `CopyByRef.copy(byte[] a)` method. 77
3.23. HDFT++'s PIP state with executed `CopyByValue.copy(byte[] a)` method. . . 77
3.24. SHRIFT's PIP-state after executing the code in Listing 3.20. 77

A.1. The complete static analysis-report from the example in Listing 3.5, where a secret value is read from a file and transmitted to a remote server. Depending on a random value, the secret value is overwritten with a public value. 103
A.2. The full analysis-report for Listing 3.19. It shows a data flow dependency which contains an inheritance relation on its path from a source to a sink. 105