Assessing Software Privacy using the Privacy Flow-Graph

Feiyang Tang
Norwegian Computing Center
Oslo, Norway
feiyang@nr.no

Bjarte M. Østvold
Norwegian Computing Center
Oslo, Norway
bjarte@nr.no

ABSTRACT

We increasingly rely on digital services and the conveniences they provide. Processing of personal data is integral to such services and thus privacy and data protection are a growing concern, and governments have responded with regulations such as the EU's GDPR. Following this, organisations that make software have legal obligations to document the privacy and data protection of their software. This work must involve both software developers that understand the code and the organisation's data protection officer or legal department that understands privacy and the requirements of a Data Protection and Impact Assessment (DPIA).

To help developers and non-technical people such as lawyers document the privacy and data protection behaviour of software, we have developed an automatic software analysis technique. This technique is based on static program analysis to characterise the flow of privacy-related data. The results of the analysis can be presented as a graph of privacy flows and operations—that is understandable also for non-technical people. We argue that our technique facilitates collaboration between technical and non-technical people in documenting the privacy behaviour of the software. We explain how to use the results produced by our technique to answer a series of privacy-relevant questions needed for a DPIA. To illustrate our work, we show both detailed and abstract analysis results from applying our analysis technique to the secure messaging service Signal and to the client of the cloud service NextCloud and show how their privacy flow-graphs inform the writing of a DPIA.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Security and privacy → Social network security and privacy**.

KEYWORDS

Program analysis, data protection and privacy, GDPR, software design documentation

ACM Reference Format:
Feiyang Tang and Bjarte M. Østvold. 2022. Assessing Software Privacy using the Privacy Flow-Graph. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3549035.3561185

MSR4P&S '22, November 18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561185

1 INTRODUCTION

Privacy has been widely discussed in recent years — with the rise in public awareness and associated legislative developments, guaranteeing privacy while processing large amounts of private user data has become an important topic. Following recent law implementations such as the GDPR, we now have a regulated and clear framework for ensuring privacy compliance, which mandates documenting software properties through, for example, a Data Privacy Impact Assessment (DPIA). Such an examination must include all parts of the software and it requires a grasp of the software as well as sufficient technical knowledge to analyse the implementation. As a result, we would anticipate a development team expert who has a brief grasp of the implementation while also having sophisticated analysis and tools at their disposal to assist ensure that critical questions in evaluation frameworks such as DPIA can be answered.

The reality, however, is considerably different. While having a privacy compliance checking process operating alongside a software development life cycle is important, analysis and tools at the code level with tailored assistance to legal experts are insufficient. In the meantime, DPIA questions require an understanding of both technical and legal aspects. This means that performing a successful DPIA cannot be done exclusively by a non-technical Data Protection Officer (DPO) who specialises in data protection policy or a technical professional from the data controller (e.g., a developer in the service provider organisation) with programming experience. Simultaneously, it is difficult for developers to keep track of every single change in terms of private data processing among hundreds of lines of code.

This raises the following question: how can we help both technical developers (from or work for data controllers) and non-technical (DPOs) individuals examine privacy compliance in software? Since tracking the flow of data originating from users is important for privacy protection, we must check sensitive user inputs to the software and use an explainable abstraction to illustrate the privacy behaviours in the software, address privacy elements, and provide assistance in producing a better privacy analysis.

We propose privacy flow-graphs as a means to help both developers and DPOs, they can adopt our technique to discover privacy-related behaviours in software. Such graphs produced by our technique enable documenting private data processing actions, assist organisations (the data controller) in showing compliance with their duties and assist the DPO in carrying out its missions. Illustrating the processes may also assist developers to construct more privacy-compliant software and achieve privacy-by-design throughout development and deployment.

Our contributions are:

- The definition of the privacy flow-graph (Section 3.2)
- How to write a DPIA informed by the privacy flow-graph (Section 4).
- A static program analysis that builds the privacy flow-graphs for Java programs (Section 5).

We demonstrate the utility of our research by examining privacy-related trends in two well-known Java applications: Signal and NextCloud (Section 6).

2 MOTIVATION

Examining data protection compliance is essential for the vast majority of software released to the market, as well as for every service update when new user data must be analysed or when the way data is handled changes. Legal regulations such as the GDPR necessitate that legal experts obtain detailed privacy-related information processes from software developers. This implementation-specific information is typically obtained through manual labour by developers, and may not include all that a legal expert needs.

However, there are developers that are unfamiliar with the existing software and might have difficulties providing in-depth information to legal experts.

This circumstance motivated us to design a lightweight, semi-automated program analysis technique that automatically analyses how and where personal data is accessed and processed, therefore providing software developers and DPOs with a great deal of ease.

3 PRELIMINARIES

In this section, we describe the preliminary aspects of our analysis: the local and global data-flow, the privacy flow-graph, the source and sink methods, and the handcrafted datasets we created to support the analysis.

Let c, d denote classes, n, m methods, and let notation $c.m$ make explicit that class c that declares m. We assume that method names are unique in a class.

3.1 Local Data-Flow in Methods

We define some notation to refer to results obtainable from the control flow graph (CFG) of a method. These results concern the kind of values that may flow between various points either inside the method body.

Definition 3.1 (Method data-flow point p). A data-flow point p associated with a method $c.m$ is one of the following:

`start` – the start of the method;
`invoke` $d.n_i$ – an invocation of method $d.n$;
$\texttt{i\_primitive}_i$ – an input primitive;
$\texttt{o\_primitive}_i$ – an output primitive;
$\texttt{return}_i$ – a return statement.

Definition 3.2 (Local data-flow F; beginning, end). Let p, p' be data-flow points, let F be $p \mapsto p'$ and let $c.m$ be a method. We write $F \trianglelefteq \mathit{CFG}(c.m)$ to means that the control-flow-graph of $c.m$ specifies a *local data-flow* F, that is, that values may flow from p to p'. We refer to p as the *beginning* of F, denoted $\mathit{begin}(F)$ and p' as the *end* of F, denoted $\mathit{end}(F)$.

An invocation can be both a beginning and an end of a flow, whereas the start of the method and an input primitive can only be a beginning, and a return statement and an output primitive can only be an end.

We are concerned with all data-flows that originate from the use of an input primitive. We now define some particular types of flows.

Definition 3.3 (Source flow, F^o). Given method $c.m$ where $(\texttt{i\_primitive}_i \mapsto \texttt{return}_j) \trianglelefteq \mathit{CFG}(c.m)$. This flow is called a *source flow*, denoted F^o.

Definition 3.4 (Sink flow, F^i). Given method $c.m$ where $(\texttt{start} \mapsto \texttt{o\_primitive}_i) \trianglelefteq \mathit{CFG}(c.m)$. The flow is called a *sink flow*, denoted F^i.

3.2 Global Data-Flow & the Privacy Flow-Graph

We now consider global data-flow, specifically data-flows between methods of different classes, those are, all data-flows that start from the use of an input primitive.

We extend the concept of a data-flow from local flows F inside methods to global flows G across methods. A global data-flow is defined by a series of local data-flows, each corresponding to a method invocation, and that satisfies certain conditions.

Definition 3.5 (Global data-flow G). A *global data-flow* G is finite series of two or more local data flows, $F_1 \cdots F_n$. The notions of beginning and end extend to G in an obvious way. Furthermore, any F_k, F_{k+1} above must satisfy the following: Let $c_k.m_k$ be such that $F_k \trianglelefteq \mathit{CFG}(c_k.m_k)$ and $c_{k+1}.m_{k+1}$ such that $F_{k+1} \trianglelefteq \mathit{CFG}(c_{k+1}.m_{k+1})$ and $\mathit{end}(F_k) = \texttt{return}_i$ and $\mathit{begin}(F_{k+1}) = \texttt{invoke}\ c_k.m_{k\,j}$ for some i, j.

A global data-flow $G = F_1 \ldots F_n$ is a *privacy flow* if F_1 is a source flow. We are especially interested in global data-flows that involve data from input primitives ending up in output primitives.

Let P be a program with privacy flows $G_1, \ldots, G_n$. The *privacy flow-graph* is a graph where there the nodes are all methods involved in a privacy flow and the edges are pairs of methods involved in successive flows F_k, F_{k+1} part of some G_j.

3.2.1 Java specifics. Here we consider some issues in adapting our data-flow definitions to Java.

First, we define rich types with the intuition that we are only interested in flows that involve values of these kinds of types.

Definition 3.6 (Rich type). A *rich type* is any of following: the primitive data types `string`, `int`, `byte`, the object types, as well as arrays of rich types.

Values of rich types are those values that may contain privacy-related information. In principle, a *boolean* could also be relevant to privacy, but we limit our scope to the rich types to simplify our task. We are concerned with the processing of privacy-related data and not with the leakage of bits of privacy information stemming from such processing.

All non-trivial programs refer to either standard libraries or third-party libraries and thus source flows and sink flows may take place inside the methods of these libraries. In order to include these flows without analyzing the libraries, we introduce the concept of source methods and sink methods where such flows happen, and we apply a separate library analysis to pre-build a collection of source and sink methods.

A *source method* is a method whose invocation results in a source flow, and we denote it as *om*. A *sink method* is a method whose invocation results in a sink flow, denoted *im*.

3.2.2 Library analysis. We have manually constructed a dataset of source and sink methods in the native Java library[1] as well as the most used third-party Java libraries across different categories[2]. The third-party libraries were selected from the Maven Repository list based on their download frequency[3]. There are 158 Java source methods and 257 third-party library methods, which are divided into five groups based on the return data type. Table 1 displays three Java source method samples and three from third-party libraries.

Similarly, we created a dataset that included 350 sink methods from the same Java and 365 sink methods from the third-party libraries we investigated for the source method. Five examples of sink methods are displayed below in Table 2.

A global privacy data-flow is made up of many nodes that represent various methods. Different methods imply different types of data processing; to help demonstrate these processes, we characterise *process* under four categories.

Definition 3.7 (Process). A process is a local data-flow F in a *privacy flow* $G = F_1 \dots F_n$ that is not a source flow F^o or a sink flow F^i.

To specify some special kinds of processes, we use the following separate terms:

- Security process, if a process involves cryptography, database, security, or network packages.
- Authentication process, if authentication is involved.
- Initialisation process, if a process initialises a class.
- Non-privacy process, if it does not belong to either of the three categories above.

4 ASSESSING DATA PRIVACY

It is challenging for software developers and legal privacy experts to have a mutual understanding and benefit from each other's expertise and insights. To address this, we examine how to leverage information from data flows in software to answer particular concerns related to GDPR rules. According to Article 4 in GDPR, *"the data controller determines the purposes for which and the means by which personal data is processed"*; hence, software providers (organisations) are data controllers if the organisation develops its own software. Otherwise, the software developers provide the implementation to the data controllers who are responsible for privacy protection. In this paragraph, we first look at the core GDPR obligations of the data controller, which serves as the duty of DPOs, and then discuss how we may help DPOs answer key DPIA questions (the document created by the approach in this study is referred to as a DPIA.).

4.1 Obligation of the Data Controller

Article 24 in the GDPR [9] states several obligations of the data controller which should be monitored by the DPO:

- by default and by design, the data controller should have a record of processing activities (Article 30);
- to ensure the security of the processing (Article 32);
- to notify personal data breaches to the supervisory authorities (Article 33);
- to communicate personal breaches to the data subject (article 34)
- to conduct DPIA (Article 35);
- to conduct prior consultation with supervisory authorities (Article 36).

The DPOs' role is to monitor whether the data controller fulfilled all of their commitments, which includes performing a DPIA when required. The writing of a DPIA is a shared duty for data controllers and DPOs.

As one of the major data protection authorities in Europe, the Irish Data Protection Commission [8] provides a short explanation of what DPIA contains:

> "*A DPIA describes a process designed to identify risks arising out of the processing of personal data and to minimise these risks as far and as early as possible.*"

Here we picked one of the most often used sample templates for generating a DPIA from the British Information Commissioner's Office (ICO) [20].

Under *Section 2: Describe the processing* of the template, there are three questions:

- Describe the nature of the processing: how will you collect, use, store and delete data? What is the source of the data? Will you be sharing data with anyone? You might find it useful to refer to a flow-graph or another way of describing data flows. What types of processing identified as likely high risk are involved?
- Describe the scope of the processing: what is the nature of the data, and does it include special category or criminal offence data? How much data will you be collecting and using? How often? How long will you keep it? and more
- Describe the context of the processing: what is the nature of your relationship with the individuals? How much control will they have?

Also under *Step 5: Identify and assess risks*, DPIA requires *"Describe the source of risk and nature of the potential impact on individuals."*

With a list of privacy data-flows listed under different categories, developers and DPOs could identify the parts of the program that collect privacy data from users and the relevant risky sinks. As a result of identifying privacy flows, they can pinpoint exposure risks and offer solutions to minimise those risks.

4.2 Answering Key DPIA Questions

Based on the previous paragraph, we now define six key questions relevant to the DPIA. Software development teams and DPOs should consider how to answer these questions when writing the DPIA. Each question is followed by an explanation of how our proposed analysis technique can help answer the questions.

Q1 *What is the source & nature of the data?*

[1] Based on JDK 8u201
[2] Jackson, Log4j2, Apache Commons, Guava, HttpClient, JMS, Joda Time, Apache MINA, Apache Commons Codec and Derby
[3] Maven Repository: https://mvnrepository.com/

Table 1: Examples of source methods

Method signature	Category
int java.io.DataInputStream.read(byte[])	I/O
java.lang.String java.net.URL.getQuery()	Network
java.sql.ResultSet java.sql.Statement.getResultSet()	Database
int org.apache.commons.io.input.ProxyInputStream.read(byte[])	I/O
org.apache.http.ssl.SSLContextBuilder org.apache.http.ssl.SSLContextBuilder.loadKeyMaterial()	Network
java.sql.ResultSet org.apache.derby.iapi.jdbc.BrokeredStatement.executeQuery(java.lang.String)	Database

Table 2: Examples of sink methods

Method signature	Category
void java.util.logging.Logger.log(java.util.logging.LogRecord)	Log
void java.io.BufferedWriter.write(int)	I/O
void javax.servlet.http.HttpServletResponse.sendRedirect(java.lang.String)	Network
void com.sun.xml.txw2.output.XMLWriter.comment(char[],int,int)	I/O
java.net.HttpURLConnection org.jsoup.helper.HttpConnection(org.jsoup.Connection)	Network

A1 We need to know where the data is acquired originally and through which way. By having privacy source methods detected from the target program, we are able to look for all the potential locations in which personal data from users might get captured by the system. Different categories of privacy source methods might also indicate the type and nature of the data. For example, a method from `java.io.File` indicates this method reads from a file in the local file system.

Q2 *How is private data processed?*

A2 We want to identify the parts of the program that involve the processing of private data. This is a discovery study based on the flows that stem from privacy source methods. There are many patterns that might provide details on the processing of privacy data, for example, data travel through multiple sources or reach into multiple different sinks.

Q3 *Will the data be transformed? If so, how to ensure privacy data quality?*

A3 Data transformation and quality control can be subtle. There are clues such as the change of data types, certain types of data manipulation methods or certain APIs that might get involved in data transformation such as encryption or database packages.

Q4 *Will the data be shared/transferred and if yes, how?*

A4 Most of the data transportation happens when the privacy data flow into a sink method. By pinpointing the location and type of sink methods, we are able to identify whether there are private data being shared or transferred out of the target program.

Q5 *Does the data collected include special/highly sensitive personal data?*

A5 The property of privacy data need to be manually identified or with the help of developers. By adopting pure logic we can pick up properties that are directly linked with specific input devices of software.

Q6 *How is the data secured?*

A6 The security of private data is ensured when there are data protection mechanisms adopted, for example, the usage of cryptographic libraries or some encrypted databases. By locating the occurrence of these methods, we are able to analyse the data security protection of the target program.

5 IMPLEMENTATION

In the following paragraphs, we explain how our program analysis technique is implemented. Our implementation is built on Soot [16], a Java optimisation framework that provides four intermediate representations for analysing and transforming Java bytecode. Our technique consists of three parts:

- Transforming program bytecode to intermediate representation;
- Finding the source and sink methods;
- Building a privacy flow-graph by constructing one privacy flow for each source method at a time;
- Producing the abstraction extracted from the privacy flow-graph.

5.1 Finding Source and Sink Methods

Soot helps us transform our target program into a 3-address intermediate representation [23]. By traversing the $CFG(c.m)$ of each method $c.m$ in the program (provided in Jimple), the local data-flow analysis helps us detect the occurrences of source and sink methods in the pre-set annotation datasets (om and im) defined in Section 3.2.1. By having a complete list of source and sink methods in the application as O and I, we now use them to start building the privacy flow-graph.

5.2 Building the Privacy Flow-Graph

For every class that includes a detected source method, we mark it as a class-of-interest (COI). For each COI, we first build a complete call-graph for it.

Definition 5.1 (Class-of-interest). A Class-of-interest (COI) is a class that contains an invocation to one of the source methods (O).

$$c \in COI \Leftrightarrow \exists o \in c, o \in O$$

Now for each source method $o \in O$, we build a global data-flow $G_o = F^o \dots F^n$ for it from the call-graphs of each class that G_o passes through. The final output is a union of all the global data-flows originating from source methods. This graph uses $A \rightarrow B$ to represent that method B invokes method A. Each G_o will be output as a separate `dot` file consisting of all the nodes (full signature of methods) and edges (invocations among the methods) which enables users to easily visualise it with simple tools.

5.3 Abstracting the Privacy Flow-Graph

Privacy flows can be lengthy and comprise a variety of non-sensitive processes, many of which are from the same class and are unrelated to privacy protection yet may confound both developers and DPOs. We want to enable DPOs to get a big picture of the important processes without getting bogged down in minutiae by creating an abstraction from the privacy-flow-graphs generated by each source method. The abstraction is powered by a simple Python script running automatically on the initial complete privacy flow-graph. We select several key parts from the complete privacy flow-graph which are listed below as symbols:

- ▲: the starting source method;
- △: a non-starting source method;
- ○: a non-special process;
  Multiple processes that belong to the same package will be grouped into one process symbol in the abstraction.
- ⊗: a security process (cryptography, database, or network);
  A security process is detected by the substring detector, we look for substrings such as 'encrypt', 'db', 'send', 'connect' in the method and its package name.
- ▼: the end sink method;
- ▽: a non-ending sink method;
- ●: the end process;
- ◊: an authentication process;
  Similar to a security process, we report an authentication process when we detect the substring 'auth' in the method or its package name.
- ⊙: initialisation process(es).
  The initialisation process has 'init' in their names which can be picked up by our substring detector.

The above key information can be interpreted to help developers pin down specific issues in code and assist DPOs to have a sketch of high-level privacy patterns in the program, to also better answer the relevant questions in DPIA.

An example abstraction output reflecting the code snippet in Figure ?? is shown below: The example has one obvious source method `read()` (line 7) which acts as the starting point of our analysis. The technique then finds the next invocation to the source method when class `Student` gets initialised (line 16). This initialisation is triggered later by another initialisation of class `Status` (line 24). Following the newly created object `Status s`, we can trace the invocations to `calculate()` (line 28), `encode()` (line 21), `findResult()` (line 31) and finally to a sink `print()` (line 31) which is invoked by the `Main()` method. Source method `read()` and sink method `print()` have their categories labelled as well as the special process `encode()`.

Along with the abstraction figure, we provide short labels with the symbols which consist of information such as 1) categories of starting source method and sink methods; 2) categories of the special processes (security, authentication, or initialisation); 3) the class name is displayed when it is an initialisation process (optional).

6 EXPERIMENT

We are looking for apps that accept raw sensitive user data and entail data transmission, often in messaging and cloud storage applications. We thus selected the following two applications: Signal[4] and NextCloud[5]. The non-profit Signal Foundation and Signal Messenger LLC created Signal, a cross-platform end-to-end instant messaging service. We intend to study how Signal processes privacy-related user data by analysing both Signal's front-end Android application and the Signal Client Service API because of its expertise in end-to-end encryption. The purpose is to figure out how data is taken from the user and sent to the server. NextCloud is a client-server software package for developing and managing file hosting services. It is free and open-source software that anybody may install and run on their own private servers. We chose an implementation of its Client API that assists developers in developing Java apps with NextCloud integration since it is highly configurable. Similar to Signal, we intend to determine how the application handles privacy-sensitive user data.

6.1 Signal

The Signal Service API contains 17,710 lines of code, which might require developers and DPOs significant time and effort to comprehend. With our samples of DPIA answers, DPOs could effortlessly use our implementation results to create a DPIA.

A total number of 11 privacy flows were detected in Signal Service API (9 out of a total 11 are displayed here), the abstraction of its privacy flow-graph is shown below as Figure 2. We categorise the 9 source methods found into four 4 different functionalities. In Signal, we have discovered a similar pattern for all types of data communication: each raw entry is instantly sent into Signal's own cryptography libraries, allowing all user entries to be completely encrypted before they reach any possible sinks or processes. *Signal: Send Message* and *Signal: Receive Message* in Figure 2 demonstrate this end-to-end encryption mechanism. As indicated by the dashed green lines, there are some source methods that accept some values from local fields which originated from source methods in other flows. `PS01`, for example, gets value from source methods `06` and `09`, which are network-related properties associated with the message object.

Now, we answer the DPIA questions we listed in Section 4.2 using the flow that originates from `01` (blue flow) in *Signal: Send Message* of Figure 2. To analyse privacy compliance, we combine the abstraction figure (which only comprises shapes and categories of critical processes) with detailed privacy flow-graphs (which contain

[4] https://signal.org/en/
[5] https://nextcloud.com/

```java
public class Student {
    double id;
    String name;

    Student() {
        //read() is a source method
        this.id = read();
        this.name = "John Doe";
    }

}
public class Status {
    double id;
    String name;
    Status() {
        Student student = new Student();
        this.id = student.id;
        this.name = student.name;
    }
    public static String findResult() {
        return "Result is: " + encode();
    }
    public static String calculate() {
        Status s = new Status();
        return String.valueOf("Out: " + s.id + s.name);
    }
    public static String encode()
        return String.valueOf(calculate().hashCode());
    }
    public static void main(String args[]) {
        System.out.println(findResult());
    }
}
```

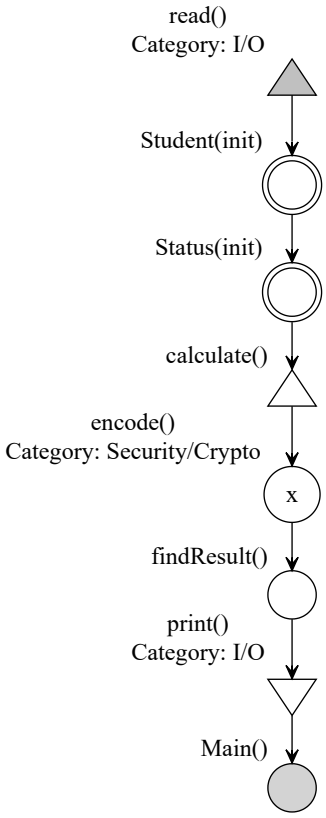

Figure 1: Example of a privacy data-flow generated for a source code fragment and its abstraction

every node in the flow-graph as well as their complete signature), shown as in Table 3.

Q1 *What is the source & nature of the data?*
A1 Android applications take text input from a TE object which is a UI fragment providing a text field for users. The message field contains the raw message users want to send out.
Q2 *How is private data processed?*
A2 The abstraction tells us that there exist multiple processes when the text message is being sent out. There are two non-privacy processes from packages `org.signal.securesms.jobs` and `org.signalservice.api.signalservicemessagesender`. The package names indicate the types of processing behind the processes. There are also highly sensitive privacy processes such as the `MessageContentProcessor()` which is a non-starting source method that takes privacy data from a local field, in this case, it combines multiple privacy data including the text message. `org.signalservice.api.crypto` shows a typical encryption process, this also demonstrates the end-to-end encryption in Signal.
Q3 *Will the data be transformed? If so, how to ensure privacy data quality?*
A3 We notice that the data type gets immediately changed after being read into the device as raw strings. Both non-privacy and privacy processes transform data in order to achieve their functionality. However, encrypted messages stay encrypted before they get sent out, which ensures the content will not get manipulated by external parties.
Q4 *Will the data be shared/transferred and if yes, how?*
A4 The final ending sink method `sendMessage()` sends encrypted message objects out to the server from the client.
Q5 *Does the data collected include special/highly sensitive personal data?*
A5 The properties of the message object are sensitive. Not only the text message body itself, its attributes such as the details of senders but receivers and timestamps also remain sensitive during the entire process.
Q6 *How is the data secured?*
A6 Data security is guaranteed here by end-to-end encryption. All the privacy data related to the message get encrypted together as an `EncryptedMessage` object. This encrypted object cannot be decrypted by the server, which remains unreadable until it reaches the destination client.

Our discovery also supports what Signal claims in its privacy policy. By supplying the aforesaid information to both developers and DPOs, they are able to receive adequate information for creating DPIA and examining the privacy protection status in Signal without having to read the original code.

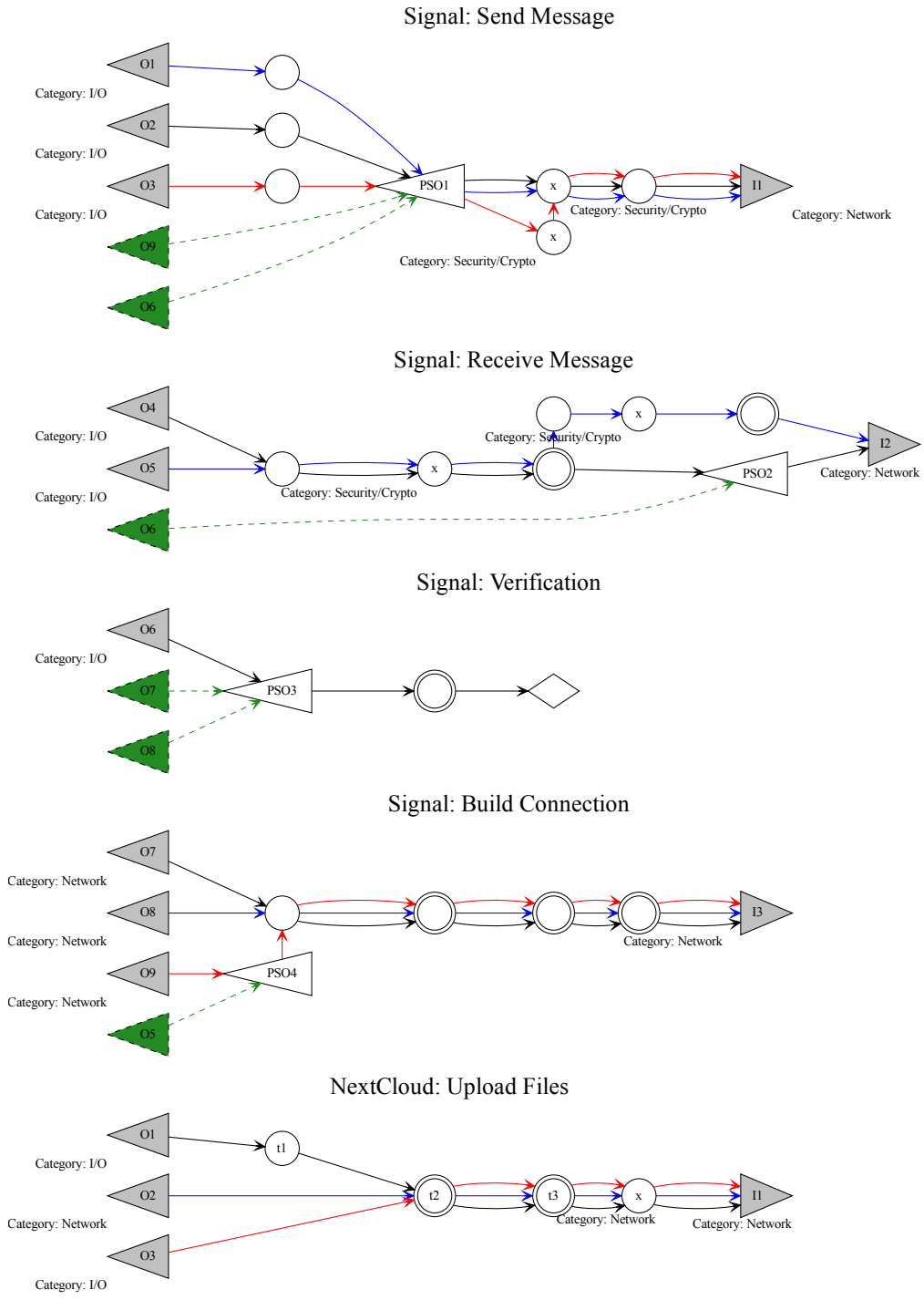

Figure 2: Sample abstract privacy flows for Signal and NextCloud

Table 3: Complete privacy data-flow with abstraction symbols for sending a text message in Signal

Abstraction	Complete privacy data-flow
▲	android.widget.EditText getText()
○	org.thoughtcrime.securesms.jobs.PushTextSendJob deliver(message)
△	org.thoughtcrime.securesms.messages.MessageContentProcessor handleMessage(content, timestamp, ...)
⊗	org.whispersystems.signalservice.api.crypto.SignalServiceCipher encrypt(destination, message, ...)
○	org.whispersystems.signalservice.api.SignalServiceMessageSender getEncryptedMessage(content, recipient, timestamp, ...)
	org.whispersystems.signalservice.api.SignalServiceMessageSender getEncryptedMessages(content, recipient, timestamp, ...)
	org.whispersystems.signalservice.api.SignalServiceMessageSender createMessageContent(message)
▼	org.whispersystems.signalservice.api.SignalServiceMessageSender sendMessage(message, recipient, ...)

6.2 NextCloud

Since NextCloud recently implemented end-to-end encryption in their products, this feature only offers on the level of 'end-to-end encrypted folders'. Hence in our analysis, we only apply the technique to the client API which is applied to the traditional version that relies on TLS communication for safely transferring files.

From a total of 8,923 lines of code, we are able to extract key information from the NextCloud Client API using a simplified privacy flow-graph along with the complete flow-graphs with full signatures, as we did with Signal. We evaluate the DPIA questions to help DPOs in getting information from a legal standpoint, using the abstraction graph derived from our technique in Figure 2.

Q1 *What is the source & nature of the data?*
A1 NextCloud Client API allows a client to upload a new file via `uploadNewFile()`. The files can be of various types but shall be categorised as the user's personal data. There is also one network source, which links with data that can be used to identify users on the Internet.
Q2 *How is private data processed?*
A2 The file is transmitted from the device to the network; this is how a file is sent from the client to the server.
Q3 *Will the data be transformed? If so, how to ensure privacy data quality?*
A3 Not only the file acquired from the user is transferred to the server, but also network data and configuration settings. These various user data are processed and loaded into multiple fields of various class objects (reflect on the two initialisation processes). During these procedures, data types must be transformed in order to be organised for transmission as a type that the server accepts.
Q4 *Will the data be shared/transferred and if yes, how?*
A4 The final node is a network sink, which indicates that the user's data has been transmitted into the network and shared with the server.
Q5 *Does the data collected include special/highly sensitive personal data?*
A5 In this example, the data comprises user files, settings, and network details. User files are highly sensitive in terms of privacy.
Q6 *How is the data secured?*
A6 The network process here depicts a TLS connection, which is a cryptographic technology meant to ensure network communications security.

With the information provided above, we provide both developers and DPOs a better understanding of how the file upload process works in the NextCloud Client API, as well as what and where are the important aspects of privacy protection for NextCloud.

Privacy flow-graphs illustrate trends in terms of privacy-related data processing, including both benign and bad practices. It can assist not just DPOs and developers in responding to DPIA questions and addressing important processing, but also in identifying potentially questionable practices and ensuring good practices on privacy-related data.

7 RELATED WORK

Using static analysis for security bug detection in software [4, 6, 10] is a source of inspiration for our work. In our work, we used handcrafted datasets of source and sink methods for Java and popular third-party libraries as the start point for our analysis. The idea of using a pre-built set as a basis of static analysis is similar to SUSI [2], IccTA [17], MudFlow [3] and AndroidLeak [11] in terms of privacy protection for Android applications. Most current work, including the above, is specific to Android sinks and sources and often uses name features as the basis of their analysis, whereas we focus on Java in general without adopting heuristics. Regarding the GDPR, we demonstrate the utility of employing privacy flow-graphs to ease the DPIA process, which saves manual labour and assists in identifying possible sensitive processes that may be missed by human eyes.

Overall, there is an increasing interest in assuring privacy protection compliance prior to or throughout the software development lifecycle [22]. Privacy-by-design (PbD) has sparked research into methodologies and models for preserving software privacy before implementation begins, as well as forecasting or managing developer privacy compliance throughout implementation [1, 12, 14]. Many of these approaches may also be employed on a regular basis during the development cycle and while updating software. In the era of GDPR in Europe, there is also prior research [5, 13, 15] that aims to provide personalised solutions for DPIA in a variety of applications. According to a survey conducted by Dias Canedo *et al.* [7], technical staff frequently lack legal knowledge regarding privacy protection. Many existing works [18, 19, 21] propose models that limit on a conceptual level, that are not tangible for both technical and non-technical people to apply to implementation, motivating us to propose an automatic technique to analyse privacy compliance in software.

8 CONCLUSION

In terms of privacy protection, there always exists a barrier between developers and DPOs. DPOs need to generate a successful DPIA to document the privacy protection behaviour of software, this requires the developer's comprehensive knowledge of code details. Our work provides a technique for detecting privacy source and sink methods in software bytecode, generating privacy flow-graphs from the discovered sources, and supporting DPOs in writing a DPIA utilising privacy flow-graphs and associated abstractions.

9 LIMITATION AND FUTURE WORK

Our present method requires predetermined source and sink lists. Given that modern applications typically contain hundreds of direct and indirect dependencies, we may miss a significant number of privacy-related sources and sinks. Therefore, we rely on the knowledge of technical specialists to create a more precise list of sources and sinks. Moreover, despite the fact that our complete privacy flow-graphs and their abstractions can express key privacy-sensitive behaviours such as data acquisition, encryption, and transportation, they are unable to provide complete information regarding which type of data manipulation was involved in terms of privacy protection; therefore, developers may be required to provide additional explanation for DPOs.

Future work includes a more detailed local flow analysis for each local data-flow in a privacy global data-flow, such as tracking how values from privacy-related data are modified in the local method and flagging sensitive manipulations such as value accumulation and separation. In the meantime, it is feasible to extract information from the manifest file on which third-party libraries are imported by the software in order to assist in the construction of a more adaptable list of sources and sinks. This procedure might be automated by including these third-party libraries (which are usually downloadable as JAR files) as a part of the input of the analysis. Additionally, since dynamically-typed languages such as JavaScript are used in many different types of modern systems, it would be advantageous to build a source code-based analyser based on tools such as Semgrep [6], which as a starting point for extending our results to web applications.

ACKNOWLEDGMENTS

We appreciate the legal insight that Jan Czarnocki and Lydia Belkadi have given. This work is part of the Privacy Matters (PriMa) project. The PriMa project has received funding from European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No. 860315.

REFERENCES

[1] Thibaud Antignac and Daniel Le Métayer. 2014. Privacy by design: From technologies to architectures. In *Annual privacy forum*. Springer, Berlin, Heidelberg, 1–17.

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks.

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Italy, 426–436. https://doi.org/10.1109/ICSE.2015.61

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[5] Shakila Bu-Pasha. 2020. The controller's role in determining 'high risk' and data protection impact assessment (DPIA) in developing digital smart city. *Information & Communications Technology Law* 29, 3 (2020), 391–402.

[6] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE security & privacy* 2, 6 (2004), 76–79.

[7] Edna Dias Canedo, Angelica Toffano Seidel Calazans, Eloisa Toffano Seidel Masson, Pedro Henrique Teixeira Costa, and Fernanda Lima. 2020. Perceptions of ICT practitioners regarding software privacy. *Entropy* 22, 4 (2020), 429.

[8] Data Protection Commission (DPC). 2022. *Data Protection Impact Assessments*. DPC. Retrieved July 6, 2022 from https://www.dataprotection.ie/en/organisations/know-your-obligations/data-protection-impact-assessments

[9] European Commission. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/eli/reg/2016/679/oj

[10] David Evans and David Larochelle. 2002. Improving security using extensible lightweight static analysis. *IEEE software* 19, 1 (2002), 42–51.

[11] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 291–307.

[12] Irit Hadar, Tomer Hasson, Oshrat Ayalon, Eran Toch, Michael Birnhack, Sofia Sherman, and Arod Balissa. 2018. Privacy by designers: software developers' privacy mindset. *Empirical Software Engineering* 23, 1 (2018), 259–289.

[13] Jane Henriksen-Bulmer, Shamal Faily, and Sheridan Jeary. 2020. DPIA in Context: Applying DPIA to Assess Privacy Risks of Cyber Physical Systems. *Future Internet* 12, 5 (2020), 93.

[14] Jaap-Henk Hoepman. 2014. Privacy Design Strategies. In *ICT Systems Security and Privacy Protection*, Nora Cuppens-Boulahia, Frédéric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 446–459.

[15] Martin Horák, Václav Stupka, and Martin Husák. 2019. GDPR Compliance in Cybersecurity Software: A Case Study of DPIA in Information Sharing Platform. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (Canterbury, CA, United Kingdom) *(ARES '19)*. Association for Computing Machinery, New York, NY, USA, Article 36, 8 pages. https://doi.org/10.1145/3339252.3340516

[16] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. IEEE, Purdue University.

[17] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Italy, 280–291. https://doi.org/10.1109/ICSE.2015.48

[18] Yod-Samuel Martin and Antonio Kung. 2018. Methods and Tools for GDPR Compliance Through Privacy and Data Protection Engineering. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, London, 108–111. https://doi.org/10.1109/EuroSPW.2018.00021

[19] Aaron K Massey, Paul N Otto, Lauren J Hayward, and Annie I Antón. 2010. Evaluating existing security and privacy requirements for legal compliance. *Requirements engineering* 15, 1 (2010), 119–137.

[20] Information Commissioner's Office. 2018. Data Protection Impact Assessments (DPIAs). https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/data-protection-impact-assessments-dpias/. (Accessed on 03/02/2022).

[21] Luca Piras, Mohammed Ghazi Al-Obeidallah, Andrea Praitano, Aggeliki Tsohou, Haralambos Mouratidis, Beatriz Gallego-Nicasio Crespo, Jean Baptiste Bernard, Marco Fiorani, Emmanouil Magkos, Andres Castillo Sanz, et al. 2019. DEFeND architecture: a privacy by design platform for GDPR compliance. In *International Conference on Trust and Privacy in Digital Business*. Springer, Springer, Bratislava, Slovakia, 78–93.

[22] Ira S Rubinstein. 2011. Regulating privacy by design. *Berkeley Tech. LJ* 26 (2011), 1409.

[23] Raja Vallée-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations.

[6] https://semgrep.dev/