Static Analysis for GDPR Compliance

Pietro Ferrara[1] and Fausto Spoto[1,2]

[1] Julia SRL, Verona, Italy
[2] Università di Verona, Italy

Abstract

Information systems might access, manage and record sensitive data about citizens. In addition, the pervasiveness of these systems is dramatically increasing and increasing thanks to the mobile and the IoT revolutions. However, several unintended data breaches are reported every week, and this might compromise the privacy, safety, and security of citizens. For all these reasons, the European Parliament approved in April 2016 the EU General Data Protection Regulation (GDPR). The main goal of such regulation is to protect the privacy of citizens with regard to the processing of their personal data. It enforces a Privacy by Design and by Default approach, where personal data is processed only when needed by the functionalities of the information system. On the other hand, static analysis aims at proving at compile time various properties on information systems. This discipline has been widely applied during the last decades to identify potential software leaks of sensitive data.

In this scenario, this paper discusses what role static analysis could play in a GDPR perspective. In particular, we introduce GDPR and static analysis, and we then propose how existing taint analyses and backward slicing algorithms might be combined to produce reports useful for GDPR compliance. We identify four main actors in the GDPR compliance process (namely, data protection officers, chief information security officers, project managers, and developers), and we propose a specific level of reporting for each of them.

1 Introduction

On April, 27th 2016 the European Parliament adopted the "General Data Protection Regulation" (GDRP). This regulation will become enforceable on May, 28th 2018, and its goal is to lay down *"rules relating to the protection of natural persons with regard to the processing of personal data and rules relating to the free movement of personal data"*[1]. During the last decades, in Europe several regulations were approved and enforced by various countries. The EU GDPR is the first effort to unify these different legislations. The pervasiveness of information systems that might collect sensitive data of citizens was definitely one of the crucial factors that pushed the European institutions to promote such regulation. Nowadays, almost all citizens have mobile devices that might be exploited to track them, and they use various IT services that contain a wide range of sensitive data (about their salary, their health, etc..). In addition, the IoT revolution is coming: *"Gartner, Inc. forecasts that 8.4 billion connected things will be in use worldwide in 2017, up 31 percent from 2016, and will reach 20.4 billion by 2020"*[11]. All these devices might record sensitive data about different environments and people. In addition, many new breaches of sensitive data are reported every week.

Roughly speaking, GDPR allows an information system to access, and manage sensitive data that is needed to perform the functionalities of the system following the Privacy by Design [5] principles. Sensitive data might be *"any information relating to an individual, whether it relates to his or her private, professional or public life. It can be anything from a name, a home address, a photo, an email address, bank details, posts on social networking websites, medical information, or a computers IP address"*. While it is still under discussion how such regulation

```
1  public void onStart() {
2          send("start", "now");
3          TelephonyManager telephonyManager = ...;
4          String imei = telephonyManager.getDeviceId();
5          send("id", imei);
6  }
7
8  public void onLocationChanged(Location location) {
9          send("latitude", location.getLatitude());
10         send("longitude", location.getLongitude());
11 }
12
13 private void send(String key, String value) {
14         new URL("http://<site>?"+key+"="+value).openConnection().connect();
15 }
```

Figure 1: Our running example.

will be checked and enforced, tools that help to check how sensitive data is processed by a software system could become a main asset for GDPR compliance.

During the last decades, static program analysis has been widely applied to various properties. From a GDPR perspective, privacy analyses aimed at detecting possible leakages of sensitive data are particularly promising, since they allow to detect potential leaks at compile time before the system is deployed. Therefore, they might help to prevent unintended data breaches because of software vulnerabilities and leaks.

Contribution
In this scenario, the main contribution of this paper is to discuss how static analysis might be applied for GDPR compliance. In particular, we discuss (i) how such tools can be applied as a privacy enhancing technology (PET) in the Privacy by Design (PbD) approach, (ii) what types of different static analyses for privacy exist, and (iii) what information might be tracked by these tools and how this might be reported to the various actors (namely, data protection officers, chief information security officers, project managers, and developers) in the GDPR compliance.

The paper is structured as follows. The rest of this Section will introduce a minimal running example. Sec. 2 presents privacy enhancing technologies, privacy by design and the GDPR. Sec. 3 discusses static analysis flavors and its use for privacy analysis. Sec. 4 advocates and formalizes its role for GDPR compliance. Sec. 5 concludes.

1.1 Running Example

Fig. 1 reports a minimal running example of Android code that processes confidential data. It leaks the user identifier, in `onStart`, and his exact location, in `onLocationChanged`. Both leaks occur in `send`, that transmits this data to `http://<site>`. If the first leak is unneeded for the functionality of the app, static analysis should infer this at compile time and be hence a PET that warns app's users that confidential data is leaked. Similarly, for PbD, static analysis can check that the implementation of the code actually matches its design. For GDPR, one can leverage such information to check that the processor deals with data as expected.

2 The EU General Data Protection Regulation (GDPR)

This section provides background about privacy regulations and technologies and the GDPR.

2.1 Privacy Enhancing Technologies (PET)

The concept of Privacy Enhancing Technologies (PET) has been around for decades [10]. ENISA [7] defines it as *"a manner of accomplishing a task especially using technical processes, methods, or knowledge, to support privacy or data protection features*, where the latter *require safeguards concerning specific types of data since data processing may severely threaten informational privacy"* [6]. The identification of such types of data is a complicated problem by itself, but orthogonal *wrt.* the scope of this article, that assumes it is a given fact. Historically, the main principles of PETs have been *data minimization and identity protection by anonymization.*

2.2 Privacy by Design (PbD)

While PETs intervene at the end of the chain of data manipulation (*e.g.*, anonymization before communicating or storing the data), *"the Privacy by Design approach is characterized by proactive rather than reactive measures. It anticipates and prevents privacy invasive events before they happen. PbD does not wait for privacy risks to materialize, nor does it offer remedies for resolving privacy infractions once they have occurred – it aims to prevent them from occurring. In short, Privacy by Design comes before-the-fact, not after"* [5]. PbD embeds privacy into the entire engineering process, from its early design phase to the operation of the productive system, that is, it embeds privacy into the design of the system (Principle 3 of [5]). In addition, Principle 1 of PbD lists, among the privacy practices, *"established methods to recognize poor privacy designs, anticipate poor privacy practices and outcomes, and correct any negative impacts, well before they occur in proactive, systematic, and innovative ways".*

2.3 The EU General Data Protection Regulation (GDPR)

The GDPR will come into enforcement on May 28th, 2018. It enforces PbD as a legal obligation for entities that control and/or process privacy data. Its Article 25 (data protection by design and by default) recites: *taking into account the state of the art, the cost of implementation and the nature, scope, context and purposes of processing as well as the risks of varying likelihood and severity for rights and freedoms of natural persons posed by the processing, the controller shall, both at the time of the determination of the means for processing and at the time of the processing itself, implement* ***appropriate technical and organizational measures****, such as pseudonymization, which are designed* ***to implement data-protection principles****, such as data minimization, in an effective manner and to integrate the necessary safeguards into the processing in order to meet the requirements of this Regulation and protect the rights of data subjects.* Article 25 further arguments that *the controller shall implement appropriate technical and organizational measures for ensuring that, by default,* ***only personal data which are necessary for each specific purpose of the processing are processed****. That obligation applies to the amount of personal data collected, the extent of their processing, the period of their storage and their accessibility. In particular, such measures shall ensure that by default personal data are not made accessible without the individual's intervention to an indefinite number of natural persons.* PETs and PbD will become mainstream, to fulfill the legal obligations imposed by GDPR (in particular, Principle 1 of PbD). Institutions, such as ENISA, already provide

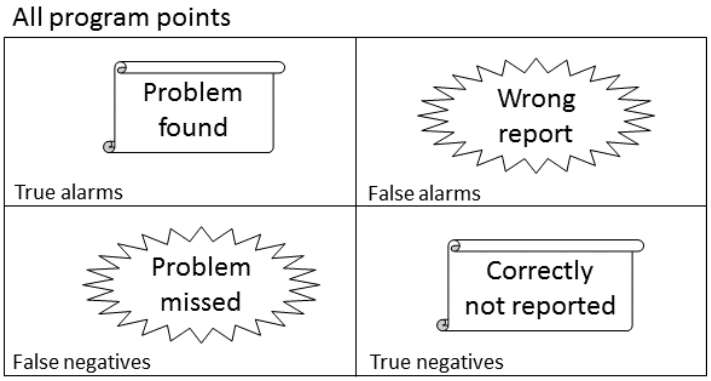

Figure 2: Alarms classification.

community standards on the evaluation of PETs [7]. Given the maturity of static analysis of privacy properties, we believe that it will play a relevant role in this context.

3 Static Analysis for Privacy

Certifications and guidelines often impose, discuss or suggest the use of static analysis [14, 15, 24]. But concrete static analyses can be very differently evaluated along a few axes. Moreover, they usually deal with reliability and security of software and not with its privacy issues.

3.1 Static Program Analysis

Wikipedia[1] defines it as *the analysis of computer software that is performed without actually executing programs*. This includes for instance manual software audit, but this article restricts its scope to *automatic* software static analysis, to focus on a technical discussion.

3.2 Alarms and Soundness

A static analyzer issues *alarms* at program points where an actual bug might occur at runtime, such as a `NullPointerException`, or the code might be inefficient, because for instance of an unused variable, or a security vulnerability might be exploited, such as an SQL injection, or private data might be leaked, by sending for instance the user identifier to a website, as in Fig. 1. Alarms might be (i) true alarms (such as signaling an information leak at line 14 in Fig. 1); (ii) false alarms, that are not actually problems (such as reporting a warning at line 2); (iii) true negatives (no alarms at non-problematic program points, such as not reporting a warning at line 2); or (iv) false negatives (missed true problems, such as not reporting an information leak at line 12). Fig. 2 depicts these concepts: the left side contains the problematic program points; the right side the non-problematic ones; the upper part is what is reported by the analysis, while the lower part is what is not reported. Ideally, a static analyzer should feature true alarms and true negatives only. This, however, is not computable in general [20].

[1] https://en.wikipedia.org/wiki/Static_program_analysis

A warning should embed contextual information about its meaning. For instance, a warning might explain where a `null` value was assigned to a variable that gets dereferenced. A privacy violation warning might report the source program point where sensitive data was disclosed and the data flow from the source to the sink. This lets the developer determine if the warning is a real issue: if the source is not deemed relevant or if the flow is sanitized in the middle, the developer might decide to ignore the warning. Privacy warnings might also report the API call that discloses the information (such as URLConnection.connect()) or might specify the exact kind of sensitive data that is disclosed (such as the longitude and the latitude of the user). For instance, a static analyzer might be able to see that the method calls at lines 5, 9, and 10 in Fig. 1 lead to a leak of confidential information, while that at line 2 does not. In addition, it might report that line 5 discloses the IMEI of the telephone, line 9 the location projected on the latitude, and line 10 the longitude.

A *sound* static analyzer considers *all possible executions* of a program and consequently does not miss any true alarm while still featuring some false alarm. If it does not issue any alarm about a property, then the program always satisfies that property. That is, sound analyzers have no false negatives. In Fig. 1, it would report that there is an information leak at line 14 when `send` is called from lines 5, 9 and 10; it might also report a false information leak alarm when `send` is called from line 2, if it does not track context-sensitive information.

Full soundness is still an open problem for real programming languages such as Java and Android, that feature complex execution behaviors. It is often compromised because of the application lifecycle (components of mobile apps use complicated event-based execution models, triggered by external actors); reflection (applications can use data as code and consequently perform arbitrary dynamic executions); multithreading (arbitrary interleavings are too many for the analysis); native code (machine language translated from any other programming language can be linked, that the analyzer does not understand); dynamic class-loading (code is loaded at runtime, unknown at analysis time). For instance, an analyzer unsound *wrt.* the Android application lifecycle could miss that the operating system calls `onLocationChanged` in Fig. 1; the analyzer could consequently miss the leak of the user location. To the best of our knowledge, current analyzers are all unsound *wrt.* native code (which is platform-dependent), while some support some of the first four features, at different degrees [2, 9, 18, 3].

3.3 Current Privacy Analyses

Privacy leaks detection in software has been widely studied. It reduces to detection of information flows from a source of confidential information (*e.g.*, `getDeviceIdentifier`) to a sink (*e.g.*, operations on Internet connections). From this perspective, it is similar to detecting injections and XSS vulnerabilities, but sources and sinks are not fixed but rather user-provided. Early information flow analyses [19, 21] considered implicit flows as well (indirect information flows through program control structures) but have been proved to be too conservative [16]. Hence other approaches have been proposed, such as quantitative information flow [22] and taint analysis [25]. Several scientific works [2, 26, 12] extend these analyses to mobile (Android) software. Information disclosure through side channels has been studied as well [17] (*e.g.*, running time of an algorithm processing confidential data). This necessarily non-exhaustive list shows that relevant scientific theoretical results exist, whose application is however limited up to now.

These analyses are the most appealing for GDPR, since they automatically identify leaks and scale (in terms of efficiency and precision) up to the size of industrial applications. However, they provide limited feedback: since a source-sensitive flow analysis would be too expensive, they only track and discover that *some* sensitive data could be leaked, without further detail;

moreover, they only report the program point where data is leaked, not the complete flow.

4 Static Analysis for GDPR Compliance

A few years ago, ENISA already underlined [6] that *it is also possible to rely on formal (mathematical) methods to prove that, based on appropriate assumptions on the PETs involved, a given architecture meets the privacy requirements of the system. To comply with accountability of practice, data controllers must be able to demonstrate that their actual data handling complies with their obligations.* However, the type of control forecast was limited to logs auditing. We envision instead to apply static analysis as a PET, to check program behaviors *wrt.* sensitive information at the different stages of the software lifecycle.

One can identify four main actors in this lifecycle, who apply static analysis for GDPR compliance. The *Data Protection Office* (**DPO**) uses a static analyzer to inspect if the whole system respects the privacy constraints identified during design. He needs a very high level view of what sensitive data is leaked (such as the user identifier or location) and to whom. At this level, no technical detail such as precise program points and chain of called methods is needed. The *Chief Information Security Officer* (**CISO**) needs a similar view of the system, but with detail about which software components leak which data, in order to identify the project manager who is responsible of leaks unexpected at design time. The *Project Manager* (**PM**) needs precise detail for each component about the program points that generated the leak (exact sources, sinks and flow of sensitive data inside the program). This lets the *Software Developer* (**DEV**) inspect the code where an alarm is issued. The developer is interested in the exact flow of sensitive data.

Higher levels require information that can be abstracted from lower levels. In particular, the precise flows of sensitive data at **DEV** level, belonging to the same subcomponent, can be abstracted (removing the exact program points accessing and leaking sensitive data) and aggregated into a unique report at **PM** level. As a further approximation, many program points can be abstracted into few sources of sensitive data (since the same information can be accessed at different program points and in different ways), leading to a higher and more readable representation at **CISO** level. Finally, information can be aggregated to represent privacy leaks of the whole system at **DPO** level.

Therefore, we start by considering how the exact flow of sensitive data can be reconstructed from the results of a taint analysis and then present how this flows can be abstracted into the information needed at different levels.

4.1 Reconstructing Sensitive Data Flow from Taint Analysis Results

Let us start by considering how one can obtain the information necessary at **DEV** level, by using static analysis. As discussed in Sec. 3, taint analysis can be instantiated to report the program points that could leak sensitive information. To achieve this, taint analysis typically tracks, for each program statement, the local variables and heap locations that might be tainted, that is, could contain sensitive information. It is too expensive to track precisely the source of sensitive data during the analysis, but one can reconstruct full data flows backwards, starting from the sinks.

Namely, for each statement, one can apply backward slicing [13] that, relying on taint information, tracks how the tainted data flows inside the program. At the end of this step, we obtain many flows of sensitive information since the data could flow from different sources (and through different paths) to the sink.

Formalization
Let St and W be the set of all program statements and warnings, respectively. Each warning refers exactly to one program statement (formally, $getStatement : \mathsf{W} \to \mathsf{St}$). We assume that each program $\mathsf{p} \in \mathsf{P}$ (where P is the set of all programs) is composed by a set of methods (formally, $\mathsf{P} = \wp(\mathsf{Me})$ where Me is the set of all methods), and each method is represented as a control flow graph (CFG) whose nodes are statements ($\mathsf{Me} = CFG(\mathsf{St})$ where $CFG(\mathsf{St})$ represents a control flow graph of statements). On this CFG, taint analysis, given a program, infers a set of warnings, and an entry and exit state for each program statement. Formally, $taint : \mathsf{P} \to (\wp(\mathsf{W}) \times \Phi)$ where Σ is the set of states of the taint analysis reporting the local variables that are tainted, and $\Phi = \mathsf{St} \to \Sigma \times \Sigma$ represents the results (that is, entry and exit states) obtained by the taint analysis on each statement of the program.

We can then reconstruct the flow of sensitive data by computing a backward slice [13] (relying also on the information inferred by the taint analysis), starting from the results of the taint analysis and the warning containing the statement and local variable that are leaked to the sink (formally, $backwardSlice : (\mathsf{W} \times \Phi) \to \mathsf{S}$ where LV is the set of local variables, and S represents the slices over a program).

Running Example: Consider the running example we introduced in Section 1.1. We assume that the location (received by the listener `onLocationChanged` at line 8) and the user IMEI (read by calling `TelephonyManager.getDeviceId` at line 4) are considered as sources of sensitive data, while sending information (through a call to `URL.openConnection().connect()` at line 14) is a sink. On this program, the taint analysis produces one warning at line 14 reporting that some sensitive data might be leaked. We then apply backward slicing to this program point and we obtain three distinct slices: (i) a slice from line 4 to 5 and then 14 (formalized by $s_1 = 4 \to 5 \to 14$) representing the leakage of the identifier, (ii) $s_2 = 9 \to 14$ (leakage of latitude), and (iii) $s_3 = 10 \to 14$ (leakage of longitude). Note that the backward slicing is in position to discard the flow coming from the call to `send` at line 2 (that is, the slice $2 \to 14$) by checking that this method call does not pass tainted (i.e., sensitive) data through the parameters.

4.2 Reporting

We now discuss four different levels of reporting targeting the four main actors we identified (namely, data protection officers, chief information security officers, project managers, and developers). Starting from the information inferred by static analysis through taint analysis and backward slicing (as formalized in Section 4.1), we present these reports as further abstractions starting from the one that contains the most detailed information (that is, the report for developers).

DEV
The developer needs to know full details about flows of sensitive data, to determine if the flow was real and which sensitive data is involved. The slices computed by the backward slicing algorithm contains the information needed at **DEV** level since, for each flow of sensitive data to a sink, it reports complete detail of the flow. Hence, one can formalize the **DEV** report as follows:

$$DEV : \mathsf{P} \to \wp(\mathsf{S})$$
$$DEV(\mathsf{p}) = \bigcup_{\mathsf{w} \in \mathsf{W}_1} backwardSlice(\mathsf{w}, \phi) \text{ where } taint(\mathsf{p}) = (\mathsf{W}_1, \phi)$$

Running Example: The **DEV** report contains the full details of the three slices introduced in Section 4.1. In this way, the developer can inspect the complete flow of sensitive.

PM

The project manager is interested in knowing what subcomponents generated the flows of sensitive data and complete detail of the sources and sinks, at source code level. His goal is to identify the developers that implemented such piece of code. One can abstract **DEV** into **PM** by (i) aggregating all leaks whose source is in the same component; and (ii) abstracting the flows into pairs composed by a source and a sink. Assume that a function $component : \mathsf{St} \to \mathsf{C}$ specifies which component a statement belongs to, and functions $source : \mathsf{S} \to \wp(\mathsf{Source})$ and $sink : \mathsf{S} \to \mathsf{Sink}$ return the possible sources and the sink of a given slice (where $\mathsf{Source} \subseteq \mathsf{St}$ and $\mathsf{Sink} \subseteq \mathsf{St}$), respectively. One can project **DEV** (that is, a set of slices) into **PM** as follows:

$$PM : \wp(\mathsf{S}) \to (\mathsf{C} \to \wp(\mathsf{Source} \times \mathsf{Sink}))$$
$$PM(\mathsf{S}_1) = [\mathsf{c} \mapsto \bigcup_{\mathsf{s} \in \mathsf{S}_1, \mathsf{source} \in source(\mathsf{s})} \{(\mathsf{source}, \mathsf{sink}) : \mathsf{sink} = sink(\mathsf{s}) \wedge component(\mathsf{source})\}]$$

Running Example: Let us assume that the running example in Figure 1 is made by two components: st concerns the start of the application (method `onStart`), while loc deals with location updates (method `onLocationChanged`). Therefore, the **PM** report will related (i) component st to the singleton $\{(4, 14)\}$ (representing the leakage of user identifier), and (ii) component loc to the set $\{(9, 14), (10, 14)\}$ (representing the leakage of latitude and longitude, respectively).

CISO

High level managers such as the Chief Information Security Officer would be interested in knowing which subcomponent of the system accessed and leaked which sensitive data. Hence, as a further abstraction, one can aggregate sources and sinks by projecting them into their types. Assume that functions $sourceType : \mathsf{Source} \to \mathsf{SourceTypes}$ and $sinkType : \mathsf{Sink} \to \mathsf{SinkTypes}$ translate sources and sinks to their type, respectively, are provided. The projection of **PM** into **CISO** is defined as follow:

$$CISO : (\mathsf{C} \to \wp(\mathsf{Source} \times \mathsf{Sink})) \to \mathsf{C} \to \wp(\mathsf{SourceTypes} \times \mathsf{SinkTypes})$$
$$CISO(\mathsf{p}) = [\mathsf{c} \mapsto \bigcup_{(\mathsf{source}, \mathsf{sink}) \in \mathsf{p}(\mathsf{c})} (source(\mathsf{source}), sink(\mathsf{sink})) : \mathsf{c} \in dom(\mathsf{p})]$$

Running Example: Let us assume that longitude and latitude are both abstracted to the same source type called Location (since they both represent very precise geographical information), while the user identifier is represented by the source type IMEI. Instead, the sinks that transmit data over Internet are represented by Internet. Therefore, the **CISO** report applied to our running example of Figure 1 will relate component st to $\{(\mathsf{IMEI}, \mathsf{Internet})\}$, and component loc to $\{(\mathsf{Location}, \mathsf{Internet})\}$. This report abstracts away the implementation details about the method calls accessing and leaking the sensitive data, as well as the flow of such data inside the program. However, it can be understood by actors in the software lifecycle that are not involved in the technical implementation details.

DPO

The data protection officer needs a high-level and legal view of how the whole software system deals with sensitive data. Hence, one can project away the information about components from

CISO and produce the **DPO** information. Formally,

$$DPO : (\mathsf{C} \to \wp(\mathsf{SourceTypes} \times \mathsf{SinkTypes})) \to \wp(\mathsf{SourceTypes} \times \mathsf{SinkTypes})$$
$$DPO(\mathsf{t}) = \bigcup_{(\mathsf{sourceType},\mathsf{sinkType}) \in dom(\mathsf{t})} (\mathsf{sourceType}, \mathsf{sinkType})$$

Running Example: This last level of abstraction merges together the two components tracked by the **CISO** into the same set, producing $\{(\mathsf{IMEI}, \mathsf{Internet}), (\mathsf{Location}, \mathsf{Internet})\}$. Through this information one might check if the information inferred by the static analysis matches what was expected by the system design.

4.3 Summary

Let us summarize the main components needed to develop the four GDPR reports **DEV**, **PM**, **CISO** and **DPO**). First of all, one needs a set of sources Source and sinks Sink. For injection analysis these sets are usually fixed (for instance, reading servlet inputs are sources and SQL methods are sinks for SQL-injection), for GDPR we expect that these sets will be specific to the software system under analysis, since different software rely on APIs providing sensitive data in various ways. In addition, one needs the abstraction of software sources and sinks (such as `getDeviceId` and `connect` in Fig.ẽflst:runningexample) to types of sensitive data (device identifier) and data leaks (leakage to `<site>`). These components are formalized by functions *sourceType* and *sinkType*. In any case, the GDPR requires to identify how sensitive data is accessed (that is, the set of sources together with their abstraction) and leaked (sinks). Finally, one needs to know how the software system is split into components (function *component*), a common information available on all structured software systems.

5 Conclusion

This paper discussed how static analysis can be applied for GDPR compliance. In particular, it introduced some aspects of the EU General Data Protection Regulation, and existing applications of static analysis to privacy properties. It then identified four main actors (developers, project managers, chief information security officers, and data protection officers) that could be involved at various stages of GDPR compliance, and formalized how combining existing static analyses and information can lead to useful reports for GDPR compliance. The final vision is that static analyzers can be applied to help GPDR compliance re-using existing static analyses (and in particular, taint analysis and backward slicing) and information (about source and sinks, and software components).

These ideas are currently being implemented into the Julia static analyzer [23], which already contains an industrial implementation of taint analysis [8, 4].

References

[1] EU Regulation 2016/679 of the European Parliament and of the Council, April 2016. http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of PLDI '14*. ACM, 2014.

[3] 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 ICSE '11*. ACM, 2011.

[4] E. Burato, P. Ferrara, and F. Spoto. Security Analysis of the OWASP Benchmark with Julia. In *Proceedings of ITASEC '17*, 2017.

[5] A. Cavoukian. Privacy by Design - The 7 Foundational Principles, January 2011.

[6] ENISA. Privacy and Data Protection by Design, December 2014. https://www.enisa.europa.eu/publications/privacy-and-data-protection-by-design.

[7] ENISA. Readiness Analysis for the Adoption and Evolution of Privacy Enhancing Technologies, March 2016. https://www.enisa.europa.eu/publications/pets.

[8] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Proceedings of LPAR '15*, Lecture Notes in Computer Science. Springer, 2015.

[9] P. Ferrara. *Static analysis via abstract interpretation of multithreaded programs*. PhD thesis, Ecole Polytechnique of Paris (France) and University "Ca' Foscari" of Venice (Italy), May 2009.

[10] S. Fischer-Hübner. *IT-security and Privacy: Design and Use of Privacy-enhancing Security Mechanisms*. Springer-Verlag, Berlin, Heidelberg, 2001.

[11] Gartner. Gartner says 8.4 billion connected "Things" will be in use in 2017, up 31 percent from 2016, February 2017. http://www.gartner.com/newsroom/id/3598917.

[12] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of TRUST '12*. Springer-Verlag, 2012.

[13] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[14] International Organization for Standardization. Space systems – Dynamic and static analysis – Exchange of mathematical models, 2005. ISO 14954:2005 standard.

[15] International Organization for Standardization. Road vehicles Functional safety, 2011. ISO 26262 standard.

[16] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit Flows: Can'T Live with 'Em, Can'T Live Without 'Em. In *Proceedings of ICISS '08*. Springer-Verlag, 2008.

[17] B. Köpf and D. Basin. An Information-theoretic Model for Adaptive Side-channel Attacks. In *Proceedings of CCS '07*. ACM, 2007.

[18] A. Miné. Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs. In *Proceedings of ESOP '11*. Springer-Verlag, 2011.

[19] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of POPL '99*. ACM, 1999.

[20] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[21] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.

[22] G. Smith. On the Foundations of Quantitative Information Flow. In *Proceedings of FOSSACS '09*. Springer-Verlag, 2009.

[23] F. Spoto. The Julia Static Analyzer for Java. In *Proceedings of SAS '16*, Lecture Notes in Computer Science. Springer, 2016.

[24] H. F. Tipton. *Official (ISC)2 Guide to the CISSP CBK*. Auerbach Publications, 2nd edition, 2009.

[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of PLDI '09*. ACM, 2009.

[26] Z. Yang and M. Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Proceedings of WCSE '12*. IEEE Computer Society, 2012.