Tailoring Taint Analysis to GDPR

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

[1] JuliaSoft SRL, Verona, Italy
{pietro.ferrara,luca.olivieri}@juliasoft.com
[2] Università di Verona, Italy
fausto.spoto@univr.it

Abstract. Static analysis is the analysis of software at compile time without executing it. Its goal is to explore all execution paths without needing specific inputs to drive the execution. Thanks to its wide coverage, this approach, and in particular taint analysis, has been widely applied to detect security vulnerabilities like SQL injections and XSS.

The European General Data Protection Regulation requires all controllers of sensitive data to enforce an approach based on privacy by design and by default. In such context, verification and testing techniques can be applied to check if the system implementation follows the constraints identified at design time. Therefore, static program analysis might be applied to track how sensitive data is automatically managed by a software, and if such software could leak some of this data.

In this paper, we formalize and discuss how taint analysis can be extended and augmented in order to detect potential unintended leakages of sensitive data. Starting from the specification of how sensitive data is retrieved and it could be leaked, and what types of leakages are allowed by the privacy policy established by the controller of sensitive data, we apply standard taint analysis to detect potential leakages, we reconstruct the flow to check if the flow is allowed or not, and we report full details about all the flows not allowed by the privacy policy. This approach has been implemented on the Julia static analysis, and we report some promising experimental results on the OWASP WebGoat benchmark.

Keywords: Static Analysis, Taint Analysis, GDPR Compliance

1   Introduction

The European General Data Protection Regulation[1] (GDPR) was adopted by the European Parliament on April 27, 2016, and will be enforced from May 25, 2018. Its main 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" (Article 1). This regulation imposes that the controller of sensible data adopts an approach based on the concepts of privacy

---

[1] https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32016R0679

by design and by default. The European Commission provided various guidelines to drive the process of GDPR compliance[2]:

> Companies/organisations are encouraged to implement technical and organisational measures, at the earliest stages of the design of the processing operations, in such a way that safeguards privacy and data protection principles right from the start ("data protection by design"). By default, companies/organisations should ensure that personal data is processed with the highest privacy protection (for example only the data necessary should be processed, short storage period, limited accessibility) so that by default personal data isn't made accessible to an indefinite number of persons ("data protection by default").

The scope of GDPR is extremely broad, and ranges from very high level organizational to deep technical procedures. This paper focuses on a relatively small and precise aspect of the regulation, that is, the automatic treatment of personal data in software. In this context, a controller of personal data should make the best effort to ensure that software processes data in the *right* way *w.r.t.* the GDPR policy (i.e., a rather standard privacy policy establishing what kinds of sensitive data might disclosed to what kinds of leakage points) of the organization identified during the design of the system. Here the main question is: how could we (hopefully automatically) check if software manages personal data correctly w.r.t. the constraints identified at design time? What tools and approaches could help?

Static analysis has been widely applied to prove various software properties, automatically [7–9]. Its main idea is to create a mathematical model of the executions of a program and to *statically* prove (*i.e.*, without executing the code) some properties on such model. A sound static analysis creates a model that covers all possible executions. Therefore, it can prove that all possible executions of the program under the analysis satisfy the given property.

Absence of runtime errors, correct synchronization between parallel threads, absence of security vulnerabilities such as SQL injection and cross-site scripting (XSS) are just some notable examples of properties that can be proven with static analysis. Here, the scientific literature is extremely broad. In particular, information flow analysis targets privacy properties since several decades, and taint analysis has been already applied for this goal. However, such analyses normally only detect if there exists a data flow from a source of personal data to a leaking point. That is, they do not tell which type of personal data flows and along which path.

This article describes an extension of standard taint analysis that proves if software complies to a given GDPR policy. Section 2 introduces background (static analysis, information flow, taint and privacy analysis); Section 3 introduces the configuration of the analysis, which reflects what is already required by

---

[2] https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations/obligations/what-does-data-protection-design-and-default-mean_en

the same GDPR compliance process; Section 4 presents how the configuration is used to instrument the taint analysis engine, how information is extracted from the results of that analysis, and how a GDPR report is built from this information. We have implemented a prototype in the Julia static analyzer [25] and applied it to the analysis of WebGoat, the motivating example introduced below and used throughout the article to show how our approach works on real-world software.

1.1 WebGoat: Motivating Example

WebGoat version 6.0.1 (the last released legacy version[3]) will be the motivating example throughout this article. It "is a deliberately insecure web application maintained by OWASP[4] designed to teach web application security lessons"[5]. Since WebGoat is a web application designed to expose various security flaws, it is a particularly good target to test and show the results of various security and privacy analysis. In addition, it is a relatively small application (about 20KLOCs of Java code), hence the results of the analysis can be manually checked.

WebGoat contains two critical points interesting from a GPDR perspective:

- class `org.owasp.webgoat.session.Employee` represents an employee and holds sensitive data, such as name, surname, SSN, credit card number, etc..;
- the lesson class `WsSAXInjection` asks the user to add or change her password, and therefore deals with this sensitive data.

Moreover, WebGoat contains many standard leakage points, such as standard DB interactions or logging calls. However, we focus on two kinds of leakage:

- into a database, through some standard APIs such as `java.sql.PreparedStatement` and `Connection`, and
- into the Internet, through some standard APIs such as `java.net.URL` or the Apache Element Construction Set library. This is a library that generates elements for a variety of markup languages. WebGoat uses it to build HTML pages, then sent and rendered.

2 Background

This section introduces background about information flow static analysis, its industrial application to the detection of various security vulnerabilities (such as SQL injection and XSS) and its current extensions to privacy properties.

---

[3] The source code can be found at https://github.com/WebGoat/WebGoat-Legacy/releases/tag/v6.0.1

[4] The Open Web Application Security Project, available at the web address https://www.owasp.org/index.php/Main_Page

[5] https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

2.1   Static Analysis

The goal of static analysis is to prove, statically (*i.e.*, without executing the code), various program properties [28]. While dynamic analysis, including testing, explores only a portion of the program, that reachable from some given inputs, static analysis can explore all possible executions. During the last decades, many different approaches have been introduced to develop static analyzers. Model checking [6], type systems [23], data and control flow analyses [18, 21], and abstract interpretation [7, 8] are the most notable and successful examples. In particular, sound static analysis guarantees that, if a property is proven on a program, then *all* possible executions of the program respect the given property. For instance, if a sound static analyzer proves that a program does not contain an SQL injection, then there exists no execution leading to an SQL injection.

Static program analysis has been widely applied to detect bugs in industrial software. Historically, its first application was to detect potential runtime errors in safety critical embedded software for avionics. In this context, various industrial static analyzers [4, 1, 2, 19] have been formalized, implemented and applied to real-world code. During the last decade, various research efforts [25, 27] have targeted the automatic detection of various kinds of injections and XSS vulnerabilities, achieving a relevant impact on industrial software.

2.2   Information Flow and Taint Analysis

Information flow analyses "can prove that a program cannot cause supposedly nonconfidential results to depend on confidential input data" [10]. They check if private input (such as sensitive data or user-controlled input) flows explicitly (that is, through assignments) or implicitly (through conditions) to a public channel (such as the Internet or an SQL query execution routine). A lattice structure defines different (hierarchical) levels of private and public channels, allowing to check rather complicated policies.

This concept has been around for more than four decades and produced an impressive amount of scientific and industrial results. As explained by Sabelfeld and Myers [24]:

> The standard way to protect confidential data is (discretionary) access control: some privilege is required in order to access files or objects containing the confidential data. Access control checks place restrictions on the release of information but not its propagation. (...) To ensure that information is used only in accordance with the relevant confidentiality policies, it is necessary to analyze how information flows within the using program; because of the complexity of modern computing systems, a manual analysis is infeasible. (...) This analysis must show that information controlled by a confidentiality policy cannot flow to a location where that policy is violated.

Many analyses (mostly focused on specific type systems) tracking both implicit and explicit flows have been formalized and developed, with JFlow [20]

being probably the most notable tool. However, they achieved relatively little industrial impact mostly because of false alarms generated by implicit flows and limited scalability. For this reason, taint analysis [27, 12], introduced more than a decade ago, relaxes standard information flow analysis by considering only explicit flows, hence reducing the number of false alarms; and using only one level of *taintedness* (that is, data can only be public or private) to improve performance. Hence the analysis checks if there is an explicit information flow from an untrusted *source* to a trusted *sink*, without intermediate *sanitization*. This generic schema has been instantiated to several critical security vulnerabilities in the OWASP Top 10 list [22], such as

- SQL injection, where sources are methods returning user's inputs, sinks are methods executing SQL queries, and sanitizers are methods escaping the input;
- cross-site scripting, where sinks are instead methods executing the given data; and
- redirection attacks, where sinks are instead parameters of methods opening an Internet connection.

Taint analysis achieved impressive industrial results, detecting many vulnerabilities in real-world software (and in particular web servers) and achieving amazing results in terms of recall and precision [5], in comparison to other (usually pattern-based) approaches. In addition, several approaches have recently applied static taint analysis to Android applications [3], a context where privacy leaks are particularly relevant.
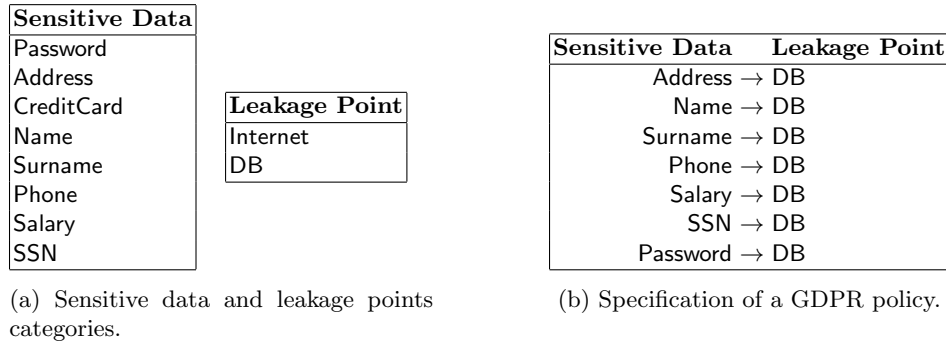
2.3   Privacy Analysis

Recent research extends static and dynamic taint analysis to detect privacy leaks in mobile applications [11, 15]. It tries to overcome two main limitations of taint analysis. Namely, it tries to devise (i) a source sensitive analysis that allows different types of taintedness and not only a unique public/private layer; and (ii) a finer-grained tracking of sensitive data; for instance, the first eight digits of an IMEI number identify the manufacturer of the device and do not contain any information about the serial number of the device. Hence, they can be freely divulged.

3   Configuration of the GDPR Analysis

This section introduces the configuration that must be provided in order to specify a static analysis for GDPR. In particular, this configuration must specify (i) what types of sensitive data and leakage points exist; (ii) how sensitive data can be accessed and leaked; and (iii) a GDPR policy that specifies the data flows that are allowed or forbidden.

3.1   Categories of Sensitive Data and Leakage Points

Not all types of sensitive data and leakage points are equal. For instance, name and surname of a person are probably sensitive data, but social security number and credit card number are definitely more critical data from a privacy perspective. Similarly, leaking sensitive data into a log could be problematic, but it is rather more dangerous to leak the same data into an insecure Internet connection. Hence, the configuration of a GDPR analysis must include a categorization of sensitive data and leakage points. Formally, it must define sets SD (Sensitive Data) and LP (Leakage Point) of the interesting categories of sensitive data and leakage points, respectively.

Sensitive Data
Password
Address
CreditCard
Name
Surname
Phone
Salary
SSN

Leakage Point
Internet
DB

(a) Sensitive data and leakage points categories.

Sensitive Data	Leakage Point
Address → DB	
Name → DB	
Surname → DB	
Phone → DB	
Salary → DB	
SSN → DB	
Password → DB	

(b) Specification of a GDPR policy.

Type	Member	Sensitive Data category
Field	WsSAXInjection.password	Password
Field	Employee.ccn	CreditCard
Field	Employee.firstName	Name
Field	Employee.lastName	Surname

(c) APIs accessing sensitive data (sources).

Type	Member	Parameter	Sensitive Data category
MethodParameter	URL.<init>	arg 0	Internet
MethodParameter	PreparedStatement.setString	arg 1	DB
MethodParameter	ecs.html.B.<init>	arg 0	Internet

(d) APIs leaking data (sinks).

Fig. 1: Configuration of a GDPR analysis for WebGoat.

Motivating Example Figure 1a reports the categories of sensitive data and leakage points that we consider interesting for a GDPR analysis of WebGoat. They have already been informally discussed in Section 1.1. In particular, column A of Figure 1a reports the categories of data considered as sensitive, while its

column B specifies that the only interesting leakage points are in categories Internet or DB.

3.2   Specification of Sensitive Data and Leakage Points

Once the interesting categories have been fixed, one needs to specify how sensitive data is read and leaked at the statements of the program. If on the one hand such information needs to be manually specified, on the other hand the GDPR compliance process requires to know how sensitive data could be accessed and leaked by the software.

In this Section, we will denote by St the set of statements.

Sensitive Data   The question of how a program can read sensitive data is equivalent to asking how software can read data programmatically. This can happen through method calls returning a value, or by reading fields, both in the code of the application (for instance through method calls that access a database) and in the code of the libraries. Formally, the sensitive data specification SDSpec is a partial function that relates statements to a sensitive data category: $SDSpec : St \rightarrow SD$.

Leakage Points   The specification of leakage points reduces to how data might be passed to components outside the bounds of the main application, programmatically. In this case, this might happen by writing a field, or passing a value to a parameter of a method call. However, this applies only to components in the libraries, since the application itself can leak data only by calling APIs of external libraries. Similarly to sensitive data, the leakage points specification LPSpec is formalized by a partial function $LPSpec : St \rightarrow LP$.

Motivating Example   Figures 1c and 1d report (a part of) the specification of sensitive data and leakage points for WebGoat, respectively. In particular, many fields of class Employee are tagged with the appropriate category of sensitive data (for instance, Employee.ccn returns sensitive data in category CreditCard) and field WsSAXInjection.password is tagged as Password. The leakage points java.sql.PreparedStatement.setString and Statement.executeQuery are tagged as DB (since data passed to those methods will be stored into a database); several other APIs that disclose data into the net are tagged as Internet, such as the constructor of java.net.URL, methods for handling cookies or Apache ECS elements (for instance, the class B that represents a bold text in an HTML page). The full specification includes 12 kinds of statements as sensitive data and 58 as leakage point (46 are in the ECS library).

3.3   GDPR Policy

The last part of the configuration of a GDPR analysis is the specification of a privacy policy. As discussed in the introduction, the GDPR obliges the controller

of sensitive data to identify, since the design phase, what type of data it manages, and how. Hence, the GDPR policy specifies what categories of sensitive data are allowed to be disclosed to what categories of leakage points. This is represented as a set of pairs relating sensitive data categories to leakage points categories. Formally, $\mathsf{GDPRPolicy} = \wp(\mathsf{SD} \times \mathsf{LP})$. For instance, the pair $(\mathsf{Name}, \mathsf{DB})$ specifies that the GDPR policy allows names to be stored into a database.

Motivating Example Figure 1b reports a GDPR policy for the analysis of WebGoat, that we consider as a sensible formalization of what is allowed in such a program. In particular, the policy specifies that address, name or surname of an employee can be stored into a database, as well as passwords. However, it is not allowed to store credit card numbers into a database and no sensitive data should ever be leaked into the Internet.

4   Static Analysis for GDPR

This section describes how the configuration in the previous section is used to tune a taint analysis and extract information from it, useful for GDPR purposes.

4.1   Sources and Sinks

Taint analysis requires to specify a set of sources and sinks ($\mathsf{Sources}$ and Sinks, respectively, see Section 2.2). They are statements in St that access sensitive data or leak information, respectively. These sets can be derived from the configuration of a GDPR analysis, that specifies SDSpec and LPSpec as shown in Section 3.2. Namely, let $\mathsf{sdspec} \in \mathsf{SDSpec}$ and $\mathsf{lpspec} \in \mathsf{LPSpec}$ be the specification of sensitive data and leakage points, respectively. Taint anaysis will be performed with $\mathsf{Sources} = \mathsf{dom}(\mathsf{sdspec})$ and $\mathsf{Sinks} = \mathsf{dom}(\mathsf{lpspec})$ (where dom is the domain of a function). There is no specification of sanitizers (as common in taint analysis) since, typically, different types of sensitive data require different sanitizers. Hence, the user must evaluate the report described in Section 4.4, to remove false alarms.

4.2   Taint Analysis

After a taint analysis is performed, one obtains (i) all calls to leakage points that might pass a tainted parameter: these are the potential leaks of sensitive data; and (ii) for each program point, the variables and (abstract) heap locations[6] that might be tainted.

The result of a taint analysis is a function $\mathsf{taint} : \mathsf{St} \to \wp(\mathsf{LocalVar} \cup \mathsf{HeapLoc})$ that, for each statement, returns the set of heap locations ($\mathsf{HeapLoc}$) and local

---

[6] How to abstract heap locations is an orthogonal problem that has been deeply investigated by the static analysis research community. We refer the interested reader to [13, 17] for more details.

variables (LocalVar) that might be tainted there (that is, might contain sensitive data) during an execution of the program. That result can be combined with the specification LPSpec of the leakage points to infer where leaks might occur. This is expressed by a function leaks(taint, lpspec) ∈ ℘(St).

4.3 Flow Reconstruction

The taint analysis described in the previous section merges all sources of sensitive data, for scalability. Hence, it cannot identify the source of sensitive data that flows into a leakage point. As observed in Section 2.2, existing approaches that track more than one Boolean taintedness flag do not scale to industrial software (that is, up to 100KLOCs or even 1MLOCs). Therefore, they cannot be considered as industrially viable solutions. Moreover, in any case they do not provide the flow (sequence of statements) that tainted data follows from a source to a sink.

   To overcome such limitations, for each statement detected as potential leak, our analysis performs a backward flow reconstruction that, according to the semantics of program statements, looks for the origin of tainted data. The result of such reconstruction is one or more (because of conditional statements) flow graphs, that is, potentially interprocedural execution paths. The set of such flow graphs is denoted as FlowGraph.

   For most statements, the backward reconstruction is straightforward and just amounts to following assignments backwards. The only operations that require careful processing are:

 – heap access. When the backwards flow reconstruction reaches an access to a heap location that returns sensitive data, it must continue with all potential writers of that (abstract) heap location, backwards. This is achieved by using the same heap abstraction described in Section 4.2;
 – method call. When the backwards flow reconstruction reaches a method call that returns sensitive data, it must continue with all possible methods that might be called there and might return sensitive data. For that, it relies on the static call graph of the program, that approximates the callers/callees relation in a program[7].

It is possible that this flow reconstruction fails, because of a very large number of alternatives that must be followed backwards. This is particularly true when heap accesses with many writers are followed. As a result, there might be leaks for which no flow graph gets reconstructed.

   Formally, we represent the backward flow rebuilder by a partial function flowRebuilder : (TaintRes × St) → ℘(FlowGraph). We assume that functions source : FlowGraph → St and sink : FlowGraph → St are defined on flow graphs, to return the source and the sink of the flow, respectively.

---

[7] The construction of the static call graph is an orthogonal problem that has been widely investigate by the static analysis community. We refer the interested reader to [26, 16] for more detail.

4.4   GDPR Report

Algorithm 1 GDPR report construction

1: **procedure** GDPRREPORT(sdSpec, lpSpec, GDPRpolicy, program)
2:      res ← taint(program, dom(sdSpec), dom(lpSpec))
3:      flows ← ∅
4:      unknown ← ∅
5:      **for** l ∈ leaks(res, lpSpec) **do**
6:          **if** (res, l) ∈ dom(flowRebuilder) **then**
7:              flows ← flows ∪ flowRebuilder(res, l)
8:          **else**
9:              unknown ← unknown ∪{l}
10:     unexpectedFlows ← ∅
11:     **for** f ∈ flows **do**
12:         **if** (source(f), sink(f)) ∉ GDPRPolicy **then**
13:             unexpectedFlows ← unexpectedFlows ∪ {f}
14:     **return** (unexpectedFlows, unknown)

After the flow reconstruction, it is possible to generate a report for the user of the GDPR analysis. It tells if the program satisfies the GDPR policy (Section 3.3) and shows the unexpected flows, in case of non-compliance.

Algorithm 1 builds the report. It requires the specification of the sources of sensitive data sdSpec ∈ SDSpec and of leakage points lpSpec ∈ LPSpec, of a GDPRpolicy ∈ GDPRPolicy and of a program ∈ Program. It runs the taint analysis with such sources and sinks (line 2). For each leak (line 5), it reconstructs and collects in flows the flows of sensitive data, by using the backward rebuilder (line 7); moreover, a set unknown collects the leakage points for which the flow reconstruction fails (line 9). Then (line 11) the algorithm checks if each flow is allowed by the GDPR policy (line 12); if not, the flow is collected into a set unexpectedFlows. At the end, the algorithm returns unexpectedFlows and unknown.
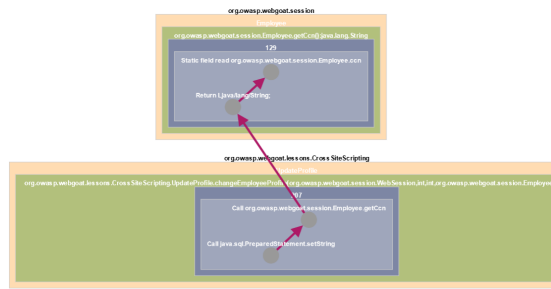
The returned information will tell the user about (i) the potential flows of sensitive data that are not allowed by the desired GDPR policy and therefore need to be corrected; and (ii) the potential leakage points for which no flow could be reconstructed and that consequently need manual inspection, to determine if they are real issues or false alarms.

4.5   The Result of the Analysis of the Motivating Example

A prototype of the analysis described in this article has been implemented in the Julia analyzer [25]. Julia already contains an industrial implementation of taint analysis [12], widely applied to the detection of security vulnerabilities such as SQL injection and XSS [5]. It also contains a heap abstraction and the construction of a static call graph (both components are used by the taint
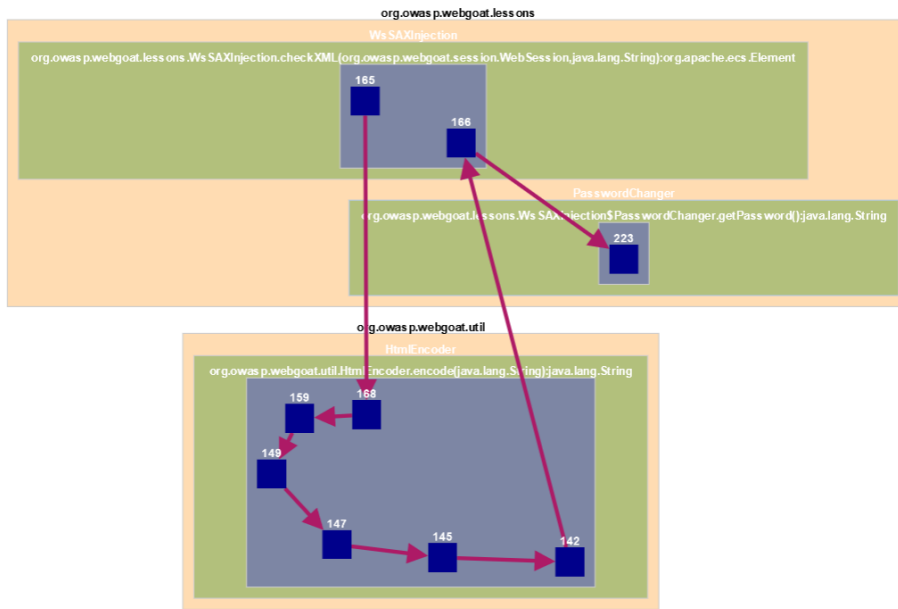
GDPR Report

(a) The GDPR report.

(b) A flow of a credit card number into the DB.

(c) A complex flow from a password into an HTML element.

Fig. 2: GDPR report of WebGoat.

analysis). We applied it to WebGoat with the specifications of sensitive data, leakage points and GDPR policy from Section 3.

Our GDPR analysis spots two flows of sensitive data that are not allowed by the GDPR policy (see Figure 2a). The first flow is from the credit card number of an employee to the database; it occurs many times in classes that update the employee's profile. An example is in Figure 2b: the credit card number is retrieved by calling Employee.getCcn() (that returns the value of the tainted field Employee.ccn, see Figure 1c); it is then passed to method setString of a java . sql . PreparedStatement (method setString is tagged as a sink in the leakage point specification in Figure 1d). In particular, line 207 of CrossSiteScripting .Up− dateProfile contains the code ps. setString (10, employee.getCcn()). The other flow is different and more complex. It involves the disclosure of a password into the Internet, in particular, into an HTML component. Figure 2c reports this flow. The access of sensitive data and its leakage occur at line 166 and 165 of class WsSAXInjection, respectively:

```
165:  return  new  B(HtmlEncoder.encode("You have changed the password for userid "
166:     + changer.getId() + " to ' " + changer.getPassword() + " ' "));
```

As pointed out by the flow graph, the password is passed to HtmlEncoder.encode, that returns the sensitive data. Below is a code snippet with only the statements identified by the flow graph in Figure 2c:

```
140: public static  String encode(String s1)
141: {
142:      StringBuffer  buf = new StringBuffer();
                ...
145:     for  (i = 0; i < s1.length (); ++i)
147:            char ch = s1.charAt( i );
148:
149:            String  entity = i2e.get(new Integer((int) ch ));
                        ...
159:                              buf . append(ch);
            ...
168:     return  buf. toString ();
169: }
```

The flow graph explains that sensitive data is passed to the beginning of this method; it is then read at line 145, later read and assigned to local variable ch at line 147; it flows into variable entity at line 149; it is appended to buf at line 159; and it is finally returned to the callee at line 168. This example shows that the flow graph provides full detail about the propagation of sensitive data. This is invaluable to understand if and how the flow might be a problematic security breach, violating the GDPR policy.

5   Conclusion

This article describes a novel solution to take advantage of static analysis inside the process of GDPR compliance. GDPR is a broad regulation that involves

many different aspects of data security. We argued that static analysis plays a relevant role in building tools that identify how sensitive data is processed in ways that do not comply to the GDPR policy identified during the design of the software system. The solution leverages many well-known and studied techniques, notably, taint analysis. It augments them in order to (i) allow the user to specify the policy, (ii) reconstruct how sensitive data flows in the program, and (iii) check which flows do not respect the GDPR policy. We formalized the approach in detail and applied it to a standard benchmark, WebGoat, often used to show the effectiveness of static analyses for security. A prototype has been implemented in the Julia static analyzer.

As future work, we are currently working at front-ends to present the results of the analysis: plugins for various IDEs (such as Eclipse and IntelliJ IDEA) and dashboards for the results. We have already studied various levels of reporting, targeting distinct actors of the GDPR compliance process [14]. Each actor will deserve his front-end view of the results.

References

1. Absint, https://www.absint.com/
2. Grammatech, https://www.grammatech.com/
3. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of PLDI '14. ACM (2014)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of PLDI '03. ACM (2003)
5. Burato, E., Ferrara, P., Spoto, F.: Security Analysis of the OWASP Benchmark with Julia. In: Proceedings of ITASEC '17 (2017)
6. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL '77. ACM Press (1977)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of POPL '79. ACM Press (1979)
9. Cousot, P., Cousot, R.: Abstract interpretation: past, present and future. In: Proceedings of CSL-LICS '14. ACM (2014)
10. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7) (Jul 1977)
11. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. 32(2) (Jun 2014)
12. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean Formulas for the Static Identification of Injection Attacks in Java. In: Proceedings of LPAR '15. Lecture Notes in Computer Science, Springer (2015)

13. Ferrara, P.: Generic combination of heap and value analyses in abstract interpretation. In: Proceedings of VMCAI '14 (2014)
14. Ferrara, P., Spoto, F.: Static analysis for GDPR compliance. In: Proceedings of ITASEC '18 (2018)
15. Ferrara, P., Tripp, O., Pistoia, M.: Morphdroid: Fine-grained privacy verification. In: Proceedings of ACSAC 2015. ACM (2015)
16. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of OOPSLA '97. ACM (1997)
17. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of PASTE '01. ACM (2001)
18. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '73, ACM, New York, NY, USA (1973)
19. Mathworks: Polyspace, https://www.mathworks.com/products/polyspace.html
20. Myers, A.C.: JFlow: Practical Mostly-static Information Flow Control. In: Proceedings of POPL '99. ACM (1999)
21. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
22. OWASP: Top   10   Project   2017   (March   2018), https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
23. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
24. Sabelfeld, A., Myers, A.C.: Language-based Information-flow Security. IEEE J.Sel. A. Commun. 21(1), 5–19 (Sep 2006)
25. Spoto, F.: The Julia Static Analyzer for Java. In: Proceedings of SAS '16. Lecture Notes in Computer Science, Springer (2016)
26. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proceedings of OOPSLA '00. ACM, New York, NY, USA (2000)
27. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: Effective Taint Analysis of Web Applications. In: Proceedings of PLDI '09. ACM (2009)
28. Wikipedia: Static program analysis, https://en.wikipedia.org/wiki/Static_program_analysis