Automatic Annotation of Confidential Data in Java Code

Iulia Bastys[1], Pauline Bolignano[2], Franco Raimondi[3], and Daniel Schoepe[2]

[1] Chalmers University of Technology
bastys@chalmers.se
[2] Amazon Prime Video
{pln,schoeped}@amazon.com
[3] Amazon Prime Video and Middlesex University
frai@amazon.com,f.raimondi@mdx.ac.uk

Abstract. The problem of *confidential information leak* can be addressed by using automatic tools that take a set of *annotated* inputs (the *source*) and track their flow to public *sinks*. Unfortunately, manually annotating the code with labels specifying the secret sources is one of the main obstacles in the adoption of such trackers.
In this work, we present an approach for the automatic generation of labels for confidential data in Java programs. Our solution is based on a graph-based representation of Java methods: starting from a minimal set of known API calls, it propagates the labels both intra- and inter-procedurally until a fix-point is reached.
In our evaluation, we encode our synthesis and propagation algorithm in Datalog and assess the accuracy of our technique on seven previously annotated internal code bases, where we can reconstruct 75% of the pre-existing manual annotations. In addition to this single data point, we also perform an assessment using samples from the SecuriBench-micro benchmark, and we provide additional sample programs that demonstrate the capabilities and the limitations of our approach.

1 Introduction

A number of information flow trackers for automatically detecting leaks of confidential data have been developed for roughly every programming language: Joana [14] or the Checker framework [1] for Java, JSFlow [15] for JavaScript, TaintDroid [13] for Android apps are just a few examples of such tools. Whether they operate dynamically, statically, or in a mixed fashion, the trackers usually require the *manual* intervention of the developer for *explicitly* marking the variables that contain confidential information (the secret sources) and the methods that output on public channels (the public sinks). Then, based on these annotations, the trackers automatically detect any (explicit or implicit) information flow from the secret sources to the public sinks.

Confidential data leak issues are difficult to catch by standard engineering testing strategies. Therefore, at first glance, information flow trackers seem to be

the ideal solution to the problem of detecting such leaks. However, in practice, a different picture is displayed. Developers are burdened with an error-prone, manual task of figuring out what is sensitive, adding annotations to their code to highlight it, and keeping them up-to-date in a consistent way. As previously highlighted [11], this manual process of annotating (or labelling) the code is one of the main obstacles in the adoption of programming analysis tools at large scale. Furthermore, annotations generate risks of their own, as they may introduce compilation issues due to lack of support for them in the future. In a number of cases, these factors tip the balance between benefit and risk in favour of avoiding the use of automated tools that require manual annotation.

In this paper, we describe a method for *automatically* detecting and annotating confidential data in Java code. Once annotated, the code can be passed on to an information flow tracker for detecting data leaks. By employing an automatic labelling mechanism, we reduce the burden for developers and remove the risk associated with code changes.

More in detail, our approach is based on a graph-based representation of Java programs and consists of rules that characterise confidentiality. We refer to these rules as the confidentiality policy. For example, the policy includes the assumption that if a variable is encrypted, then it is highly likely that is confidential and it should be labeled as such. Our analysis is parametric in the confidentiality policy, so the policy can be extended or modified for different application domains.

Naturally, without any input from the developer, not *all* confidential data will be annotated. For example, variables that are not encrypted, or that do not match our algorithm's "selection" criteria will not be detected. Developers can still extend the policy with other cases, or even resort to manual annotations.

The paper is structured as follows: we introduce background material on graph-based representations for Java programs and the underlying Datalog-based solver in Section 2. Our method is described in Section 3, while details about its implementation and evaluation are reported in Section 4. A discussion on its limitations and possible extensions is presented in Section 5, while related work is discussed in Section 6. Finally, we conclude in Section 7.

2 Background: graph-based representations for Java

Several graph-based representations of Java objects have been used in the past and their variations have appeared under different names such as Groums (Graph-based Object Usage Models) [21], BigGroums [19], and AUGs (API Usage Graphs) [7]. These representations are typically directed acyclic graphs capturing control and data flows, and interactions within and between objects, such as object instantiations, method calls, and data field accesses. While previous work has focussed mainly on detecting mis-uses of APIs [19, 7], our aim is slightly different: we employ the graph-based representation to construct a set of potentially sensitive variables based on their usage in the code. We also extend

previous representations by introducing *inter-procedural edges* (Section 3.4). For simplicity, we further refer to our graphs as *Groums*.

In the following, we give a brief overview of Groums, and for more details we refer the reader to the original work [7, 19, 21].

Definition 1 (Groum). *A Groum is a directed acyclic multi-graph with a single entry node and a single exit node. Nodes can be of three types: action, control, and data. Edges can be of two types: control- and data-flow.*

Nodes. There are three types of nodes in a Groum: action, control, and data. Data nodes (depicted as ellipses) denote the program literals and variables, control nodes (depicted as diamonds) denote the instructions altering the control flow of the program (such as conditional and loop statements, but also exception raising), and action nodes (depicted as boxes) denote all other instructions, such as method invocation (MI), assignments, etc. As a convention, each Groum has a single start and exit node, which have no corresponding instruction in the program they model, and are represented as data nodes.

Edges. A Groum has two types of edges: data flow and control flow. Data flow edges (depicted as directed dotted edges) are either outward edges connecting to an action or control node if the literal or variable they represent is used in that action or control statement, or inward edges if the data they represent is a result of an action, such as method return. Control flow edges (depicted as directed solid edges) connect action and control nodes and denote the order of instruction execution in the program.

Data flow edges are refined further, as follows: condition (`cond`) between a data node and a control node denoting the result of expression guarding the conditional or loop statement or the exception raised, definition (`def`) between an MI action node and a data node, parameter (`param`) between a data node and an MI action node, and receiver (`recv`) between a data node depicting an instance of a class object and a method of that class.

Control flow edges are also refined further, as follows: dependence (`dep`) between two action nodes or between an action node and a control node (not necessarily in that order) denoting the order of instruction execution in a program, exception throwing (`throw`) between an MI action node and a control node representing a `try` statement or `catch` clause, true/false (`T`/`F`) between a control node denoting the guard of a conditional or loop statement and the action/control node denoting the instruction to be executed after the guard evaluation.

An example of Groum, together with the corresponding Java code it models, is depicted in Figure 1.

3 The Algorithm for Automatic Annotations

In our implementation, we extend the code developed for AUGs in [7], which is publicly available [4]. Since the Groum extraction algorithm has been designed

```
5 ...
6 public String myMethod() {
7   String high = getData();
8   String low = encrypt(high);
9   return low;
10 }
```

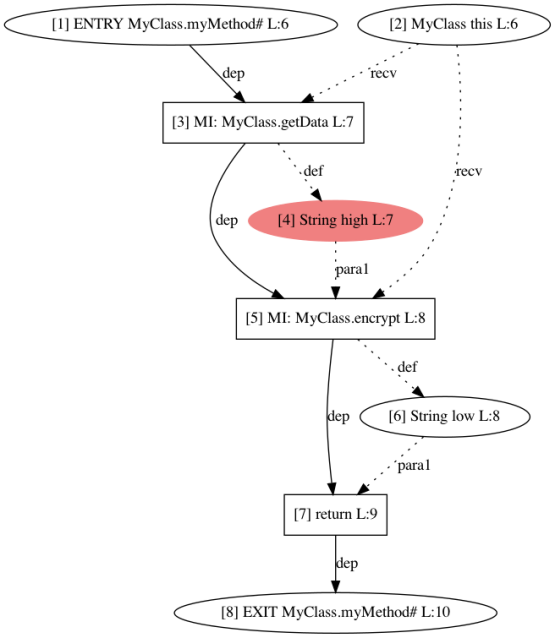

Fig. 1: Java method and its corresponding Groum.

with an interest only in intra-procedural analysis, a separate Groum is extracted for every method and no support for inter-procedural analyses is provided. In this section we describe in more detail our extension which allows for an inter-procedural analysis on Groums.

We employ Datalog and the tool Soufflé as the underlying reasoning engine for our approach. Datalog is a declarative, Prolog-style programming language "introduced as a query language for deductive databases in the late 70s", and Soufflé [6] is an open-source engine for Datalog that has been employed successfully for, among other things, static analysis of Java [2] and vulnerability detection [3].

Our algorithm employs three stages, as depicted in the diagram of Figure 2. Grey boxes represent external programs, while white boxes refer to our implementation. Initially, a Groum is generated (a) for every method in the Java code base. Additional details on the extraction step can be found in previous work [21, 7]. Also here, the Datalog generator (b) encodes the Groums as Datalog facts.

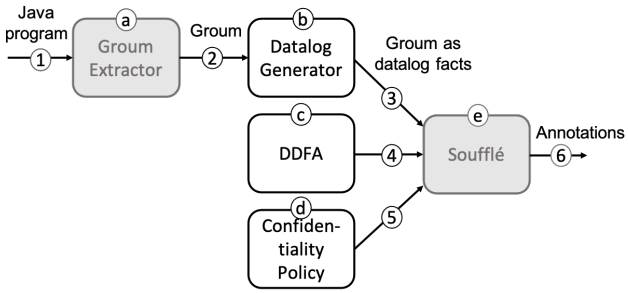

Fig. 2: Stages of our method.

Next, we send these facts to Soufflé, along with the Datalog-based data flow analysis (DDFA) (c), and a confidentiality policy (d) used for specifying the confidentiality criteria. Soufflé evaluates (e) the rules of the DDFA based on the given facts and policy, and outputs the data to be labeled (6).

The last step deals with the actual labelling of the confidential data in the Java source code. Currently, we implement this final step manually, presenting results to developers in textual form.

3.1 Datalog facts extraction

For our purposes, we create a hierarchy of Datalog relations for the Groum nodes, edges and methods for which a Groum is constructed: at the top level, we define relations `GroumNode`, `GroumEdge`, and `Groum` respectively. We use the information contained in `GroumNode` and `GroumEdge` to create more specific relations concerning the nodes and edges. E.g., relation `GroumDefinitionDFEdge` captures `def` edges, and `GroumMethodCallActionNode` represents an MI action node. In this way, we build a one-to-one correspondence between the AUG representation from [7] described in Section 2 and a set of Datalog relations.

3.2 Confidentiality policy

The automated process for deciding which data to label needs some heuristics to base its decisions on. A reasonable indication that a piece of data is confidential is whether it is encrypted, or if it is the result of a decryption method. This represents what we refer to as the *confidentiality policy*.

As such, in our confidentiality policy we include Java APIs implementing cryptographic methods for encryption and decryption. These are methods that either have confidential *parameters* (encryption APIs) or confidential *returns* (decryption APIs). The policy can be extended by the developer with other cryptographically-related APIs or even with other methods known to return confidential data (e.g., `getDeviceId()`) or to have arguments referring to confidential data (e.g., `processUserOrder(userId)`).

Our algorithm further employs the confidentiality policy to detect the starting nodes for the DDFA (Section 3.3). A forward annotation propagation phase detects the data nodes influenced by these initial nodes (Section 3.4).

3.3 Initial data annotation phase

As described in Section 2, a Groum contains parameter `param` and definition `def` data flow edges. These are the edges whose connecting data nodes we target, depending on whether the adjacent action nodes correspond to calls of methods contained in the confidentiality policy. As a result, in the phase of the DDFA for initial data annotation we retain all data nodes connected via a `param` edge to an MI action node denoting an encryption method invocation. The Datalog relation `ConfidentialVarsFromMethodParams` captures this.

Listing 1.

```
ConfidentialVarsFromMethodParams(method, id) ←
    MethodWithConfidentialParams(method, from),
    ParameterDFEdge(method, to, from).
```

Further, we retain all data nodes connected via a `def` edge to an MI action node representing a call to a decryption method. The Datalog relation `ConfidentialVarsFromMethodReturn` captures this.

Listing 2.

```
ConfidentialVarsFromMethodReturn(method, id) ←
    MethodWithConfidentialReturn(method, to),
    DefinitionDFEdge(method, from, to).
```

For example, in the code below, `h` is annotated by our algorithm as confidential as it is the argument of encryption function `encrypt`.

```
String h = getData();
String l = encrypt(h);
```

Observation. The cryptographic methods (or methods added by the developer in the confidentiality policy) whose implementation is part of the codebase under investigation are treated differently, as a Groum is generated for them. This is in contrast with the case when the methods are just API calls and hence no Groum is generated. In the former case, we do not use the intra-procedural `def` and `param` edges to mark the data nodes denoting confidential data, but instead the inter-procedural data flow edges `InputParamEdge` and `OutputParamEdge` which we describe in more detail in paragraph *Inter-procedural DFA* of the next subsection.

3.4 Data annotation propagation phase

In order to evaluate our approach we also implement a forward propagation of the labels, as not all taint trackers support this step. The nodes retained during the initial data annotation phase are used as starting nodes for propagating the confidential labels forward in the graph, by following the data flow paths.

Put rather simply, Groums are control flow graphs extended with data nodes and contain no explicit data flow edges, i.e., there are no edges connecting data nodes with other data nodes. However, this is exactly what we need for our second stage of the DDFA—data annotation propagation through the data flow path.

Hence, we extend Groums with additional edges connecting data nodes, both intra- and inter-procedurally. Thus, two data nodes are connected (intra- or inter-procedurally) if there is a data dependence relation between the `from` node and the `to` node, i.e., the value of node `from` flows-to or influences the value of node `to`.

We discuss each case of dependence, intra- and inter-procedurally separately, starting with the former.

Intra-procedural DFA. At the moment, we support the intra-procedural cases listed below. Note we also model data flows via exceptions (not listed in the rules below).

Listing 3.

```
IntraDFEdge(method, from, to) ←
  (ReceiverDFEdge(method, from, recv),
   DefinitionDFEdge(method, recv, to))
  ;
  (ParameterDFEdge(method, from, m),
   DefinitionDFEdge(method, m, to),
   ¬IsGroum(method, m))
  ;
  (ConditionDFEdge(method, from, cond),
   ControlFlowBlock(method, cond, join),
   cond < id <= join,
   DefinitionDFEdge(method, id, to)).
```

Observe from the last case of relation `IntraDFEdge` that our analysis takes into account control dependencies, whereas typical taint analyses consider only data dependencies for tainting. This means that a control flow block (such as conditional branches or loops) guarded by confidentially-labeled data will taint everything (re-)defined inside it. More specifically, assuming `h` is marked as confidential in the program below, `l` will be marked as confidential as well, as their corresponding data nodes will be connected through an `IntraDFEdge`.

```
if (h > 0) { l = 1; } else { l = 0; }
```

In this regard, our analysis performs an over-approximation, as in the example which follows, a slight variation of the previous one, `l` is marked as confidential, although at runtime it will be influenced by `h` only if `h > 0`.

```
if (h > 0) { l = 1; }
```

Inter-procedural DFA. Unfortunately, the original implementation of Grooms in [7] does not provide support for inter-procedural analyses, as a separate graph is generated for every method of the program being analysed and no relation between them is provided. Thus, there are no inter-procedural (data flow) edges, and no call-graph is given.

In order to capture inter-procedural data flows, we extend the initial Groom analysis with three new types of edges that connect previously disconnected Grooms by creating three new Datalog relations:

- `CallDependenceEdge` — between an MI action node in the caller Groom and the start node of the corresponding callee Groom of the method invoked in the action node.
- `InputParameterEdge` — between a data node denoting a parameter to an MI action node in the caller Groom and its corresponding argument node in the callee Groom of the method invoked in the action node.
- `OutputParameterEdge` — between a return action node in the callee Groom and the data node defined by an MI action node in the caller Groom denoting the method depicted by callee Groom.

Further, based on these new edges, we define relation `InterDFEdge` for connecting data nodes in different Grooms:

Listing 4.

```
InterDFEdge(caller, from, callee, to) ←
  (InputParameterEdge(caller, from, callee, param),
   DefinitionDFEdge(callee, param, to))
  ;
  (OutputParameterEdge(caller, to, callee, return),
   ParameterDFEdge(callee, from, return)).
```

Annotation propagation. We obtain all data nodes originating in the nodes computed during the initial phase by following the data flow paths obtained from relations `IntraDFEdge` and `InterDFEdge` (a path is defined as the transitive closure of an edge relation). The relation `ConfidentialDFPath` is responsible for this.

Listing 5.

```
ConfidentialDFPath(caller, from, callee, to, cxt) ←
  (DFPath(caller, from, callee, to, cxt),
   NodeFromInitialPhase(caller, from)
   ;
   ConfidentialDFEdge(caller, from, callee, to, cxt)
   ;
```

```
1 public void backwardInter(String s) {
2   String h1 = "high";
3   String l = myMethod(h1);
4 }
5
6 public String myMethod(String h2) {
7   return encrypt(h2);
```

Fig. 3: Inter-procedural example.

```
  ConfidentialDFPath(caller, from, m, id, cxt),
  DFPath(m, id, callee, to, _)),
 ¬IsDeclassified(callee, to).
```

Note that not all data nodes belonging to a data flow path originating in the data nodes returned by the initial phase of DDFA may require annotations. Assume the following code:

```
enc = encrypt(pwd);
```

DDFA will rightfully mark `pwd` as in need of annotation, as it is the argument of an encryption method. In addition, the DDFA will create a data flow edge between the parameter node `pwd` and the defined variable `enc`. Since `pwd` is annotated, `enc` would become annotated as well, although there is no need for it, as encryption methods act as declassifiers and no information can be learned about the encrypted value.

This is the role of relation `IsDeclassified` called during the creation of a `ConfidentialDFPath`, to check whether a data node should be marked as declassifier. If a node is marked as such, then all the nodes on the data flow path are discarded and as consequence, not marked for receiving annotations.

This backward step also works inter-procedurally. For example, in function `backwardInter` in Figure 3, `h1` is properly marked as confidential, because it is used as a parameter of `myMethod`, and the parameter of `myMethod` is marked as confidential as an argument of a sanitiser function.

Observe relation `ConfidentialDFPath` takes a 5th argument—`cxt`, which is used to distinguish between different calls to a certain callee method taking place in the same caller method. E.g., our analysis is able to distinguish between the two calls to the method `foo` in the snippet below:

```
int x = foo(a);
int y = foo(b);
```

4 Evaluation

We have implemented the DDFA analysis in Datalog. The actual Datalog code for the deduction rules consists of approximately 650 lines of code. The Datalog

```
protected void doGet(HttpServletRequest req, HttpServletResponse
    resp) throws IOException {
  String name = req.getParameter(FIELD_NAME);
  Object o1 = name;
  Object o2 = name.concat("abc");
  Object o3 = "anc";

  PrintWriter writer = resp.getWriter();
  writer.println(o1);                          /* BAD */
  writer.println(o2);                          /* BAD */
  writer.println(o3);                          /* OK */
}
```

Fig. 4: Test case Aliasing4 from SecuriBench-microbenchmark.

facts generator is implemented on top of the existing AUG Java implementation from [7] and consists of approximatively 350 additional lines of code. In this section we report results obtained in two scenarios: using a publicly available benchmark and on previously annotated Java code within Amazon code bases.

4.1 SecuriBench

In addition to programs extending the basic structure of the examples described in the previous sections, our analysis was tested on the SecuriBench-microbenchmark [5]. SecuriBench-microbenchmark contains minimal test cases, each of them checking a specific ability of the static analyser. For example, Aliasing4 (depicted in Figure 4) checks for simple aliasing with casts. The test case is annotated with "BAD" or "OK", indicating what should be flagged or not. In this case, our analysis behaves correctly, it detects the two illicit outputs but not the last one which is valid.

Note that this benchmark is not designed for assessing how precise the labelling is performed, it only evaluates the label propagation. For example, in Aliasing4, we have marked `req.getParameter` as being a method with confidential return. Therefore the labelling part of our algorithm marks `name` as confidential, and the label propagation part then propagates it forward.

The results of our analysis are shown in Table 1, by reporting on 12 categories. The first column presents the category, the second the number of true positives (TPs) detected by our analysis compared to the total, while the last column depicts the false positives (FPs) given by our analysis.

Our analysis was able to flag most of the *aliasing* (10/12) and *basic* (54/60) cases, with only 2 FPs. 5 of the missed cases and the 2 FPs are due to lack of field and array sensitivity, other 3 are due to the fact that we do not mark constructors, such as `new FileWriter` as public sinks. These results show that our DDFA analysis is able to handle complex control flows such as the one in example

Table 1: SecuriBench-micro benchmark

Category	**TP/Total**	**FP**
Aliasing	10/12	0
Arrays	2/9	1
Basic	54/60	2
Collections	0/14	1
Data Structures	0/5	0
Factory	3/3	0
Inter	8/16	0
Pred	3/3	4
Sanitizer	3/4	3
Session	0/3	0
Strong Updates	0/1	0

Table 2: Reconstructed annotation

Service	**Found/Total**	**Analysis time (s)**
S1	0/1	5.53
S2	1/1	3.85
S3	1/2	3.86
S4	2/2	3.71
S5	1/1	3.72
S6	2/2	3.99
S7	2/3	4.11

Basic28, in which there are 39 branchings, nested in various combinations up to 9 times deep.

4.2 Reconstructing existing annotations

A further data point for the evaluation of our approach is provided by considering code that has been previously annotated with labels to characterise confidential information. In particular, we have considered 7 existing software packages implementing Amazon services and we have extracted the Java implementation of classes that contained annotated variables using the Checker framework [1]. Overall, we identified seven files containing 12 annotated variables. Our analysis was able to find 9 out of the 12 annotated variables.

Table 2 reports the number of annotations found by our algorithm versus the total number of annotations present and the execution time (all the experiments have been performed on a standard 2019 Macbook laptop with 16 Gb of Ram). The size of each class ranges between 60 and 426 lines of code; the names of services have been anonymised.

5 Discussion and limitations

One key feature of our method resides in working with a graph-based representation of the program, and its modeling in Datalog. This renders our approach (almost) language-independent. Once a Groum conversion is applied to a program expressed in a language other than Java, our Datalog analysis would require minimal interventions before it could annotate those programs as well.

5.1 Limitations

Our analysis is work in progress and, as discussed below, it cannot provide completeness guarantees and it does not deal with persistent memory storage. However, as seen in the preliminary results discussed in the previous section, it already shows some promising results. There are several limitations worth mentioning.

First, with the exception of the backward propagation of declassifiers, our framework performs a forward analysis only, so it misses to label data when backwards steps are required. For instance, in the program below, the DDFA will label as confidential the return value of `foo(pwd)`, but not `pwd`.

```
encryptedPassword = encrypt(foo(pwd));
```

Second, when performing the backward step for detecting the arguments of encryption methods, our analysis only looks at the last definition of those arguments, and it does not inspect how they were formed. For example, in the program below, our analysis only annotates `h2`.

```
String l1 = "Something_Public";
String h1 = "Something_Secret";
String h2 = l1 + h1;
String l2 = encrypt(h2);
```

The analysis could be extended to cover this case by performing a backwards analysis as well, but without additional information provided by the developer, it would lead to additional false positives. E.g., in the program above, it would falsely annotate `l1`.

Consider again function `backwardInter` from Figure 3. Although `myMethod` is considered a declassifier, as it returns the encryption of its argument, due to our computing of the transitive closure of the edge relations, `l` ends up falsely marked as confidential.

The approach presented in this paper targets Java and therefore we support dynamic memory allocation, even if we are not fully precise in terms of context sensitivity. For instance, adding call-sensitivity context would further improve DDFA's precision. Consider the program below:

```
String userId = getUserId();
String l1 = foo("abc");
String h = foo(userId);
String l2 = foo("xyz");
```

First, the user ID (returned by method with confidential returns `getUserId`) is stored in variable `userId`, then method `foo` is invoked three times each with parameters `"abc"`, `userId`, and `"xyz"` respectively, and its results are stored in variables `l1`, `h`, and `l2` respectively. The analysis should only label as confidential `h`, but it labels as confidential `l2` as well, as the returned value of method `foo` is marked as confidential in its Groum due to the dependency to confidential `userId`.

Finally, as we previously mentioned, our analysis does not currently support field sensitivity.

5.2 Other approaches

Improving precision. As discussed in the previous sections, our algorithm uses a single Groum for every method invoked and encodes additional information to capture context-call sensitivity and to distinguish between different invocations of the same method.

Another approach would be to use a Groum for every method invocation. The resulting inter-procedural graph may *explode*, but the algorithm's precision would improve. An investigation on how the performance may be affected in this case would also be required. The implementation of this variant, as well as an analysis on the trade-offs between the two approaches is left for future work.

Upgrade to information flow analysis tool. A natural extension of our algorithm is to transform it into an information flow analysis tool, by expanding the confidentiality policy to include methods that should be considered as public sinks. Then, we could get an information flow analysis by extending the algorithm with a relation which simply checks that no annotated nodes in the graph are parameter nodes of the public methods.

6 Related Work

There is a substantial body of work in this area. In this section, we discuss and compare our method with some of the related work.

Automatic labelling of confidential data. Merlin [18] infers information-flow specifications in .NET code using a data propagation graph to model inter-procedural data flows. In contrast to our approach, Merlin uses probabilistic constraints, potentially resulting in an exponential number of constraints that are then approximated to achieve scalability. Zhu et al. [27] present an approach to infer confidentiality annotations for library calls without the corresponding source code being available, but still assumes other sources and sinks in the program to be annotated.

Groums. Groums (Graph-based Object Usage Model) [21], which form the basis of our approach, were initially designed for automatically inferring API usage patterns from an API's usage in a code base. Groums were later also used for detecting API-misuse [7].

Information-flow control. Information-flow control [16, 23] is an active area of research focused on detecting information leaks in programs providing stronger security guarantees than taint trackers. There exist both dynamic and static approaches to information-flow control for many languages, such as Jif [20], Joana [14], and Paragon [9] as extensions of Java, LIO [10, 26] and FlowCaml [22] for languages in the ML family, as well as JSFlow [15], a dynamic information-flow tracker for EcmaScript [12]. All of the above approaches require some amount of user annotation to indicate which inputs to a program are confidential. The approach presented in this paper can be used to automate this annotation process, assuming the availability of Groums, and can potentially simplify the use of information-flow control in practice.

Taint tracking. Taint tracking is a practical approach to information-flow control that intentionally ignores [24] some information leakage resulting from less explicit features of program semantics such as control-flow, termination, and concurrency. Taint tracking can be applied both statically [17] as well as dynamically [25]. Similar to the approach here, Li et al. [17] present a static taint tracking system based on program dependency graphs (PDGs), which have similarities with Groums. This representation would allow an approach similar to the one presented here to automate the labelling of confidential inputs and outputs. Many taint-tracking systems have been applied to real-world applications: TaintDroid [13] and FlowDroid [8] are taint-tracking systems for Android applications. The Checker Framework [1] allows building custom type checking extensions for Java programs and includes support for taint tracking. Similar to information-flow control approaches, such systems typically require manual annotation to indicate which sources and sinks are confidential. The approach here can be used to lessen the annotation burden to developers, potentially enabling an easier use of taint tracking on real world software.

7 Conclusion

We have presented a method for automatically annotating confidential data in Java programs. Our method uses a graph-based program representation based on Groums to mark the data nodes denoting the confidential information, based on a confidentiality policy. This policy is designed to mark as confidential data which either is encrypted or results from decryption operations. The confidentiality policy also allows for developer extensions to capture more cases of interest. We have implemented our approach using Datalog and we have assessed the current features and limitations against publicly available examples. We have also validated the approach using existing internal code bases, reproducing 75% of the existing annotations.

We see our work as an initial step in the construction of a fully automated tool to generate annotations for confidential data, with the long-term goal aim of enabling zero-touch information flow analysis.

Acknowledgments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Checker framework. `https://checkerframework.org/manual/`
2. Doop framework. `https://bitbucket.org/yanniss/doop/src/master/`
3. Java Vulnerability Detection. `https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49\_PROJECT\_ID:122`
4. MUDetect. `https://github.com/stg-tud/MUDetect`
5. SecuriBench-micro. `https://github.com/too4words/securibench-micro`
6. Soufflé. `https://souffle-lang.github.io`
7. Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M.: Investigating next steps in static API-misuse detection. In: MSR 2019, 26-27 May 2019, Montreal, Canada (2019)
8. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 259–269 (2014)
9. Broberg, N., van Delft, B., Sands, D.: Paragon - practical programming with information flow control. J. Comput. Secur. **25**(4-5), 323–365 (2017)
10. Buiras, P., Vytiniotis, D., Russo, A.: HLIO: mixing static and dynamic typing for information-flow control in haskell. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. pp. 289–301 (2015)
11. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. p. 332–343 (2016)
12. ECMA International: Standard ECMA-262 - ECMAScript Language Specification. 5.1 edn. (June 2011)
13. Enck, W., Gilbert, P., Chun, B., Cox, L.P., Jung, J., McDaniel, P.D., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings. pp. 393–407 (2010)
14. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security **8**(6), 399–422 (Dec 2009)
15. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking Information Flow in JavaScript and its APIs. In: SAC (2014)
16. Hedin, D., Sabelfeld, A.: A perspective on information-flow control. In: Software Safety and Security - Tools for Analysis and Verification, pp. 319–347 (2012)

17. Li, B., Ma, R., Wang, X., Wang, X., He, J.: DepTaint: A Static Taint Analysis Method Based on Program Dependence. In: Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences. pp. 34–41 (2020)
18. Livshits, V.B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 75–86 (2009)
19. Mover, S., Sankaranarayanan, S., Olsen, R.B.P., Chang, B.E.: Mining framework usage graphs from app corpora. In: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018 (2018)
20. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow (July 2006), `http://www.cs.cornell.edu/jif`
21. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: ESEC/FSE, 2009, Amsterdam, The Netherlands, August 24-28, 2009 (2009)
22. Pottier, F., Simonet, V.: Information flow inference for ML. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 319–330 (2002)
23. Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers. pp. 352–365 (2009)
24. Schoepe, D., Balliu, M., Pierce, B.C., Sabelfeld, A.: Explicit secrecy: A policy for taint tracking. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016. pp. 15–30 (2016)
25. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. pp. 317–331 (2010)
26. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in haskell. In: Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011. pp. 95–106 (2011)
27. Zhu, H., Dillig, T., Dillig, I.: Automated inference of library specifications for source-sink property verification. In: APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. pp. 290–306 (2013)