Privacy Property Graph: Towards Automated Privacy Threat Modeling via Static Graph-based Analysis

Immanuel Kunz
Fraunhofer AISEC
Garching b. München, Germany
immanuel.kunz@aisec.fraunhofer.de

Konrad Weiss
Fraunhofer AISEC
Garching b. München, Germany
konrad.weiss@aisec.fraunhofer.de

Angelika Schneider
Fraunhofer AISEC
Garching b. München, Germany
angelika.schneider@aisec.fraunhofer.de

Christian Banse
Fraunhofer AISEC
Garching b. München, Germany
christian.banse@aisec.fraunhofer.de

ABSTRACT

Privacy threat modeling should be done frequently throughout development and production to be able to quickly mitigate threats. Yet, it can also be a very time-consuming activity. In this paper, we use an enhanced code property graph to partly automate the privacy threat modeling process: It automatically generates a data flow diagram from source code which exhibits privacy properties of data flows, and which can be analyzed semi-automatically via queries. We provide a list of such reusable queries that can be used to detect various privacy threats. To enable this analysis, we integrate a taint-tracking mechanism into the graph using privacy-specific labels. Since no benchmark for such an approach exists, we also present a test suite for privacy threat implementations which comprises implementations for 22 privacy threats in multiple programming languages. We expect that our approach significantly reduces time consumption of threat modeling and show that it also has potential beyond the threat categories defined by LINDDUN, e.g. to detect privacy anti-patterns and verify compliance to privacy policies.

KEYWORDS

Privacy Threat Modeling, Cloud Privacy, Automated Risk Assessment, Static Code Analysis

1 INTRODUCTION

Threat modeling heterogeneous architectures like cloud systems for privacy and security risks has become increasingly harder. This problem has been explained partly with the shift towards compartmentalized, service-driven architectures which has made end-to-end data flow analyses more difficult [23]. Moreover, distributed systems change frequently—during development and in production—and so do their attack surfaces and the privacy threats their users are exposed to [42]. Risk analyses should therefore be done frequently to identify risks as early as possible. Yet, performing a thorough risk analysis, including threat modeling and impact evaluation, can be very time-consuming as the system's data flows have to be modelled, possible attack vectors identified, impacts evaluated, and risks prioritized, until finally, mitigative measures can be derived.

Especially in agile development it is difficult to integrate this time-consuming process into the short development sprints [14, 22]; but also the maintenance of large applications in general requires privacy risks to be re-assessed frequently to check whether the system still conforms to initially established privacy requirements.

The standard approach to modeling privacy threats is LINDDUN [9, 49] which—similarly to STRIDE [35] in the context of security threat modeling—describes a threat modeling methodology for privacy. Its practical application, however, can suffer from the problems described above. This conflict calls for an automated approach to the identification of privacy threats that reduces time spent on threat modeling, and allows the frequent identification of threats across multiple services (see also [14, 23, 42]). Previous works have already proposed automated analyses on extended data flow diagrams [6, 47] but they do not generate these diagrams automatically and do not include privacy properties. Thus, they also suffer from the limitations mentioned above.

In this paper, we leverage an existing code property graph library [48] to enable (semi-)automated privacy analyses of distributed applications. A code property graph includes data flow information—similar to a data flow diagram—that is generated automatically, and can be extended with dedicated nodes and edges, to represent privacy-related properties. Based on the threats proposed in LINDDUN GO [49], we show which privacy threats can be detected automatically in a code property graph. To make privacy properties, like identifiers, visible in the graph we use a language-independent taint-tracking approach for code property graphs that extracts privacy labels from code annotations and comments. We also extend the graph with further privacy-related properties, like database operations. The graph can then be inspected via reusable queries that identify generic or application-specific privacy threats. We propose a number of such reusable and adaptable queries.

Our approach not only partly automates the privacy threat modeling process, but also allows to identify privacy issues that would not be detectable as easily in a manually created data flow diagram. For example, it can be used to identify entry points and exit points of personal data, which can be utilized for a granular data flow tracking across services.

Proceedings on Privacy Enhancing Technologies 2023(2), 171–187
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2023-0046

We also propose a test suite of 22 privacy threat implementations as a benchmark for the evaluation of static analysis tools. Our own implementation performs with a precision of 0.65 and recall of 0.91 on this test suite.

In summary, we present the following contributions:

- An operationalization of LINDDUN privacy threats for their detection in a code property graph,
- an enhanced code property graph implementation (called the *Privacy Property Graph*) that allows to detect the identified properties, partly via taint-tracking, as well as a number of reusable graph queries, and
- a test suite of privacy threats that for the evaluation of graph-based static analysis tools.

2 BACKGROUND

2.1 Code Property Graphs

Yamaguchi et al. first introduced the concept of a Code Property Graph (CPG) [51], a directed attributed graph representing source code that allows the analysis of large source code projects. It consists of nodes that represent the syntactic elements of the source code, as well as edges that put the nodes into relation. The representation of source code in a code property graph allows to search large code bases for vulnerabilities and patterns that can give insight into whether a program behaves in accordance to compliance criteria. Constructed from source code or an intermediate representation, it is stored in a graph database and can then be queried with declarative languages.

A CPG's edge types are used to represent different aspects of source code: First, the program's syntactic structure, in the form of an Abstract Syntax Tree (AST), second, the program's execution order, e.g. as a Control Flow Graph (CFG) between the AST-nodes, and third, the program's data flow, e.g. as a Data Flow Graph (DFG). The dedicated DFG allows to track data flows and is therefore essential for our approach to track personal data. Furthermore, AST and CFG are important to follow the control flow and thus achieve flow-sensitivity which is important for taint tracking. Alternative solutions include, e.g., Program Dependence Graphs (PDG) which can be seen as a combination of a CFG and DFG. Note that we will consider adding a PDG to our approach in future work, since it could help to avoid false negative results.

Practical implementations of code property graphs differ in the abstraction level and the languages that they support: *Joern* [52] is a platform for robust analysis of C/C++ code that also supports other languages. Joern as well as *Plume* [10] and *Graft* [21] support the analysis of Java bytecode. The *cpg* [48] represents control flow on a finer granularity, also capturing the order in which expressions are evaluated. It can represent Java, C/C++ as well as Python, Go and TypeScript code in a unified format, allowing the analysis of heterogeneous services written in multiple languages.

2.2 The Cloud Property Graph

The Cloud Property Graph (CloudPG) [5] is an extension of the cpg. Among others, it adds runtime information of cloud deployments, e.g. security features like encryption properties and logging functionalities, as well as deployment information like container image and orchestration configurations, to the graph. The information is added according to an ontological description of cloud resources and security features. Since it adds data flows between endpoints, it also allows to detect security threats across services.

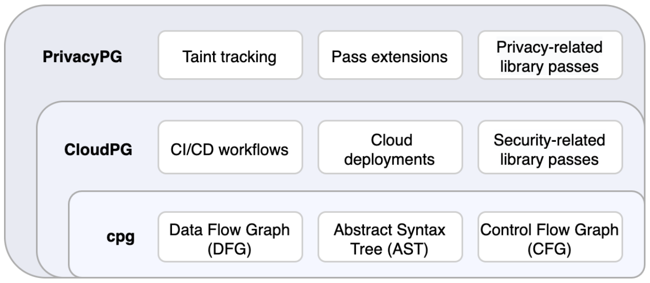

Figure 1: PrivacyPG, CloudPG, and cpg: the basic code property graph (cpg) provides node relations of three types (see Section 2.1), while the Cloud Property Graph (CloudPG) adds nodes and edges e.g. regarding CI/CD workflows. The Privacy Property Graph (PrivacyPG, or PPG) adds taint tracking, library-specific passes and extensions of existing passes for more languages and implementation variations.

Furthermore, the CloudPG is extendible, e.g. to add dedicated nodes for specific technologies. Since the detection of privacy threats requires special properties in the data, we leverage its extendibility in this paper to add privacy-relevant concepts. Note that using the CloudPG to analyze an application is not limited to those deployed in cloud systems. The resulting analysis, however, benefits from additional cloud-specific information as described above in case of a cloud deployment.

In this paper, we refer to the CloudPG as the implementation basis described above and in [5], while we refer to the Privacy Property Graph (PPG) as the extended version of the CloudPG which reflects properties that allow to detect privacy threats in the graph. Figure 1 shows the relation between cpg, CloudPG and PPG.

2.3 Privacy Threat Modeling

In this paper we aim at automating several steps of the privacy threat modeling process. A popular method for conducting privacy threat modeling is LINDDUN [9]. LINDDUN considers the privacy protection goals anonymity, unlinkability, undetectability, plausible deniability, confidentiality, policy non-compliance, and content unawareness, based on the definitions by Pfitzman and Hansen [34]. A more leightweight approach is LINDDUN GO which uses a reduced scope and makes LINDDUN easier to apply for non-experts. Still, the LINDDUN GO authors claim that it is comprehensive if all proposed threat types are considered. In this paper, we use the LINDDUN GO threat descriptions [49], since it is the more recent version and describes the threats in a more compact form. This way, we can operationalize them one by one without neglecting any of the original LINDDUN threats.

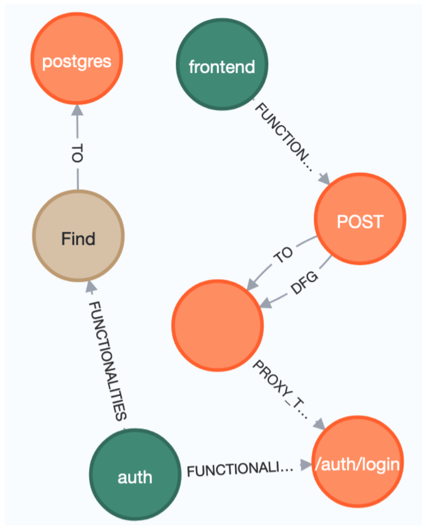

Figure 2: A screenshot of a part of the code property graph generated for an example application. It , which shows a basic data flow diagram. It can be generated with a reusable query and shows the microservices of the application, the HTTP calls that exist between them, as well as the databases they access: in this case the *frontend* service employs an HTTP POST request to call the login endpoint of the *auth* service which in turn has a database operation on a PostgreSQL database defined.

3 APPROACH: GRAPH-BASED PRIVACY THREAT MODELING

In this section we analyse privacy threats to show if and how they can be detected in a static, graph-based code analysis. The results are summarized in Appendix A.

3.1 Creating a Data Flow Diagram

The first step to detect privacy threats is to create a data flow diagram. We use an existing code property graph library, the *cpg* (see Section 2.1) ,to generate a code property graph—essentially an enriched data flow diagram—via static source code analysis, and store the graph in a graph database. Figure 2 shows an excerpt of the graph generated for an example application.

Furthermore, using reusable queries, one can obtain a high-level graph that resembles a manually created data flow diagram including microservices, data flows between them, databases they access, and further information.

3.2 Operationalizing Privacy Threats

In the following, we identify the general data properties that are required to detect these privacy threats automatically, ordered by the LINDDUN GO threat categories. Appendix A includes a short technical description of each LINDDUN GO threat as well as a summary of these data properties.

We differentiate between contextual data, i.e. metadata, and transactional data, i.e. payload data. While contextual data can usually be predicted in a static code analysis, e.g. due to the communication protocol that is used, transactional data cannot always be predicted, since it often depends on user inputs. We also identify the technical operations that need to be recognized to detect privacy threats, e.g. database operations.

Note that many of the LINDDUN threats [49] aim at helping users to think of possible threats, partly through scenario descriptions. While this is helpful for brainstorming sessions, we show in the following that they are redundant in several cases. For example, threat *ID1 Identifying Credentials* suggests to investigate if user credentials contain identifiers or unique information that can result in profiling. Technically, however, it can be seen as a more specific threat than *ID3 Identifying Inbound Data* which concerns all personal data flowing from users to the service.

3.2.1 *Identifiability.*

Identifiability is the feasibility of the data controller to identify a user from the contextual or transactional data the user sends to the service (or that is collected from the user).

LINDDUN GO identifies seven identifiability threats and differentiates between threats that stem from credentials (ID1), inbound data (ID3), i.e. data that flows from the user to the server, sharing identifiable data with third parties (ID5), and storing (ID6) and retrieving (ID7) identifiable data in/from databases. Furthermore, users may be identified by side-channel information of their actions (ID2), or through identifiable contextual data (ID4).

Generally speaking, this category includes threats that result from identifiable transactional data, identifiable contextual data, or from side-channel threats. In this paper, we do not consider side-channel attacks (ID2), since the identification of such attacks requires a broader analysis of the system and its context, e.g. expected user behavior, which is not available in a static analysis.

Thus, the technical properties that need to be detected, are the following. First, identifiers need to be detected in the code. For transactional data (ID1, ID3, ID5, ID6, ID7) we need to detect declarations, e.g. variables, that hold identifiable information, and endpoints that send and accept such declarations, and we need to detect and follow the data flows between them. Second, threats result from identifiers included in contextual data (ID1, ID4). These are usually determined by the protocol that is used, e.g. IP addresses are included in metadata of the HTTP protocol. Since this metadata is not directly available in a code property graph, we need to detect the relevant protocols and libraries, and detect their usage in the graph. In our implementation, we therefore use protocol-specific extensions to determine if a certain technology is used and detect threats accordingly (see Section 4). Third, database interactions need to be recognized (ID6, ID7). Since the implementation of database interactions varies with the used technology, they need to be detected per database technology.

3.2.2 *Linkability.*

Linkability is the feasibility of the data controller to link together two pieces of information about a user, e.g. two requests or two pieces of data about the user which are stored in different databases.

LINDDUN GO lists 7 linkability threats which are analogous to the identifiability threats: linkability of credentials (L1), user actions (L2), inbound data (L3), contextual data (L4), shared data with third parties (L5), and stored (L6) and retrieved data (L7).

While identifiability results from the single transmission of an identifier, linkability results from multiple transmissions of pseudo-identifiers. If the transmission of a pseudo-identifier is implemented in code, it can be assumed that it can also be transmitted more than one time, i.e. a potential linkability threat can be derived in this case.

Additionally, pseudo-identifiers may be transmitted from different sources, e.g. from several devices pertaining to the same user. In this case a static analysis needs to detect the services that send and consume the pseudo-identifiers to detect the linkability threat. Note that identifiability always entails linkability as well, since identifiers are always linkable.

The properties we need to detect are therefore similar as for identifiability, i.e. pseudo-identifiers in transactional and contextual data, as well as database interactions. Again, we do not consider linkability threats that result from side-channel information (L2).

3.2.3 Non-Repudiation. Non-repudiation is the inability of the user to deny an action, such as sending a message. It is the opposite of plausible deniability. Often, non-repudiation is treated—under certain assumptions—as the undisputable attribution of an action to a person, e.g. a message is attributed to a person who is believed to hold the private key that was used to cryptographically sign a message. LINDDUN also considers non-cryptographic measures, such as logs that record actions and messages, which still can present credible proof of an action.

The following 5 non-repudiation threats are described in LINDDUN GO: Non-repudiation of credentials (NR1), sending a message (NR2), receiving a message (NR3), storing data (NR4), and retrieving data (NR5). Analogous to other categories, NR4 and NR5 are extensions of the other threats, but add a database operation. For example, a non-repudiable credential may be used to log in to a service (NR1), an event log is created and saved (NR4), and later it is retrieved by an administrator (NR5).

Similarly to ID1 and L1, NR1 requires the detection of identifiers, but specifically in credentials. Threat NR2 may occur in various scenarios of which we target two important ones: First, the usage of private keys for digital signatures, which presents the classic non-repudiation property in the cryptographic sense. Second, we cover logging mechanisms that record the sending of messages. These two scenarios are usually not self-implemented in an application but are usually implemented using third-party libraries. Thus, the usage of such libraries needs to be detected in combination with (pseudo-) identifiers. Finally, NR4 and NR5 again require the detection of database operations.

For NR3, we need to detect automatic acknowledgments of message receipts. This threat implies that a message is sent to the user where it triggers an automated response, indicating the reception. It therefore requires the detection of a very specific heuristic of data flows which we do not cover in this paper—encoding this heuristic in a static analysis can lead to either many false positive or false negative results, and should be addressed in future work.

3.2.4 Detectability. Detectability is the ability of an attacker to distinguish a specific user action from others in a set of actions.

LINDDUN GO differentiates between 5 detectability threats: detectable credentials (D1), communication (D2), outliers (D3), storage operations (D4), and retrieval operations (D5).

Detectability threats mostly concern responses of a service's APIs and what information can be derived from these responses. For instance, an account registration interface may return an HTTP 409 ("Conflict") header, indicating that a certain email address is already in use for the service (D1). Depending on the kind of service, this information can violate the user's privacy. Similarly, information can be derived from API responses to database queries (D4, D5).

To detect an API that leaks personal information, we need to detect several code operations in combination. First, APIs need to be identified that accept identifiers or pseudo-identifiers. Second, a database operation using that (pseudo-)identifier needs to be detected, and third, a response that leaks information about the DB state needs to be recognized, i.e. a response of the type "Not Found" or "Conflict", i.e. 404 or 409 in the case of HTTP headers. Note that we focus on HTTP connections here, but expect that the approach works for other protocols as well[1].

We furthermore assume that a detectable communication threat occurs when an HTTP connection from one trust domain to another is established, resulting in an observable data flow (D2). Also, database operations need to be identified (D4, D5). As explained above, threats based on side-channel information are out of scope for this paper, which is why detectable outliers (D3) is not considered here.

3.2.5 Unawareness. Unawareness threats result from data subjects' insufficient knowledge or control over their personal data after its collection. More specifically, this category includes the threats of insufficient transparency, i.e. insufficient information given to the user about the handling of personal data (U1), missing mechanisms for users to access (U3) and modify (U4) their data, as well as missing checks for user consent (U5). Finally, threats to usable privacy in general are addressed (U2), including, for instance, privacy-unfriendly default settings.

To check for mechanisms for managing personal data (U3, U4), we need to combine the detection of database operations on personal data with a role-based access control check, since we only want to detect database operations meant for users on *their* data.

We would argue that transparency information (U1) should not be addressed by a static analysis. It can rather be addressed once the implementation is considered (mostly) finished, since it can easily be added without impacts on other parts of the code or the architecture. Still, a graph-based analysis can give guidance on which transparency notices should be created, e.g. by showing which personal data flows exist in the system. Similarly, it can help to uncover threats to usable privacy (U2), which, however, should be addressed in respective validation tests with users.

Consent checks (U5) are usually obtained once before data collection starts, e.g. when a user installs a mobile app. This threat is thus not relevant for our approach, but should be considered together with the legal ground for the data collection (see NC2 below), e.g. once before and verified after the implementation.

3.2.6 Policy Non-Compliance. Policy non-compliance refers to the data controller's adherence to privacy policies, for instance policies based on data protection regulations.

[1]Most file transfer protocols use status codes that indicate availability of a file or correctness of a username, for example see FTP status 331 ('User name okay, need password') and 450 ('Requested action not taken. File unavailable'), and SCP 6 ('File does not exist') and 79 ('Invalid user name')

LINDDUN GO lists violations of the purpose limitation principle, including disproportionate collection (NC1), processing (NC3), and storage (NC5), as well as processing personal data without a legal basis (NC2). Finally, automated decision making about users' personal data is stated as a threat (NC4).

To detect NC5, we can check if a personal datum is stored in a storage, e.g. a certain database table, but no access is implemented which could retrieve it. Regarding NC1, we need to detect if a (pseudo-)identifier is sent from user to server, and if it is processed there or not (apart from storage operations). To detect such a processing step, we can use the code property graph's node types to check, e.g., if the datum is assigned to a variable, but not used in a processing expression.

The other threats are out-of-scope for our approach: To detect NC3, one needs to know the purpose of the data collection and how it should be reflected in code, which cannot be generally addressed in our approach. Regarding NC2, the legal basis for collecting personal data does not have to be addressed in a regular threat modeling process, but can be a one-time activity before development of the system starts. Furthermore, automated decision making (NC4) targets situations where an impactful decision, e.g. the decision about a loan application, is made completely automatically. This is rather a design decision that should be addressed by other means since it requires a contextual consideration of what is "impactful".

3.2.7 *Confidentiality.*

Confidentiality is not explicitly considered in LINDDUN GO[2]. The CloudPG we use as the basis for our implementation already covers the detection of some types of security threats (see Section 2). In Section 5 we propose a test library that also includes test cases that implement unencrypted HTTP connections. We do not, however, go into further detail regarding security threat modeling in this analysis.

Summarizing, it can be said that many privacy threats have common properties in source code, such as the usage of (pseudo-) identifiers, database interactions, role checks, and the usage of dedicated privacy-relevant libraries. These can be detected in a static analysis, which can therefore support threat modeling activities and reduce their time consumption.

3.3 Prioritizing Threats

After detecting potential threats, the next step is to prioritize them according to their relevance. We distinguish general and application-specific prioritization criteria.

Generally, it is meaningful to prioritize (1) threats that result from data flows from user to server domain over (2) threats that result from data flows from the server domain to a third party domain, and over (3) threats that result from data flows within the server domain itself, e.g. threats from malicious insiders.

We consider threats of category (1) the most severe since they entail that users lose control over their personal data in the first place (see also Gürses et al. [16]). Threats of category (2) further undermine the users' control over their data by introducing more entities that potentially can misuse the data. Threats of category (3) include threats that, e.g., result from a database access of identifiable data within the system, and can be considered downstream threats from (1) and (2).

Furthermore, there can be application-specific prioritization criteria. Evidently, threats that concern more sensitive data should be prioritized over those that concern less sensitive data. It is, however, not trivial to classify data regarding their sensitivity in a consistent way that is also easy to apply for non-experts, and compliant with data protection regulations (see also our discussion in Section 6.2). We therefore use the classification by Roßnagel et al. [36] who developed *data protection classes* for the GDPR[3]. They define three such data protection classes[4], since their assignments have the goal of defining protection measures for the respective data. They also propose lists of data items for each category that make it better understandable for non-experts to assign data to the three classes.

4 IMPLEMENTATION

In the previous section we have identified data and code properties that determine privacy threats: identifiers and pseudo-identifiers in transactional and contextual data, database operations, data flows, API responses, role checks, the usage of certain libraries, e.g. for logging, and data sensitivities. In this section we show which extensions the PPG implements to enable the detection of potential privacy threats, and we present a set of queries to reveal such threats in the generated graph.

4.1 Exposing Privacy Properties in Transactional Data

To expose privacy properties of transactional data in the graph we use a taint-tracking approach with systematic labels. Two strategies to introduce such taints are possible: First, a naming convention can be defined to mark code entities containing personal data, e.g. adding a prefix to a variable or class name. The labels would then become visible in the graph as it reveals such declarations and their names. Second, the respective code entities can be annotated with code annotations or comments, e.g. `@Identifier`. While the former may introduce less overhead, the latter does not interfere with existing naming conventions and is less prone to unintentional mistakes, for example when renaming a variable. In our implementation we mainly use annotations to introduce taint labels and use comments where annotations are not available, for instance in the Go programming language.

As shown in the previous section, such labels are required for identifiers and pseudo-identifiers, while other properties, like database operations, can be detected without dedicated labels. Code entities that contain (pseudo-)identifiers are usually variables, e.g. a variable that holds a string from a user input and is sent to the server via an HTTP request. The PPG parses these annotations, adds them as nodes to the graph, and connects them via `ANNOTATIONS` edges. This is done via dedicated *passes*. Passes are modular extensions of the CloudPG that add special nodes and edges to the graph[5].

[2]We assume that this is the case because confidentiality is also covered by security threat modeling, which commonly uses the security goals *confidentiality*, *integrity*, and *availability*.

[3]The classification was developed in the AUDITOR project which is a candidate scheme for a GDPR certification.

[4]They also define a special class "3+" that is not relevant in the context of this paper.

[5]Further information regarding the development of custom passes is included in the open-source GitHub project of our implementation; we will submit the source code for the artifact review and will add a link here.

The PPG's pass for parsing comments and annotations introduces the following labels:

- **Identifier:** A code entity that holds data which represents an identifier. Every identifier is implicitly also handled as a pseudo-identifier.
- **PseudoIdentifier:** A code entity that holds data which can be linked to other pseudo-identifiers and to a person.
- **PrivacyLabel:** An annotation that holds an additional positive integer representing an ordinal scale to capture different sensitivity levels of private information.

The resulting nodes and edges can be used in combination with the data flow graph (DFG, see Section 2). The DFG connects declarations, statements, expressions and calls that exchange data and we can use it to track the flow of labeled entities through the code. We additionally use `@Anon<Placeholder>` labels that indicate the sanitization of a tainted code entity. These can be introduced analogously via annotations. We introduce them as dedicated *AnonLabel* nodes in the graph and `[:ANONYMIZES]` edges to the respective `DataLabel` they are supposed to anonymize. This way, we can easily exclude data flows that are processed by an anonymization function, effectively reducing false-positive results (see Listing 1).

To query the graph, we use Neo4j's query language Cypher [13]. The Cypher syntax uses round brackets to describe nodes and their types (*(:Expression)*), dashes for undirected edges (--), arrows for directed edges (-->), and square brackets in between dashes to describe edge types (*-[:ANONYMIZES]-*). The *-operator denotes a path of arbitrary length[6].

Listing 1: Cypher query which finds tainted data flows that are used in a function but which excludes paths that contain a node that anonymizes the taint. The *-operator denotes paths of arbitrary length.

```
MATCH p=(n:DataLabel)-[:LABELEDNODE]->()-[:DFG*]->(:
      ParamVariableDeclaration)<-[:PARAMETERS]-(:
      FunctionDeclaration)
WHERE all(al in nodes(p)
WHERE NOT (al)<-[:LABELEDNODE]-(:AnonLabel)-[:ANONYMIZES]->(n))
RETURN p
```

4.2 Exposing Privacy Properties in Contextual Data and Code Operations

To expose privacy-relevant code operations, like database interactions, we have implemented further passes which add dedicated nodes to the graph. These are database read and write operations, requests between services (e.g. HTTP requests), the use of dedicated libraries, e.g. for logging and cryptographic signatures, as well as role checks. Some of these properties would also be detectable without dedicated nodes in the graph, but such dedicated nodes represent the desired framework-specific properties in a more abstract form, allowing to write more generic and reusable queries.

4.2.1 Database Operations: To expose database operations in the graph, we detect the usage of language-specific calls for PostgreSQL and MongoDB. For instance, we create dedicated nodes for a PostgreSQL database and its tables, and add DFG edges from the database operation to the table in the direction of the data flow—depending on if it is a read or a write access.

4.2.2 HTTP Calls and Responses: To correctly identify an endpoint of an HTTP request in a static analysis, their endpoints need to be resolved: In our example application (described in the next section), for instance, a request's URL is not a static String but a variable that is initialized via an external configuration file. Consequently, the URL needs to be resolved during the analysis. We implement such a path-sensitive value resolver for variables that are initialized via external configuration files[7]. We have also added DFG edges to account for the logic of specific HTTP libraries. For instance, we add DFG edges from an HTTP endpoint to the variable that is initialized with the HTTP request's body. This way, we allow to easily track data flows across HTTP requests with generic queries.

4.2.3 Contextual data: Contextual data is metadata that can contain (pseudo-)identifiers depending on the communication protocol that is used. We use the HTTP protocol as an example, which contains pseudo-identifiers by design, i.e. IP addresses. Using the PPG, a simple query, similar to the one shown in Listing 2, can reveal all HTTP requests with their sources and targets. To cover more frameworks, only minor additions to the query are necessary.

4.3 Developing Queries

Having exposed the privacy properties in the graph, we can develop queries that detect potential privacy threats. In these queries we utilize the nodes and edges that the passes described above add to the graph, e.g. identifier nodes, database operation nodes, and respective DFG edges. We use Neo4j's query language Cypher to write the queries which can be executed manually via the Neo4j user interface, or automatically via the respective API. We aim at developing all queries in a way that reveals vulnerable paths in the graph with high precision and recall. At the same time, we aim at writing them in a generic, i.e. application-independent, fashion.

4.3.1 Identifiability and Linkability. As seen in Section 3, identifiability and linkability threats are highly dependent on transactional data properties. Since we use generic labels that taint personal data in code, queries for these threats are simple: We follow DFG edges of tainted elements and check if they are transmitted, e.g., via HTTP calls, see Listing 2.

Note that an *application*, as referred to in different queries, means one microservice, e.g. a frontend or a database. These are identified by the CloudPG via CI/CD definitions, e.g. GitHub workflow files[8]. The query in Listing 2 can therefore easily be limited to, e.g., the user frontend to limit the query to the identifiable data flows originating from the user domain.

Listing 2: Cypher query for HTTP data flows of (pseudo-)identifiers that are transmitted to an HTTP endpoint. The PPG implicitly treats identifiers also as pseudo-identifiers so querying for a pseudo-identifier also reveals identifiers.

```
MATCH p=(:PseudoIdentifier)--()-[:DFG*]->(:HttpEndpoint)
RETURN p
```

[6]See also https://neo4j.com/developer/cypher/querying/

[7]Note that the CloudPG already implemented such value resolvers, which, however, were improved for the PPG, e.g. to resolve URLs defined in separate configuration files.

[8]https://docs.github.com/en/actions/using-workflows

The query above also covers D2, detectable communication, since it reveals potentially observable data flows. Furthermore, it can be extended to find services that read personal data from more than one database, i.e. they can link data from multiple sources, as shown in Listing 3.

Note that we have not yet implemented dedicated passes and queries for detecting contextual data, i.e. the LINDDUN threats I4 and L4. Their development would require an in-depth analysis of communication protocols and the identifiers and pseudo-identifiers they use which we consider future work.

Listing 3: Query that finds linkability threats resulting from access to multiple databases that contain personal data. It first matches requests that include a database read operation. Then it limits the results to those database storages that contain (pseudo-)identifiers and to paths where at least two different such storages are used.

```
MATCH p=(a:Application)--(:HttpRequest)<-[:DFG*]-()--(:
        DatabaseOperation)<--(d1:DatabaseStorage),
(a:Application)--(:HttpRequest)<-[:DFG*]-()--(:DatabaseOperation)
        <--(d2:DatabaseStorage)
WHERE (:PseudoIdentifier)--()-[:DFG*]-()--(:DatabaseOperation)-->(
        d1:DatabaseStorage) AND (:PseudoIdentifier)--()-[:DFG*]-()
        --(:DatabaseOperation)-->(d2:DatabaseStorage)
RETURN p
```

As an example, consider a scenario where a microservice accesses both a database of pseudonymized patient health records as well as the database which holds the assignment of pseudonyms to real user names. Such a threat may result from design mistakes or forgotten legacy code, and could be detected with the query above.

We do not differentiate here between the flow of credentials and other data (see threats ID1, L1, D1, NR1), since we assume that all relevant data is labeled. Still, the queries can easily be modified to limit them, e.g., to a certain variable name. Similarly, the queries can be limited to flows that include a database operation (as required in ID6, L6, ID7, I7).

4.3.2 Non-Repudiation. As discussed in Section 3, in practice most non-repudiation threats stem from the usage of certain libraries, like logging and cryptographic libraries in combination with personal data. Listing 4 shows a query which detects operations that log tainted personal data. In our implementation the underlying pass is implemented for the Go zerolog[9] library, and is easily extendible for other libraries and languages. The respective pass creates abstract `LogOutput` nodes for the library calls, and adds data flows accordingly.

Listing 4: Query that finds logging calls which use identifiers.

```
MATCH p=(:Identifier)--()-[:DFG*]->()-[:ARGUMENTS]-()-[:CALL]-(:
        LogOutput)
RETURN p
```

Consider that many log outputs are sent to centralized logging services where they may be accessible to other roles. Such a threat could be uncovered using the query above.

[9]https://github.com/rs/zerolog

4.3.3 Detectability. Several detectability threats result from API responses that potentially leak personal information. In our implementation we cover these threats by detecting respective responses in HTTP APIs: we harmonize the naming for HTTP status codes across languages, and thus make them detectable in generic queries as shown in Listing 5.

Listing 5: Cypher query that finds tainted entities which are transmitted via an HTTP request and may trigger an HTTP status code of 404 or 409 as response. These responses could indicate that the transmitted data entity does not or already exists in the database.

```
MATCH p=(:PseudoIdentifier)--()-[:DFG*]->(h:HttpEndpoint)--(:
        FunctionDeclaration)-[:EOG*]->({name:"HttpStatus.CONFLICT"})
RETURN p
```

An example threat that could be detected via this query is a leaking account registration function for a potentially sensitive service like a crypto currency exchange.

4.3.4 Unawareness. To detect the unawareness threats U3 and U4 we need to check if users can read and modify their personal data. This detection requires three steps. First, we check which storages contain personal data by following tainted elements to database write operations. We then check if for those storages, read operations are implemented, i.e. the data can potentially be accessed afterwards. Third, it needs to be checked if these operations are restricted to the users whose data is stored in these storages. To perform the last step, we need to identify role checks not only in the application code, but potentially also mechanisms on other layers, e.g. access control lists on the cloud configuration layer. The PPG can potentially cover role checks on several such layers. The implementation, however, is outside the scope of this paper.

Listing 6 shows how the first two steps can be implemented in a Cypher query.

Listing 6: Cypher query that finds services that write personal data to database storages but do not have read access to that data.

```
MATCH p=(:PseudoIdentifier)--()-[:DFG*]->(h1:HttpRequest)-[:DFG
        *]->(ds:DatabaseStorage), (a:Application), (h2:HttpRequest)
WHERE NOT EXISTS ((:HttpRequest)-[:DFG*]->()-[:CALLS]-()<-[:DFG]-(
        ds:DatabaseStorage)) AND ((h1)--(a)) AND ((h2)--(a))
RETURN p
```

As discussed in Section 3, further unawareness threats concern missing transparency information about the handling of personal data. While the detection of such threats is difficult to cover with a static analysis tool, their mitigation can be supported: using queries as presented in Listing 2, a list of flows of personal data can be compiled and used as the basis of creating transparency notices.

4.3.5 Policy Non-Compliance. Policy non-compliance mainly targets disproportionate collection, processing, and storage. Since these threats may take various forms in code, many different queries can be developed to detect such threats. In the graph generated by our implementation, the data flow graph can be used to track, e.g., if personal data that is stored in a database is also retrieved, as shown in Listing 7.

Listing 7: Cypher query that finds database tables (in PostgreSQL) and collections (in MongoDB) which are written (pseudo-)identifiers to but never read from indicating a disproportionate storage threat.

```
MATCH p=(:PseudoIdentifier)--()-[:DFG*]->(:DatabaseOperation)-->(s
        :DatabaseStorage)
WHERE NOT EXISTS((:DatabaseOperation)<-[:DFG]-(s)) RETURN p
```

To detect disproportionate collection, we check if a (pseudo-)identifier flows to another service via an HTTP request, but does not flow through a processing node subsequently. The cpg uses different node types that can be interpreted as processing nodes. The query in Listing 8 uses a list of such nodes, e.g. If-Statements and certain types of Expressions, to reduce false-positive results.

Listing 8: Cypher query that finds flows of personal data to an HTTP endpoint where its further data flow does not cross a list of nodes that are considered processing steps like certain Expressions or If Statements.

```
MATCH p=(:PseudoIdentifier)--()-[:DFG*]->(h:HttpEndpoint)
WHERE NOT EXISTS { MATCH (h)-[:DFG*]->(i) WHERE (i:Expression) AND
        NOT (i:DeclaredReferenceExpression) AND (NOT (i:
        BinaryOperator) OR i.operatorCode <> "=") OR (i:IfStatement)
        OR (i:WhileStatment) }
RETURN p
```

4.3.6 Confidentiality. As mentioned in Section 3 we do not focus on confidentiality in this paper. Still, we have implemented a pass that exposes information about transport encryption of HTTP connections (see Listing 9), to make our approach better comparable to the LINDDUN example presented in the next section.

Listing 9: Cypher query that finds HTTP endpoints which have no TransportEncryption node attached i.e. they are not secured using TLS. TransportEncryption nodes are added for example if an HTTP request uses an "https" URL.

```
MATCH p=(h:HttpEndpoint) WHERE NOT ((h)--(:TransportEncryption))
        RETURN p
```

For more example use cases for the queries please see also the test cases we have developed (Section 5).

4.4 Developing Further Analysis Queries

Beyond the threat categories targeted above, our approach allows to analyze source code in various other ways to detect privacy problems. In the literature, different anti-patterns for privacy problems have been proposed, e.g. privacy smells [25] and privacy dark patterns [7]. Some of these can be detected as well using our approach: For example, The *data lakes* smell refers to databases that are used by many services for the storage of different personal data. This smell can be detected by the query shown in Listing 10.

Listing 10: Cypher query that finds database storages (e.g. tables) that are written to by more than one service.

```
MATCH p=(:DatabaseOperation)-[:DFG]->(d:DatabaseStorage) WITH d,
        COUNT(*) AS num
WHERE num > 1 RETURN d
```

The graph furthermore allows to validate custom policies. For example, an organization could define a policy regarding which services are allowed to handle which kind of sensitivity level as defined by the respective label, see Section 4.1. One can then check which services handle data of a certain sensitivity level as shown in Listing 11.

Listing 11: Cypher query that finds all services that accept data of a protection level > 1 via an HTTP endpoint.

```
MATCH p=(a:Application)--(:HttpEndpoint)<-[:DFG*]-()--(l:
        PrivacyLabel)
WHERE l.protectionlevel > 1 RETURN p
```

5 EVALUATION

In this section we present a privacy threat[10] test suite for evaluating the effectiveness of static analysis tools, e.g. CPG- or PDG-based tools. It is implemented as an open-source project[11]. The results of applying our own tool on this test suite are summarized in Section 5.2.

5.1 Test Suite Implementation

The test suite includes 21 implemented threats which cover all of the six LINDDUN GO categories. Additionally, there is one test case targeting a Disclosure threat. An overview with short descriptions is included in Appendix B.

At the time of writing the test suite implements each test case for both Python and Go, and we plan to extend it for more languages, e.g. Java and TypeScript. Furthermore, almost each test case has a validation test case which includes code that does *not* exhibit the respective threat. These test cases therefore validate the correct implementation of the tests and queries. In total, the test suite comprises 86 test cases.

In the implementation of the test cases, we have aimed at writing short, standard code that is largely similar to examples given, e.g., in the libraries' documentation.

The tests are structured as follows. Since privacy threats result from the transmission of personal data from client to server, each test case consists of at least one client and one server file. Furthermore, each test case includes a textual description of the concrete threat it implements, and the expected outcome that an analysis tool should provide. Also, every test case includes a configuration file that contains mock deployment information, such as databases and server URLs. Finally, every test case has a Neo4j-specific test implementation which executes the PPG on the files described above and compares the result to the expected outcomes.

For example, one Policy Non-Compliance test case implements a client-server interaction where a tainted and an untainted datum are sent to the server, but only the untainted one is further processed. A *README* file defines the test conditions, e.g. that the tainted datum is detected and that its data flow to the server is detected, while the actual testing logic that executes the test is included in the PPG's testing directory.

We consider this test suite an initial set that should be extended in future work. For example, more complicated scenarios including authentication and role checks should be created. Note that

[10]Note that we use the term *threat* here interchangeably with *weakness* or *vulnerability*.
[11]We will submit the source code for the artifact review and will add the link to the GitHub repository here.

we discuss further limitations of the testing library and the PPG implementation in the next section.

5.2 Results

Table 1 summarizes the results of applying the PPG to the test cases. The results are the same for both Go and Python tests, which is why they are generalized in the table. The results show that most implemented threats are detected, with an overall precision of 0.65, a recall of 0.91 and an accuracy (F-measure) of 0.76. Yet, there are also false positives and in some cases false negatives. In the 22 test cases, two false negatives (U4) resulted from missing differentiation of database operations: the PPG currently only differentiates between read and write operations, but does not specifically detect modifications to existing data or deletions as such.

Threats related to missing possibilities to erase or rectify personal data can therefore not be detected in the current state. Eleven false positives (L6/7, I6/7, NR4/5, D4/5, NC1/5) result from missing field sensitivity in the detection of HTTP request bodies: the PPG detects data entities that are transmitted in an HTTP request body, but when multiple entities are transmitted, they cannot be distinguished anymore at the receiver side. Thus, when tainted and untainted data is transmitted in one request and stored at the server-side, both are flagged as a potential threat.

The above-mentioned deficits are addressable with additional engineering effort. Adding the missing database operations, for example, requires to create classes for the create, read, update, and delete (CRUD) operations and to write passes for the relevant database technologies, like MongoDB and PostgreSQL. Then, the passes can create CRUD-nodes when the respective operations are detected and add them to the graph.

This example demonstrates an important balance to be made in the engineering of the proposed tool: On the one hand, it is theoretically possible to prevent any false positives and false negatives if passes for all existing libraries and programming languages are implemented. On the other, a reasonable balance has to be found between engineering effort of the tool and the number of false results that can be tolerated. In the context of threat modeling, the focus should be on reducing false negatives: It can be considered more important to detect all threats (i.e. to reduce false negatives) than to only detect valid ones (i.e. to reduce false positives), because it is better to sort through false positives than to risk implementing undetected weaknesses that may cause data breaches in the future.

5.3 Performance and Scalability

Generally, CPGs scale well with application size, since in contrast to symbolic execution or model checking, a CPG does not simulate representations of program state, but performs searches on structural information. This results in trading accuracy for more efficiency and avoids state explosion problems that other methods are susceptible to. Yamaguchi et al., for example, have analyzed the Linux kernel with their CPG in under 40 seconds [51].

To evaluate the performance of our implementation, we have conducted three benchmark tests[12] as described in the following.

[12]We conducted the benchmarks on a MacBook Pro with an Apple M1 chip, using the kotlinx-benchmark toolkit, see https://github.com/Kotlin/kotlinx-benchmark.

Table 1: Results of evaluating our tool on the test suite. ● = threat correctly detected, ○ = false positive, ○ = false negative

Test Case	Result
L3 Linkability of inbound data	●
L5 Linkability of shared data	●
L6 Linkability of stored data	● ○
L7 Linkability of retrieved data	● ○
I3 Identifying inbound data	●
I5 Identifying shared data	●
I6 Identifying stored data	● ○
I7 Identifying retrieved data	● ○
NR2 Non-repudiation of digital signature	●
NR2 Non-repudiation of logging	●
NR4 Non-reputable storage	● ○
NR5 Non-repudiation of retrieved data	● ○
D2 Detectable communication	●
D4 Detectable at storage	● ○
D5 Detectable at retrieval	● ○
U3 No access or portability	●
U4 No erasure	○
U4 No rectification	○
NC1 Disproport. collection	● ○
NC5 Disproport. storage (w/o retrieval)	● ○
NC5 Disproport. storage (w/o processing)	● ○
Disclosure	●
Precision p (● / (● + ○))	0.65
Recall r (● / (● + ○))	0.91
F-measure (2pr / (p+r))	0.76

5.3.1 Benchmarks of Test Cases. First, we have conducted benchmarks for all test cases, with 10 executions per benchmark. The average execution times ranged from 0.475 to 2.463 seconds per benchmark, i.e. including the analysis of the source code, building the graph, storing it into the Neo4j database, and querying the database. The average execution time was 0.632 seconds with a median of 0.583. The benchmark with the maximum execution time therefore is an outlier which can be explained by multiple HTTP clients and requests as well as multiple database operations included in this test case. The standard deviation values were consistently below 0.1 seconds, with two exceptions at 0.19 and 0.76.

5.3.2 Benchmarks of Increasing Code Size. Second, we conducted benchmarks with an increasing number of lines of code (LoC), between 100 and 1.000.000 LoC, to show how the implementation scales with real-world code sizes. The code was auto-generated by duplicating the encryption function of the Ceasar Cipher in Python with 10 LoC. To void heap overruns, multiple files were created with 100 LoC each. The code did not include PPG-specific code, like HTTP connections or database operations, and it did not include

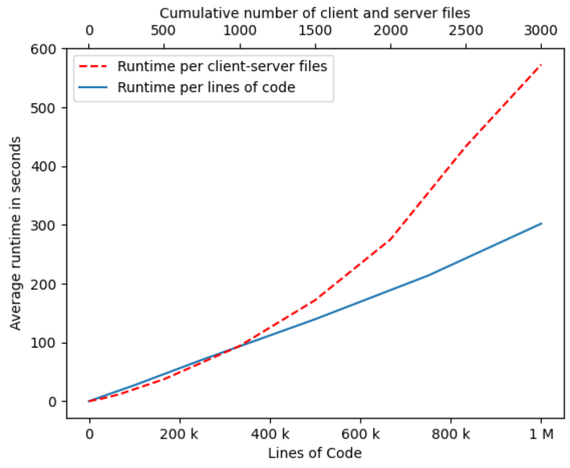

Figure 3: Performance benchmark results: The graph plots the execution times in seconds, and lines of code (lower x-axis) and number of client-server connections (upper x-axis). In the former case, depicted by the blue, solid line, a linear growth in execution times can be observed. This growth underlines that code property graphs only process structural information and avoid state explosion problems. In the latter case, the red, dotted line depicts the growth in execution times depending on the cumulative number of client-server connections. This growth can be explained with the quadratic growth in HTTP connections that are made in the respective tests, since all clients establish HTTP connections to all servers.

storing results to a database or querying it. Figure 3 presents the results of these benchmarks (blue solid line).

5.3.3 Benchmarks of Increasing Client-Server Connections. Finally, we conducted a benchmark that tests a more resource-intensive scenario. In this scenario, the analyzed code contains an increasing number of clients and servers, i.e. between a total of 2 and 3.000 clients and servers, where all clients create HTTP connections to all servers. These were generated by duplicating the client and server files from the D2 Detectable Communication test case. Again, this benchmark did not include storing or querying the results. Figure 3 shows the execution times.

Overall, the benchmarks demonstrate that performance primarily depends on specific passes introduced in the PPG, rather than storage and retrieval of results or the pure parsing of code. As Figure 3 shows, even with thousands of clients and servers, the execution time still ranges within minutes, which we assume to be a practical execution time in most real-world scenarios.

6 DISCUSSION

We first discuss limitations regarding completeness, manual effort, bias, and other limitations and then discuss our approach in the context of agile development.

6.1 Completeness

Overall, our approach as presented in Section 3 addresses 24 out of 34 LINDDUN GO threats (see Appendix A), of which 18 are addressed in the current implementation of the PPG. Threats that cannot meaningfully be covered in a static analysis include, e.g., side-channel-based and legal threats. We would argue that these threats do not have to be assessed regularly: In an agile development process with short development sprints, for example, the lawful ground of processing (NC2) for the data collection can be assessed and documented once before development starts.

Furthermore, not every threat that is addressed is fully covered by our approach. A disproportionate processing threat (NC3), for instance, cannot be detected if the respective datum is processed, but the processing is not necessary for the actual purpose. We hope to advance the implementation to cover as many threats as possible in an open-source community effort. Future work needs to show to what extent the threats can be covered by the PPG in practice.

6.2 Manual Effort

Based on the cpg library, we automate the generation of a data flow diagram as well as the detection of threats as discussed above. Still, the PPG may also incur overhead in the following steps: (1) Developing new queries and passes, for instance for frameworks we do not cover yet, (2) adding labels to the code to expose transactional privacy properties, (3) setting up the tool, (4) executing the tool, and (5) analyzing the results.

We expect the overhead introduced in the development (1) to decrease as the implementation and catalog of queries improve. Regarding manual labeling (2), we would argue that a mapping from data to privacy labels should not be created by developers but should be created by legal experts. In standards and in the literature, similar lists already exist, for example in the ISO 29100 [19] standard, the HIPAA privacy rule [29], and proposed in [36]. While organization-specific mappings may be more appropriate, developers can use these existing mappings even if no dedicated mapping is available.

While the set-up (3) is a one-time activity, the execution (4) can be automated, e.g. integrating it into a continuous build chain.

Finally, when applying the PPG (5) to a large code base that has many transmissions which include personal data, the results may become too large for manual analysis. Note that this is a common problem in threat modeling tools. Considering our test cases presented in Section 5, a typical test case results in a graph with 100-300 nodes and 400-1200 edges, which would greatly increase in large applications. Yet, the query results for the test cases typically result in approximately 10 nodes and 20 edges—and we do not expect these to increase significantly with the general code size.

In future work, we still plan to add filters for the results, e.g. for sensitivity classes and threat types, as well as a more sophisticated automatic prioritization. Overall, we expect the benefits of generating privacy threats semi-automatically to outweigh the costs of adding annotations to code, since the former does not lie in the expertise of developers, while the latter does.

6.3 Bias and Generality

In general, a bias in our implementations of the PPG and the test suite cannot be ruled out, which may impact their real-world applicability. In the following, we therefore discuss the real-world applicability, i.e. the generality, of our implementations.

6.3.1 Linkability and Identifiability. The detection of linkability and identifiability threats in our approach is based on a taint-tracking mechanism. Since this mechanism works by detecting labels attached as comments or annotations in code, we would argue that the detection of these threats is independent from the implementation of test cases. We therefore assume a high generality of our implementation and test cases in this category of threats.

6.3.2 Non-Repudiation. Non-repudiation threats largely depend on the usage of certain libraries and their specific methods. Here, our implementation relies on the detection of these method calls—the PPG and the testing library are therefore currently limited to a small set of these libraries. The generality of our approach, however, is not impacted by this limitation.

6.3.3 Detectability. Detecting threats of this category partly depends on the correct identification of data transmissions. This includes, for instance, linking the tainted datum correctly to an HTTP request body, linking that body correctly to the receiving function on the server side, etc. Therefore, our implementations are biased to some extent towards HTTP requests and the way they are written in the languages we cover. Note that this detection also depends on the correct functionality of the underlying *cpg*. Also, the detection of API responses is important for the detection of some detectability threats. Their potential for bias is similar as described above, since we focus on the detection of HTTP status codes in API responses.

6.3.4 Unawareness. Threats in this category concern missing access, i.e. read or write access, by users to their data. While we do not see reasons for considerable bias in this category, we do make certain assumptions in our implementation, for instance that such an access is implemented via a direct HTTP request that triggers a database access. Yet, access requests could also be implemented via other channels, like email. Also, it may be reasonable that certain data is not modifiable, e.g. financial data may need to be retained for tax reasons.

6.3.5 Policy Non-Compliance. Threats around disproportionate collection, processing, and storage can take many forms: a datum may be collected as part of a larger object, but may not be processed; a datum may be collected, stored, and retrieved, but not meaningfully be processed; or a datum may be collected and forwarded to a third party where it is stored without a purpose. Our test suite currently only includes a small set of tests in this context, which cannot comprehensively cover policy non-compliance as a category. Therefore, we acknowledge the our implementations are somewhat biased in this category, and the test cases should be extended especially in this category in future work.

6.4 Other Limitations

One general limitation of our approach is that the correct functionality of queries relies on the correct implementation of the underlying passes. Bugs in the implementation or the underlying CPG library could therefore lead to false negative results and mislead users into thinking that no threats exist.

A further limitation is that human errors can happen in the assignment of labels, especially when they are forgotten, which can lead to wrongfully trusted data flows. However, such errors can still be caught when inspecting the graph manually. Label assignments should therefore be checked by quality assurance measures, for instance code reviews.

A general limitation of the proposed test suite is that it is not clear to what degree it captures real-world threat implementations. The proposed test cases may be rather simple threat implementations, while real-world threats may be more hidden and more difficult to detect, for instance when data is pre-processed or aggregated.

6.5 Agile Applicability

In Section 1 we have highlighted the problem of applying threat modeling in agile development processes. To apply our approach, some initialization tasks are required first to set up the tooling and to adapt queries to the system. This approach, however, is easier to apply than manual threat modeling, since the set-up phase can be integrated into one of the early development sprints, while the analysis of the results can be integrated as single tasks.

One disadvantage of using static code analysis for privacy threat detection is that threats are detected only after they have already been implemented. This limitation implies a trade-off between the lower time consumption of our approach on the one hand, and the earliest possible identification of threats using a manual analysis on the other. In an agile environment, we expect an automated ex-post analysis still to be more efficient and effective than infrequent manual threat modeling sessions, since agile development aims at delivering working software early and frequently [12]—allowing our approach to be applied early and frequently as well.

Alternatively, an agile development team may start with a manually created data flow diagram and LINDDUN analysis, and move to our approach later as the code base grows. Note, however, that agile projects do not always develop a detailed architecture before starting with the implementation. The two approaches could also be combined, using our semi-automated approach frequently as a form of regression test, while conducting a manual analysis only for certain milestones of a project, e.g. before certain quality gates.

Overall, static graph-based analysis can detect many privacy threats, assuming that labels have been introduced correctly. Especially threats that result from code operations can be detected well in such an approach because code property graphs represent an abstraction of code that allows detailed analysis, but abstracts away unnecessary details. Thus, a graph-based analysis is well usable for automatic analysis as well as for manual inspection, while threats related to external data, side-channel information, as well as legal threats, should be covered differently.

7 RELATED WORK

7.1 Privacy Threat Modeling

The state-of-the-art in privacy threat modeling is largely focused on the LINDDUN methodology [9]. LINDDUN GO presents an updated version of this methodology with consolidated threat descriptions and improved usability, making it more easily applicable for non-experts [49]. Alternative proposals include, for instance, different protection goals to be used and other approaches to eliciting privacy requirements [17, 20, 28, 39].

In the LINDDUN extension by Sion et al., threats are elicited based on interactions between elements rather than data flow diagram elements themselves. They show that an interaction-based threat modeling results in less false-positives and less false-negatives. We follow a similar approach, since we do not identify threats based on isolated elements, like databases, but we identify threats based on data flows, and their data properties and code operations, which effectively represent interactions between entities.

The overall purpose of threat modeling and risk assessment can also be understood as the elicitation of requirements for the system under consideration. In this context, our approach can be seen as a risk-based engineering approach. Goal-based engineering approaches are proposed as an alternative to risk-based ones, see e.g. PRIPARE [28]. Various other works have addressed privacy engineering from a goal perspective [15, 16, 44].

Shapiro [39] classifies privacy engineering methods into the two dimensions analytical and instrumental, as well as programmatic and technical. He points out that methods at the analytical-technical intersection are missing. We see our approach as such a technical-analytical method, since it applies expert knowledge (technical) in the examination of source code (analytical).

7.2 Automated Threat Modeling

The automation of threat modeling in general has been investigated for a long time, e.g. for network security threats [30]. For cloud systems, many approaches use publicly available information about vulnerabilities to automatically detect security threats [1, 8].

Markovic et al. [27] propose an architecture for an automated privacy assessment service for IoT systems. They also employ graph-based analysis, but they analyze IoT systems using execution traces, rather than using source code analysis.

Berghe et al. [47] propose modified data flow diagrams that include detailed security properties to enable their automatic verification. Their approach is also more formalized, which may be an approach that is not applicable as easily to express privacy properties, but can be a topic of future work, for instance to indicate linkable attributes. A similar approach is followed by Berger et al. [6]. In contrast, we generate data flow diagrams automatically, enhanced with code-level privacy properties, to allow for an automatic detection of privacy threats. Consider that the creation of such a diagram can be very complex and time-intensive if the application is large and/or the architecture changes frequently.

In general, code property graphs have been used mainly for security analysis, for example by Yamaguchi et al. [51]. Simeonovski et al. [41] also use tainted property graphs to model attacks on the global internet infrastructure. Graph-based security analysis has also been combined with artificial intelligence [50]. To the best of our knowledge, the application to privacy threat modeling, however, has not been proposed before.

7.3 Other Automated Modeling Approaches

In the area of security and safety engineering, similar approaches have been proposed, e.g. by Thiagarajan et al. [45]. They use the Architecture Analysis and Design Language to create an architecture model which is then translated into a graph structure that resembles a program dependence graph (see also Section 2.1). The model can then also be queried via a query language.

Other approaches include, for example, security and safety analysis models like STAMP, STPA [26], and STPA-Sec [53], which also have been applied to privacy [39, 40].

7.4 Data Flow Analysis

In a static analysis, various types of sensitivities can improve the quality of analysis methods on that graph (see [38]). FlowDroid [2] is a static analysis method for Android applications that claims to be object-, field-, context-, and flow-sensitive.

The cpg (see Section 2), which serves as basis for our implementation, is inter-procedural and can track data-flows from object instantiations. The graph is by design not object- or context-sensitive, which would be desirable to reduce false-positives generated by the PPG. It is, however, field-sensitive and flow-sensitive. The former allows us to differentiate between different fields of an object and determine whether a field holds privacy-related data or not. The latter considers the sequence of statements and possible control-flow structures, and thus allows us to consider only privacy labels that can be valid in a possible control-flow path.

The annotation approach we propose to track personal data through the graph is a taint-tracking mechanism as it has been used in other tools as well. With flow-sensitivity, it is possible to then track the labels throughout the graph. For labeling data flows, some propose domain-specific labels [33], while others use generic taint labels (see [3]). Taint-tracking is, e.g., often used in the analysis of mobile applications, especially for security [11, 37, 46].

Slavin et al. [43] also propose a semi-automated tool that is able to check mobile apps for consistency with a privacy policy. It is based on matching names in the policy and in API methods. In our approach, we use labels to be introduced in code, rather than a naming schema which, however, could also be used.

Apart from static analysis, various works have proposed real-time data flow tracking approaches for cloud systems, e.g. on the cloud management plane [24] or using VM introspection [4], which could be complementary to our approach.

7.5 Labels for Tracking Personal Data

Similar to our approach, Hjerppe et al. [18] propose code annotations to document code entities relevant to the protection of personal data, and to detect violations using static analysis tools. They propose three annotations, *@PersonalData*, *@PersonalDataHandler*, and *@PersonalDataEndpoint*, to mark personal data or their context in the code, and they implement their approach for Java. In contrast, we use annotations that indicate the quality of the personal identifier. We also do not annotate handlers or endpoints as such, but analyze the flow of annotated entities. In future work, however, we will consider further annotations, e.g. to allow for the compliance verification of specific policies.

8 CONCLUSIONS

Privacy threat modeling requires the creation of a data flow diagram, frequent updates, and the evaluation of many potential threats in time-consuming brainstorming sessions.

To automate this process partly, we have first presented an analysis of privacy threats, operationalizing them for their detection in static analysis tools. We have also proposed labels for privacy properties and sensitivity classes, and have presented an implementation of a code annotation approach to introduce these labels into a code property graph. We have then presented a benchmark privacy threat test suite for the evaluation of static code analysis tools, and applied the PPG to this test suite.

8.1 Benefits and Potentials

One central benefit of our approach is the simplification of privacy threat modeling for non-experts: It turns the threat modeling process from a manual brainstorming that requires considerable experience into a semi-automatic detection that only requires users to elicit threats from a given selection. This way, the expertise behind the semi-automatic detection is decoupled from its application. A further central benefit is the reduced time-consumption of the threat modeling process once the set-up is completed.

Our approach is furthermore language-independent which solves another hurdle for threat modelers who may struggle with detecting threats in code of different programming languages.

Our approach and implementation offer various potentials for further development and exploitation. For example, our taint-tracking mechanism can be extended with different labels to track other kinds of data properties, e.g. security and compliance properties. It also allows for a combined security and privacy threat modeling. Such a combined analysis can result in reduced overall effort, since the two areas overlap regarding data protection measures, like access control and encryption, and repudiation properties.

It has furthermore the potential to improve the understanding of third-party dependencies of an application, assuming their source code is available. Their assessment is a common, time-consuming activity foreseen in several secure development lifecycles [31, 32].

Our approach and implementation also has advantages over existing tools for the creation and analysis of data flow diagrams. The Microsoft Threat Modeling Tool, e.g., allows to manually create such a diagram and automatically generate a list of standard threats, e.g. based on the specified protocol. In comparison, our tool automates the creation of the data flow diagram, and detects threats based on the actually used technologies rather than manually specified information—which could quickly be outdated.

Finally, consider that large organizations often employ a central department or external consultants who are responsible for supplying security and privacy expertise to development teams. For them, our approach is beneficial since they can develop reusable queries to be tested across applications, effectively standardizing the privacy level of applications.

8.2 Future Work

All in all, our approach has the potential to detect most LINDDUN privacy threats, as well as other potential problems like privacy smells and anti-patterns, and to apply it frequently throughout the development and maintenance process. Since it allows to standardize and share knowledge about privacy threats on a technical level, we envision our approach being used and extended collaboratively within the public community, creating reusable detection queries and implementations that reduce threat modeling effort in software development.

In future work, we plan to extend our implementation with passes for more frameworks and technologies, and develop more queries and test cases. In particular, we want to add detection of role-based access mechanisms across several layers, including software and cloud infrastructure layer. As seen in the evaluation, the PPG should also be improved with regards to field sensitivity in HTTP requests and more granular detection of database operations. Some practical advancements of our implementation will include the possibility to manually exclude certain threats from future executions, e.g. when a risk has been accepted, and the ability to integrate it into a CI/CD pipeline. To improve the degree of automation, also the creation of labels could be automated, e.g. via natural language processing. Our approach also opens up the possibility to define restrictive policies for the flow of personal data which could be complementary to identifying risky data flows. To enable the verification of such policies, we consider adding a constraint solver that can evaluate such policies. Currently, we are using the existing cpg edges, especially its data flow graph, to detect privacy threats—it could, however, be beneficial to create a dedicated privacy graph that includes and connects privacy-related operations. This would further simplify the queries required to detect privacy threats. As mentioned in Section 2, we will also explore the addition of a Program Dependence Graph (PDG). PDGs contain data flow edges as well as control dependence edges which can improve the sensitivity to indirect data flows, i.e. when personal data is not transmitted directly, but when personal data leaks as the result of another, conditional data flow. Further exploitation strategies of our operationalization and implementation include, for example, the automatic generation of privacy notices for users.

ACKNOWLEDGMENTS

This work was partly funded by the European Union Horizon 2020 project MEDINA, Grant No. 952633. We also thank Andreas Binder for his valuable support in developing the test cases.

REFERENCES

[1] Seongmo An, Taehoon Eom, Jong Sou Park, Jin Bum Hong, Armstrong Nhlabatsi, Noora Fetais, Khaled M Khan, and Dong Seong Kim. 2019. Cloudsafe: A tool for an automated security analysis for cloud computing. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 602–609.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.

[3] Jean Bacon, David Eyers, Thomas FJ-M Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter Pietzuch. 2014. Information flow control for secure cloud computing. *IEEE Transactions on Network and Service Management* 11, 1 (2014), 76–89.

[4] Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E Porter. 2014. CloudFlow: Cloud-wide policy enforcement using fast VM introspection. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, 159–164.

[5] Christian Banse, Immanuel Kunz, Angelika Schneider, and Konrad Weiss. 2021. Cloud Property Graph: Connecting Cloud Security Assessments with Static Code

Analysis. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 13–19. https://doi.org/10.1109/CLOUD53861.2021.00014

[6] Bernhard J Berger, Karsten Sohr, and Rainer Koschke. 2016. Automatically extracting threats from extended data flow diagrams. In *International Symposium on Engineering Secure Software and Systems*. Springer, 56–71.

[7] Christoph Bösch, Benjamin Erb, Frank Kargl, Henning Kopp, and Stefan Pfattheicher. 2016. Tales from the Dark Side: Privacy Dark Strategies and Privacy Dark Patterns. *Proc. Priv. Enhancing Technol.* 2016, 4 (2016), 237–254.

[8] Chi-An Chih and Yu-Lun Huang. 2015. An adjustable risk assessment method for a cloud system. In *IEEE International Conference on Software Quality, Reliability and Security-Companion*. IEEE, 115–120.

[9] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. 2011. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering* 16, 1 (2011), 3–32.

[10] David Baker Effendi, Fabian Yamaguchi, and Jaden Jung. 2021. Plume. Retrieved May 18, 2022 from https://github.com/plume-oss/plume

[11] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[12] Martin Fowler, Jim Highsmith, et al. 2001. The agile manifesto. *Software Development* 9, 8 (2001), 28–35.

[13] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*. 1433–1445.

[14] Rafa Galvez and Seda Gurses. 2018. The odyssey: Modeling privacy threats in a brave new world. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 87–94.

[15] Seda Gürses, Carmela Troncoso, and Claudia Diaz. 2011. Engineering privacy by design. *Computers, Privacy & Data Protection* 14, 3 (2011), 25.

[16] Seda Gürses, Carmela Troncoso, and Claudia Diaz. 2015. Engineering privacy by design reloaded. In *Amsterdam Privacy Conference*. 1–21.

[17] Marit Hansen, Meiko Jensen, and Martin Rost. 2015. Protection goals for privacy engineering. In *2015 IEEE Security and Privacy Workshops*. IEEE, 159–166.

[18] Kalle Hjerppe, Jukka Ruohonen, and Ville Leppänen. 2019. Annotation-based static analysis for personal data protection. In *IFIP International Summer School on Privacy and Identity Management*. Springer, 343–358.

[19] ISO/IEC 29100:2011 2011. *Information technology - Security techniques - Privacy framework*. Standard. International Organization for Standardization, Geneva, CH.

[20] Christos Kalloniatis, Evangelia Kavakli, and Stefanos Gritzalis. 2008. Addressing privacy requirements in system design: the PriS method. *Requirements Engineering* 13, 3 (2008), 241–255.

[21] Wim Keirsgieter. 2019. Graft. Retrieved May 18, 2022 from https://github.com/wimkeir/graft

[22] Hossein Keramati and Seyed-Hassan Mirian-Hosseinabadi. 2008. Integrating software development security activities with agile methodologies. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*. IEEE, 749–754.

[23] Blagovesta Kostova, Seda Gürses, and Carmela Troncoso. 2020. Privacy Engineering Meets Software Engineering. On the Challenges of Engineering Privacy By Design. *arXiv preprint arXiv:2007.08613* (2020).

[24] Immanuel Kunz, Valentina Casola, Angelika Schneider, Christian Banse, and Julian Schütte. 2020. Towards Tracking Data Flows in Cloud Architectures. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 445–452.

[25] Immanuel Kunz, Angelika Schneider, and Christian Banse. 2020. Privacy Smells: Detecting Privacy Problems in Cloud Architectures. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 1324–1331.

[26] NG Leveson and Engineeringa SaferWorld. 2011. Systems Thinking Applied to Safety.

[27] Milan Markovic, Waqar Asif, David Corsar, Naomi Jacobs, Peter Edwards, Muttukrishnan Rajarajan, and Caitlin Cottrill. 2018. Towards automated privacy risk assessments in IoT systems. In *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things*. 15–18.

[28] Nicolás Notario, Alberto Crespo, Yod-Samuel Martín, Jose M Del Alamo, Daniel Le Métayer, Thibaud Antignac, Antonio Kung, Inga Kroener, and David Wright. 2015. PRIPARE: integrating privacy best practices into a privacy engineering methodology. In *2015 IEEE Security and Privacy Workshops*. IEEE, 151–158.

[29] HHS Office for Civil Rights. 2002. Standards for privacy of individually identifiable health information. Final rule. *Federal register* 67, 157 (2002), 53181–53273.

[30] Xinming Ou, Sudhakar Govindavajhala, Andrew W Appel, et al. 2005. MulVAL: A Logic-based Network Security Analyzer.. In *USENIX security symposium*, Vol. 8. Baltimore, MD, 113–128.

[31] OWASP. 2006. OWASP CLASP v1.2. *https://owasp.org/www-pdf-archive/Us_owasp-clasp-v12-for-print-lulu.pdf* (2006). https://owasp.org/www-pdf-archive/Us_owasp-clasp-v12-for-print-lulu.pdf

[32] OWASP. 2020. OWASP SAMM v2.0. *https://github.com/OWASP/samm/blob/master/Supporting%20Resources/v2.0/OWASP-SAMM-v2.0.pdf* (2020). https://github.com/OWASP/samm/blob/master/Supporting%20Resources/v2.0/OWASP-SAMM-v2.0.pdf

[33] T Pasquier, B Shand, and J Bacon. 2013. Information flow control for a medical web portal. *e-Society, IADIS* (2013).

[34] Andreas Pfitzmann and Marit Hansen. 2010. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf. http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf v0.34.

[35] Bruce Potter. 2009. Microsoft SDL threat modelling tool. *Network Security* 2009, 1 (2009), 15–18.

[36] A. Roßnagel, A. Sunyaev, S. Lins, N. Maier, and H. Teigeler. 2019. *AUDITOR Concept of Protection Categories – Draft version 0.9*. Technical Report. Karlsruher Institut für Technologie (KIT). https://doi.org/10.5445/IR/1000092273

[37] Julian Schütte, Alexander Küchler, and Dennis Titze. 2017. Practical Application-Level Dynamic Taint Analysis of Android Apps. In *2017 IEEE Trustcom/BigDataSE/ICESS*. 17–24. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.215

[38] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[39] Stuart S Shapiro. 2016. Privacy risk analysis based on system control structures: Adapting system-theoretic process analysis for privacy engineering. In *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 17–24.

[40] Stuart S Shapiro. 2017. Addressing Early Life Cycle Privacy Risk. In *2017 International Workshop on Privacy Engineering-IWPE*, Vol. 17.

[41] Milivoj Simeonovski, Giancarlo Pellegrino, Christian Rossow, and Michael Backes. 2017. Who controls the internet? analyzing global threats using property graph traversals. In *Proceedings of the 26th International Conference on World Wide Web*. 647–656.

[42] Laurens Sion, Dimitri Van Landuyt, and Wouter Joosen. 2020. The Never-Ending Story: On the Need for Continuous Privacy Impact Assessment. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 314–317.

[43] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*. 25–36.

[44] Sarah Spiekermann and Lorrie Faith Cranor. 2008. Engineering privacy. *IEEE Transactions on software engineering* 35, 1 (2008), 67–82.

[45] Hariharan Thiagarajan, John Hatcliff, et al. 2021. Awas: AADL information flow and error propagation analysis framework. *Innovations in Systems and Software Engineering* (2021), 1–20.

[46] Dennis Titze and Julian Schütte. 2015. Apparecium: Revealing data flows in android applications. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE, 579–586.

[47] Alexander van Den Berghe, Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2017. A model for provably secure software design. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. IEEE, 3–9.

[48] Konrad Weiss, Christian Banse, and Maximilian Kaul. 2019. Code Property Graph. Retrieved May 18, 2022 from https://github.com/Fraunhofer-AISEC/cpg

[49] Kim Wuyts, Laurens Sion, and Wouter Joosen. 2020. LINDDUN GO: A Lightweight Approach to Privacy Threat Modeling. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 302–309.

[50] Wang Xiaomeng, Zhang Tao, Wu Runpu, Xin Wei, and Hou Changyu. 2018. CPGVA: Code property graph based vulnerability analysis by deep learning. In *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*. IEEE, 184–188.

[51] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[52] Fabian Yamaguchi, Markus Lottmann, Niko Schmidt, Michael Pollmeier, Suchakra Sharma, and Claudiu-Vlad Ursache. 2019. Joern Code Property Graph. Retrieved May 18, 2022 from https://cpg.joern.io

[53] William Young and Nancy G Leveson. 2014. An integrated approach to safety and security based on systems theory. *Commun. ACM* 57, 2 (2014), 31–35.

A OPERATIONALIZATION OF LINDDUN GO

Table 2: An overview of LINDDUN GO threats and our operationalization described in Section 3.

ID	Name	Technical Description	Required Properties
L1	Linkability of credentials	Data including a reused credential is sent from user to server	(Pseudo-)Identifiers (in credentials)
L2	Linkability of user actions	Behavioral information makes data flows linkable	*out of scope*
L3	Linkability of inbound data	Data is sent from user to server where it can be linked with other existing data	(Pseudo-)Identifiers, DB read operations
L4	Linkability of context	Behavioral information makes data flows linkable	Identifiers (in contextual data, e.g. communication protocol)
L5	Linkability of shared data	Data including (pseudo-)identifiers is sent from server to third party	(Pseudo-)identifiers
L6	Linkability of stored data	Data including (pseudo-)identifiers is sent from user to server and stored in a DB	(Pseudo-)identifiers (in DB write op)
L7	Linkability of retrieved data	Data including (pseudo-)identifiers is sent from server to third party	(Pseudo-)identifiers (in DB read op)
ID1	Identifying credentials	The use of (non-anonymous) credentials allows identification of the user	Identifiers (in credentials)
ID2	Actions identify user	The user is identified via his requests to the system	*out of scope*
ID3	Identifying inbound data	The data sent to the system can be used to identify the user	Identifiers (in transactional data)
ID4	Identifying context	The contextual information of the communication can be identified	(in contextual data, e.g. communication protocol)
ID5	Identifying shared data	Communicated content can be used by receiving party to identify individuals	Identifiers (linked with stored data)
ID6	Identifying stored data	Personal data being stored can be identified	Identifiers (in DB write operations)
ID7	Identifying retrieved data	Personal data being retrieved from persistent storage can be used to identify the data subject	Identifiers (in DB read operations)
NR1	Credentials non-repudiation	Person cannot deny having authenticated to a service	Identifiers (in credentials), logs, digital signatures
NR2	Non-repudiation of sending	The user cannot deny having sent a message	Logs, digital signatures
NR3	Non-repudiation of receipt	The user cannot deny having received a message	*out of scope*
NR4	Non-reputable storage	The data in storage cannot be denied	Logs, digital signatures (in DB write operations)
NR5	Non-repudiation retrieved data	The retrieved data contains undeniable information	Logs, digital signatures (in DB read operations)
D1	Detectable credentials	Response of a request allows detection of existence of a user	API responses (e.g. HTTP 404)
D2	Detectable communication	Communication between the user and the server can be observed	Data flows between user and server
D3	Detectable outliers	Behavioral information makes data flows linkable	*out of scope*
D4	Detectable at storage	DB write operation returns “user exists”	API responses, DB operations
D5	Detectable at retrieval	DB read operation returns empty set or identifier	API responses, DB operations
U1	No transparency	The data subject is insufficiently informed about the collection and processing of personal data	*out of scope*
U2	No user-friendly privacy control	The system does not provide user-friendly privacy control	*out of scope*
U3	No access or portability	The data subject does not have access to or is not able to port their personal data	DB read operations, user role
U4	No erasure or rectification	The data subject cannot request erasure or rectification of personal data	DB write operations, user role

An overview of LINDDUN GO threats and our operationalization described in Section 3 (cont.).

ID	Name	Description	Required Properties
U5	Insufficient consent support	Data subject consents are not properly taken into account by the relevant processes	*out of scope*
NC1	Disproportionate collection	More personal data are being collected than required for the purpose	(Pseudo-)identifiers
NC2	Unlawful processing	There is no lawful ground for the collection, processing, and storage of PD	*out of scope*
NC3	Disproportionate processing	More personal data are being processed than required for the purpose	*out of scope*
NC4	Automated decision making	A decision is made solely on automated processing of PD which significantly affects the data subject	*out of scope*
NC5	Disproportionate storage	More personal data are being stored than required for the purpose	(Pseudo-)identifiers, DB read operations, DB write operations

B TEST CASES

Table 3: Threat descriptions of the test cases implemented in the test suite.

LINDDUN GO Threat	Description
L3 Linkability of inbound data	A pseudo-identifier is sent to a server where it can be linked to other pseudo-identifiable data (which may be submitted via the same request).
L5 Linkability of shared data	A pseudo-identifier is sent to a server where it can be linked to other pseudo-identifiable data (which may be submitted via the same request). The server furthermore shares the data with a third party (where it also may be linked to other pseudo-identifiable data).
L6 Linkability of stored data	A pseudo-identifier is sent to a server where it is stored and can be linked to other pseudo-identifiable data (which may be submitted via the same request).
L7 Linkability of retrieved data	A pseudo-identifier is sent to a server which stores it in a database. Another client can access the datum via a GET request.
I3 Identifying inbound data	An identifier is sent to a server where it can be linked to other pseudo-identifiable data (which may be submitted via the same request).
I5 Identifying shared data	An identifier is sent to a server where it can be linked to other pseudo-identifiable data (which may be submitted via the same request). The server furthermore shares the data with a third party (where it also may be linked to other pseudo-identifiable data).
I6 Identifying stored data	An identifier is sent to a server where it is stored and can be linked to other pseudo-identifiable data (which may be submitted via the same request).
I7 Identifying retrieved data	An identifier is sent to a server which stores it in a database. Another client can access the datum via a GET request.
NR2 Non-repudiation of digital signature	A cryptographically signed, i.e. non-reputable, message including personal data is sent from client to server.
NR2 Non-repudiation of logging	A message including personal data is sent from client to server where it is logged.
NR4 Non-reputable storage	A cryptographically signed, i.e. non-reputable, message including personal data is sent from client to server where it is stored in a database.
NR5 Non-repudiation of retrieved data	A cryptographically signed, i.e. non-reputable, message including personal data is sent from client to server where it is stored in a database and is retrieved by another client.
D2 Detectable communication	A (pseudo-)identifier is sent from client to server. This may be observed by other network participants, which then know that the person is using the service provided by the server.
D4 Detectable at storage	The server offers an API to a database which leaks information about personal data it holds: When the client tries to store data, the server may respond by indicating a conflict.
D5 Detectable at retrieval	The server offers an API to a database which leaks information about personal data it holds: When the client tries to access data, the server may respond by a not found message.
U3 No access or portability	A (pseudo-)identifier is sent from client to server where it is stored in a database and cannot be accessed by the client again.

Threat descriptions of the test cases implemented in the test suite. (cont.).

LINDDUN GO Threat	Description
U4 No erasure	A (pseudo-)identifier is sent from client to server where it is stored in a database and cannot be deleted again by the client.
U4 No rectification	A (pseudo-)identifier is sent from client to server where it is stored in a database and cannot be modified again by the client.
NC1 Disproport. collection	A message including personal data is sent from client to server where it is not further processed in a meaningful way.
NC5 Disproport. storage (w/o retrieval)	A (pseudo-)identifier is sent from client to server where it is stored but not retrieved afterwards.
NC5 Disproport. storage (w/o processing)	A (pseudo-)identifier is sent from client to server where it is stored and retrieved, but not processed afterwards.
Disclosure	A (pseudo-)identifier is sent from client to server without transport encryption.