Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis

Katja Tuma*, Musard Balliu†, Riccardo Scandariato*
* Chalmers | Gothenburg University, Gothenburg, Sweden, [katja.tuma, riccardo.scandariato]@cse.gu.se
† KTH Royal Institute of Technology, Stockholm, Sweden, musard@kth.se

Abstract—**This paper presents a practical and formal approach to analyze security-centric information flow policies at the level of the design model. Specifically, we focus on data confidentiality and data integrity objectives. In its guiding principles, the approach is meant to be amenable for designers (e.g., software architects) that have very limited or no background in formal models, logics, and the like. To this aim, we provide an intuitive graphical notation, which is based on the familiar Data Flow Diagrams, and which requires as little effort as possible in terms of extra security-centric information the designer has to provide. The result of the analysis algorithm is the early discovery of design flaws in the form of violations of the intended security properties. The approach is implemented as a publicly available plugin for Eclipse and evaluated with four real-world case studies from publicly available literature.**

Index Terms—**Secure design, Data Flow Diagram, Confidentiality, Integrity**

I. Introduction

Security and privacy threats to software systems are a significant concern in many organizations, in particular due to recent legislations regarding data privacy (e.g., GDPR) and upcoming standards about security engineering (e.g., ISO 21434). In essence, there is an increasing push towards adopting methods and techniques that provide security and privacy assurance from the very beginning of a software development project. In particular, this paper focuses on the (architectural) modeling phase, when the structure of a software system is defined. In this context, threat analysis techniques like STRIDE [1] and LINDDUN [2] have gained significant popularity in the industry. Such techniques use Data Flow Diagrams (DFDs) as the notation of choice to represent the key computing elements in a system and to describe how information flows among them. The intrinsic value of such approaches is not under dispute. However, these approaches have two limitations. First, the DFD notation is geared towards business analysts. As such, the notation is very informal as there is no formal semantics attached to the model elements in a DFD. Second, threat analysis techniques like STRIDE hinge on the expertise of the analyst and do not provide any guarantee about the correctness and completeness of the analysis results [3].

Contribution. This paper has the ambition to lift the level of preciseness and automation used in the security-centric validation of design models. In particular, we are inspired by code-level information flow analysis techniques [4], [5], which are here raised to the level of abstraction of DFD-like design models. To this aim, the first contribution of this paper is a lightweight extension of the modeling capabilities provided by DFDs. In particular, the designer (e.g., software architect) has to provide the intended security policy for the information assets that flow in the system. For instance, the designer could specify that the geo-location of the user is private (high confidentiality), and the social network feed is public (low confidentiality). This is achieved by adding a security label to the data flows in the design diagram. In this paper, we focus on data confidentiality and data integrity properties. Additionally, the designer has to specify an abstract input-output security contract for the computational nodes. For simplicity, this is done by choosing from a small set of predefined options. For instance, the designer can specify that the asset on an input flow is copied to an output flow. The second contribution of this paper is a tool-supported, formally-based flow analysis technique that leverages the above-mentioned information to propagate the security labels across the design model (similar to taint analysis [6]). The result of the analysis algorithm is the early discovery of design flaws in the form of violations of the intended security policies. The analysis technique enables the enforcement of security contracts by means of static analysis and for security properties such as non-interference and declassification [7], [5]. The approach is implemented using the Viatra framework and packaged as a publicly available plugin for Eclipse [8]. Further, the approach is evaluated with four real-world case studies from publicly available literature.

Novelty. As discussed in the related work, some approaches have automated the threat analysis of DFDs by means of pattern-matching techniques, however, they do not provide soundness guarantees [9]. On the other hand, previous attempts to provide a formal semantics for DFDs have often resulted in a complicated language, hence losing the intuitive flexibility of DFDs [10]. This paper aims at achieving the benefits of a formal analysis technique by retaining the simplicity and intuitiveness of DFDs.

Relevance. Our approach is related to existing secure design practices and can be used to support secure software development. Further, it could easily synergize with existing code-level analysis techniques. In particular, our technique rests on the assumption that the contracts specified by the designer are correct, namely for what concerns the input-output relationships at each processing node. These contracts could be translated to code-level properties and verified at the implementation level. The rationale for a two-tier analysis approach (i.e., model and code) lies on the observation that

TABLE I
TYPE OF NODES AND THEIR SEMANTICS.

Node type	Security contract
Compare, Use, Join, Split	Join
Forward, Copy	Copy
Encrypt	Encrypt
Decrypt	Decrypt

code-level information flow analysis techniques do not scale well to large systems. Hence, it is preferable to verify localized contracts on smaller, amenable code units and delegate the verification of global, end-to-end policies to the model level.

In conclusion, we remark that our approach can also be used in the context of privacy, e.g., to analyze unintended flows of personal information in a system. Interestingly, as the modeling notation is intentionally kept simple, the approach can be used as a communication medium among several stakeholders: the privacy officers (who identify the privacy-sensitive information), the software architects (who specify the input-output behavior of the processing nodes), and the developers (who have to enforce such behavior in the code).

The rest of the paper is organized as follows. Section II presents an overview of the approach. Section III describes a formal model for the security analysis of DFDs, including a description of a security specification language and the underlying semantics of Security Data Flow Diagrams (SecDFD) labels. Section V describes the real-world case studies and evaluation results. Section VI discusses the relation between the global design-level analysis and the local code-level analysis, and considers limitations of our work. Finally, Section VII discusses the related work, while Section VIII presents future work and concluding remarks.

II. Overview of the Approach

The analysis approach relies on modeling a Security Data Flow Diagram (SecDFD) using a Domain Specific Language (cf. Section IV). Designers are required to invest some effort in modeling the appropriate components to use our plugin. Example SecDFDs used for the evaluation are available in the repository [8]. This section describes the SecDFD and gives an overview of the analysis approach by means of an example.

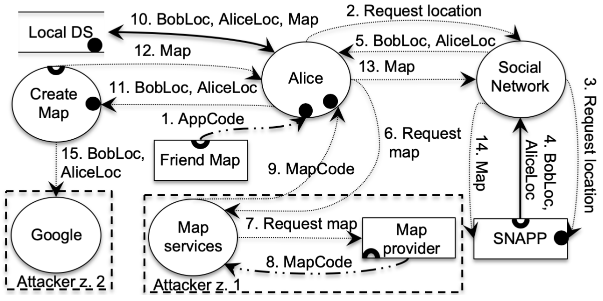
SecDFD. In a nutshell, a SecDFD is composed of interconnected nodes enriched with security concepts. Generally, nodes represent a piece of code as a series of commands. Each command might take some input from an incoming flow and transform it into an output. Inspired by the model in [11], we define the security semantics for an initial set of commands (dubbed ‘node types’). The latter differ with respect to the security contracts, as depicted in Table I. We define four such security contracts.

- Encrypt or hash contract. The contract for encrypting (possibly several) assets always results in propagating a low (public) label on the output flow(s).
- Decrypt contract. If the input asset is low, decrypting it results in propagating a low label on the output flow. However, if the input asset is high, decrypting it results in propagating a high label on the output flow.
- Join contract. The propagation function for joining two or more assets propagates the label equivalent to most restrictive input asset. This contract is applied to nodes comparing, using, joining, and splitting assets.
- Copy contract. This propagation function will copy the labels of the input assets to the corresponding output flows. This contract is applied to nodes copying and forwarding assets.

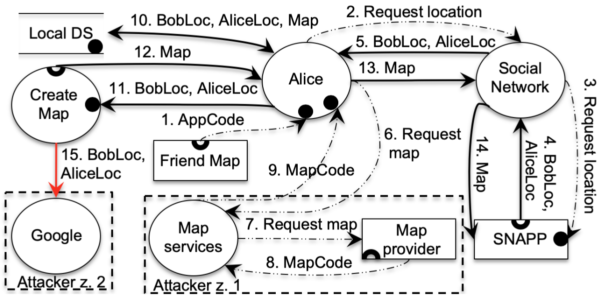
Inspired by eDFDs [12], SecDFDs are enriched with assets, their traces and security objectives, and security policies. We focus on tangible information assets and track them from the source node to target nodes. The asset source is a node in the diagram where an asset is first created. The asset target(s) are either nodes in the diagram where an asset rests (where it is stored permanently) or nodes in the diagram where a functionality makes use of an asset (where it has an impact on the application logic). We distinguish between a *global* security policy and a *local* security policy. The designer defines the global security policy by specifying security objectives of initial system assets and attacker observations (i.e., attacker zones). The global security policy stipulates that no information from a confidential input asset flows to a public output asset. The local security policy is defined at the level of nodes, and it can be parametric on the security labels of input and output flows. Attackers are commonly modeled as individual malicious nodes interacting with the system on a particular level of granularity. In this work, we explore the possibility to model the attacker as a set of nodes. We refer to the “attacker zone” as a non-empty set of node elements whose vulnerabilities can be exploited by attackers. For each attacker zone the designer can specify the capabilities of attackers launching attacks in that zone. For confidentiality, the attacker can either observe (read) an asset or not. Designers are able to also run the analysis with a more conservative attacker model, where the attacker can observe assets at all nodes. Finally, the analysis identifies design flaws where the global policy fails in the model. A formal specification language for the SecDFDs is described in Section III.

Local temporal dependencies. The standard DFD does not require to define the sequence of events. However, if a node has several propagation functions the order in which these functions are executed is important. For instance, $forward(c); join(a, b) \mapsto c$ is not equivalent to $join(a, b) \mapsto c; forward(c)$. In the first case, c is forwarded before it is created, thus a label propagation of such a sequence would give a faulty result. In the second case, however, a and b are first joined into a new asset c. Only after the join function is executed, the asset c is forwarded. Our model allows to specify such temporal dependencies.

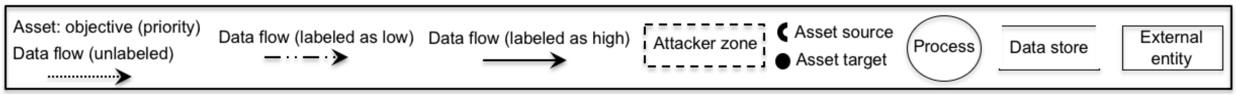
FriendMap example. Figure 1(a) depicts the SecDFD of a real-world application for a distributed computing platform developed by Liu et al. [13]. The platform’s programming language is based on Java Information Flow (JIF) [14] and it controls the computation and data through security type



(a) The SecDFD of FriendMap before the analysis.



(b) The SecDFD of FriendMap after the analysis.



(c) Legend.

Fig. 1. A SecDFD of FriendMap before the analysis (a), after the analysis (b), and legend (c).

annotations representing security policies. The application enables users (e.g., "Alice") to create a map of their friends (e.g., "Bob") and to post it on a social network. A client app on Alice's device first downloads the application code and executes it locally. The application fetches the locations of Alice and Bob from the social network and requests a map's code from a third-party map provider (e.g., Google Maps). The map is then created with the respective locations. Finally, Alice can also choose to post this map on the social network.

Analysis algorithm. Algorithm 1 shows the procedure for the analysis. In essence, the analysis propagates security labels

```
Data: SecDFD
Result: SecDFD'
derive security labels of initial flows;
while every graph node is not visited do
    for function : ordered list of propagation functions do
        propagate security label on the output flow;
    end
    visit next node;
end
if end-to-end view then
    query graph for end-to-end flow;
end
SecDFD' = fire global policy constraints on graph;
```

Algorithm 1: Analysis of a SecDFD.

according to local security policies. The input of the analysis is an instance of a SecDFD (similar to the one presented in Figure 1(a)). First, the security labels of initial flows are derived from the global security policy. After this step all flows are unlabeled, except for the initial flows (see flows 1, 4, 8, 10 and 15 in Figure 1(a)). If the most restrictive security objective on the flow is low, the label is derived as low (e.g. flow 1). If there is one confidentiality objective on a flow the label is derived as high (e.g. flow 4). The global policy dictates that no confidential assets are allowed to be revealed to the attacker.

TABLE II
THE SEQUENCE OF EVENTS FOR NODE ALICE.

	Event	Input asset	Output asset
1	Execute	AppCode	-
2	Forward	-	Request location
3	Forward	-	Request map
4	Execute	MapCode	-
5	Forward	BobLoc, AliceLoc	BobLoc, AliceLoc
6	Store	BobLoc, AliceLoc, Map	-
7	Forward	Map	Map

Thus, the security labels of input flows to malicious nodes are derived based on the attacker zone (flow 15). Second, all nodes of the SecDFD are visited. At each visit, propagation functions are executed in the proper order. Every propagation function propagates the security labels according the respective security contract. For instance, in Figure 1(a) the Alice node triggers several events for which the correct order is listed in Table II. If the event to forward the location (event 5) triggers right after the AppCode is executed (event 1), then the labels will first propagate on flow 11 (Figure 1(a)). When propagating labels, the assets determine which input flows will affect the propagation. In this case, flows 5 and 10 are the input flows transporting the locations. The current labels of flows 5 and 10 are initialized as low, therefore the propagation would incorrectly label flow 11 as low. Instead, the correct propagation requires event 2 to occur before event 5. As a result, flow 11 is correctly labeled as high. After this step, all the flows in the graph are labeled and the analysis terminates. Optionally, designers can trace the assets with an end-to-end view. Figure 1(b) shows the state of the SecDFD after label propagation. The analysis outcome is the result of verifying the global policy over the annotated SecDFD'. Design flaws are identified where the verification fails (e.g., flow 15).

III. Security Analysis for DFDs

This section presents a formal model underpinning the security analysis at the level of Data Flow Diagrams, as described in Section II. The analysis focuses on data confidentiality and data integrity objectives of a system at the time of system's architecture design. The main objective is to introduce a lightweight model that features the advantages of DFDs such as simplicity and usability, yet it contains enough information to reason about the security aspects of a system in a formal manner. Drawing on the theory of information-flow analysis [5], we introduce a security specification language for a system's design that describes the security objectives at the level of individual processes and functions of the system, and their interactions. The specification language supports a system designer in expressing flexible security policies locally for each process. Further, it enables an automatic procedure to analyze system-wide security objectives in an end-to-end fashion, thus unveiling potential security flaws at design time.

A. A security specification language

We now introduce the security specification language for DFDs. A process consists of a set of function signatures $f(i_1, \cdots, i_n : o)$ using a (possibly empty) set of input assets $i_1, \cdots, i_n$ to compute an (possibly empty) output asset o. We write $f(i_1, \cdots, i_n :)$ and $f(: o)$ whenever the set of input and output assets is empty, respectively. We define the interaction between processes through function composition, by linking the output of a function to the input of another function. As an example, consider the function signatures $getAliceLocation(i_A : o_A)$ and $getBobLocation(i_B : o_B)$ that retrieve the location i_A of Alice and location i_B of Bob, and forward them to the output channels o_A and o_B, respectively. Consider also a function signature $computeDistance(loc_1, loc_2 : dist)$ that computes the distance $dist$ between locations loc_1 and loc_2. We can model a DFD that uses the locations of Alice and Bob to compute their distance by composing function signatures as follows: $getAliceLocation(i_A : o_A); getBobLocation(i_B : o_B)$; $computeDistance(o_A, o_B : dist)$. Note that the matching between function inputs and outputs allows to model temporal dependencies between different functions in the DFD. Specifically, the outputs of the first two functions are used as inputs to the third function. In general, these dependencies induce a partial order between function signatures, which we use to capture the dependencies in a DFD (as represented by arrows). For simplicity, we assume a linearization of evaluation order between function signatures of a DFD, as denoted by the sequential composition of function signatures.

To reason about the security objectives of a system, we enrich function signatures with *security contracts* that enable a system designer to express the security policies *locally*, for each function and process. We then leverage the sequential composition of security contracts to analyze system-wide security policies over DFDs.

Concretely, we enrich the inputs and outputs of a function signature with security labels for confidentiality and integrity.

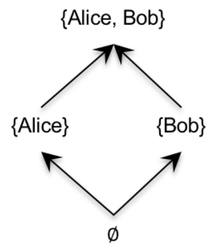


Fig. 2. Security lattice for confidentiality.

For a given (input or output) asset x, we write $(C(x), I(x))$ to denote the pair of confidentiality and integrity labels of asset x, respectively. For confidentiality, this means that the information stored in the asset x can only *flow to* assets that are at least as confidential as $C(x)$, and, for integrity, it means that the information stored in the asset x can only *affect* assets that are at most as trustworthy as $I(x)$. In this section we focus on confidentiality, noting that integrity is similar through dualization [15].

We assume a bounded lattice of security labels $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$. A label $\ell \in \mathcal{L}$ represents the confidentiality level of an asset. We write $\sqsubseteq$ to denote the ordering relation between security labels and, $\sqcup$ and $\sqcap$ to denote the *join* and *meet* lattice operators, respectively. We write $\top$ and $\bot$ to denote the top and the bottom element of the lattice. Figure 2 depicts a security lattice that models the confidentiality objectives for two principals, Alice and Bob. The lattice consists of 4 elements $\mathcal{L} = \{\{Alice, Bob\}, \{Alice\}, \{Bob\}, \varnothing\}$, where $\top = \{Alice, Bob\}$ and $\bot = \varnothing$. The ordering relation $\sqsubseteq$ (displayed by arrows) is set inclusion $\subseteq$, and the join and meet operators correspond to set union $\cup$ and set intersection $\cap$, respectively. For instance, an asset x labeled as $\{Alice, Bob\}$ is more confidential than an asset y labeled as $\{Alice\}$, as defined by the ordering relation $\{Alice\} \subseteq \{Alice, Bob\}$. Hence, the asset y can flow to the asset x, but not vice versa. In fact, the asset x requires the security clearance of both Alice and Bob in order to be observable, while the asset y only requires the security clearance of Alice. An attacker is an observer that can see information with a given security label from the lattice. For instance, if an attacker has security label $\{Alice\}$, the attacker has the security clearance to observe information that is at most as confidential as $\{Alice\}$. In particular, the attacker cannot observe assets labeled as $\{Bob\}$ or $\{Alice, Bob\}$. We remark that by fixing the security label of the attacker, the lattice can be reduced to a two-element lattice, where any asset below the attacker's label in the lattice is considered as public, otherwise it is considered as confidential. In what follows, we use the two-level security lattice $\mathcal{L} = \{\mathbf{L}, \mathbf{H}\}$ consisting of level $\mathbf{H}$ (high) for assets containing confidential information and level $\mathbf{L}$ (low) for assets containing public information. Further, we have that $\mathbf{L} \sqsubseteq \mathbf{H}$ and, for all $\ell_1, \ell_2 \in \{\mathbf{H}, \mathbf{L}\}$, $\ell_1 \sqcup \ell_2 = \mathbf{L}$ only if $\ell_1 = \ell_2 = \mathbf{L}$. Finally, we assume that the attacker has security label $\mathbf{L}$, hence the goal is to prevent the attacker from learning any information about assets labeled as **H**.

We lift function signatures to *security contracts* $f(i_1^{\ell_1}, \cdots, i_n^{\ell_n} : o^{lbl(\ell_1,\cdots,\ell_n)})$, where $\ell_1, \cdots, \ell_n \in \mathcal{L}$, and $lbl : \mathcal{L} \times \cdots \times \mathcal{L} \mapsto \mathcal{L}$ is a *labeling* function, mapping input labels to an output label. We sometimes write lbl or its definition for $lbl(\ell_1, \cdots, \ell_n)$. Security contracts allow a system designer to assign security labels to the input and output assets of a function. Moreover, the labeling function allows to specify how the security label of input assets affects the security label of an output asset. Security contracts have the unique property of being parametric on the security labels of input assets, and enforcing relationships between input and output labels. Label parametricity is an important feature at the design phase where processes and functions are designed in isolation and the system's security policy may still be unknown. In fact, the same process or function can have different security labels, depending on the context in which it is used. Finally, we remark that concrete (non-parametric) security labels can still be expressed through constant labeling functions, as in $lbl(\ell_1, \cdots, \ell_n) = \mathbf{H}$.

Because security contracts are an extension of function signatures with security labels, they can be composed through function composition in a similar manner. To prevent information leaks from confidential assets to public assets, we can ensure that the output label of a security contract is at most as confidential as the input label of the corresponding security contract. We achieve this by using the ordering relation from the security lattice, as in $f(: x^{\ell_1}); g(x^{\ell_2} :)$, where $\ell_1 \sqsubseteq \ell 2$. The following example elucidates the security contracts.

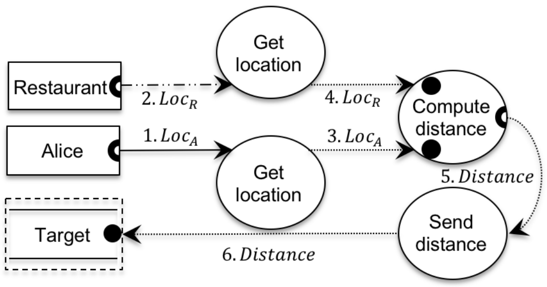


Fig. 3. A security-centric Data Flow Diagram before the analysis.

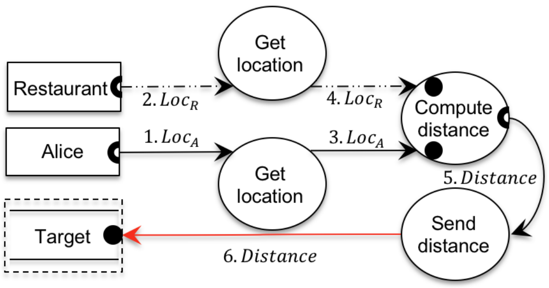


Fig. 4. A security-centric Data Flow Diagram after the analysis.

Example 1: Consider the design of a system that calculates the distance between a public location, e.g., the location of a restaurant labeled as **L**, and a confidential location, e.g., the location of user Alice labeled as **H**, and sends the distance to the user Charlie labeled as **L**. A system designer specifies the following security contracts:

- $getLocation(i^{\ell_1} : o^{lbl(\ell_1)})$ and $lbl(\ell_1) = \ell_1$, constraining the output label to be the same as the input label.
- $computeDistance(loc_1^{\ell_1}, loc_2^{\ell_2} \; : \; dist^{lbl(\ell_1,\ell_2)})$ and $lbl(\ell_1, \ell_2) = \ell_1 \sqcup \ell_2$, constraining the output label to be the same as the join of input labels.
- $sendDistance(i^{\ell_1} : o^{lbl(\ell_1)})$ and $lbl(\ell_1) = \ell_1$, constraining the output label to be the same as the input label.

The system can be designed by compositing the security contracts as follows: $getLocation(i_A^{\ell_A} : loc_A^{\ell_A}); getLocation(i_R^{\ell_R} : loc_R^{\ell_R})$; $computeDistance(loc_A^{\ell_A}, loc_R^{\ell_R} : dist^{\ell_A \sqcup \ell_R})$; $sendDistance(dist^{\ell_C} : o^{\ell_C})$, as displayed in Figure 3.

A designer can instantiate the DFD with a security policy for the source and the destination assets. Concretely, the designer defines the security policy by instantiating the input assets as $\ell_A = \mathbf{H}, \ell_R = \mathbf{L}$ and the output asset as $\ell_C = \mathbf{L}$ (cf. Figure 3, top). Intuitively, the design above is not secure since, by observing the distance between the confidential location of Alice and the public location of the restaurant, Charlie (an attacker with security clearance **L**) can learn Alice's location. The design flaw can be discovered by solving the constraints between the security labels globally in the DFD. The output label of the distance contract is $\ell_A \sqcup \ell_R = \mathbf{H}$, however the input label $\ell_C = \mathbf{L}$, which violates the security constraint $\ell_A \sqcup \ell_R \sqsubseteq \ell_C$, since $\mathbf{H} \not\sqsubseteq \mathbf{L}$ (cf. Figure 3, bottom).

B. Semantics of SecDFD labels

We now present a formal account of security contracts and their use in unveiling security flaws in DFDs. Figure 5 displays the syntax of our security specification language. We fix a two-level lattice $\mathcal{L} = \{\mathbf{L}, \mathbf{H}\}$. Security labels l consist of concrete labels (**L** and **H**) and label variables ($\ell \in \mathcal{L}$). Security label *expressions* e consist of security labels l and lattice operations over security labels ($e_1 \oplus e_2$), where $\oplus \in \{\sqsubseteq, \sqcup\}$. We use label expressions to define the security labeling function lbl, and to enforce constraints between security contracts. A Security Data Flow Diagram (SecDFD) consists of (sequential compositions of) security contracts $f(i^{l_1}, \cdots, i_n^{l_n} : o^e)$, where e is the definition of the labeling function connecting input labels to output labels. We use two special security contracts, $src(: o^l)$ and $dst(i^l :)$, to represent explicitly the source and destination assets of a SecDFD, respectively. We use source and destination assets to define a *global* security policy over a SecDFD. Moreover, we use the declassification contract $decl(i^{\mathbf{H}} : o^{\mathbf{L}})$ to downgrade the security of confidentiality information and consider it as public, e.g., after an encryption or hashing operation. We remark that our specification language captures the DFD node types from Section II.

Figure 6 depicts the semantics of labels for SecDFD. For a SecDFD dfd, security *configurations* have the form $\sigma \; \vdash \Gamma \; \{dfd\} \; \Gamma'$, where Γ and Γ' are *security environments* of type $Var \mapsto e$ and σ is a *label environment* of type $LVar \mapsto \{\mathbf{L}, \mathbf{H}\}$. We write Var for the set of variables that

$$
\begin{aligned}
l ::= {} & \mathbf{L} \mid \mathbf{H} \mid \ell \\
e ::= {} & l \mid e_1 \oplus e_2 \\
dfd ::= {} & f(i_1^{l_1}, \cdots, i_n^{l_n} : o^e) \mid src(: o^l) \mid dst(i^l :) \\
& \mid decl(i^{\mathbf{H}} : o^{\mathbf{L}}) \mid dfd_1; dfd_2
\end{aligned}
$$

Fig. 5. SecDFD Grammar.

are used in the security contracts, and $LVar$ for the set of parametric labels, i.e., $\ell \in LVar$. The security environment Γ keeps track of security labels during the execution of a SecDFD. The label environment σ instantiates parametric labels with concrete labels. We write $\Gamma(i)$ for the value of a variable i in Γ, and $\sigma(e)$ for the value of an expression e in σ. Moreover, $\Gamma[i \mapsto l]$ denotes a security environment Γ with variable i assigned the security label l. We also write _ (don't care), whenever a symbol is not important.

Intuitively, Γ describes the security labels of variables before the analysis of dfd, and Γ' describes the security label of variables after the analysis. Moreover, σ describes an instantiation of parametric labels with concrete labels. The rules can be read as follows: If the premises of a rule are satisfied in a security environment Γ and a label environment σ, i.e., none of the security constraints fails, the security contract executes and yields the security environment Γ'.

Each rule models the process of label propagation, the generated security constraints, and the update of the security environment. The rule CONTR ensures that whenever a security contract is executed, the input labels of the matching contracts are upper bounded by the security labels of input of the current contract (cf. $\Gamma(i_k) \sqsubseteq \sigma(l_k)$). This constraint prevents insecure flows from an output of a confidential contract to an input of a public contract. Moreover, we update the security environment by first updating the label of the input variable (cf. $\Gamma'' = \Gamma[i_k \mapsto \sigma(l_k)]$), and then evaluating the label expression in the new security environment Γ'' (cf. $\Gamma' = \Gamma''[o \mapsto \sigma(e)]$). Finally, the analysis produces the security environment Γ'. The rule SRC updates the security environment with the label of an input asset. We use this rule to define the security policy for the input assets of a SecDFD. The rule DST checks whether or not the security label of the input to an output asset is upper bounded by the security label of that asset. We use the DST rule to prevent insecure flows from confidential inputs to public assets, e.g., attacker zones. The rule DECL downgrades the security label of confidential input by making the output public. Finally, the rule SEQ models the sequential composition of two SecDFDs by matching the corresponding security environments.

The security analysis targets insecure flows of information from sources labeled as confidential to destinations labeled as public. To achieve this, a system designer defines a global security policy by specifying security labels for sources and destinations. Following the rules in Figure 6, we implement a static analyzer that infers a label environment σ (if it exists) and verifies the correctness of a security policy over the SecDFD. The inference algorithm uses basic constraint solving over the security lattice [4]. Section IV provides details about our implementation.

Definition 1 (Security policy for SecDFD): Given a set of source assets $\{src(: o_i^{l_i})\}$ and a set of destination assets $\{dst(i_j^{l_j} :)\}$, a security policy is an assignment of concrete security labels to the source and destination assets, i.e., $\Gamma_0[o_i \mapsto l_i, i_j \mapsto l_j]$ and $l_i, l_j \in \{\mathbf{L}, \mathbf{H}\}$.

Definition 2 (SecDFD security): A SecDFD dfd is secure wrt. security policy Γ_0 if there exists a label environment σ such that for any pair of matching security contracts $c_1 = \_(\_ : x^{l_1})$ and $c_2 = \_(x^{l_2} : \_)$, $\sigma(l_1) \sqsubseteq \sigma(l_2)$.

Intuitively, the security condition requires that there is never a flow in the SecDFD from a confidential output of a contract to a public input of a matching contract. In particular, this ensures the absence of flows from confidential sources to public destinations, thus enforcing the security policy. The following theorem shows that the rules in Figure 6 enforce the security condition in Definition 2.

Theorem 1: Given a SecDFD dfd, a label environment σ, and security policy Γ_0, then dfd is secure wrt. Γ_0 only if $\sigma \vdash \Gamma_0 \{dfd\} \Gamma'$, for some Γ'.

PROOF. The theorem can be proved by case analysis on the type of matching security contracts and the rules in Figure 6. Assume that $\sigma \vdash \Gamma_0 \{dfd\} \Gamma'$ for the given dfd. We show that for any $c_1 = \_(\_ : x^{l_1})$ and $c_2 = \_(x^{l_2} : \_)$ we have that $\sigma(l_1) \sqsubseteq \sigma(l_2)$. We illustrate a few cases.

- If $c_1 = f(i_1^{l_1}, \cdots, i_n^{l_n} : x^e)$ and $c_2 = f(i_1^{l_1}, \cdots, x^{l_2}, \cdots, i_n^{l_n} : o^{e_1})$, then it follows from the premise of rule CONTR that $\Gamma(x) = \sigma(e)$ and $\Gamma(x) \sqsubseteq \sigma(l_2)$, hence $\sigma(e) \sqsubseteq \sigma(l_2)$.
- If $c_1 = f(i_1^{l_1}, \cdots, i_n^{l_n} : x^e)$ and $c_2 = dst(x^{l_2} :)$, then it follows from the premise of rule CONTR that $\Gamma(x) = \sigma(e)$, and from the premise of rule DST that $\Gamma(x) \sqsubseteq \sigma(l_2)$, hence $\sigma(e) \sqsubseteq \sigma(l_2)$.
- If $c_1 = src(: x^{l_1})$ and $c_2 = dst(x^{l_2} :)$, then it follows from the premise of rule SRC that $\Gamma(x) = \sigma(l_1)$, and from the premise of rule DST that $\Gamma(x) \sqsubseteq \sigma(l_2)$, hence $\sigma(e) \sqsubseteq \sigma(l_2)$.
- If $c_1 = decl(i^{\mathbf{H}} : x^{\mathbf{L}})$ and $c_2 = f(i_1^{l_1}, \cdots, x^{l_2}, \cdots, i_n^{l_n} : o^{e_1})$, then $\mathbf{L} \sqsubseteq \sigma(l_2)$, since $\mathbf{L}$ is the bottom element of the lattice.
- If $c_1 = src(: x^{l_1})$ and $c_2 = decl(i^{\mathbf{H}} : x^{\mathbf{L}})$, then $\sigma(l_1) \sqsubseteq \mathbf{H}$, since $\mathbf{H}$ is the top element of the lattice.

□

IV. IMPLEMENTATION

Figure 7 shows the plugin toolchain. The approach is implemented as a publicly available Eclipse plugin [8] using the Eclipse Modeling Framework (EMF). We build two metamodels: for modeling SecDFDs, and for end-to-end security views. In addition, we built an external Domain-Specific Language (DSL) using the Xtext framework for modeling SecDFDs. The DSL is accompanied by a simple grammar

$$\textsc{Contr}\ \frac{\Gamma(i_k) \sqsubseteq \sigma(l_k) \qquad \Gamma'' = \Gamma[i_k \mapsto \sigma(l_k)] \qquad \Gamma' = \Gamma''[o \mapsto \sigma(e)] \qquad k \in \{1, \cdots, n\}}{\sigma \vdash \Gamma\ \{f(i_1^{l_1}, \cdots, i_n^{l_n} : o^e)\}\ \Gamma'} \qquad \textsc{Src}\ \frac{\Gamma' = \Gamma[o \mapsto \sigma(l)]}{\sigma \vdash \Gamma\ \{src(: o^l)\}\ \Gamma'}$$

$$\textsc{Dst}\ \frac{\Gamma(i) \sqsubseteq \sigma(l)}{\sigma \vdash \Gamma\ \{dst(i^l :)\}\ \Gamma} \qquad \textsc{Decl}\ \frac{\Gamma' = \Gamma[o \mapsto \mathbf{L}]}{\sigma \vdash \Gamma\ \{decl(i^{\mathbf{H}} : o^{\mathbf{L}})\}\ \Gamma'} \qquad \textsc{Seq}\ \frac{\sigma \vdash \Gamma\ \{dfd_1\}\ \Gamma'' \qquad \sigma \vdash \Gamma''\ \{dfd_2\}\ \Gamma'}{\sigma \vdash \Gamma\ \{dfd_1; dfd_2\}\ \Gamma'}$$

Fig. 6. Semantics of SecDFD labels.

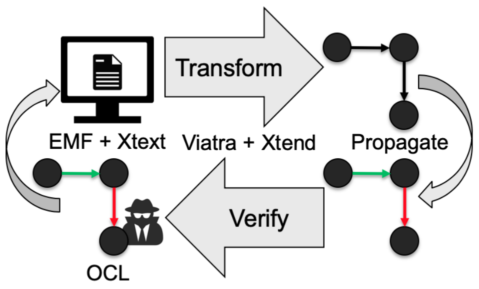


Fig. 7. The plugin toolchain.

and a textual syntax. We use the Viatra query engine and the Xtend language to transform SecDFDs to simple graphs. The new graph is visited with a recursive Depth First Search (DFS) algorithm. At each node, the labels of outgoing flows are propagated according to the node semantics. Afterwards, we statically validate the global policy over the resulting graph model. To this aim we write constraints in the Object Constraint Language (OCL). Optionally, the graph model is queried for end-to-end asset traces, namely, all graph elements handling a particular asset. Note that our implementation currently supports specifying local temporal dependencies (at node level) with flow enumeration.

V. Evaluation

We evaluate our approach by running the analysis on several open source projects. First, we test our approach on microbenchmarks from DroidBench[1], an open test suite for evaluating the effectiveness of taint-analysis tools for Android apps. Second, we model four realistic applications, namely FriendMap, Hospital, JPmail, and WebRTC. The analysis of all four applications does not produce any false positives. Further details about the evaluation are available in the repository [8].

We have tested our plugin on several inter-component communication examples. These small examples have built-in design flaws. They were used to verify that the propagation functions work as expected and are able to identify built-in design flaws. For instance, the model of the first example consists of two processes, a source and a sink data store. A sensitive asset is read from a source data store, sent via the activities with a *forward* responsibility, and then written to a sink data store. The attacker zone models APIs able an attacker to observe (read) information, and it is referenced to one of the activities. After the flow labels have been propagated, a static constraint model detects the leak on the elements flowing to the attacker zone. The propagation functions were able to identify design flaws in all initial tests. Together with the core security analysis from Section III, the microbenchmark increased our confidence on the correctness of our analysis tool.

[1]https://github.com/secure-software-engineering/DroidBench

A. FriendMap

We have evaluated our approach on the example described in Section II. Figure 1(b) depicts the results of the analysis.

Since the AppCode is not considered confidential, the flow from FriendMap to Alice is labeled as low. Similarly, the requests (for locations and MapCode) that are forwarded from Alice cause the labels on the underlying flows to be low (public). However, Alice's and Bob's locations are sensitive, therefore the resulting flow (flow 4) is labeled as high (private). The locations are forwarded (Social Network, Alice), copied (Alice) and stored (Local DS) in other parts of the model, causing high labels on flows 5, 10, 11, 15. The Map is created and is forwarded (via nodes Alice, Social Network) to a social network server where it is stored. Because the Map is sensitive, it causes the propagation of high labels on flows 12, 13 and 14. In this scenario, a part of the map code could be malicious and attempt to leak the locations from the browser. After the label propagation, a static validation of the model instance identifies a potential design flaw on flow 15. This is the only design flaw the analysis discovers under attacker zones 1 and 2 (see Figure 1(b)). A possible security solution is to obfuscate and then declassify the assets (Map, BobLoc, AliceLoc) before sending them to Create Map. An analysis on the new model (under the same attacker zone) does not identify any design flaws [8].

B. Hospital

We evaluated the plugin on an application for controlling access to sensitive patient data [13]. Figures 8 and 9 depict the SecDFD for the Hospital application before and after the analysis, respectively. The access control policy and the application code are first loaded to the Employee client (i.e.

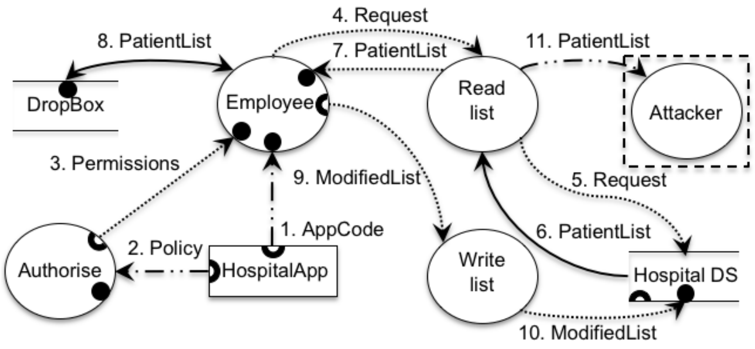


Fig. 8. A SecDFD for Hospital before the analysis.

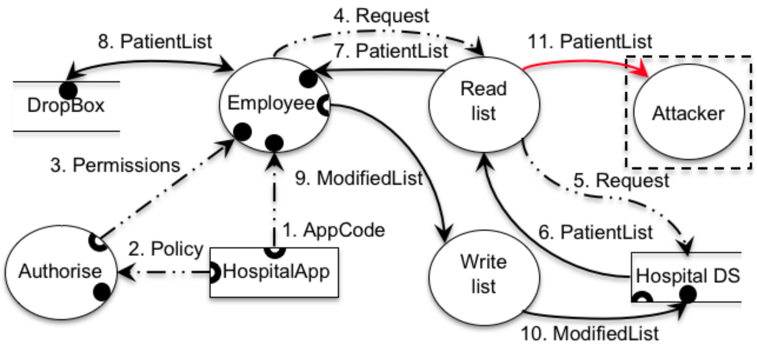


Fig. 9. A SecDFD for Hospital after the analysis.

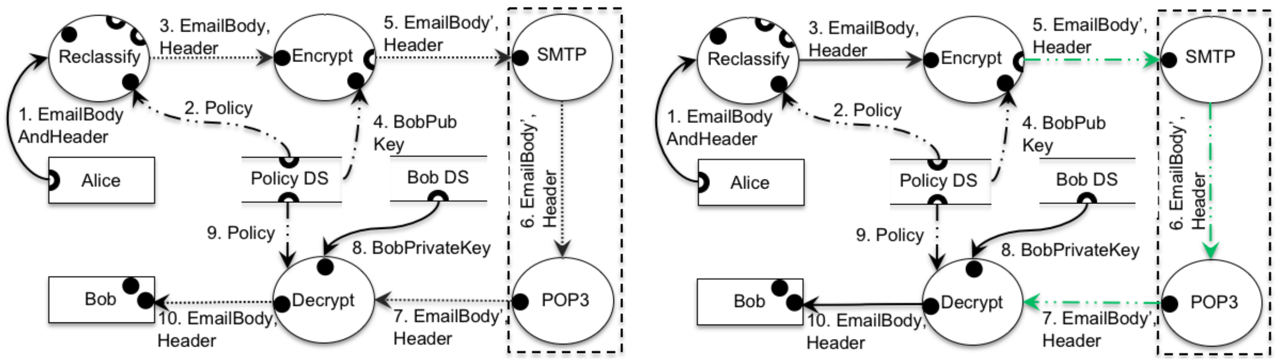
mobile application). The employees can sent a request to read the list of patients, including the patient HIV status. The request is forwarded to a node that handles read requests. Depending on the given permissions, the node retrieves the sensitive list of patients and forwards it to the employee node, where it is stored on a server. A similar node handles requests for modifying the patient list. We model the attacker as a node attempting to observe the list of patients. The global policy causes flows transporting the Application Code and Policy as low, since the assets are modeled as non-confidential [8]. Further the labels of flows transporting the Patient List and Modified List are derived as high, since these assets are modeled as confidential [8]. The label propagation causes the flows containing the patient list and the modified patient list to be labeled as high. The final verification of the global policy discovers a design flaw in the node retrieving the patient list from the Hospital DS. Namely, if the attacker is able to spoof the Read list node by injecting false requests, he could gain access to the patient list. An alternative model [8] includes a declassification of the patient list before it is sent to Read list. An analysis on the alternative model (under the same attacker zone) does not identify any design flaws.

C. JPmail

JPmail[2] is an email client implementing a subset of the MIME protocol. Developed in JIF, it leverages information-flow control to enforce the following policy: *"The body of an email should be visible only to the authorized senders and receivers."* Figures 10(a) and 10(b) depict the SecDFD for the JPmail application before and after the analysis, respectively.

The mail client consists of three main software components: a POP3-based mail reader, an SMTP-based mail sender and a policy store. In essence, to send an email the user ("Alice") specifies the email body and header. Both email body and header are passed through a classifier to reclassify the header as public (headers must be readable by the mail server). The email body however, is encrypted using symmetric encryption. Finally, the email is sent to the SMTP server, which delivers it to the POP3 server. The email recipient ("Bob") retrieves the email from a POP3 server. During this procedure, the email header remains public, whereas the email body is decrypted using Bob's private key, which is read from Bob's local storage. Upon a successful decryption, Bob is able to read the contents of the email body. In this scenario, the SMTP and POP3 servers are common targets of attack as they are exposed to open networks. Thus the respective nodes are modeled in the attacker zone. The global policy causes flows 2, 4 and 9 to be labeled as low, since the assets on the flows (Policy and Bob's public key) are not modeled as confidential. Further the email body and header are confidential, causing the label on flow 1 to be derived as high. At this point, the graph is visited and the propagation functions propagate labels accordingly. No information can be leaked to the attacker zone due to the nodes for encrypting and decrypting the email body. The analysis confirms that JPmail is flawless by design. We have

[2]http://siis.cse.psu.edu/jpmail/jpmaildetails.html



(a) A SecDFD for JPmail before the analysis.

(b) A SecDFD for JPmail after the analysis.

Fig. 10. A SecDFD for JPmail before and after the analysis.

injected a design flaw (omitted encryption of email body) and the analysis could identify it correctly.

D. WebRTC

We extend our validation by applying our approach on a model of the WebRTC project [3] which facilitates real-time communication capabilities for browsers and mobile applications. In a nutshell, WebRTC provides the infrastructure needed for developing applications that require sharing user media (voice, video, files) between two or more parties. Figure 11 depicts the SecDFD for WebRTC after the analysis. A peer-to-peer network communication is established via a signaling server using the HTTPS protocol (see flows 5-8 in Figure 11). Afterwards the client browsers (see Browser A, Browser B in Figure 11) obtain Browser IDs via the browser identity providers (see IDPX, IDPY in Figure 11). The browsers verify the identity of other clients involved in the media exchange. In the next step, the calling browser (e.g., Browser A) triggers an attempt and establishes a secure connection via STUN/TURN servers. Datagram Transport Layer Security (DTLS) and Secure Real-Time Transport Protocol (SRTP) are used to secure data transfers between the browsers. Once a secure connection is established, the media can be transfered between the clients (see flows 21-26 in Figure 11).

The analysis was performed under a conservative attacker model. At each node, we assume the attacker is able to observe the assets. The global policy causes flows 9, 12, 21, and 24 in Figure 11 to be labeled as high, since the assets on the flows (AliceBrowserID, BobBrowserID, AMedia, and BMedia) are modeled as confidential. Upon visiting the graph, the security labels are propagated as follows. First, the AliceBrowserID is generated and stored to a data store on G-mail servers (flows 1-2 in Figure 11). Since AliceBrowserID is labeled as confidential and IDPX forwards the assets, the copy security contract is applied. An analogous propagation happens for flows 9-10, 3-4, 10-11, and 12-14 in Figure 11. Upon propagating labels on flows 5-8 in Figure 11, the nodes are forwarding encrypted session data via the HTTPS protocol. Therefore the copy contract is applied and the flows are labeled as low. A similar propagation happens for flows 15-20 in Figure 11. Since AMedia is encrypted in Browser A, the encrypt security contract is applied to the outgoing flow 22. The EncrAMedia is decrypted in Browser B causing the use of the decrypt security contract on flow 23. Similar reasoning is applied to BMedia that is sent to Alice in the other direction. Due to the conservative attacker model, our analysis identifies potential design flaws at all nodes with high input/output flows.

A second analysis of the WebRTC was performed using a different tool as a sanity-check of our results. We compared our results to the results obtained by an Eclipse plugin introduced by Sion et al. [16]. We remark that their approach performs the analysis (similarly to Microsoft Threat Modeling tool [4]) by applying the STRIDE threat-to-element mapping table to each model element. Further, our approach limits the analysis to the confidentiality concern, where as Sion et al. [16] also consider integrity, availability and accountability. Thus, the outcomes of our analysis are complementary. Due to extending the model with security solutions, Sion et al. [16] identify several threats as less critical. An interesting remark is that these less critical threats align with our analysis results. Namely, the nodes where no confidential assets can be observed (STUN TURN A, Signaling server, STUN TURN B in Figure 11) do not violate the global security policy.

VI. Discussion and limitations

A mismatch between intended architecture and implemented architecture is a source of frustration in many organizations. It manifests itself in a loss of resources for large code refactorings, loss of functionality, architectural decay and technical debt. Further, assuring architectural compliance is a hard problem [17]. In the following, we discuss challenges and opportunities of our approach.

Secure model to code. SecDFDs provide a simple and intuitive model for analyzing security policies at the design level. They can help a system designer to uncover security flaws or prove that the design is secure. On the other hand, the

[3]https://webrtc.org/

[4]https://www.microsoft.com/en-us/download/details.aspx?id=49168

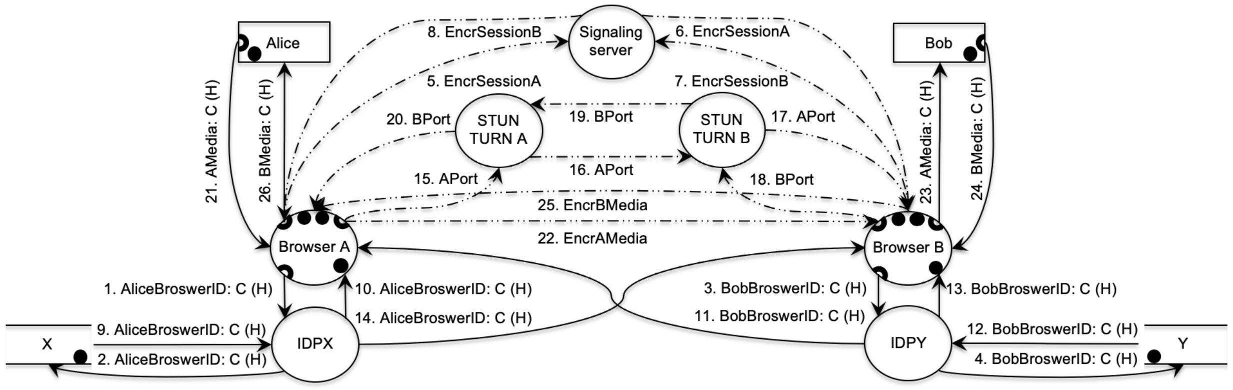


Fig. 11. A SecDFD for WebRTC after the analysis.

security guarantees provided by this approach hold under the assumption that the code implementing the security contracts does not violate the security labels. Such implementation can be verified in a second phase by using existing code-level security analysis. For our case studies, we leverage existing implementations in security-typed languages such as JIF[14].

A major advantage that comes with the two-phase security analysis is *compositionality*. By decoupling the security analysis into a global design-level analysis and a local implementation-level analysis, we only need to check the security of implementations locally, and obtain end-to-end security assurance for the entire system. Compositionality is an important and desirable property for scaling a security analysis to real-world systems. Further, it enables us to relate the results of the verification analysis to semantic security conditions such as noninterference [7], thus providing provable security guarantees. Intuitively, the security condition requires that confidential information is released to attacker zones only in a controlled manner. The design-level security analysis enforces such invariant for a given security policy. Further, the code-level security analysis ensures the implementation of a security contract entails the contract's security labels. Thus it follows that the system satisfies the security condition.

Secure code to model. Existing approaches and tools for code-level security analysis may suffer from scalability issues. In particular, such tools may not scale to cyber-physical systems, large-scale information management systems or cloud-based systems. Thus, a higher level of abstraction is necessary for the security analysis of such systems. Significant effort has been put into reverse engineering the architecture (design recovery) from the implementation [18], [19]. Further, existing work aims to extract and analyze the security concepts in the implemented architecture [20], [21]. Yet, existing tools for extracting the architectural design also have scalability issues. Further, they do not provide formal guarantees. We see potential benefits in leveraging our approach for lifting implementation to the design level and analyzing security on that level of abstraction. For instance, a call graph can be extracted from code. In this case, the call graph would represent the SecDFD at the lowest level of granularity. By means of specifying the node types of the call graph (or possibly annotating code methods with node types) and specifying a global security policy, the SecDFD could be extracted from the code. Once the node types are specified, the compositionality properties of our approach enables further abstraction. The security analysis can then be performed on the *implemented* architectural design. Under the assumption that the node types are specified correctly and that the implemented methods do not violate the security labels, such an approach could provide not only a semi-automatic architecture compliance, but also certain guarantees for the security analysis.

Limitations. The challenge of identifying the sources and sinks of information, including attacker zones, is also applicable to our work. At the current stage, our approach is best suitable for top-down architectural security design, where arguably the designer is aware of sources and destinations of information and their security requirements. Alternatively, we can leverage existing work on automatic discovery of source and sinks, e.g., the SuSi tool for the Android framework [22]. To our best knowledge, SecDFDs keep the desired simplicity of the DFD notation. However, the usability aspect was not subject to our validation. In the future we plan to conduct user studies to validate the difficulty of performing a SecDFD threat analysis with practitioners. Finally, we recognize potential subjectivity in reporting the evaluation results and remark that a thorough validation is planned for future work.

VII. Related work

This section discusses related work in automated security analysis, DFD semantics, and information flow analysis.

Automating security analysis of diagrams. Almorsy et al. [23] propose an approach for automating the security analysis by capturing vulnerabilities and security metrics. It is beneficial to analyze system vulnerabilities and system defenses side by side. Similarly to our approach, their model-based approach requires input from the designer to discover vulnerabilities. However, Almorsy et al. [23] do not capture information disclosure threats and security policies. Further, the correctness of analysis results relies on the soundness of the signatures, which are very generic. More importantly, the described signatures for identifying the vulnerabilities are not supported by formal reasoning.

Berger et al. [24] propose an approach for semi-automated architectural risk analysis. This approach leverages a subset of open source vulnerability repositories and a rule checker for pattern matching the system model represented with EDFDs. The use of knowledge-bases has shown to be successful in finding security threats in the past. Yet, it is challenging to interpret vulnerability descriptions from open source repositories into graph query rules. Like us, Berger et al. [24] extend the DFD with data source, target, channels, and confidentiality objectives. In contrast, the SecDFD has more semantical flavor and includes the ability to provide security specifications and attacker zones.

Sion et al. [16] present an approach for a risk-centric threat analysis of DFDs, which enables threat elicitation and risk analysis. The ability to model security solutions in the form of architectural patterns is useful for a more comprehensive analysis. The approach relies on the security expert to provide the initial distribution estimates for the Monte-Carlo simulation which may result in low confidence and precision. Similarly, the authors enrich the DFD with security solutions. Further, their tool uses the Viatra query engine and a graph-based pattern language. In contrast, their approach is risk-centric whereas this work aims to provide security semantics to DFDs.

Jürjens et al. [25], [26] have proposed UMLSec, an extension for UML to model security aspects in system design and prove security properties, such as secrecy. UMLSec's formal semantics scales well as it applies to a variety of model types, e.g., activity diagrams or statecharts. Like SecDFD, UMLSec defines a system as a composition of subsystems and enables modeling a security policy and attackers. In contrast, Jürjens et al. [25] focus the analysis on attacker behavior, rather than the semantics of security labels on flows.

Guerriero et al. [27] propose a privacy-by-design approach for specifying and enforcing privacy policies by code rewriting. Similarly to our work, the authors propose a model and a specification language for policies over sensitive data flows. In addition, the privacy policies are enforced algorithmically on an application data flow model. They introduce privacy-aware operators which enable policy enforcement. Further, their approach is focused on preventing disclosure of sensitive data under certain contextual conditions (i.e., when and how much data can be observed), rather than the way sensitive data is transformed by the system (i.e., security contracts of operations).

Breaux et al. [28] develop a methodology for mapping privacy requirements from natural language text to a formal language. Interestingly, the authors define traces of requirements around particular data (similar to our end-to-end view). However, Breaux et al. [28] focus on specifying policies and identifying conflicts between policies of different actors.

Security semantics of DFDs. Several works approach DFDs from a formal angle, by associating a formal semantics to the model. They aim at extending DFDs with lightweight specifications for expressing functional correctness properties. For instance, Leavens et al. [29] propose a DFD semantics that allows to specify the dynamic behavior of a concurrent system, and Larsen et al. [30] leverage formal specifications in the VDM language to formally reason about DFDs. We refer to the work by Jilani et al. [31] for an overview. In contrast, our work focuses on the *security* semantics of DFDs and it presents a simple label model that enables security analysis at the design level. The simplicity stems from the fact that our label model only focuses on security and ignores the functional correctness of the system.

Information flow analysis at code level. Abdellatif et al. [32] present an approach accompanied by a toolkit to automate information flow control in component-based systems. Their approach requires developers to specify the security properties with a configuration file, which in turn is used to validate the system for potential data leaks, before the security code is generated. Similarly to our work, the authors identify security leaks by automatically checking the security policy at the level of components. Yet, our work is unique with respect to label propagation and label specification for system components. In addition, our work defines attacker zones as part of a global security policy, while Abdellatif et al. [32] do not model the attacker explicitly.

Information flow control is a well-studied research area. A large array of security conditions and enforcement mechanisms have been proposed to address different computational models, languages and systems [7], [5], [6]. Our contribution is orthogonal to these works and it can leverage their results as discussed in Section VI.

VIII. Conclusion

In this paper we have presented a formal approach to analyze security objectives of information flows at the design level. The approach focuses the analysis of confidentiality and integrity objectives. We provide a formal definition of a security specification language for DFDs. In addition, we introduce the Security Data Flow Diagram (SecDFD) and provide semantics of SecDFD security labels. We prove security for the SecDFD with respect to a global security policy and a security label environment. We have implemented our approach using the Viatra framework and packaged it as a publicly available plugin for Eclipse. The approach is evaluated on four open source applications. The underlying compositionality of our approach provides opportunity to refine the analysis from a global design level to a local implementation level analysis. In the future we plan to implement mechanisms to refine the analysis on the level of implementation by combining existing information flow analysis techniques such as static, dynamic and hybrid analysis. Further, we plan to extend the node semantics, covering additional node types, such as authentication, authorization, and verification. Finally, we plan further validation efforts with respect to the usability aspects and the analysis of larger open source projects with (and without) known design flaws.

Acknowledgment

This research was partially supported by the Swedish VINNOVA FFI project "HoliSec: Holistic Approach to Improve Data Security". Musard Balliu was partially supported by the JointForce project financed by the Swedish Research Council and the TrustFull project financed by the Swedish Foundation for Strategic Research.

References

[1] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[2] K. Wuyts, R. Scandariato, and W. Joosen, "Empirical evaluation of a privacy-focused threat modeling methodology," *Journal of Systems and Software*, vol. 96, 2014.

[3] R. Scandariato, K. Wuyts, and W. Joosen, "A descriptive study of Microsoft's threat modeling technique," *Requirements Engineering*, vol. 20, no. 2, 2015.

[4] D. M. Volpano and G. Smith, "A type-based approach to program security," in *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, 1997, pp. 607–621.

[5] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[6] M. Balliu, D. Schoepe, and A. Sabelfeld, "We are family: Relating information-flow trackers," in *Proceedings of the 22nd European Symposium on Research in Computer Security, ESORICS*, 2017.

[7] J. A. Goguen and J. Meseguer, "Security policies and security models." in *S&P*, 1982.

[8] K. Tuma, M. Balliu, and R. Scandariato, "Flaws in flows: Unveiling design flaws via information flow analysis," https://github.com/anonymous-anon/flaws-in-flows, 2018.

[9] B. Berger, K. Sohr, and R. Koschke, "Automatically extracting threats from extended data flow diagrams," in *Engineering Secure Software and Systems (ESSoS)*, 2016.

[10] A. A. A. Jilani, A. Nadeem, T. hoon Kim, and E. suk Cho, "Formal representations of the data flow diagram: A survey," in *Advanced Software Engineering and Its Applications (ASEA)*, 2008.

[11] A. van den Berghe, K. Yskout, W. Joosen, and R. Scandariato, "A model for provably secure software design," in *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*. IEEE Press, 2017, pp. 3–9.

[12] K. Tuma, R. Scandariato, M. Widman, and C. Sandberg, "Towards security threats that matter," in *Computer Security*. Springer, 2017, pp. 47–62.

[13] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 321–334.

[14] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif 3.0: Java information flow," July 2006. [Online]. Available: http://www.cs.cornell.edu/jif

[15] K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Tech. Rep., 1977.

[16] L. Sion, D. Van Landuyt, K. Yskout, and W. Joosen, "Sparta: Security & privacy architecture through risk-driven threat assessment," in *IEEE 2018 International Conference on Software Architecture (ICSA 2018)*. IEEE, 2018.

[17] S. Jasser, K. Tuma, R. Scandariato, and M. Riebisch, "Back to the drawing board-bringing security constraints in an architecture-centric software development process." in *ICISSP*, 2018, pp. 438–446.

[18] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Advances in Engineering Software*, vol. 41, no. 4, pp. 519–526, 2010.

[19] G. Rasool and D. Streitferdt, "A survey on design pattern recovery techniques," *J. Comp. Sci*, vol. 8, no. 2, 2011.

[20] M. Abi-Antoun and J. M. Barnes, "Analyzing security architectures," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 3–12.

[21] B. J. Berger, K. Sohr, and R. Koschke, "Extracting and analyzing the implemented security architecture of business applications," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 285–294.

[22] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[23] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 662–671.

[24] B. J. Berger, K. Sohr, and R. Koschke, "Automatically extracting threats from extended data flow diagrams," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 56–71.

[25] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *International Conference on The Unified Modeling Language*. Springer, 2002, pp. 412–425.

[26] J. Jürjens, J. Schreck, and Y. Yu, "Automated analysis of permission-based security using umlsec," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2008, pp. 292–295.

[27] M. Guerriero, D. A. Tamburri, and E. Di Nitto, "Defining, enforcing and checking privacy policies in data-intensive applications," in *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2018, pp. 172–182.

[28] T. D. Breaux, H. Hibshi, and A. Rao, "Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements," *Requirements Engineering*, vol. 19, no. 3, pp. 281–307, 2014.

[29] G. T. Leavens, T. Wahls, and A. L. Baker, "Formal semantics for sa style data flow diagram specification languages," in *Proceedings of the 1999 ACM Symposium on Applied Computing*, ser. SAC '99, 1999, pp. 526–532.

[30] P. G. Larsen, N. Plat, and H. Toetenel, "A formal semantics of data flow diagrams," *Form. Asp. Comput.*, vol. 6, no. 6, pp. 586–606, Dec. 1994.

[31] A. A. A. Jilani, A. Nadeem, T.-h. Kim, and E.-s. Cho, "Formal representations of the data flow diagram: A survey," in *Proceedings of the 2008 Advanced Software Engineering and Its Applications*, ser. ASEA '08. IEEE Computer Society, 2008, pp. 153–158.

[32] T. Abdellatif, L. Sfaxi, R. Robbana, and Y. Lakhnech, "Automating information flow control in component-based distributed systems," in *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. ACM, 2011, pp. 73–82.