Contextualisation of data flow diagrams for security analysis.

FAILY, S., SCANDARIATO, R., SHOSTACK, A., SION, L. and KI-ARIES, D.

2020

Contextualisation of Data Flow Diagrams for security analysis

Shamal Faily[1][0000−0002−2859−1143], Riccardo Scandariato[2], Adam Shostack[3],
Laurens Sion[4][0000−0002−8126−4491], and Duncan Ki-Aries[1][0000−0001−8114−2737]

[1] Department of Computing & Informatics, Bournemouth University, Poole, UK
{sfaily,dkiaries}@bournemouth.ac.uk
[2] Chalmers and University of Gothenburg, Gothenburg, Sweden
riccardo.scandariato@cse.gu.se
[3] Shostack & Associates, Seattle, USA
adam@shostack.org
[4] imec-DistriNet, KU Leuven, Leuven, Belgium
laurens.sion@cs.kuleuven.be

Abstract. Data flow diagrams (DFDs) are popular for sketching systems for subsequent threat modelling. Their limited semantics make reasoning about them difficult, but enriching them endangers their simplicity and subsequent ease of take up. We present an approach for reasoning about tainted data flows in design-level DFDs by putting them in context with other complementary usability and requirements models. We illustrate our approach using a pilot study, where tainted data flows were identified without any augmentations to either the DFD or its complementary models.

1   Introduction

Data Flow Diagrams (DFDs) are useful as a sketch that explores how a system and its elements might be exploited; their simplicity makes it possible for different people with different levels of expertise to contribute to the security analysis of a system as it evolves.

As DFDs become more critical to security design practices, so too is the need to reason about their properties using software tools. Limitations around cognitive ability, expertise and time constrain the effectiveness of modellers when scaling up or making decisions around DFDs [16]. However, their limited semantics makes reasoning with DFDs alone difficult; this leads to an inherent trade-off between using easy to adopt notations and those that afford automated reasoning but are more elaborate [17].

Data flows are analogous with information flows. Information flow analysis (like taint analysis) is a long established technique for reasoning about the interactions of data within entities, and their impact on security as the data flows through the system [4,23]. Unfortunately, visual inspection alone is insufficient for spotting potential issues with data inside data flows. Formal policy specifications and binary instructions provide the context necessary to reason about

tainted information flows, but DFDs lack this level of precision. The options are either (i) adding additional information to the diagram itself, or (ii) providing context via other models aligned with DFDs. In the related work, the first route has been extensively explored [20,19], so this paper takes the less followed second path. *Usability models* could play a particularly important role in providing such context. For example in [7], usability models describe the main tasks performed by a software system, and the roles associated to those tasks. The models relate to the overall goals and requirements of the system. Just as DFDs provide early insights into how systems might be exploited, usability models indicate where interaction problems might subsequently facilitate exploitation. These different models might be produced independently and, with inter-operable tools, we can reason about the security impact these models have on DFDs, and vice-versa.

Contribution. In this short paper, we present an approach for identifying potential taint in design-level DFDs. Our guiding principle is that, to encourage adoption, DFDs should be no more graphically complex than they currently are. Instead, we should leverage the alignment between DFDs and other usability and requirements models. We present the related work upon which our approach is based in Section 2 before presenting the key concepts and algorithms in our approach in Section 3. We illustrate our approach in Section 4 by using it to identify pre-process and post-process taint in a critical infrastructure pilot study, before discussing the implications of this work in Section 5.

2 Related Work and Background

2.1 Reasoning about Data Flow Diagrams in Threat Modelling

Data Flow Diagrams (DFDs) graphically model flows of information (data flows) between human or system actors external to a system (entities), activities that manipulate data (processes), and persistent data storage (data stores) [24]. This notation is often extended with trust boundaries: dotted boxes encompassing DFD elements operating at the same level of privilege. Trust boundaries help identify data flows that cross privilege levels [15].

DFDs have overlapping functions. Diane (a diagram creator) creates a DFD that diagrammatically represents her mental model. On viewing the DFD, Elaine (an engineer) internalises this mental model and requests changes. Dialogue around their differences subsequently brings both mental models closer together. Francis (a formal modeller) crafts a structured representation of a system, from which subsequent reasoning can be performed. This relationship between a mental model, a diagram, and a formal model has not been well explored.

Tuma et al. [21] first examined the potential of using information flow analysis to reason about DFDs. They extended the DFD notation by labelling data flows with assets and their security properties, indicating the source and target of assets, including domain properties and assumptions from the KAOS modelling language [12]. In later work, Tuma et al. [19] further illustrate the potential for using DFDs for design-level information flow analysis. In their approach, a

domain specific language is used to model DFDs annotated with security labels. The model is subsequently rendered as a graph and statically analysed.

Antigac et al. [1] examined how certain properties of a DFD can be hotspots for further investigation. For example, a usage hotspot corresponds with 3 DFD elements: data flow d into process p, process p, and data flow d' from p. Antigac et al. showed how such hotspots bridge the gap between different models, and provide a basis for subsequent model transformation without fundamentally changing the visual semantics of DFDs.

2.2 Security and Software Design Meta-models

Meta-models specify how model concepts are associated. In doing so, they guide analysts in collecting and analysing model data, and guide tool builders in constructing tools to support them. The software engineering community has examined the relationship between software and requirement modelling approaches and security, as summarised by [14]. These approaches do not, however, account for the role played by usability data and models. The **IRIS (Integrating Requirements and Information Security)** meta-model was devised to provide guidance on how early-stage design concepts from usability as well as security and requirements engineering might be aligned [7]. A sub-set of the IRIS concepts relevant to this paper is provided in Figure 1.
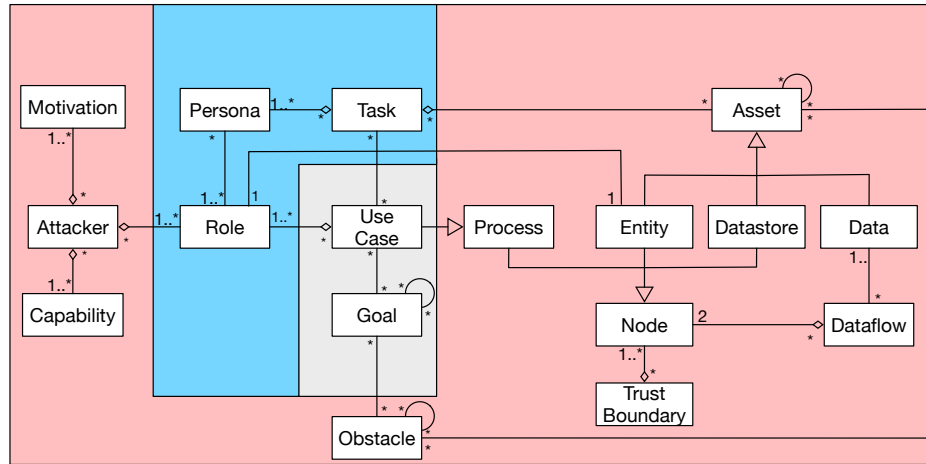
Fig. 1. A UML class diagram showing the IRIS concepts related to threat modelling (red), usability (blue) and requirements modelling (grey)

Coles et al. [2] demonstrated how use cases and assets provide the concepts necessary to threat model with data flow diagrams, and how – in addition to modelling system goals – the KAOS modelling language [12] is also suitable for modelling attack trees as obstacles. To make attacker assumptions more explicit,

IRIS supports the specification of attackers. Attackers need not be intrinsically malicious, but they will have some motivations as drivers for carrying out an attack, and capabilities that provide the knowledge and resources necessary to mount and sustain any threat. IRIS draws its taxonomy of motivations from [13], and capabilities from [10]. An additional motivation of productivity was also added to better reflect non-malicious attackers who intentionally or unintentionally commit harm to get their job done.

To leverage the outputs of user research in security design, two popular usability modelling concepts are supported by IRIS. Personas are specifications of archetypical user behaviour [3]; they not only capture user goals and expectations, but their construction and usage helps elicit security requirement [8]. Tasks are narrative scenarios that describe both the personas and the broader system – including use cases – in context.

3 Approach

Our approach focuses on how *tainted* data flows cast doubt on the safety of the data they carry. Unlike traditional taint analysis on program source code, the origins of data flow taint in our approach could be human error resulting from human entities and processes, or issues resulting from the DFDs and associated specifications. These problems could have an indeterminate impact on affected endpoints, thereby warranting further investigation. Aligning DFDs with usability and requirements models provides context to assist such an investigation.

Assuming the pre-requisite models exist, our approach validates them using the analysis checks described in Section 3.2. Because of its alignment with the DFD concepts as shown in Figure 1, our approach relies on the IRIS meta-model. DFD processes are analogous with use cases, and actors in use cases could be human or system entities. DFDs directly link to usability models because use cases, as processes, put tasks in context. DFDs are also indirectly linked because roles constituting use case actors are also fulfilled by personas – who interact in tasks – putting these roles in context.

3.1 Dataflow specification

DFDs are graphs, but can be specified as a set of data flow types. In our approach, a data flow consists of a label, names of the DFD elements data flows from and to, and the types of these elements, where *NODE* is either an *entity*, a *process*, or a *datastore*. Data flows also specify the information assets (as *DATA*) they carry. Using Z [22], we can express a data flow formally, where the predicate part of the schema contains the well-formedness constraints:

```
┌─ DataFlow ─────────────────────────────────────────
│  label, from, to : STRING
│  fromType, toType : NODE
│  assets : ℙ DATA
├────────────────────────────────────────────────────
│  assets ≠ ∅
│  ((fromType = entity) ∧ (toType = process)) ∨
│  ((fromType = process) ∧ (toType = entity)) ∨
│  ((fromType = datastore) ∧ (toType = process)) ∨
│  ((fromType = process) ∧ (toType = datastore)) ∨
│  ((fromType = process) ∧ (toType = process))
└────────────────────────────────────────────────────
```

3.2 Pre-Process and Post-Process analysis

For each entity in the DFD, our approach first visits the entity's data flows using the *dataFlows* recursive graph traversal function described in Algorithm 1. The function populates a persistent array of unique data flow sequences (*allSeqs*), and a persistent set of previously visited DFD elements (*visited*).

Algorithm 1: Identification of data flows

Input : currentNode - *NODE*, prefix - seq *DataFlow*
Data: allSeqs - seq(seq *DataFlow*), visited - ℙ *NODE*, nodeFlows - ran *Node* ↔ *DataFlow*
1 **Function** *dataFlows(currentNode, prefix)* **is**
2   *visited*.add(*currentNode*);
3   *dfs* ← nodeFlows *currentNode*;
4   **if** *dfs* = ∅ **then**
5   **if** *prefix.length* > *0* **then**
6   *allSeqs*.append(*prefix*);
7   **end**
8   **else**
9   **while** *df* ← *dfs* **do**
10   *newPrefix* ← *prefix*;
11   *newPrefix*.append(*df*);
12   **if** *df.to* ∈ *visited* **then**
13   *allSeqs*.append(*newPrefix*);
14   **else**
15   dataFlows *df.to newPrefix*;
16   **end**
17   **end**
18   **end**
19   **return**;
20 **end**

Each sequence in *allSeqs* is then enumerated to identify and log potential data *pre-process* and *post-process* taint as described in Algorithm 2. The types mentioned in the algorithm can be found in Figure 1, with the exception of *VALUE*, where *VALUE* ::= *Low* | *Medium* | *High*.

Pre-process taint checks (lines 3–15) identify instances where means, motives, and opportunity are present for human errors and violations. The checks are performed on data flows going from human entities to processes contextualised as tasks; these processes are use cases linked to tasks as indicated in Figure 1. Tasks become a possible source of human error when three conditions hold. First, roles fulfilled by personas in a task are shared with roles fulfilled by attackers. Second,

---

Algorithm 2: Taint analysis

Input : dfSeq - seq *DataFlow*

Data: contextualisedTask - ran *UseCase ↔ Task*, taskAsset - ran *Task ↔ Asset*,
personaRoles - ran *Persona ↔ Role*, taskPersonas - ran *Task ↔ Persona*,
roleAttackers - ran *Role ↔ Attacker*, allAttackerRoles - ran *roleAttackers~*,
attackerMotivation - ran *Attacker ↔ Motivation*, attackerCapability -
ran *Attacker ↔ Capability*, taskDemand - ran *Task ↔ Value*, goalConflict -
ran *Task ↔ Value*, processExceptions - ran *UseCase ↔ Obstacle*, obstructedGoals -
ran *Obstacle ↔ Goal*, obstacleAssets - ran *Obstacle ↔ Asset*, nameToProcess -
String ↛ UseCase, logPreProcessTaint - logs taint to process resulting from named
task, logPostProcessTaint - logs taint to process resulting from named obstructed goal

1 **Function** *analyseDataFlows(dfSeq)* **is**

2   **while** *df ← dfSeq* **do**

            /* Check for pre-process taint   */

3   **if** *df.fromType = entity ∧ df.toType = process ∧ df.fromName ∈ Role* **then**

4   **while** *t ← contextualisedTask (nameToProcess df.toName)* **do**

5   **if** *df.assets ∩ taskAssets t* **then**

6   **while** *r ← (personaRoles (taskPersonas t) ∩ allAttackerRoles)* **do**

7   **while** *a ← roleAttackers r* **do**

8   **if** *(Productivity ∈ attackerMotivation a) ∧ (Low Time ∈ attackerCapability a) ∧ ((taskDemand t ∩ {Medium,High}) ∨ (goalConflict t ∩ {Medium,High}))* **then**

9 | logPreProcessTaint (nameToProcess *df.*toName) *t*;

10   **end**

11   **end**

12   **end**

13   **end**

14   **end**

15   **end**

            /* Check for post-process taint   */

16   **if** *df.fromType = process* **then**

17   **while** *o ← processExceptions df.fromName* **do**

18   **if** *(obstacleAssets o ∩ df.assets) ≠ ∅* **then**

19   **while** *g ← obstructedGoals o* **do**

20   **if** *isObstacleObstructed o = true* **then**

21 | logPostProcessTaint (nameToProcess *df.*fromName) *g*;

22   **end**

23   **end**

24   **end**

25   **end**

26   **end**

27   **end**

28   **return**;

29 **end**

---

attackers have a non-malicious motive and are constrained in the means available; we define such attackers as motivated by productivity and, as a capability, a limited amount of time. Finally, affected tasks are demanding to the affected personas, or in tension with their personal goals.

Post-process taint checks (lines 16–26) identify instances where exceptions resulting from processes are unresolved, and these exceptions impact information flowing from processes. Exceptions are modelled as obstacles obstructing one or more system goals operationalised as the affected processes. An obstacle impacts an out-going data flow if assets associated with the obstacle intersect with information assets in the data flow. An exception is unresolved if these obstacles are not resolved by another goal, as determined by the *isObstacleObstructed* function defined in Algorithm 3. It begins by determining whether the input obstacle has been resolved by another goal. After evaluating whether the obstacle has been resolved, the check enumerates both obstacles that are or-refined and and-refined. In the case of or-refined obstacles, an obstruction on *any* of the refined obstacles is enough to consider the obstacle obstructed. Conversely, in the case of and-refined obstacles, an obstruction is present only if *all* refined obstacles are obstructed.

Algorithm 3: isObstacleObstructed check

Data: resolvedObstacles - ran $Obstacle \leftrightarrow Goal$, orRefinedObstacles -
ran $Obstacle \leftrightarrow Obstacle$, andRefinedObstacles - ran $Obstacle \leftrightarrow Obstacle$
Input : o - the obstacle name
Output: isObstructed - indicates if obstacle o is obstructed

1 **Function** *isObstacleObstructed(o)* **is**
2      $ros \leftarrow$ resolvedObstacles o;
3   **if** $ros \neq \varnothing$ **then**
4   isObstructed $\leftarrow$ false;
5   **else**
6          $obs \leftarrow$ orRefinedObstacles o;
7   **while** $oro \leftarrow obs$ **do**
8              $isObstructed \leftarrow$ isObstacleObstructed oro;
9   **if** $isObstructed = true$ **then**
10   break;
11   **end**
12   **end**
13          $obs \leftarrow$ andRefinedObstacles o;
14   **while** $aro \leftarrow obs$ **do**
15              $isObstructed \leftarrow$ isObstacleObstructed aro;
16   **if** $isObstructed = false$ **then**
17   break;
18   **end**
19   **end**
20   **end**
21   **return** *isObstructed*;
22 **end**

3.3 Implementation

We have demonstrated the feasibility of our approach by implementing it in CAIRIS release 2.3.3. CAIRIS (Computer-Aided Integration of Requirements

and Information Security) is an open-source software platform for eliciting, specifying and validating secure and usable system specifications [6] developed as an exemplar for IRIS tool-support.

CAIRIS models, once imported into the platform, are implemented as relational databases. Graphical models in CAIRIS are automatically generated using a pipeline process, where a declarative model of graph edges is generated by CAIRIS; this is processed and annotated by graphviz [5] before being subsequently rendered as SVG. SQL stored procedures implement a suite of security and privacy model validation checks. Algorithms 1 - 3 were implemented as SQL stored procedures; these are executed during a normal model-validation check. No changes were made to pre-existing visual models and the IRIS meta-model.

4 Pilot Study: Modifying telemetry outstation software

We used our approach to identify process taint in a partial specification of a software repository for industrial control software. While based on a hypothetical water treatment company, this anonymised specification is drawn from a more complete specification model created for a UK water treatment company. The CAIRIS model[5] of this partial specification consists of 1 attacker, 1 role, 1 persona, 1 task, 1 use case, 28 goals, 17 obstacles, 58 goal and obstacle associations, 11 assets, 11 asset associations, and 7 data flows. Creation of the model is not the subject of this paper, but further details of how the broader model was created are provided in [9].

The specification captures the system goals and complementary model elements associated with modifying software running on telemetry outstations. Such outstations provide the means for remotely monitoring and controlling physical infrastructure such as water pumps. Malicious tampering of such outstations contributed to the well publicised Maroochy Water Breach [18].

Dataflow	Assets
job	Job
software (to Sandbox)	Telemetry Software File
software (from Sandbox)	Telemetry Software File
updated software	Telemetry Software File
current software	Telemetry Software File
alarm	Alarm
update	Software Change

Table 1. Dataflows and assets

Id	Sequence	Pre-Proc.	Post-Proc.
1	⟨job, alarm⟩	✗	✗
2	⟨job, update⟩	✗	✓
3	⟨job,updated software,current software⟩	✗	✓
4	⟨job,software, software⟩	✗	✓
5	⟨current software⟩	✓	✓

Table 2. Dataflow sequences and results of pre-process and post-process taint checks

Our pilot study considers the impact of human error by an overworked technician focusing on the intricate task of updating software on telemetry outstations

---
[5] Available from `https://doi.org/10.5281/zenodo.3872071`

(Outstation update). This task puts in context the use case Modify Telemetry Software as shown in Figure 2 (top), which is carried out by an instrument technician persona (Barry). Details of how the persona and tasks were constructed are described in more detail in [8]. The task model provided the context necessary to model the DFD generated by CAIRIS in Figure 2 (bottom). Table 1 specifies the assets carried in each data flow.
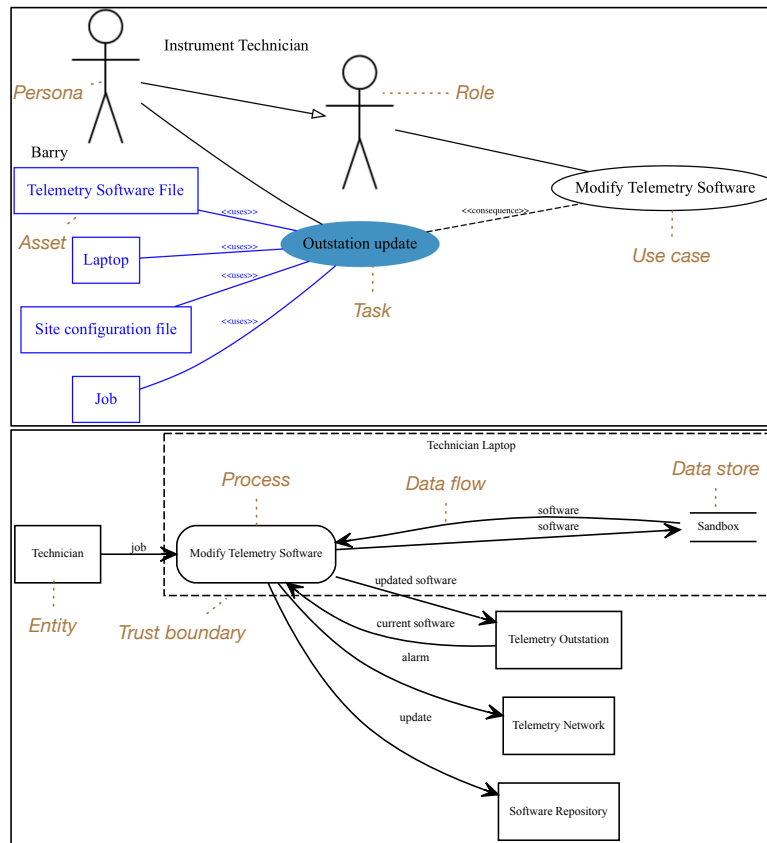

Fig. 2. Usability model (top) and DFD (bottom) of Modify Telemetry Software generated by CAIRIS

Not shown in the visible models is an attacker (Unintentional Barry). This attacker's motivation and resources are specified as 'Productivity' and 'Low Resources/Personnel and Time' to reflect non-malicious intent and a busy schedule. The task model also indicates the assets that Barry directly or indirectly interacts with in completing this task. The relationship between these and other assets associated with the specification are shown in Figure 3 (right).
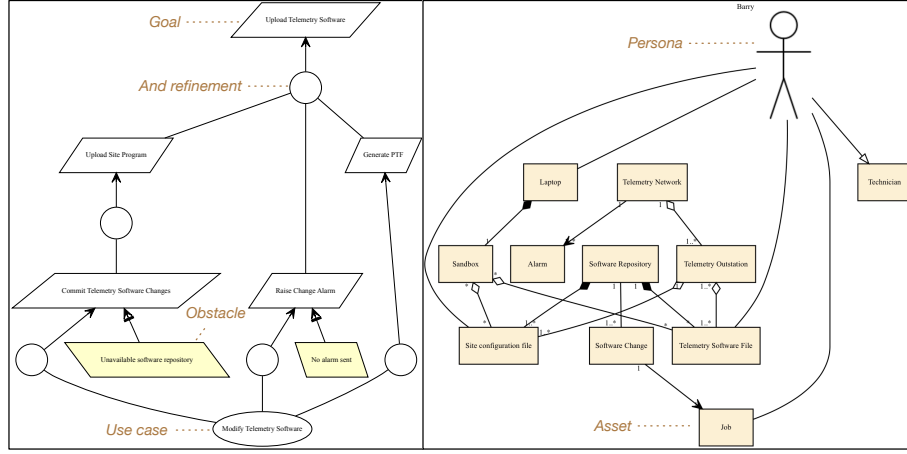
Fig. 3. Complementary KAOS goal model (left) and UML class diagram-based asset (right) model generated by CAIRIS

On performing a model validation check, five unique sequences of data flows were generated as shown in Table 2. The check indicates pre-process taint associated with sequences 1, 2, 3, and 4 resulting from the flow between the technician and the process. This was due to the *job* flow carrying alarm information associated with the task and the potential for error. The task narrative describes how Barry needs to raise an alarm to validate the setup is correct; the alert draws attention to the implications of not safeguarding this information asset.

The model validation check also indicates post-process taint associated with Sequence 1; this outgoing process flow carries alarm information. An exception is associated with the second step of the process, where the system sends a change alarm. As a cut of the goal model in Figure 3 (left) shows, the associated obstacle remains unresolved and, although not visible, the obstacle is concerned with the alarm asset carried in *job*.

5   Discussion and Conclusion

This short paper showed how, by putting DFDs in context, we can identify process taint without changing any DFD semantics. CAIRIS demonstrates the feasibility of our approach, but it could be adapted to any inter-operable combination of tools. Solutions for resolving the problems are not prescribed besides changing the attacker model and tasks, or resolving exceptions. However, by indicating otherwise invisible problems, our approach sheds light on why problems exists, and how a system or its context of use might need to change to address them. This approach is contingent on specifications containing the concepts in Figure 1 that might be created before, during, or after DFD creation. Small or poorly resourced teams may lack the resources to maintain such models

given the user research investment required. However, this approach does allow human factor experts to become more engaged with threat modelling. We are currently working with system engineering teams with such expertise to evaluate the impact this approach has on increasing such engagement.

A threat to validity is the small size of the pilot study specification. However, we have also evaluated our approach using a more complex military medical evaluation system model described in [11] consisting of 10 attackers, 14 roles, 9 personas, 12 tasks, 29 use cases, 46 goals, 25 obstacles, 167 goal and obstacle associations, 82 assets, 388 asset associations, and 134 data flows. No differences in model validation performance were noted for this larger model, but a detailed evaluation of this and other larger models will be the subject of future work.

Our approach only considers non-malicious attackers engaging in difficult tasks. However, Algorithm 2 can be extended to consider alternative attacker and task attributes corresponding with different means, motives, and opportunities. For example, an inside attacker might be motivated by improved esteem or thrill seeking, and participate in tasks with differing levels of goal conflict.

References

1. Antignac, T., Scandariato, R., Schneider, G.: Privacy compliance via model transformations. In: Proceedings of the 2018 IEEE European Symposium on Security and Privacy Workshops. pp. 120–126 (April 2018)
2. Coles, J., Faily, S., Ki-Aries, D.: Tool-Supporting Data Protection Impact Assessments with CAIRIS. In: Proceedings of the 5th International Workshop on Evolving Security & Privacy Requirements Engineering. pp. 21–27 (2018)
3. Cooper, A., Reimann, R., Cronin, D., Noessel, C.: About Face: The Essentials of Interaction Design. John Wiley & Sons (2014)
4. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM **19**(5), 236âĂŞ243 (May 1976)
5. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz— open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) Graph Drawing. pp. 483–484. Springer (2002)
6. Faily, S.: CAIRIS web site. `https://cairis.org` (April 2018)
7. Faily, S.: Designing Usable and Secure Software with IRIS and CAIRIS. Springer, 1st edn. (2018)
8. Faily, S., Fléchais, I.: Barry is not the weakest link: Eliciting secure system requirements with personas. In: Proceedings of the 24th BCS Interaction Specialist Group Conference. pp. 124–132. BCS '10, British Computer Society (2010)
9. Faily, S., Fléchais, I.: Towards tool-support for Usable Secure Requirements Engineering with CAIRIS. International Journal of Secure Software Engineering **1**(3), 56–70 (July-September 2010)
10. Jones, A., Ashenden, D.: Risk management for computer security : Protecting your network and information assets. Elsevier Butterworth-Heinemann (2005)

11. Ki-Aries, D., Faily, S., Dogan, H., Williams, C.: Assessing System of Systems Security Risk and Requirements with OASoSIS. In: Proceedings of the 5th International Workshop on Evolving Security & Privacy Requirements Engineering. pp. 14–20 (2018)
12. van Lamsweerde, A.: Requirements Engineering: from system goals to UML models to software specifications. John Wiley & Sons (2009)
13. Van der Linden, M.A.: Testing code security. Auerbach Pub (2007)
14. Matulevičius, R.: Fundamentals of Secure System Modelling. Springer (2017)
15. Shostack, A.: Threat Modeling: Designing for Security. John Wiley & Sons (2014)
16. Simon, H.A.: Rational decision making in business organizations. The American Economic Review **69**(4), 493–513 (sep 1979)
17. Sion, L., Yskout, K., Van Landuyt, D., van den Berghe, A., Joosen, W.: Security Threat Modeling: Are Data Flow Diagrams Enough? In: Proceedings of IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20). IEEE (2020), to Appear
18. Slay, J., Miller, M.: Lessons learned from the maroochy water breach. In: Goetz, E., Shenoi, S. (eds.) IFIP series in Critical Infrastructure Protection, vol. 253, pp. 73–82. Springer (2007)
19. Tuma, K., Scandariato, R., Balliu, M.: Flaws in flows: Unveiling design flaws via information flow analysis. In: Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA). pp. 191–200 (2019)
20. Tuma, K., Kalikli, G., Scandariato, R.: Threat analysis of software systems: A systematic literature review. Journal of Systems and Software **144** (2018)
21. Tuma, K., Scandariato, R., Widman, M., Sandberg, C.: Towards security threats that matter. In: Katsikas, S.K., Cuppens, F., Cuppens, N., Lambrinoudakis, C., Kalloniatis, C., Mylopoulos, J., Antón, A., Gritzalis, S. (eds.) Computer Security. pp. 47–62. Springer (2018)
22. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall (1996)
23. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 116–127. Association for Computing Machinery (2007)
24. Yourdon, E., Constantine, L.L.: Structured design: fundamentals of a discipline of computer program and systems design. Prentice Hall (1979)