Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security

Marwan Abi-Antoun
Carnegie Mellon University
mabianto@cs.cmu.edu

Daniel Wang*
Microsoft Corporation
daniwang@microsoft.com

Peter Torr
Microsoft Corporation
ptorr@microsoft.com

September 2006
CMU-ISRI-06-124
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Center for Software Excellence

Abstract

Threat modeling analyzes how an adversary might attack a system by supplying it with malicious data or interacting with it. The analysis uses a Data Flow Diagram (DFD) to describe how data moves through a system. Today, DFDs are represented informally, reviewed manually with security domain experts and may not reflect all the entry points in the implementation.

We designed an approach to check the conformance of an implementation with its security architecture. We extended Reflexion Models to compare as-built DFD recovered from the implementation and the as-designed DFD, by increasing its automation and thus its adoptability. We also designed an analysis to assist DFD designers validate their initial DFDs and detect common security design flaws in them.

An evaluation of the approach on subsystems from production code showed that it can find omitted or outdated information in existing DFDs.

Parts of this work were conducted while the first author was an intern in the Center for Software Excellence at Microsoft. Abi-Antoun's work is supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

Keywords: Threat modeling, data flow diagrams, reflexion models, architecture-level security analysis, spoofing, tampering, information disclosure, denial of service

1 Introduction

Security researchers have long recognized that secure software designs are those that make their assumptions explicit, and called for methodologies to help a development team examine its assumptions in a systematic manner [2].

For several years, companies such as Boeing, Microsoft, and others have been using *threat modeling* [43, 44] as a lightweight approach to reason about security, to capture and reuse security expertise and to find security design flaws during development [21, 28]. Threat modeling looks at a system from an adversary's perspective to anticipate security attacks and is based on the premise that an adversary cannot attack a system without a means of supplying it with data or otherwise interacting with it. Documenting the system's *entry points*, i.e., interfaces it has with the rest of the world, is crucial for identifying possible vulnerabilities.

Threat modeling adopts a data flow approach to track the adversary's data and commands as they are processed by the system and determine what assets can be compromised. Any transformation or action on behalf of the data could be susceptible to security threats. For instance, an adversary can jeopardize the application's security by directly invoking some functionality or by supplying the application with malicious data, e.g., by editing a configuration file.

A Data Flow Diagram (DFD) [49] with security-specific annotations is used to describe how data enters, leaves and traverses the system: it shows data sources and destinations, relevant processes that data goes through and trust boundaries in the system. There are many modeling techniques to reason about security more formally [5]. The current threat modeling process uses DFDs [21], perhaps because they are informal, they have a graphical representation, and they are hierarchical, thereby supporting modular decomposition [27]. One large project at Microsoft has over 1,400 completed and reviewed threat modeling DFDs, so we needed a semi-automated approach to support and enhance the current threat modeling process.

DFDs are typically constructed by development teams and later reviewed by security experts. In fact, it is not uncommon for a developer to produce during a security review meeting a DFD of how they think the system works. In many cases, the mere act of having application developers specify the DFD for a subsystem can make several of its security flaws apparent. Although this approach has found a significant number of security design flaws, it suffers from the following two important problems.

First, the DFD used in threat modeling (the as-designed DFD) is often documenting how its author thinks the system works and not how the system actually works (the as-built DFD). The security analysis is less robust if there are entry points in the implementation that are not represented in the as-designed DFD, or if an attacker can force new data flows or bypass existing data flows by calling a different API or taking actions in an unexpected order. Our first contribution is a semi-automated approach to check the conformance between an as-built DFD and an as-designed DFD (Section 2). Our tool identified interesting discrepancies between several DFDs and the implementation (Section 3).

Second, DFDs are initially produced by application experts who have little experience in threat modeling, may not be aware of the rules that good threat modeling DFDs should obey or may not apply them in a systematic way. Furthermore, building high-quality DFDs requires security knowledge. Because teams do not receive feedback from security experts until late in the process, these initial DFDs are often of low quality. High-quality DFDs are also annotated with security relevant properties, but knowing what to annotate with what information typically requires threat modeling experience. We implemented an analysis to assist novice DFD designers by warning about possible threats and providing hints for mitigation. Checking DFDs under development found anomalies ranging from failed sanity checks to missing critical security information.

The rest of the report is organized as follows. Section 2 discusses an analysis to check the conformance between an as-built DFD and an as-designed DFD. Section 3 validates the approach on real DFDs and real implementations. Section 4 discusses analyzing a DFD for well-formedness and common security flaws. Section 5 discusses some limitations and future work. Finally, Section 6 surveys related work.

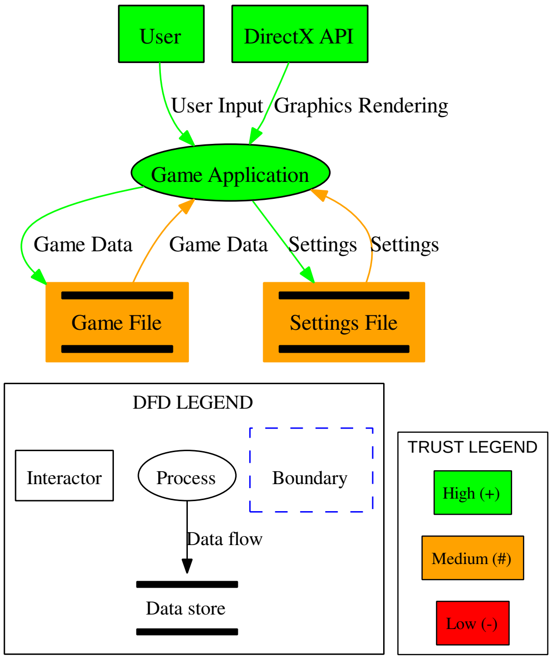

Figure 1: Minesweeper as-designed DFD.

2 Implementation Conformance

To check conformance, the as-built DFD is extracted from the implementation and compared with the as-designed DFD using extensions of the Reflexion Models technique [35]. In the latter, a *source model* is extracted using a third-party tool from the source code. The user posits the as-designed *high-level model* and then a *mapping* between the source and high-level models. Pushing each interaction described in the source model through the map produces *inferred* edges between high-level model entities. A computation then compares the inferred edges with the edges stated in the high-level model to produce the *reflexion model*. The user can modify the high-level model, the source model, or the mapping to iteratively compute additional reflexion models.

2.1 The High-Level Model: the Data Flow Diagram (DFD)

A Data Flow Diagram (DFD) for Minesweeper, a game that ships with Windows^{TM}, is shown in Figure 1. Circles represent logical processes within the subsystem being modeled; doublecircles represent processes refined into other DFDs; rectangles represent external entities; double horizontal lines represent passive data stores; and arrows represent data flows. A data flow represents the flow of information from one entity to another; ideally, it is labeled with an optional sequence number and the information being sent, e.g., "1 : Logon Information".

Data flows should not be labeled with "Read" or "Write". A large number of discrete data items (e.g., username; password; server name; authentication scheme) can be consolidated into a single label (e.g., "Logon Information"). Data flows are always unidirectional; if data flows back and forth between two entities on the diagram, two separate flows are typically shown.

We represent a DFD as a runtime view following the Component-and-Connector viewtype [9, pp. 364–365]. A DFD has a fixed set of component types: Process, HighLevelProcess, DataStore, and ExternalInteractor [43]. A Process represents a task in the system that processes data or performs some action based on the data. A DataStore represents a repository where data is saved or retrieved, but not changed. Examples of data stores include a database, a file, or the Registry — a database of configuration settings in Windows

```
<dfdmodel>
  <settings> <!-- Match rules, function groups, and conformance exceptions -->
    <matchrule binary="minesweeper.exe" regex=".+" shapeType="Process" name="Game Application" />
    <matchrule binary="advapi32.dll" regex="Reg.+" shapeType="DataStore" name="Registry" />
    <matchrule binary="kernel32.dll" regex="CreateFile.+" shapeType="DataStore" name="Game File" />
    ...
  </settings>
  <dataflowdiagram>   <!-- As-designed DFD without any traceability -->
    <process id="1" name="Game Application" trustLevel="High" />
    <datastore id="2" name="Game File" trustLevel="Medium" />
    <interactor id="4" name="User" trustLevel="High" />
    ...
  </dataflowdiagram>
  <reflexionmodel> <!-- Results of conformance checking of as-built and as-designed with traceability -->
    <datastore id="2" name="Game File" finding="Convergent" trustLevel="Medium">
      <service binary="kernel32.dll" function="CreateFile" />
    </datastore>
    <datastore id="97" name="Registry" finding="Divergent" />
    <interactor id="4" name="User" finding="Absent" trustLevel="High" />
    <dataflow id="107" name="Game Data" finding="Convergent" trustLevel="High" source="1" destination="2">
      <detail sourceBinary="minesweeper.exe" sourceFunction="MaX_Unknown"
              destinationBinary="kernel32.dll" destinationFunction="CreateFile" />  </dataflow>
    <dataflow id="108" name="Game Data" finding="Convergent AmbiguousDirection" source="2" destination="1">
      <detail sourceBinary="minesweeper.exe" sourceFunction="MaX_Unknown"
              destinationBinary="kernel32.dll" destinationFunction="CreateFile" />  </dataflow>
      ...
  </reflexionmodel>
</dfdmodel>
```

Figure 2: Extracts from the DFD model for the Minesweeper application.

operating systems. An ExternalInteractor represents an entity that exists outside the system being modeled and which interacts with the system at an entry point: it is either the source or destination of data. Typically, a human who interacts with the system is modeled as an ExternalInteractor. One connector type, DataFlow, represents data transferred between elements.

A DFD used in threat modeling often separates elements that have different privilege levels using a Boundary to describe locations where a privilege impersonation on the part of an adversary could occur, a machine or process boundary may be crossed, etc. [42, p. 90]. A Boundary is modeled as a TrustBoundary (the base type), ProcessBoundary, MachineBoundary or OtherBoundary.

In a hierarchical DFD, a HighLevelProcess is expanded into a lower level DFD. Typically, a *Context Diagram* shows the entire functionality of the system represented as a single node. That node is then broken into multiple elements in a *Level-0* diagram. From there, *Level-1*, *Level-2*, etc., diagrams can be constructed to more precisely model security-critical processing [20, p. 76]. Since developers do not use hierarchical decomposition rigorously [1], we check conformance between two Level-n DFDs, as opposed to starting at the context diagrams of two hierarchical DFDs.

2.2 The Source Model

In our approach, the source model consists of binary dependency information provided by a proprietary framework on Windows, MaX [41]. Similar analyses are available on other platforms, e.g., ATOM [40] on UNIX. Some of the advantages of using binary dependency information include analyzing an system compiled from different high-level languages and shielding the analysis from a complex build process. The disadvantage of not working at the source code level is not having access to information such as the declared types of function parameters and return values.

MaX provides mainly call dependency information but also handles dynamic loading of code and resolves Globally/Universally Unique Identifiers used in the COM and RPC protocols. For each binary, MaX creates a database that includes: (a) Inputs, the services a binary provides as exported functions, COM interfaces, etc.; (b) Outputs, the services that a binary requires from other binaries; (c) NamedObjects, any system-wide entities referenced by name such as Registry entries; (d) CallGraph, the call dependency information; and

(e) Sources, the names to the underlying source files, information that is critical for tracing back from the dependency information down to the source code.

2.3 Mapping from Source Models to High-Level Models

It is common to have for each application scenario or usecase one or more Level-n DFDs. Typically, Level-0 and Level-1 DFDs would be shared across application scenarios, and Level-2 DFDs would be scenario-specific. For instance, a peer-to-peer file sharing application could have four scenarios: a) Send Request; b) Receive Request; c) Send File; and d) Receive File. Each scenario is modeled separately with an as-designed DFD and a mapping from the source model to the as-built DFD. Match Rules and Function Groups, discussed below, map from the source model to the high-level model.

Match Rules. In Reflexion Models, the mapping between the source model and the high-level model is specified entirely each time. In our approach, a reusable catalog of match rules captures, for a given platform, some of the security domain expertise regarding security-relevant APIs for file access, registry access, network access, command-line access, event log access, etc. A reusable catalog of match rules is similar to reusable attack libraries for generating attack graphs [38].

Accessing the Registry has security implications since the registry contains personally identifiable information and since corrupting certain registry keys can render the system unusable. The Registry API consists of several functions in the `advapi32.dll` binary: for instance, `RegOpenKeyEx` opens a registry key, `RegQueryKeyValueEx` retrieves the value of the key, and `RegCloseKey` closes the key. The corresponding Match Rule in the catalog creates a DataStore named "Registry" if there are calls to binary "advapi32.dll" and the function name matches the regular "Reg.+" — taking advantage of naming conventions of the Registry API. When an API does not follow a strict naming convention, individual function names can be used in place of regular expressions.

A given binary hosts many functions so multiple match rules can refer to the same binary. For instance, a similar match rule is created for the same `advapi32.dll` binary and functions that access the system Event Log, e.g., `OpenEventLog`, `CloseEventLog` and `ReadEventLog`.

Function Groups. Match Rules are often sufficient to obtain a Level-1 DFD (See Figure 5). However, obtaining a Level-2 DFD for an application often requires looking at the internals of the application binaries. When threat modeling, the exposed API of a subsystem is broken up into a set of logically-related functions and each set — rather than each individual function, is modeled as an entry point.

A Function Group logically clusters the services provided by a set of binaries into a DFD element. Unlike Match Rules which involve system binaries and are reusable across applications, Function Groups are application- and scenario-specific, so they are added to the DFD model for a given scenario. One function group can involve different binaries as well, since one may want to group in the same logical entry point, a wrapper function defined in a binary that simply calls another function in another binary.

The match rules and function groups produce the nodes of the as-built DFD. The dependency information infers the edges in the as-built DFD. The as-built DFD is then compared against the as-designed DFD using an extension of Reflexion Models. As in Reflexion Models, there is a one-to-one mapping between the nodes of the as-built DFD and the nodes of the as-designed DFD. If a match rule or a function group refers to a node that is not present in the as-designed DFD, that node is automatically added to the as-designed DFD. In addition, Boundary elements defined in the as-designed DFD are automatically added into the as-built DFD since they are not inferred.

Even for an application as simple as Minesweeper, checking the conformance revealed several noteworthy absences: the as-designed DFD (Figure 1) failed to mention two entry points, the Registry and the Game Explorer (See Figure 3). Since Game Explorer enforces parental controls, it must be included as an entry point in the DFD.

2.4 Reflexion Models + Extensions

Our early experiments showed that using the base Reflexion Models technique produced a large number of false positives and required excessive manual input. We needed to extend it make it practical to check

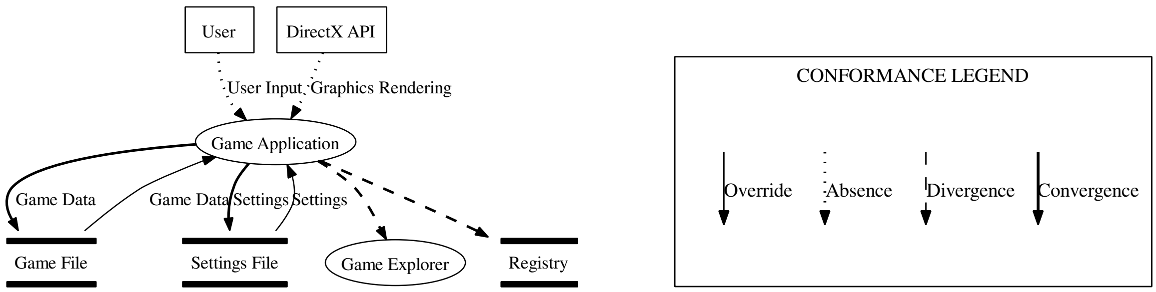

Figure 3: Checking conformance for Minesweeper

existing threat modeling DFDs. Although some of these extensions are specific to DFDs, others are more widely applicable.

First, in Reflexion Models, map entries are considered an *ordered sequence* and not an *unordered set*. As a result, a greedy regular expression placed in one of the early map entries can drastically change the as-built DFD, thus affecting the stability of the map across invocations. Using a set enables a given source model element to map freely to multiple elements in the high-level model.

Second, building a map from scratch each time is a laborious process — in one Reflexion Models case study, the map of one product contained more than 1,000 entries [35, p. 372]. We also found ourselves defining similar maps for different systems. Although our match rules and function groups could produce the same flat map as Reflexion Models, they are more structured and separate the shared parts of the map that are reusable across different applications from the application-specific ones.

Finally, the Reflexion Models approach uses as many as eight different inputs and outputs, which makes the interaction needlessly complex: (1) source model; (2) high-level model; (3) mapping; (4) configuration, i.e., node shapes, colors, etc.; (5) exclusions or exceptions; (6) reflexion model; (7) traceability information; and (8) errors or unmapped entries. In later implementations of the tool [22], some of these inputs and outputs were unified, e.g., Inputs 2,3,4.

The computation of the Reflexion Model for DFDs considers the node type defined in the high-level model, and thus requires a more unified representation. This change and our other requirements, such as having traceability in the high-level model, resulted in the integrated representation shown in Figure 9: BasicEntity, the base component type, stores the extended Reflexion Model finding in the finding property and the traceability information in the services property.

2.5 Conformance Findings

In Reflexion Models, an edge can be any of the following:

- *Convergent*: if it is both in the as-built DFD and in the as-designed DFD;
- *Divergent*: if it is in the as-built DFD , but not in the as-designed DFD;
- *Absent*: if it is in the as-designed DFD, but not in the as-built DFD.

We defined the following additional classifications:

- *Excluded:* Self-edges are prohibited in a threat modeling DFD. To reduce the number of false positives, a self-edge that is automatically inferred is automatically marked as *Excluded*.
- *AmbiguousDirection:* in some cases, the analysis cannot determine the directionality of a data flow. To reduce the number of false positives, if an edge exists but its direction cannot be determined, the analysis automatically accepts it as a convergence, but flags it differently to distinguish it from a true convergence. An edge with an ambiguous direction is shown as a solid edge, as opposed to a bold edge for a true convergence. An example of such a finding is the "Game Data" DataFlow directed from the "Game File" DataStore to the "Game Application" Process in Figure 3.
- *ExternalEntity:* to reduce the number of false positives, we extended Reflexion Models such that the node types in the high-level model can affect the mapping from the source model to the high-level

model entities. A DFD does not show a DataFlow between an ExternalInteractor and a DataStore or another ExternalInteractor since those would belong to the DFD of the ExternalInteractor. Such an edge, when inferred, is automatically classified as *ExternalEntity*.

Without these extensions, the base Reflexion Models technique generated a large number divergences and absences and would have required excessive manual user input to account for false positives, to the point of making a semi-automated approach impractical.

Conformance Exceptions. We generalized Reflexion Models annotations [34, pp. 84–88] into conformance exceptions. The user can manually override any finding in the Reflexion Model and specify a reason for the override. In the DFD model (Figure 9), each conformance exception maintains the original finding, the overridden finding and the reason for the override.

2.6 Traceability

Building a threat model is a significant investment that captures security design intent. Because of the earlier informal representation of DFDs and the lack of traceability to the code, much of the security design intent in DFDs had not been easily accessible to other code quality tools. Checking conformance between the as-designed and as-built DFDs produces traceability information that other static analyses can use. For instance, an analysis checking for buffer overruns [19] can use this traceability to assign more appropriate priorities based on a more holistic view of the system.

In Reflexion Models, the traceability information is computed on-demand and discarded between invocations. Our DFD representation adds traceability information to the high-level model.

Traceability information between the as-designed DFD and the as-built DFD is represented using *provided* and *required* services. In procedural code, a service is identified with the pair (binary, name). The binary corresponds to a deployment unit of the application. A name could be a function defined in a binary or a Globally/Universally Unique Identifier used in the Component Object Model (COM) [31] and Remote Procedure Call (RPC) protocols. For instance, the pair (`advapi32.dll`, `RegOpenKeyEx`) identifies the function `RegOpenKeyEx` in the `advapi32.dll` binary.

Each Process, ExternalInteractor and DataStore maintains a list of provided services, in the services property. Each DataFlow shows the services involved in the details property, where a connection binds a required service to a provided service (See Figure 9).

During conformance checking, traceability information from the source model is added to each DataFlow that is not Absent. For each conformance exception, traceability information consisting of the call information is saved with the conformance exception in the DFD representation. This is critical for re-running the analysis for different versions of the application: if there is a conformance exception associated with a computed edge, the call information of the computed edge is compared to the previously saved call information. If a discrepancy is detected between the current traceability information and that previously stored with the conformance exception, the exception is flagged as suspicious.

Conformance checking is an iterative process. Missing elements may be added to the as-designed DFD. The construction of the as-built DFD may be modified by changing the mapping of the implementation to the design using match rules and function groups. Innocuous divergences and absences can be documented using conformance exceptions. In a few cases, any code that seriously violates the security architecture may need to be changed. Finally, a new security review may be required if there are many unexpected differences between the as-built and the as-designed DFDs.

3 Evaluation

3.1 Tool Support

We implemented a tool, DFD Studio, to check the conformance of DFDs and analyze them for security design flaws. DFD Studio also produces a graphical display of the DFD representation using GraphViz [15] (Figure 4-C). Except for Figure 6, all diagrams in this report were obtained using DFD Studio.

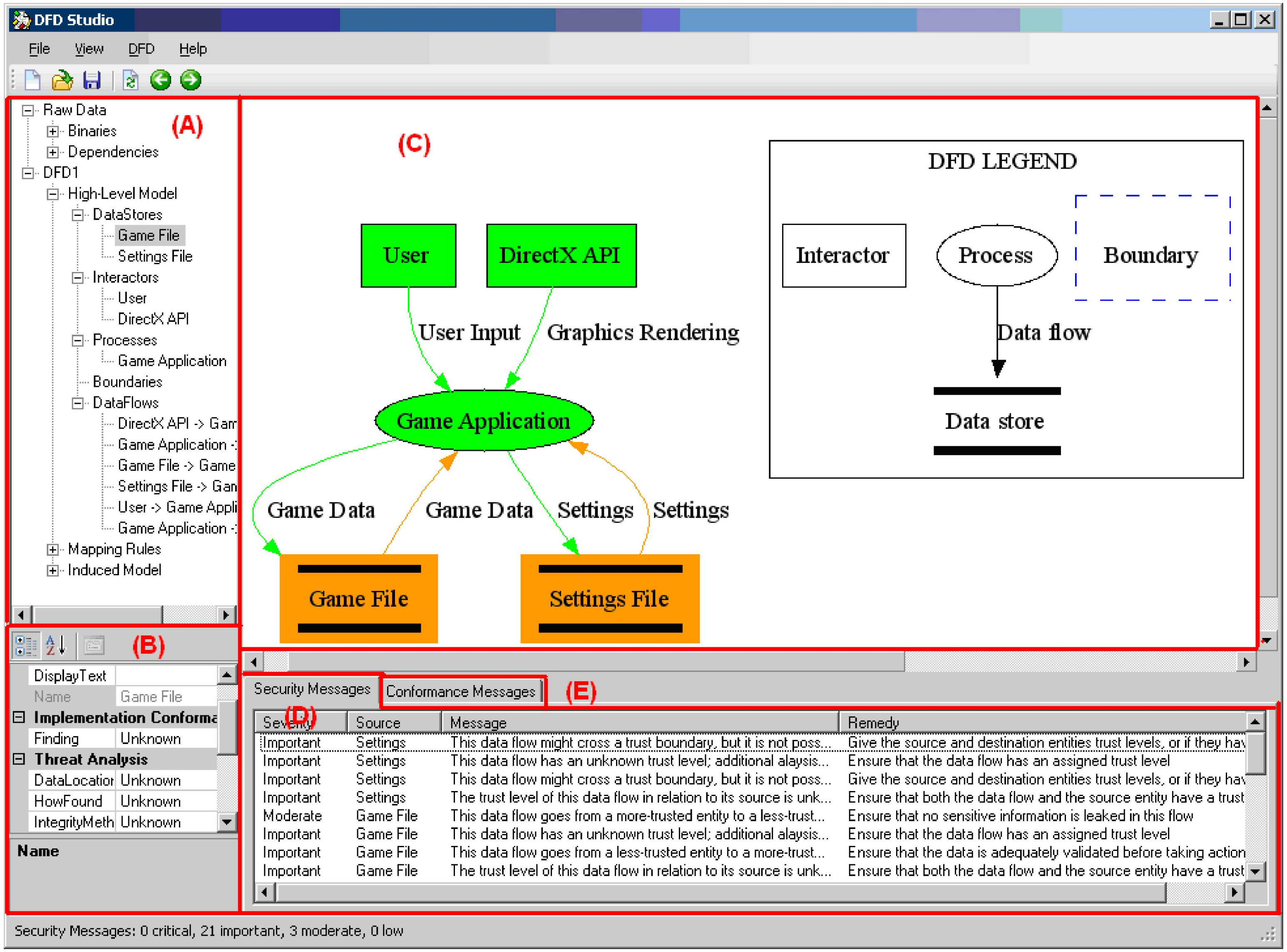

Figure 4: Screenshot of the DFD Studio tool.

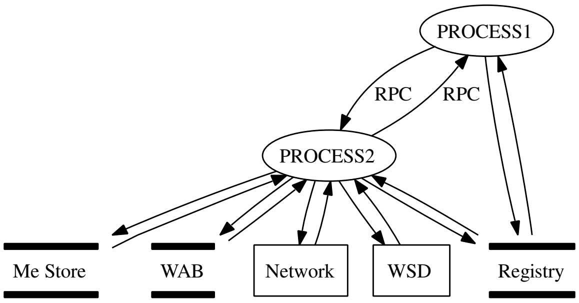

Figure 5: As-built Level-1 DFD for System A.

DFD Studio allows the user to load an as-designed DFD; specify the application binaries; import selected match rules from a shared catalog, or optionally define new ones; and cluster interactively the entry points of the selected application binaries into function groups. The DFD settings are then saved in the same file as the as-designed DFD.

If the as-designed DFD is not specified, DFD Studio can infer the Observed DFD based on the match rules and function groups. If a Target DFD is specified, the user can run the conformance checking tool. The tool displays the results of the conformance checking graphically using the conventions shown in the legend below Figure 3.

The results of running the conformance checking, including any warnings related to suspicious conformance exceptions, are shown in the "Conformance Messages" window (Figure 4-E) and in the status bar. The user can interactively select a data flow and view its traceability to the code. Finally, the user can manually override a Reflexion Model finding by defining a conformance exception.

The Properties window (Figure 4-B) allows the user to set the various security properties for the selected element under the "Threat Analysis" heading. The user can override the conformance findings for the selected element under the "Implementation Conformance" heading. The results of the security analysis discussed in Section 4 are shown in the "Security Messages" window (Figure 4-D) and in the status bar.

One large project at Microsoft already has over 1,400 completed DFDs in Microsoft Visio — a diagramming software that exposes a programmable interface. Since many of existing DFDs use a common template, the one used by the threat modeling tool [42], we implemented a Visio plug-in to convert these as-designed DFDs into the proposed DFD representation.

3.2 System A

This case study illustrates the role of an application developer interacting with security domain experts. The subject system — referred to herein as System A — is part of a large operating system under development.

As-designed DFDs. First, we obtained a set of DFDs for System A, including a Context Diagram and several Level-2 DFDs. The list of binaries for System A was obtained from a central repository which records all the binaries that constitute a given subsystem.

As-built Level-1 DFD. Using only match rules from the catalog, an as-built Level-1 DFD was easily derived (See Figure 5): it was slightly more detailed than the documented DFD but did not hold any surprises.

As-built Level-2 DFD. We then investigated a Level-2 DFDs for an application scenario (Figure 6). To produce a Level-2 DFD, function groups had to be defined for the binaries involved. Interactively clustering entry points can be tedious and require application knowledge — which the person conducting the evaluation did not have, so the initial set of function groups was obtained from the Application Module Definition (DEF) files for System A's binaries.

A DEF file lists all the exports of a given library, and typically clusters them by logical groups with optional comments describing the set of functions. Excerpts from the `advapi32.dll` DEF file are shown below (lines preceded by a semi-colon are comments):

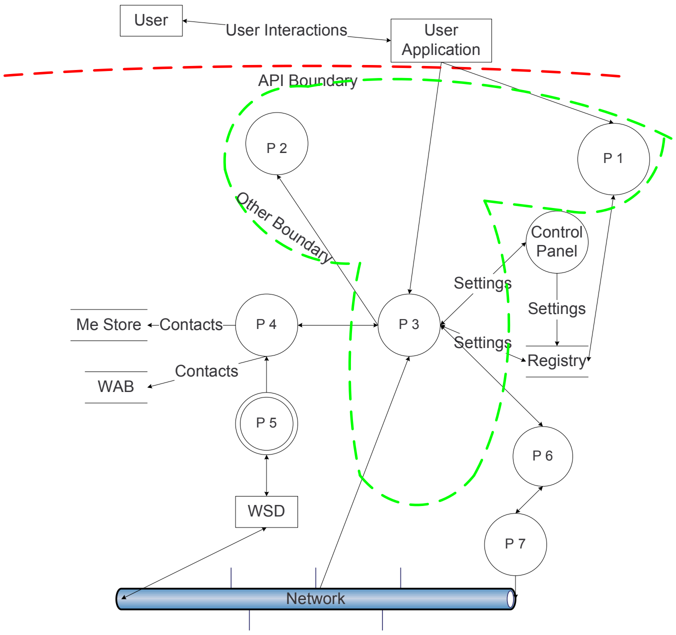

Figure 6: As-designed Level-2 DFD for a scenario of System A: some labels are obfuscated.

```
LIBRARY  ADVAPI32
EXPORTS
; Eventlog functions
    OpenEventLog
; Registry functions
    RegOpenKeyEx
...
```

It would not be hard to have a tool parse these files and obtain default function groups. Alternative clustering information could be provided using an external file. Initial clustering information could also be potentially extracted from the directory structure in the source tree.

DFD Studio was then used to check the conformance between the as-designed and the as-built Level-2 DFDs. The first few attempts yielded a large number of divergences and absences (11 convergences, 44 divergences and 30 absences), which was unsurprising since the function groups were designed without any developer input. As is often the case in architectural recovery, talking to the developers can help better map the source model to the high-level model.

Meeting with Security Experts. The security experts we met with were of the opinion that the as-designed Level-2 DFD was too fine-grained. As a rule of thumb, a DFD should have around half a dozen nodes, and the DFD under study had many more than that.

Meeting with Developers. We met with the developer – also the author of the Level-2 DFD, and discussed with him the conformance findings. This made us realize that the as-designed Level-2 DFD was exposing too many implementation details: in particular, it showed three distinct DFD Process entities when only one was expected.

When the developer built the as-designed Level-2 DFD, he was thinking in terms of the code organization, and the implementation did have three layers. However, the communication between these three layers in the code was not externally accessible. The developer also admitted being unsure about the granularity of a Level-2 DFD. As a result, the three DFD Process entities were merged into one.

The developer confirmed that the information in the DEF file was mostly accurate, but some of the following changes were needed: (a) Merge separate clusters of functions from the DEF file into one DFD

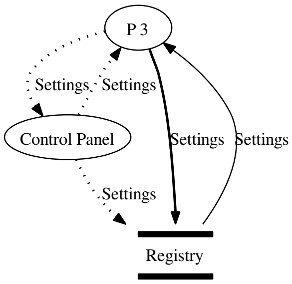

Figure 7: Partial results of checking the conformance of the Level-2 DFD for System A.

Process; (b) Move individual functions between clusters; (c) Hide clusters temporarily if their functions are not exercised in the given scenario; and (d) Exclude clusters containing utility functions that are not entry points, e.g., "Picture API".

The meeting with the developer lasted one hour and was followed up with a few email exchanges with another developer who was responsible for other parts of the code. All absences and divergences were accounted for by making minor modifications to the as-designed DFD, updating the function groups or documenting expected or innocuous differences as conformance exceptions.

Evaluation. During the triage of the conformance findings, we discovered an interesting set of absences between the Process "P3", Process "Control Panel" and the DataStore "Registry" (See Figure 7). The traceability information also identified the originating source files. A visual inspection revealed that Process "P3" was saving its "Settings" in DataStore "Registry" by launching another executable (using the `ShellExecuteEx` system call) with a command-line argument to make it display its "property pages" (tabbed dialogs), which in turn were setting values in the "Registry". The MaX data did show a dependency on `ShellExecuteEx` (the name of the executable that was launched was hard-coded) but did not provide information about the arguments passed to the function call.

Although this conformance finding did not reveal a security flaw in the implementation, the threat model for System A should have at least mentioned how the settings were being saved to abide by the principle of making security assumptions explicit. More importantly, the traceability information enabled us to quickly focus the manual code review on suspected anomalies between the design and implementation rather than potentially the entire code base.

3.3 System B

This case study illustrates the role of a security expert facing an unfamiliar subsystem. Often when a security exploit is discovered in the field, a security expert must quickly understand its impact by examining the threat models. However, many legacy applications have outdated threat models. Although reverse engineering DFDs from implementations can produce abstractions that do not map to the developer's mental model, inferring the as-built DFD from the implementation is implicitly part of conformance checking but can be also useful in its own right when the DFD is missing or outdated. The shared catalog of match rules enables reusing existing security knowledge.

We chose a resolved security bulletin for an implementation of the Domain Name System (DNS) [32] which can suffer from many security vulnerabilities [3]. After we located the relevant binaries, we used the catalog of match rules and added a match rule for each application binary. This enabled inferring an approximation of the as-built DFD in a few minutes (Figure 8). The security experts who examined the as-built DFD agreed that the diagram would be a useful quick approximation when responding to a security bulletin, considering how easily it was produced, and added that it would be easy to manually label the edges. In this case, the threat model referred to binaries that no longer existed.

An unexpected edge between the "DNS API" Process and a Process with the obfuscated label '?' (See Figure 8) warranted an investigation. The traceability information helped us narrow that flow to a specific

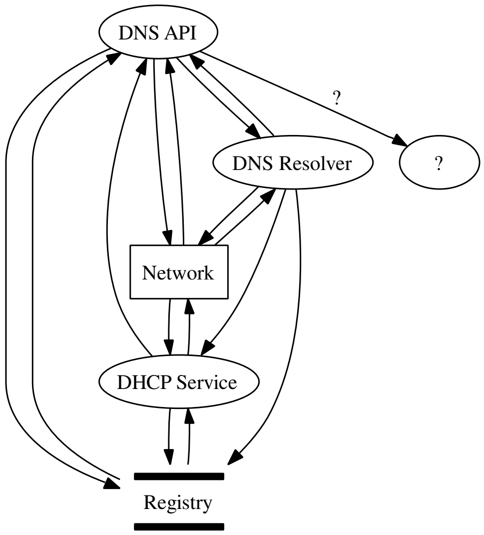

Figure 8: As-built Level-1 DFD for System B.

function call. Upon closer examination, we discovered that the documentation explicitly discouraged the use of that function due to its security loopholes. The appropriate developers were informed of this potential problem and they agreed to take action.

4 Security Analysis

High-quality DFDs are often annotated with security relevant properties, but knowing what to annotate with what information typically requires security modeling experience. We implemented an analysis to check syntactic rules and to look for security flaws at the DFD level. The analysis checks the provided information and automatically requests more information where applicable.

Syntactic Rules. Several structural rules dictate how elements in a DFD may be connected together [20, p. 78]: (1) A Process must have at least one DataFlow entering and one DataFlow exiting; (2) A DataFlow starts or stops at a Process; (3) A DataFlow cannot have the same source and destination; and (4) A DataFlow cannot be bi-directional, since the data in each direction is often different.

Security Properties. Trust boundaries are one kind of security-specific annotations that are added to a DFD. High-quality DFDs are also annotated with security relevant information. We formalized a core set of security properties in the DFD representation. Many of these properties are enumerations with pre-defined values and a default value of Unknown. The complete DFD representation with the conformance findings, traceability to code and security properties is shown in Figure 10.

For instance, all DFD elements have a trustLevel property. In addition to the common properties, there are type-specific properties. Some of the possible DataStore-specific properties include readProtection, writeProtection, secrecyMethod and integrityMethod.

If no meaningful value for a property is specified and if providing that additional information can enable additional checks, the analysis requests more information from the user, e.g., by suggesting entering a value for the trustLevel of a DataFlow in relation to its source (other than Unknown).

Since several security checks are based on the value of trustLevel, it is represented graphically: High trustLevel is green, Medium trustLevel orange, and Low or None trustLevel red (Figure 1). Since these colors are indistinguishable by colorblind readers or in a black and white printout, we add to the label of an entity (-) for a Low, (#) for a Medium, and (+) for a High trustLevel (See the Trust Legend in Figure 1) [1].

[1]As of this writing, GraphViz does not support cross-hatching fill patterns.

- **Model**
  - settings: Settings
  - dfd: List<BasicEntity>
- **Settings**
  - binaries: List<String>
  - matchrules: List<MatchRule>
  - functiongroups: List<FunctionGroup>
  - exceptions: List<ConformanceException>
- **MatchRule**
  - name: String // *e.g.,* "Registry"
  - type: ShapeType // *e.g.,* DataStore
  - binary: String // *e.g.,*"advapi32.dll"
  - match: String // *e.g.,* "Reg.+" or "Init"
- **FunctionGroup**
  - name: String // *e.g.,* "Contact APIs"
  - type: ShapeType // *e.g.,* Process
  - services: List<Service>
- **Service**
  - binary: String // *e.g.,* "`app.dll`"
  - function: String // *e.g.,* "`Initialize`"
- **ConformanceException**
  - source: BasicEntity
  - destination: BasicEntity
  - details: List<Connection>
  - finding: Finding
  - reason: String
- **Connection**
  - provided: Service
  - required: Service
- **Finding**: Absent | Convergent | Divergent | Excluded | Overridden | AmbiguousDirection | ExternalEntity | Unknown
- **ShapeType**: Process | DataStore | DataFlow . . .
- **BasicEntity**
  - name: String
  - shapeType: ShapeType
  - services: List<Service>
  - trustLevel: TrustLevel
  - howFound: HowFound
  - owner: Owner
  - finding: Finding
- **Process** extends BasicEntity
  - inputTrustLevel: TrustLevel
  - performsAuthentication: boolean
  - authenticationMethod: AuthenticationType
  - performsAuthorization: boolean
  - authorizationMethod: AuthorizationType
  - performsValidation: boolean
  - validationMethod: ValidationType
- **DataFlow** extends BasicEntity
  - source: BasicEntity
  - destination: BasicEntity
  - readProtection: DataAccessProtection
  - writeProtection: DataAccessProtection
  - secrecyMethod: InformationSecrecy
  - integrityMethod: InformationIntegrity
  - details: List<Connection>
- **DataStore** extends BasicEntity
  - readProtection: DataAccessProtection
  - writeProtection: DataAccessProtection
  - secrecyMethod: InformationSecrecy
  - integrityMethod: InformationIntegrity
- **ExternalInteractor** extends BasicEntity

Figure 9: Extended DFD representation.

- **TrustLevel**: None | High | Medium | Low | Unknown
- **HowFound**: HardCoded | ... | Mixed | Unknown
- **Owner**: ThisComponent | CompanyTeam | Anybody | ... | Mixed | Unknown
- **DataAccessProtection**: None | SystemACLs | CustomAccessControl | ... | Other | Unknown
- **InformationSecrecy**: None | Encryption | Obfuscation | ... | Other | Unknown
- **InformationIntegrity**: None | DigitalSignature | ... | Other | Unknown
- **AuthenticationType** None | Windows | Mixed | ... | Other | Unknown
- **AuthorizationType** None | RoleBased | Mixed | ... | Other | Unknown
- **ValidationType** None | RegularExpressions | ManualParsing | Mixed | ... | Other | Unknown

Figure 10: Extended DFD representation (continued).

By definition, the trustLevel of a Process — i.e., code that is shipped part of the application, must be High. If that is not the case, then that element must be represented as an ExternalInteractor. A DataStore, ExternalInteractor or a DataFlow can have any trustLevel. Since the value of trustLevel for an element controls several security checks, it is also represented graphically (See Figure 1).

The analysis looks for security flaws in the STRIDE model which include Spoofing, Tampering, Repudation, Information Disclosure, Denial of Service and Elevation of Privilege [20, pp. 83–87], and are based on the principles of information confinement [13].

At a high-level, the analysis works as follows, for each of the rules discussed below. We illustrate it with the tampering rule **T1**:

- **Analyze threats:** an attacker tampers with the contents of a DataStore whose trustLevel is High;
- **Analyze mitigations:** if a readProtection and writeProtection are SystemACLs, assume that the threat of tampering is reduced;
- **Suggest remedies**: if readProtection and writeProtection are None, suggest the remedy in the rule: "use Access Control Lists (ACLs)". A remedy is often just an informational message for the modeler. Unless the remedy requires changing the DFD or its security properties, the tool cannot always check whether they are performed.

We list below some of the rules currently implemented, organized by category.

Spoofing. An attacker pretends to be someone else.

- **S1.** Threat: If a DataFlow's trustLevel is higher than the trustLevel of the DataFlow's source, the source can potentially spoof the trusted data;
  Remedy: ensure that the flow is not treated as more trusted than the source entity.
- **S2.** Threat: An ExternalInteractor with a trustLevel other than None can be easily spoofed;
  Remedy: ensure that strong authentication and authorization constraints are in place;
  Mitigation: if performsAuthentication and performsAuthorization are set to true, the methods of authentication and authorization must be set using authenticationMethod and authorizationMethod.
- **S3.** Threat: If howFound property is set to Unknown, the entity has no defined mechanism for being located;
  Remedy: set the howFound property.
- **S4.** Threat: If howFound is set to HardCoded, the location of entity is hard-coded into the system binaries, so it cannot be spoofed.
- **S5.** Threat: If howFound is set to Pointer, the location of this entity is pointed to by another entity;
  Remedy: ensure that the referring entity cannot spoof the name or location of this entity, or cause the system to access an unexpected entity.
- **S6.** Threat: If howFound is set to Mixed, the entity has some hard-coded and some dynamic entities;
  Remedy: include a more detailed diagram that breaks the entity up into individual parts.

Tampering. Data is changed in transit or at rest.

- **T1.** Threat: If the trustLevel of a DataStore is other than None, it is possible for the contents to be tampered with or read by an attacker;
  Remedy: add Access Control Lists (ACLs) to the DataStore or take other precautions;

Mitigation: readProtection and writeProtection are set to values other than Unknown or None.

Information Disclosure. An attacker steals data while in transit or at rest.

- **I1.** Threat: If the trustLevel of a DataFlow's source is higher than that of its destination, information disclosure is possible;
  Remedy: ensure that no sensitive information is leaked in this flow.
- **I2.** Threat: If the trustLevel of a DataFlow's source has a value that is lower than the trustLevel of the destination, look for potential flaws;
  Remedy: ensure that the destination does not implicitly trust the input as it could be tainted.

Denial of Service. An attacker interrupts the legitimate operation of a system. Such a threat may arise if messages are not validated before use (e.g., by stripping prohibited escape characters), thus allowing a rogue client to crash the system and cause a denial of service for other valid clients.

- **D1.** Threat: An ExternalInteractor with a trustLevel other than High can launch a denial of service attack.

Ownership. To avoid security flaws that can arise when different development teams make different security assumptions about subsystems that may interact (e.g., [43, p. 75]), each element is assigned an owner:

- **O1.** Threat: An ExternalInteractor's owner is set to ThisComponent;
  Remedy: mark the entity's owner as being external or convert the entity into a process or other type.
- **O2.** Threat: If an entity's owner is marked as Anybody, its trustLevel must be None since this code can be written by anyone and must be untrusted.
  Remedy: either update the entity's owner or change its trustLevel.
- **O3.** Threat: An entity's owner is CompanyTeam.
  Remedy: trade threat models with the other team so each team is aware of the other's assumptions.
- **O4.** Threat: An entity's owner is Mixed.
  Remedy: include a more detailed diagram where the entity is expanded to show each individual entity with a single owner.

In *repudiation*, an attacker performs an action that cannot be traced back to them. However, the analysis currently does not check repudiation since it requires audit trails and there is no easy way to mark that a datum is auditable. A repudiation threat could arise if data that was supposed to be audited was easily tampered with or there were flows that did not include the auditing step. Similarly, in *elevation of privilege*, an attacker performs actions they are not authorized to perform. Currently, our properties do not track when a *lower-trust* entity or data can influence a *higher-trust* entity that does not perform sufficient validation.

The goal of the DFD-level analysis is not to replace a review by security experts. Threat model analysis can only be partially automated since it still requires human creativity and intuition to uncover subtle security flaws that cannot easily be generalized. Nevertheless, the analysis can help inexperienced threat modelers identify many of the small issues by providing suggestions on how to fix them in some cases and encouraging designers to focus on the complex interactions between subsystems. Over time, more properties can be added and additional checks could improve the quality of the analysis. Currently, the topological constraints and the security checks are hard-coded in the tool, but this was not a crucial issue in our evaluation. Similar rules could be defined as logic predicates using the Acme ADL [16, 46] (D. Garlan, Personal Communication, Sept. 29, 2006).

Once the as-designed DFD has been updated to reflect the as-built DFD as discussed earlier, re-running the security analysis on the as-designed DFD would at least be using an up-to-date model of the application's entry points, one that does not leave out registry accesses, configuration file accesses, command-line access, network access, etc.

Even though a system can be designed to be secure, the implementation must still correctly implement the security scheme. One of the major security blunders of the past 30 years was to introduce a bug in the implementation of encrypted telnet by upgrading to a cryptography library that verified the parity of the bits used in the keys — unaware that the last bit of each key byte was used as a parity bit. Since the keys were selected at random, only 1 in 256 keys had the correct parity and the rest were silently dropped. As a result, 255/256 telnet sessions used an all-zero key (discovered by Blaze and reported in [18]). Building secure software still requires testing the implementation, manual code reviews or various analyses, e.g., a static analysis to prevent denial of service or elevation of privilege attacks through buffer overruns [19].

5 Limitations

There are several limitations to our approach that we plan on addressing in future work.

Control Flow vs. Data Flow. Our analysis uses the binary dependency information provided by MaX, which is mainly control flow information and limited data flow information based on communication through shared registry entries. In particular, MaX does not currently have information about communication through global variables. Our evaluation has shown that although our analysis does not work at the level of abstraction of an ideal DFD, there is enough accuracy in the dependency information to check the conformance between the as-built and as-designed DFDs.

In practice, few DFDs have ordered data flows. DFDs are often imprecise about data flow directionality and include undirected edges or bi-directional edges. DFDs are often omit data flow labels, mix between control flow and data flow, or label data flows with function names such as "Send" or "Receive". Although flow ordering is important when problems arise because of differences between the time of check and the time of use, this precision seems to be mostly needed when formally modeling freshness in a protocol, in which case, it may be necessary to use models that are more formal than DFDs [5].

Although MaX is not precise about data flow ordering, directionality or labeling, this was not a serious limitation in our evaluation. When threat modeling how data or information flows during execution, it is the *classification* of the data that is more interesting than the actual information [17]. The DFD representation captures the classification information using the Boundary construct and by assigning a trustLevel to each DFD entity. In fact, the DFD security analysis flags a Dataflow that does not cross any trust boundaries as uninteresting and encourages the designer to look for DataFlows that do cross boundaries. In future work, we may define heuristics to get data flow ordering and directionality using calling context information (call site and call stack information). We could also get precise flow information by dynamically observing data flow with scenario-driven tests or using a sound source-level static analysis [7, 25].

Mapping Language. We envision several extensions to the mapping language to further reduce the number of false positives when applying Reflexion Models; some of these were alluded to by Murphy but without giving concrete examples of when they might be needed [34, p. 98].

A mapping rule may want to express that an edge should be shown only if the call originates from a given binary. For instance, a rule could be defined to show a dependency to the encrypted section of the Registry known as the "Cert[ficate] Store" only if it originates in the binary of interest (`app.dll`):

```
<rule fromBinary="app.dll" toBinary="crypt32.dll" fromFunction=".+"
  toFunction="CertOpen.+" mapTo="Cert Store" shapeType="DataStore"/>
```

An explicit edge rule could create for System A an edge between "Control Panel" and "Registry" in Figure 7 when a call to `ShellExecuteEx` is encountered:

```
<edgerule fromBinary="app.dll" toBinary="shell.dll" fromFunction="X"
 toFunction="ShellExecuteEx" edge="Settings" from="Control Panel" to="Registry"/>
```

Finally, the edge directionality could be encoded in a match rule. For instance, the Registry would have a match rule for *reads* – `RegQueryValueEx` and `RegOpenKeyEx`, and one for *writes* – `RegCreateKeyEx` and `RegSetValueEx`.
Instead of the current match rule for the Registry:

```
<matchrule binary="advapi32.dll"  function="Reg.+" mapto="Registry"/>
```

a *read* match rule would consist of:

```
<dirmatchrule binary="advapi32.dll" function="RegQueryValueEx"
  edgeFrom="Registry" edgeDirection="Incoming" />
```

Regarding the Reflexion Models technique, there are two venues we would like to explore in future work. Researchers [24] have pointed out the need for Hierarchical Reflexion Models on the basis of scalability. Our experience provides another justification for supporting hierarchical reflexion models since DFDs are naturally hierarchical. Finally, we plan to look into automated clustering techniques, e.g., [8].

6 Related Work

Structured analysis DFDs were widely used by practitioners in the 1970's and 1980's [49]. Researchers have formalized DFDs [27, 26] and attempted reverse engineering DFDs from code [6]. Liu et al. proposed Conditional Data Flow Diagrams (CDFDs) [29] to resolve many of the ambiguity problems in Yourdon's DFDs. In particular, the CDFD adds operational semantics, pre- and post-conditions and well-defined types. However, the well established threat modeling process does not use CDFDs and there is no checking of the conformance of a CDFD to an implementation. Existing DFDs can be readily converted from the current template [43] to our DFD representation.

The idea of encoding security properties in an architecture is not new, e.g., [46]. Our approach was designed to support and enhance the existing, currently manual, threat modeling process [21]. Our security analysis is novel in being the first to define STRIDE [20] security properties on individual DFD elements, and rules in terms of these elemental properties, to make a semi-automated analysis possible.

Microsoft has made publicly available a threat modeling tool [42]. Similarly, the EU-funded CORAS Project [10] proposed its description and tool support for threat models using UML. However, neither tool currently analyzes a DFD or checks its conformance with an implementation. An experimental methodology similar to threat modeling but not as widely adopted, Trike [38], employs data flow diagrams, state machines and use flows and attempts to build attack graphs automatically from attack libraries, focusing on elevation of privilege and denial of service.

The threat modeling process builds on previous work in the area of generating attack graphs [39] and has similarities with specifying security requirements [45].

Our XML-based DFD representation is similar to other XML-based Architecture Description Languages (ADLs) such as xADL [11] (a Boundary is similar to a *group*). Most ADLs also support hierarchy in the form of component decomposition. We adopted a connection model that does not use ports or roles as in some other ADLs [37]. AADL is one of the few ADLs where connections can specify patterns of control or data flow between components at runtime [14].

Various architectural-level security analyses have been proposed using different ADLs [4, 12, 33]. Secure xADL [36] also encodes a Role Based Access Control (RBAC) security model which views a system as a set of users, roles, privileges and resources. Secure xADL can be complementary to our approach when the effort required by the modeler to setup the RBAC model is warranted. Representations other than DFDs, such as UML constructs, can be used for threat modeling: one such approach based on UML, SecureUML [30], also uses RBAC.

UMLsec [23] adds to UML security extensions for secrecy, integrity, and authenticity, etc. and has an analysis to check for security weaknesses at the design level. However, conformance between the architecture and the implementation is achieved using code generation, code analysis, and test-sequence generation. Code generation, while potentially guaranteeing the correct refinement of an architecture into an implementation, is often too restrictive to be fully adopted on a large scale and cannot account for millions of lines of legacy code. The traceability information obtained using our approach can actually help guide the static analyses and penetration testing of exposed sections of the code.

Formal security analyses have been proposed, e.g., [47], but those heavyweight techniques have not been demonstrated to scale to systems with millions of lines of code. Formal methods are probably best suited for debugging compact subsystems such as authentication protocols [2, 5]. Finally, Yee [48] uses "private data flow charts" similar to DFDs for visualizing the flow of privacy-sensitive data.

7 Conclusion

A semi-formal approach based on a lightweight representation of a Data Flow Diagram (DFD) is proposed and used to check the conformance between an as-built DFD and an as-designed DFD. An evaluation of the approach on subsystems from production code found omitted or outdated information in existing DFDs. The checking also yields valuable traceability information between the threat model and the implementation that can be used by other code quality tools. Finally, extending the DFD representation with security properties enables an analysis of early DFDs produced by developers with limited threat modeling experience to warn about possible security attacks.

Acknowledgements

The authors thank the members of the Security Development Lifecycle (SDL) team at Microsoft for their useful input and the anonymous Microsoft developers for participating in the System A case study. The first author thanks Jonathan Aldrich, David Garlan, Thomas LaToza and Bradley Schmerl for detailed comments on earlier drafts of this report.

References

[1] M. Adler. An Algebra for Data Flow Diagram Process Decomposition. *IEEE Transactions on Software Engineering*, 14(2), 1988.

[2] R. Anderson. Why Cryptosystems Fail. In *ACM Conference on Computer & Communication Security*, 1993.

[3] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833, 2004.

[4] C. Bidan and V. Issarny. Security Benefits from Software Architecture. In *International Conference on Coordination Languages and Models*, pages 64–80, 1997.

[5] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1), 1990.

[6] G. Canfora, L. Sansone, and G. Visaggio. Data Flow Diagrams: Reverse Engineering Production and Animation. In *International Conference on Software Maintenance (ICSM)*, 1992.

[7] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Symposium on Operating systems Design & Implementation (OSDI)*, 2006.

[8] A. Christl, R. Koschke, and M.-A. Storey. Equipping the Reflexion Method with Automated Clustering. *Working Conference on Reverse Engineering*, pages 89–98, 2005.

[9] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.

[10] CORAS. `http://coras.sourceforge.net`, 2006.

[11] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001.

[12] Y. Deng, J. Wang, J. J. P. Tsai, and K. Beznosov. An Approach for Modeling and Analysis of Security System Architectures. *IEEE Trans. on Knowledge and Data Engineering*, 15(5):1099–1119, 2003.

[13] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications ACM*, 19(5):236–243, 1976.

[14] P. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL). Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, 2006.

[15] E. R. Gansner and S. C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software – Practice & Experience*, 30(11), 2000.

[16] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[17] D. A. Gustafson. Control flow, Data flow & Data Independence. *SIGPLAN Notices*, 16(10):13–19, 1981.
[18] P. Gutmann. Secure Internet-based Electronic Commerce: The View from Outside the US, 1998.
[19] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular Checking for Buffer Overflows in the Large. In *International Conference on Software Engineering (ICSE)*, pages 232–241, 2006.
[20] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2nd edition, 2003.
[21] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.
[22] jRM Tool. `http://jrmtool.sourceforge.net`, 2003.
[23] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.
[24] R. Koschke and D. Simon. Hierarchical Reflexion Models. In *Working Conference on Reverse Engineering*, page 36, 2003.
[25] S. Kowshik, G. Rosu, and L. Sha. Static Analysis to Enforce Safe Value Flow in Embedded Control Systems. In *Intl. Conference on Dependable Systems and Networks*, pages 23–34, 2006.
[26] P. G. Larsen, N. Plat, and H. Toetenel. A Formal Semantics of Data Flow Diagrams. *Formal Aspects of Computing*, 6(6):586–606, 1994.
[27] G. T. Leavens, T. Wahls, and A. L. Baker. Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages. In *SAC*, 1999.
[28] S. Lipner. The Trustworthy Computing Security Development Lifecycle. In *Annual Computer Security Applications Conference*, pages 2–13, 2004.
[29] S. Liu, A. J. Offutt, C. Ho-Stuart, M. Ohba, and Y. Sun. SOFL: a Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):24–45, 1998.
[30] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *International Conference on the Unified Modeling Language*, pages 426–441, 2002.
[31] Microsoft Corporation. The Component Object Model Specification. Version 0.9, 1995.
[32] P. Mockapetri. Domain Names – Implementation and Specification. (RFC 1035), 1987.
[33] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure Software Architectures. In *IEEE Symposium on Security and Privacy*, page 84, 1997.
[34] G. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.
[35] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27(4), 2001.
[36] J. Ren and R. Taylor. A Secure Software Architecture Description Language. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, 2005.
[37] J. Ren, R. Taylor, P. Dourish, and D. Redmiles. Towards an Architectural Treatment of Software Security: a Connector-Centric Approach. In *Workshop on Software Engineering for Secure Systems – Building Trustworthy Applications (SESS)*, pages 1–7, 2005.
[38] P. Saitta, B. Larcom, and M. Eddington. Trike v1 Methodology Document. `www.octotrike.org`, 2005.
[39] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. In *IEEE Symposium on Security and Privacy*, 2002.
[40] A. Srivastava and A. Eustace. ATOM: a System for Building Customized Program Analysis Tools. In *Programming Language Design and Implementation (PLDI)*, 1994.
[41] A. Srivastava, J. Thiagarajan, and C. Schertz. Efficient Integration Testing using Dependency Analysis. Technical Report MSR-TR-2005-94, Microsoft Research, 2005.
[42] F. Swiderski. Threat Modeling Tool. `www.microsoft.com/downloads`, June 2004.
[43] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.
[44] P. Torr. Demystifying the Threat-Modeling Process. *IEEE Symposium on Security and Privacy*, 03(5):66–70, 2005.
[45] A. van Lamsweerde. Elaborating Security Requirements by Construction of Intentional Anti-Models. In *International Conference on Software Engineering (ICSE)*, 2004.
[46] D. S. Wile. Revealing Component Properties through Architectural Styles. *J. Systems & Softw.*, 65(3), 2003.
[47] D. Xu and K. E. Nygard. Threat-Driven Modeling and Verification of Secure Software Using Aspect-

Oriented Petri Nets. *IEEE Transactions on Software Engineering*, 32(4):265–278, 2006.
[48] G. Yee. Visualization for Privacy Compliance. In *International Workshop on Visualization for Computer Security*, pages 117–122, 2006.
[49] E. Yourdon. *Structured Analysis.* Prentice Hall, 1988.