

# Analyzing Security Architectures

Marwan Abi-Antoun  
Department of Computer Science  
Wayne State University  
mabiantoun@wayne.edu

Jeffrey M. Barnes  
Institute for Software Research  
Carnegie Mellon University  
jmbarnes@cs.cmu.edu

## ABSTRACT

We present a semi-automated approach, SECORIA, for analyzing a security runtime architecture for security and for conformance to an object-oriented implementation. Type-checkable annotations describe architectural intent within the code, enabling a static analysis to extract a hierarchical object graph that soundly reflects all runtime objects and runtime relations between them. In addition, the annotations can describe modular, code-level policies. A separate analysis establishes traceability between the extracted object graph and a target architecture documented in an architecture description language. Finally, architectural types, properties, and logic predicates describe global constraints on the target architecture, which will also hold in the implementation. We validate the SECORIA approach by analyzing a 3,000-line pedagogical Java implementation and a runtime architecture designed by a security expert.

## Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures

## General Terms

Design, Security, Verification

## 1. INTRODUCTION

Companies such as Boeing and Microsoft have been using *threat modeling* [38] as a lightweight approach to reason about security, to capture and reuse security expertise and to find security design flaws during development. Proponents of threat modeling have employed it in a number of industrial-scale projects and report impressive results. Microsoft, for example, claims that threat modeling led to a 50% reduction in security vulnerabilities [16], and around 1,500 data flow diagrams (DFDs) were manually reviewed during the development of Windows Vista.

When a security expert asks a developer to build a security architecture for a system under study, the developer

typically produces a diagram mostly from his recollection of how the system works, with little tool support to extract such an architecture from the code. Then, during the security review, the experts study the architecture, assign to the components different architectural properties such as *trustLevel* or *privacyLevel*, and enumerate all possible communication between the more trusted and the less trusted components of the system. They also analyze the communication between components that contain personally identifiable information, and those that do not.

One potential hurdle to achieving significantly better defect reduction is that the architecture may not show all the communication that is present in the system. As a result, an architectural-level analysis may be incomplete. While any architecture-based approach suffers from these problems, security architectures pose special challenges.

A security architecture<sup>1</sup> is an example of a *runtime architecture*, which shows runtime components and connectors, uses hierarchical decomposition, and partitions a system into tiers [8]. Today, the tools for extracting and analyzing conformance of runtime architectures are immature [22, 11], compared to tools for the code architecture [31].

Moreover, a security analysis must consider the worst and not the typical case of possible component communication. The analysis results are valid only if the architecture reveals all objects and relations that may exist at runtime—in any program run. This requires a static analysis, which can capture all possible executions. In contrast, a dynamic analysis, which extracts an architecture or analyzes conformance based on one or more program runs [34], may miss important objects or relations that arise only in other executions.

The contributions of this paper are:

- The first architecture-centric approach, SECORIA<sup>2</sup>, that enables both reasoning at the level of a security runtime architecture, and relating it to the code at the same time, as compared to previous approaches which do one or the other. The approach can enforce both code-level and global architectural constraints.
- An end-to-end validation of SECORIA using a real, 3,000-line Java implementation of a runtime architecture designed by a security expert.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

Copyright 2010 ACM 978-1-4503-0116-9/10/09 ...\$10.00.

<sup>1</sup>Threat modeling typically uses a *data flow diagram*, which describes how data enters, leaves and traverses the system, shows data sources and destinations, trust boundaries and processes that data goes through [38]. Here, a *security architecture* shows points-to rather than data flow connectors.

<sup>2</sup>SECORIA stands for Security conformance of object-oriented runtime views of architecture.

SECORIA specializes the SCHOLIA approach by Abi-Antoun and Aldrich to analyze, at compile time, communication integrity between arbitrary object-oriented code and a hierarchical, target runtime architecture [2]. *Communication integrity* [25] means that each component in the implementation may only communicate directly with the components to which it is connected in the architecture. SCHOLIA requires adding typecheckable annotations, to establish traceability between the target architecture and the code.

Our design intent-based analysis involves an iterative process with two main stages: the *conformance* stage (Sect. 2), which relates a security architecture to code, and the *enforcement* stage (Sect. 3), which enforces architectural intent. We evaluate the SECORIA approach by applying it to a real system (Sect. 4). Finally, we conclude with a discussion (Sect. 5) and a brief survey of related work (Sect. 6).

## 2. CONFORMANCE STAGE

Architectural reasoning about security is best accomplished with a runtime architecture rather than a code architecture. Unfortunately, extracting the runtime architecture is difficult. At runtime, an object-oriented system can be represented as an *object graph*: nodes correspond to objects, and edges correspond to relations between objects. Taking a snapshot of the heap at runtime reveals the structure at that instant in great detail, but the profusion of objects makes it difficult to get a high-level picture, without extensive graph summarization and manipulation [28]. Moreover, such a snapshot shows only one or more executions, meaning the developer may miss important objects or relations that arise only in other executions. On the other hand, a sound static analysis can extract an object graph that captures all executions. But previous static analyses produce non-hierarchical object graphs that explain runtime interactions in detail but convey little architectural abstraction. A flat object graph mixes low-level objects such as `HashMap` with architecturally relevant objects such as `CryptoReceipt`, and a developer has no easy way to distinguish them. Even for a small program, a flat object graph typically does not convey sufficient architectural abstraction to be used for conformance analysis.

### 2.1 Ownership Domain Annotations

A central difficulty is that architectural hierarchy is not readily observable in arbitrary code. To achieve hierarchy in an object graph, SCHOLIA requires that a developer pick a top-level object as a starting point, then use modular ownership annotations in the code [6] to impose a conceptual hierarchy on objects.

The annotations specify object encapsulation, logical containment and architectural tiers within the code, constructs which are not explicit in most programming languages. The SCHOLIA tools use existing language support for annotations. In addition, the annotations implement a type system, so a typechecking tool can validate the annotations and identify inconsistencies between the annotations and the code.

An *ownership domain* is a conceptual group of objects with an explicit name and explicit policies that govern how a domain can reference objects in other domains [6]. Each object is assigned to a single domain that does not change at runtime. A developer indicates the domain of an object by annotating each reference to that object in the program.

The annotations define two kinds of object hierarchy, logical containment and strict encapsulation, defined below.

**Logical Containment.** A public domain provides *logical containment*, thus making an object conceptually “part of” another object. Having access to an object gives the ability to access objects inside all its public domains. For example, `LocalKeyStore` has a public domain, `KEYS`, to hold `LocalKey` objects (Fig. 1).

**Strict Encapsulation.** A private domain provides *strict encapsulation*. Thus, a public method cannot return an alias to an object in a private domain, even though the Java type system allows returning an alias to a field marked `private`. For example, `LocalKeyStore` stores the `ArrayList` of `LocalKey` objects, `keys`, in a private domain, `OWNED` (Fig. 1).

### 2.2 Object Graph Extraction

SCHOLIA extracts an Ownership Object Graph (OOG) that provides architectural abstraction by ownership hierarchy, by showing architecturally significant objects near the top of the hierarchy and data structures further down. An object graph can also provide abstraction by types, by collapsing objects further based on their declared types.

The visualization uses box nesting to indicate containment of objects inside domains and domains inside objects (Fig. 1). Dashed-border, white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as “object `obj`” or as “`T` object,” meaning “an instance of the `T` class.” E.g., `LocalKey` is inside `KEYS`. A private domain has a thick, dashed border; a public domain, a thin one. A (+) symbol on an object or a domain indicates that it has a collapsed substructure.

An extracted object graph is *sound* in two respects. First, each runtime object has exactly one representative in the object graph. Second, the object graph has edges that correspond to all possible runtime points-to relations between those objects. Soundness is important for an architectural-level security analysis. For instance, if an architecture showed the same runtime object as two components in the architecture, an analysis could assign those two components different values for the key `trustLevel` property and potentially invalidate the analysis results.

In addition, ownership-parametric library code, such as `ArrayList`, often creates interesting architectural relationships in application objects, when formal parameters are bound to actual domains on specific objects created by the application. The object graph resolves these parameters to ensure that the relevant object relations appear at the level of the global application object structures; hence, there is an edge from `keys` to `localKey` (Fig. 1).

### 2.3 Communication Integrity

To analyze communication integrity using SCHOLIA, a developer follows the extract-abstract-check strategy [31], as follows: (1) the developer documents the designed runtime architecture; (2) she adds annotations to the code and typechecks them using `ARCHCHECKJ`; (3) then, she uses `ARCHRECJ` to *extract* an object graph; (4) she then uses `ARCHCOG` to *abstract* an object graph into a built architecture; (5) finally, she uses `ARCHCONF` to structurally *compare* the built and the designed architectures, and *check* and enforce *communication integrity* in the designed architecture.

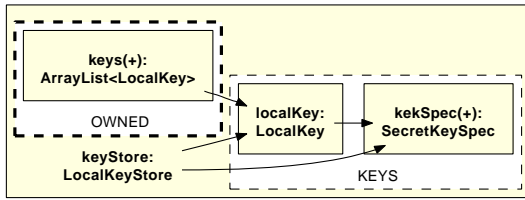


Figure 1: The LocalKeyStore OOG.

A developer can perform any of the following: (a) Refine the annotations iteratively based on visual inspection of an extracted object graph before it is abstracted; (b) Fine-tune the abstraction of an object graph into an architecture; (c) Manually guide the comparison of the built and the designed architecture, if the structural comparison fails to perform the proper match; (d) Correct the code if she decides that the designed architecture is correct, but that the implementation violates the architecture; or (e) Update the designed architecture if she determines that the implementation highlights an important omission in the architecture.

Following Murphy’s terminology [31], the analysis finds:

- **Convergence:** a node or an edge that is in *both* the built architecture and the designed architecture;
- **Divergence:** a node or an edge that is in the built architecture, but *not* in the designed architecture;
- **Absence:** a node or an edge that is in the designed architecture, but *not* in the built architecture.

When adding annotations and extracting OOGs, the goal is to minimize the number of annotation warnings and the number of objects in the top-level domains. When analyzing communication integrity, the goal is to minimize the number of divergences and absences, or to ensure that they do not correspond to cases where the implementation violates the architectural intent.

### 3. ENFORCEMENT STAGE

Having analyzed conformance and established traceability between target architecture and code, SECORIA now requires us to identify implementation-level violations of architectural intent by enriching the designed architecture with types, properties, and constraints.

#### 3.1 Code-Level Constraints

At the code level, we use annotations to enforce local, *modular* constraints, i.e., ones that are checked one class at a time. A *domain link* annotation specifies policies that govern how a domain can reference objects in other domains [6], which we illustrate by example. A *List* *assumes* that its owning domain can access the list elements stored in the ELTS domain parameter. In turn, when a *LocalKeyStore* instantiates a *List*, and binds ELTS to KEYS, *LocalKeyStore* must satisfy that assumption. Hence, *LocalKeyStore* declares a *domain link* from OWNED to KEYS.

#### 3.2 Architectural Description

We describe the target architecture in an Architecture Description Language (ADL) and set architectural types, properties, and predicates to enforce global constraints as well as security design intent. We use Acme, a general-purpose ADL with mature, publicly available tool support [12].

Most ADLs support the following elements. A *component* is a unit of computation and state. A *connector* represents an interaction between components. A component can op-

tionally be decomposed into a nested subarchitecture. A *property* is a name-and-value pair associated with an element. A *group* is a named set of elements, such as a tier. To specify architectural intent, architectural types, properties and constraints can be set.

**Architectural Types.** Many ADLs, including Acme, support a type system for architectural elements. Thus, there may be many component instances that all belong to a single *component type*, which defines properties and constraints that are common to all instances, and likewise for connector types. Acme’s type system for architectural elements supports inheritance, so one element type may inherit from another. Types are helpful for architectural description not only because they save work (by abstracting out constraints common to multiple instances), but also because they match the way architects naturally think about software systems. These element types are used to build up *families*, which encapsulate element types that are applicable to a whole class of software architectures, such as all multi-tier architectures or all pipe-and-filter architectures.

In SECORIA, we model a DFD as a component-and-connector runtime architecture [8, pp. 364–365]. A DFD has a fixed set of component types: *Process*, *HighLevelProcess*, *DataStore* and *ExternalInteractor* [36]. 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, such as a database or file. 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.

DFDs used in threat modeling often separate elements that have different privilege levels using a *Boundary* to describe locations where an adversary could mount a privilege impersonation attack, or where a machine or process boundary may be crossed [36, p. 90]. Here, we model a *Boundary* as an architectural runtime tier, which is represented in the annotations as a domain.

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, diagrams at Level-1, Level-2, and so on can more precisely model security-critical processing [15, p. 76]. This model naturally fits with a hierarchical representation of runtime architecture.

Fig. 4(a) shows several element type definitions from the Acme DFD family. The component type *DataStore* inherits from *DFDComponent*, which in turn inherits from *DFDTypeT*, a base element type that defines common properties such as *trustLevel* and *howFound*. Similarly, we defined property types (mostly, enumerated types), properties and rules (predicates) that use properties and types to analyze and enforce security qualities. The DFD family is reusable for other systems, and is available in an online appendix [1].

**Architectural Properties.** SECORIA assigns element-level properties to the various architectural component instances. For example, several rules for checking information disclosure rely on a key *trustLevel* property. Because all DFD elements must have such a property, the property is defined

on the base architectural type, and thus inherited by all its subtypes. Usually, distinct component instances have different values for architectural properties such as `trustLevel`.

In addition to the common properties, there are type-specific properties. For instance, `DataStore`-specific properties include `readProtection`, `writeProtection`, `secrecyMethod`, and `integrityMethod`. Many of these properties are enumerations with pre-defined values and a default value of `Unknown`.

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 users, e.g., by suggesting they enter a value for the `trustLevel`.

**Well-Formedness Constraints.** A DFD must obey several well-formedness rules. For example, two `DataStores` cannot be connected directly. Indeed, data cannot travel from one `DataStore` to another without the aid of a `Process`.

**Information Flow Constraints.** Threat modeling looks for vulnerabilities in the areas of `Spoofing`, `Tampering`, `Repudiation`, `Information Disclosure`, `Denial of Service`, and `Elevation of Privilege` (STRIDE) [15, pp. 83–87].

The constraints analyze the architecture for security flaws. The analysis checks the provided properties and automatically requests more information where applicable. At a high level, the analysis works as follows:

1. **Analyze threats:** one rule checks for the threat of tampering. For example, an attacker can tamper with the contents of a `DataStore` whose `trustLevel` is `High`.
2. **Analyze mitigations:** for the tampering rule, the analysis checks if `readProtection` and `writeProtection` are `SystemACLs`. In that case, the threat of tampering is reduced.
3. **Suggest remedies:** for the tampering rule, the analysis checks if `readProtection` and `writeProtection` are `None`. In that case, the analysis suggests the following remedy: “Use access control lists.” A remedy is often just an informational message for the modeler. Unless the remedy requires changing the DFD or its security properties, the analysis cannot always check that the remedy is performed.

We formalized constraints that focus mostly on global information flow vulnerabilities (`Spoofing`, `Tampering`, `Information Disclosure`). Checking for flaws such as `Denial of Service` or `Elevation of Privilege` requires tracking state changes in the system, which the extracted architecture does not presently have. A description of the full set of rules currently implemented is in a companion technical report [4].

**General Constraints.** First-order logic predicates [29] enforce structural, application-specific constraints such as:

- Components  $c_1$  and  $c_2$  are never connected directly:  
`!connected(c1, c2)`
- No Component of type  $T_1$  directly connects to one of type  $T_2$ ;  

```
forall c1: Component in self.COMPONENTS |
forall c2: Component in self.COMPONENTS |
connected(c1, c2) -> !(declaresType(c1, T1)
AND declaresType(c2, T2));
```
- No component in Group  $g_1$  communicates directly with any component in Group  $g_2$ .  

```
forall m1 in g1.members |
forall m2 in g2.members | !connected(m1, m2);
```

During this stage, the goal is to reduce the number of violations of architectural types and constraints.

Enforcing predicates at the architectural level is not novel. But since SECORIA has the traceability between architecture and code, these architectural constraints enforce global constraints on the application structure in the code.

### 3.3 Constraining Evolution

During software evolution, it is common for developers to introduce architectural violations that may lead to security vulnerabilities. Relating the target architecture and the code, together with effective change management, can help detect unwanted architectural violations more effectively than inspecting the program, with or without annotations. In the unannotated program, changing the runtime architecture is as simple as storing or passing a reference to an object. The ownership annotations help to constrain the evolution of the program somewhat. But a developer can still add communication paths, e.g., by adding domain links. Admittedly, manual code inspections could be used to audit more closely revisions that modify domain link annotations. Even so, the annotations can enforce only modular constraints. So it is still necessary to identify code modifications that impact the global architectural structure, ideally using an automated approach.

Extracting the up-to-date built architecture, analyzing its conformance to a target architecture, and checking architectural constraints can make it easier to trigger an architectural review. Structural constraints in the target architecture can enforce various security policies. Indeed, empirical evidence suggests that such policies are frequently needed during software evolution. For instance, a study using a well-designed framework showed that students subverted the framework’s design by passing to and storing additional objects in the constructors of classes that implemented the core framework interfaces [20]. Such unconstrained evolution of a system might lead to serious architectural violations, and as a result, to security vulnerabilities.

## 4. EVALUATION

We validate the end-to-end SECORIA approach using `CryptoDB`, a secure database system designed by security expert Kevin Kenan [19]. `CryptoDB` follows a database architecture that provides cryptographic protections against unauthorized access, and includes a 3,000-line sample implementation in Java. The presence of both a Java implementation and an informal architectural description makes `CryptoDB` an appropriate choice to demonstrate our approach.

### 4.1 Conformance Stage

During this stage, we iterated the process of annotating the code and extracting OOGs until an extracted OOG and the target architecture had roughly similar tiers, similar hierarchical decomposition, and a similar number of components in each tier. We then modeled the target architecture, analyzed its communication integrity and established traceability to the code.

#### 4.1.1 Gather Available Documentation

We studied the architectural documentation available for `CryptoDB`, which included DFDs (e.g., Fig. 2) and accompanying, explanatory text [19]. We used these materials only as a guide, as the implementation departed from this documentation in some respects (see Sect. 4.1.4 for an example). We mined the diagrams for the architecturally significant



```

Family DFD = { // Snippets from security family DFD.acme
  Property Type TrustLevelT =
    Enum {Low, Medium, High, None, Unknown};
  Property Type HowFoundT =
    Enum {HardCoded, FromPointer, Mixed, Unknown};
  ...
  Element Type DFDTTypeT = {
    Property trustLevel : TrustLevelT;
    Property howFound : HowFoundT;
    Property owner : OwnerT;
    Property name : string;
    ...
  }
  Component Type DFDCComponent extends DFDTTypeT;
  Component Type DataStore extends DFDCComponent with {
    Property readProtection : DataAccessProtectionT;
    Property writeProtection : DataAccessProtectionT;
    Property secrecyMethod : InformationSecrecyT;
    Property integrityMethod : InformationIntegrityT;
    ...
  }
  analysis notFoundFromPointer(c:DFDCComponent):boolean =
    hasValue(c.howFound) AND !(c.howFound==FromPointer);
  rule NoComponentsFoundFromPointer = heuristic
    forall c: DFDCComponent in self.COMPONENTS |
    declaresType(c,DFDCComponent)->notFoundFromPointer(c)
    <<label : string = "Ensure there is no pointer that
    would allow one entity to spoof another.">>;
  ...
}
(a) Type definitions from our DFD family description.

System CryptoDBTarget: SyncFamily, DFD = new ... {
  ...
  Component KeyVault : DataStore = new DataStore ... {
    Property label = "KeyVault";
    Property trustLevel = High;
    Property howFound = HardCoded;
    Property owner = ThisComponent;
    ...
  }
}
(b) A component from our CryptoDB system description.

```

Figure 4: Excerpts from our Acme specification.

has these substructures collapsed, while the Level-0 graph shows only the top-level domains (Fig. 6).

#### 4.1.4 Model the Target Architecture

We then documented the target architecture for CryptoDB using Acme, basing it largely on the aforementioned DFDs. We also represented the subarchitectures corresponding to second-level DFDs. We used Acme groups to partition the architecture into broad areas of responsibility. We added directional connectors based on the text describing the architecture. In many cases, the points-to connectors were the reverse of the data flow connectors in the DFDs.

We used the DFD Acme family (Section 3.2). Fig. 4(b) shows the Acme description of the CryptoDB KeyVault, illustrating how the architectural types are instantiated and the properties set on the individual component instances. SyncFamily is a “mix-in” family used by the conformance analysis. In particular, SyncFamily defines properties that maintain the traceability between the target architecture and the code. The full Acme specification of the CryptoDB target architecture is in an online appendix [1].

Getting the target architecture right required a process of iteration. This was due in large measure to the ways in which the implementation departed from the architecture.

The implementation, in our case, was a demonstrative implementation found in a security book, not a fully faithful implementation of the design. In particular, the implementation was simplified in many respects. For instance, Kenan identifies in principle a number of subcomponents of the cryptographic provider: an initializer, an encoder, a receipt manager, an engine interface, and others [19, §6.1]. In the implementation, the provider was nearly monolithic; few of these distinct responsibilities were actually allocated to separate objects. We had to modify our target architecture to accommodate the casual way in which the implementation realized the described architecture. (If we had not done so, we would have had to deal with these discrepancies later in the conformance stage.) In a system in which the implementation more faithfully realized the design, less iteration would be required.

This iteration was also necessitated by the mismatch between conceptual and implementation-level architectures. In Acme, a component may have a *representation*, which is a decomposition of that component into its constituent parts. Thus, a component may be a subsystem with its own internal structure. In an OOG, and the resulting built architecture, there is also a natural notion of decomposition—a component collapses one or more objects, which are its constituent parts—but it is governed by ownership and type structures.

Another change we made in the process of iteration was to delete the external interactors. Although useful for showing the endpoints of the system, they did not correspond to any code elements (since they were, of course, external to the system), so they were always going to show up as absences.

#### 4.1.5 Analyze Communication Integrity

We then analyzed communication integrity using ARCH-CONF. The resulting *conformance view* of the target architecture (Fig. 5) shows convergences, divergences, and absences (see Sect. 2.3) and enables tracing from each architectural element to the corresponding lines of code.

## 4.2 Enforcement Stage

### 4.2.1 Define Code-Level Constraints

We defined domain links and assumptions, as discussed in Sect. 3.1. The resulting domain link declarations in the top-level class were largely expected (Fig. 6). There were bidirectional links between PROVIDERS and CONSUMERS. But the links were unidirectional from PROVIDERS and KEY-MANAGEMENT to KEYSTORAGE. There were no links from CONSUMERS to KEYSTORAGE. Note that domain link permissions are not transitive.

### 4.2.2 Set Architectural Types

We then imported the DFD security family into the CryptoDB system, and assigned to components and connectors in the system the corresponding types we had defined earlier. For example, we assigned to the component instance KeyVault the DataStore type. In Acme, an element can have multiple types. See Fig. 4(b).

### 4.2.3 Set Architectural Properties

By assigning to the component KeyVault the DataStore type, all the properties defined on the DataStore type and its supertypes, e.g., howFound, become available on KeyVault

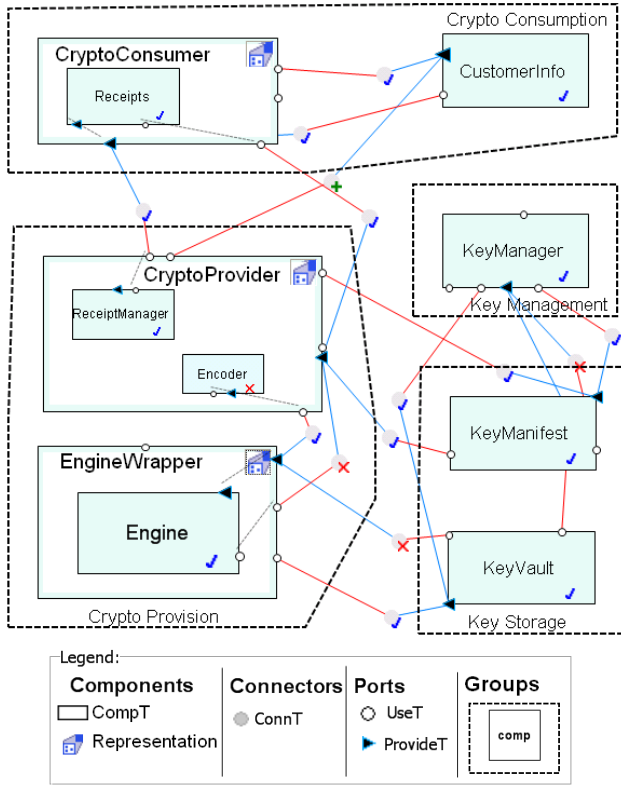


Figure 5: CryptoDB conformance view in AcmeStudio, showing convergences (✓), divergences (+), and absences (✗).

and their values can be set independently on each instance. For example, we set the `trustLevel` to be `High` (see Fig. 4(b)).

#### 4.2.4 Set Architectural Constraints

**Security Constraints.** Similarly, once we import the family and assign types to architectural elements, all the security-related predicates are immediately enforced.

**Application-Specific Constraints.** For CryptoDB, we found several restrictions on the communication allowed in the architecture. We formalized these as constraints and added them to the target architecture. Some of the constraints include:

1. `KeyManager` should not connect to `EngineWrapper`;
2. `KeyVault` should not point to `KeyManifest`;
3. Only `KeyManager` and `EngineWrapper` should have access to `KeyVault`.

All these constraints reflect our understanding of the security requirements of the target architecture, and indeed

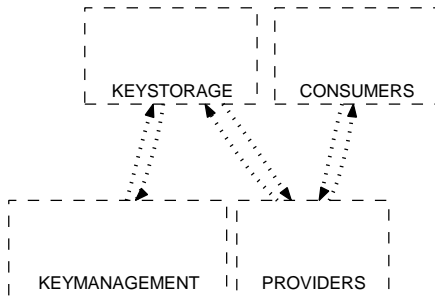


Figure 6: CryptoDB object graph (Level-0).

they are all derived from commentary in Kenan’s book [19]. For example, constraint 3 is an adaptation of the following remark: “Access to the key vault [...] should be granted to only security officers and the cryptographic engine” (p. 71). The key manager is the component through which security officers access the system, hence we arrive at constraint 3.

We formalized these constraints using the Acme predicate language [29]:

1. forall c : Component in KeyManagement.MEMBERS | !connected(c, EngineWrapper)
2. !pointsTo(KeyVault, KeyManifest)
3. forall c : SyncCompT in self.COMPONENTS | pointsTo(c, KeyVault) -> c.label == "KeyManager" or c.label == "EngineWrapper"

### 4.3 Evaluation Summary

We successfully related the security architecture and the implementation.

**Renames.** Because SCHOLIA uses a structural comparison algorithm to compare the built and designed architectures, we were able to analyze conformance despite naming discrepancies—e.g., `KeyManager` versus `KeyTool`.

**Conformance Findings.** Overall, the top-level components in the target architecture and the implementation were mostly consistent, as indicated by the large number of convergences (Fig. 5).

One interesting divergence corresponds to communication in the implementation that is omitted from the target architecture, namely the reference from `CryptoProvider` to `CustomerInfo`. This divergence really exists in the implementation and is not a false positive. We traced the edge to the code and realized that the provider maintains a list of all the `EncryptionRequest` objects it receives.

Looking inside some of the top-level components revealed more interesting differences. For example, a Level-2 DFD (Fig. 2) shows an `Encoder` component inside the `Provider`. However, the `Encoder` is implemented using a helper class `Utils`, which is never instantiated, hence the corresponding absence in the conformance view. We could resolve this absence by modifying the code to instantiate a singleton `Utils` object without affecting the system behavior.

While modeling the target architecture, we confronted several architecture–implementation discrepancies of this nature. We ultimately dealt with them, in most cases, by modifying the target architecture to match the implementation. This was necessary because of the departures that the CryptoDB implementation made from the cryptographic database architecture. Had we not reconciled the differences at that stage, we would have had much more noise to sort through in the conformance operation.

For example, we could have added the external interactors to the target architecture. However, because external interactors represent components that are outside of the system, they will always show up as absences. In the end, we decided to exclude the external interactors.

Naturally, distinguishing between deliberate departures from the architecture and genuine architecture violations requires careful judgment. However, we view it as a strength of our approach that architects have the opportunity to exercise their judgment to exclude uninteresting violations.

In other cases, we refined the annotations. For instance, we had initially modeled all instances of `CryptoReceipt` and

`CompoundReceipt` in a `RECEIPTS` domain inside `CustomerManager`. As a result, the analysis flagged the `ReceiptManager` inside `CryptoProvider` as an absence. Then we looked more carefully at how the `Provider` and the `CustomerManager` exchanged these objects. This led us to define a `RCPTMGR` domain inside `provider` for `CompoundReceipts`, and left the `CryptoReceipts` in the `RECEIPTS` domain inside `mgr` (Fig. 3).

**Constraint Violations.** We then added the aforementioned constraints (Section 4.2.4) to the target architecture and used the `AcmeStudio` tool to verify them. The tool thus gave us a positive assurance that the implementation had no architectural violations.

## 4.4 Defect Prevention

To further validate our approach, we manually injected a manufactured architecture violation into the `CryptoDB` code to confirm that our constraints would catch it. Specifically, we coupled the `Provider` and the `LocalKeyStore`. According to constraint 3 above, the `Provider` is not allowed to point to the `LocalKeyStore` in this way. In the architecture, access to the `KeyVault` is highly restricted due to the sensitivity of the contents.

When we modified the code in this way and ran our analysis, the conformance view showed an additional divergence between `provider` and `keyVault`, and the predicate raised a warning about the architectural violation in the conformance view. In this case, the domain link checks alone would not have caught this violation. Both `engine` and `provider` are peers in the same `PROVIDERS` domain (Fig. 3). So, there was already a domain link from `PROVIDERS` to `KEYSTORAGE` for `engine` to access `keyStore`. But we did not want `provider` to access `keyStore`.

Admittedly, the annotations could place `provider` and `engine` in separate domains, and use domain links to enforce restrictions. However, putting each object in a separate domain means losing a key abstraction that domains provide, namely grouping related objects into tiers. In addition, this would require annotations that are more verbose.

## 5. DISCUSSION

**Validity.** *Internal threats* may indicate that factors other than the technique determined the results. *External threats* limit the extent to which the results can be generalized.

**Internal Validity.** One threat to internal validity is whether the security runtime architecture `SECORIA` uses truly serve the purposes of threat modeling. For the target architecture to be comparable to the one `SCHOLIA` extracts from the implementation, it must follow certain conventions. There could be some DFDs that are not easily expressible in this manner. For instance, we currently restrict a given `DFD Process` to belong to a single `Boundary`. In addition, there could be boundaries that may not be expressible using domains or tiers, e.g., network boundaries, if they are not directly visible in the implementation.

The `SECORIA` security analysis covers only a small subset of architectural defects and design flaws. Only defects related to local or global, structural connectivity in the runtime architecture are in scope. When conducted by humans, threat analysis incorporates more subtle checks, admittedly in a much more informal or ad-hoc way. The goal of any architecture-level analysis is not to replace a review by security experts. Threat model analysis can be only partially

automated since it still requires human creativity and intuition to uncover subtle security flaws that cannot easily be generalized. Nevertheless, `SECORIA` can help identify many of the small issues, provide suggestions on how to fix them in some cases, and encourage designers to focus on the complex interactions between subsystems.

**External Validity.** *Can the approach find architectural violations in other systems?* The underlying `SCHOLIA` approach has been evaluated on several systems, some of which are at least an order of magnitude larger than `CryptoDB`. So we are confident that the architectural extraction and conformance analysis can scale.

In order to evaluate the security analysis, however, the challenge is to find an implementation together with a non-proprietary security runtime architecture, ideally one that is documented by a domain expert. Moreover, the tools that currently support the approach—while not Java-specific—are implemented in the Java Eclipse development environment. On the other hand, the biggest producer of DFDs, Microsoft, uses mostly the Visual Studio development environment for C++ and C# programming.

**Limitations.** `SECORIA` suffers from several limitations.

**Annotation Overhead.** The main effort in `SECORIA` is in adding the annotations. Inferring the annotations automatically, while still an open research problem, is becoming increasingly feasible. Still, even without inference, the cost of applying `SECORIA` may be justified for a security architecture, since security exploits can be quite expensive. For example, the Microsoft Security Response Center estimates that a security bug which requires a security bulletin may cost Microsoft alone hundreds of thousands of dollars [15].

**Distributed Systems.** A technical limitation of `SECORIA` is that the underlying `SCHOLIA` approach cannot extract the built architecture of a distributed system. It is precisely such systems that exhibit many security vulnerabilities.

**Scalability.** Admittedly, evaluating `SECORIA` on a 3-KLOC system does not demonstrate scalability. However, the scalability of `SECORIA` is limited mostly by that of `SCHOLIA`'s whole-program static analysis for extracting the as-built runtime architecture. The static analysis of runtime architecture is much less mature than that of code architectures. For example, even 3,000 lines of code is considered out of reach for some heavyweight static analyses.

## 6. RELATED WORK

**Previous Work.** In earlier work [2], Abi-Antoun and Aldrich presented the `SCHOLIA` approach to extract a hierarchical object graph, abstract it and use it to analyze conformance. `SECORIA` specializes `SCHOLIA` to security runtime architectures and focuses on enforcement at the code and architectural levels, once the architecture and the implementation have been related. `SCHOLIA` analyzes communication integrity. `SECORIA` reasons about security using architectural types, properties and constraints that the previous architectures did not have. For instance, the `CryptoDB` system was designed and documented by a security expert, unlike some of the previous systems which were documented by university professors. This increases the external validity of the result.

Also in earlier work [5], Abi-Antoun, Wang and Torr defined a model for reasoning about security at the architectural-level, following the `STRIDE` methodology

commonly used in threat modeling. The previous security model and checker were implemented using custom code. SECORIA formalizes the same security model using ADL support for architectural types and properties, and defines the checks as predicates. Using an ADL gives the benefit of having a declarative model, with less room for error compared to custom code. Moreover, with a declarative model, power users can add properties and predicates to extend or customize the security analysis, something they cannot do with a monolithic tool. Finally, this enables SECORIA to reap additional benefits when representing an extracted architecture in an ADL. By enriching the target architecture with the security model, SECORIA can find security vulnerabilities in the implementation.

**Architectural Extractors.** With the exception of object graph analysis, points-to analysis and shape analysis, which we discuss below, most static extractors do not track objects precisely. Instead, they represent structural information with respect to files, directories, packages or classes, rather than objects. For example, they express that a class is part of some package, or a package is nested inside another package. In SCHOLIA, ownership type annotations provide containment information at the level of objects, i.e., an object is “part of” another object. SCHOLIA can thus distinguish between different instances of the same class that are in different domains. Many papers often implicitly use the term “component” to mean a “package,” “module” or a collection of classes [11].

**Static Analysis.** A number of existing approaches extract, fully automatically, various non-hierarchical object graphs [17]. While these approaches can be useful for showing object interactions, they share a fundamental scalability limitation: for programs of any size, they produce a diagram with so many objects that, in practice, the diagram is unusable for architectural purposes.

Many of these approaches rely on points-to analysis, a fundamental static analysis to determine the set of objects whose addresses may be stored in variables or fields of objects. All previous points-to analyses produce non-hierarchical graphs [37, 27]. Also, most points-to analyses achieve a granularity that is no coarser than an object or a set of objects. Shape analysis is a kind of points-to analysis that summarizes possible relationships among objects at run-time in a precise way [33]. A shape graph also does not collapse some objects underneath other objects.

**Security Testing.** Analysis offers substantial benefits beyond those of testing alone. Perhaps most significantly, since SECORIA is based on static analysis, it can reveal information about all possible runs of a program, while testing is limited to a small number of runs. This difference is particularly important in the security domain. Similar to testing is dynamic conformance analysis, which instruments and monitors a system [35, 34].

**Conformance Analysis.** There are many approaches for analyzing architecture conformance to a code architecture (see Knodel and Popescu [21] for a comparative analysis, as well as the extensive survey by Ducasse and Pollet [11]). Indeed, SCHOLIA is modeled after, and complements, Reflexion Models (RM) [31]. RM, however, cannot handle runtime architectures [2].

**Consistency Management.** Many approaches analyze inconsistencies between different but related views, that are typically at the same level of abstraction, such as a UML

class diagram and a corresponding UML sequence diagram. That is a problem in *horizontal conformance* [11]. Here, we check conformance between views at different levels of abstraction, namely an implementation and a target architecture, which is a problem in *vertical conformance* [11].

**Code Generation.** SecureUML [24] recommends a model-driven approach in which security constraints are imposed on a model that is later elaborated into code. Like all model-driven approaches, it is useful only for construction of new systems, not for analysis of existing implementations. Our approach is appropriate for use on existing code, requiring only annotations. Another difference is that SecureUML is based on a code architecture.

**Architectural Analysis.** Various architectural-level security analyses have been proposed [30, 10]. For example, UMLsec [18] extends UML with secrecy, integrity and authenticity, to allow design-level analysis of security weaknesses. 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 and cannot account for legacy code. SECORIA can analyze an existing system after the fact.

**Code-Level Analyses.** Architectural analysis matches the way experts reason about security or privacy better than a purely code-based strategy. SECORIA complements, and does not supplant, code-level analyses. Moreover, the traceability between a security architecture and the code that our approach derives can benefit other static analyses. Until now, due to the lack of traceability, much of the security design intent generated during threat modeling has not been easily accessible to other code quality tools. For instance, a static analysis checking for buffer overruns [13] can use this traceability to assign to its warnings more appropriate priorities based on a more holistic view of the system.

Many code-level analyses use type systems to track information flow or check other security properties [32]. SECORIA is a complementary, architecture-centric analysis.

**Design Enforcement.** Many approaches can enforce local, modular, code-level constraints, e.g., [14]. SECORIA is complementary and can enforce structural constraints on the global architectural structure, which may not correspond to any explicitly declared code structure.

**Threat Modeling Tools.** Microsoft has made publicly available a threat modeling tool [36, 26]. Similarly, the EU-funded CORAS Project [9] proposed its description and tool support for threat modeling. While these tools recently added enhanced architectural-level analyses, neither tool currently relates a security architecture to an implementation. Moreover, the tools implement their analyses using custom code, rather than declarative types and predicates. We believe SECORIA significantly improves the current state of the art in threat modeling tools.

**Operating Systems.** The operating systems research community has developed information flow control techniques to provide for the security of applications built from potentially untrusted components [39, 23, 7]. This work is similar to ours in motivation but quite different in approach. These approaches rely on the introduction of new operating systems designed to enforce security requirements, while we approach the problem at the application level by tying architecture to implementation.

## 7. CONCLUSION

We presented the first approach, SECORIA, to analyze, entirely statically, a security runtime architecture for some information flow vulnerabilities and for conformance to an object-oriented implementation.

Our evaluation showed how SECORIA can detect code changes that introduce architectural violations on a real system. SECORIA's architecture-based analysis matches the way experts reason about security during threat modeling, and nicely complements other code-oriented strategies.

**Acknowledgements.** Abi-Antoun was supported by his faculty startup fund at Wayne State University. Barnes was supported in part by the Office of Naval Research, United States Navy, N000140811223, as part of the HSCB project under OSD; by the US Army Research Office under grant numbers DAAD19-02-1-0389 to Carnegie Mellon University's CyLab and DAAD19-01-1-0485; by the National Science Foundation under grant 0615305; and by the Software Engineering Institute at CMU.

## 8. REFERENCES

- [1] [www.cs.wayne.edu/~mabianto/cryptodb/](http://www.cs.wayne.edu/~mabianto/cryptodb/), 2010.
- [2] M. Abi-Antoun and J. Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In *OOPSLA*, 2009.
- [3] M. Abi-Antoun and J. M. Barnes. Enforcing conformance between security architecture and implementation. Technical Report CMU-ISR-09-113, Carnegie Mellon Univ., 2009.
- [4] M. Abi-Antoun and J. M. Barnes. STRIDE-based security model in Acme. Technical Report CMU-ISR-10-106, Carnegie Mellon Univ., 2010.
- [5] M. Abi-Antoun, D. Wang, and P. Torr. Checking threat modeling data flow diagrams for implementation conformance and security. In *ASE*, pages 393–396, 2007. Short paper/poster.
- [6] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.
- [7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.
- [8] P. Clements et al. *Documenting software architectures: views and beyond*. Addison-Wesley, 2003.
- [9] CORAS. <http://coras.sourceforge.net>, 2006.
- [10] Y. Deng, J. Wang, J. J. P. Tsai, and K. Beznosov. An approach for modeling and analysis of security system architectures. *Knowledge & Data Eng.*, 15(5), 2003.
- [11] S. Ducasse and D. Pollet. Software architecture reconstruction: a process-oriented taxonomy. *TSE*, 35(4), 2009.
- [12] D. Garlan, R. Monroe, and D. Wile. Acme: architectural description of component-based systems. In *Component-based systems*. Cambridge Univ., 2000.
- [13] B. Hackett et al. Modular checking for buffer overflows in the large. In *ICSE*, 2006.
- [14] H. J. Hoover and D. Hou. Using SCL to specify and check design intent in source code. *TSE*, 32(6), 2006.
- [15] M. Howard and D. LeBlanc. *Writing secure code*. Microsoft Press, 2nd edition, 2003.
- [16] M. Howard and S. Lipner. *The security development lifecycle*. Microsoft Press, 2006.
- [17] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *TSE*, 27(2), 2001.
- [18] J. Jürjens. *Secure systems development with UML*. Springer, 2004.
- [19] K. Kenan. *Cryptography in the database*. Addison-Wesley, 2006. Code at [http://kevinkenan.blogspot.com/downloads/cryptodb\\_code.zip](http://kevinkenan.blogspot.com/downloads/cryptodb_code.zip).
- [20] D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in object-oriented framework reuse. *Empirical Softw. Eng.*, 12(3), 2006.
- [21] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *WICSA*, 2007.
- [22] R. Koschke. Architecture reconstruction: Tutorial on reverse engineering to the architectural level. In *Intl. Summer School on Software Engineering*, 2008.
- [23] M. Krohn et al. Information flow control for standard OS abstractions. In *SOSP*, 2007.
- [24] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *UML*, 2002.
- [25] D. C. Luckham and J. Vera. An event-based architecture definition language. *TSE*, 21(9), 1995.
- [26] Microsoft threat modeling tool. <http://msdn.microsoft.com/en-us/security/sdl-threat-modeling-tool.aspx>, 2007.
- [27] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 14(1), 2005.
- [28] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *ECOOP*, 2009.
- [29] R. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163R, Carnegie Mellon Univ., 2001.
- [30] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure software architectures. In *IEEE Security & Privacy*, 1997.
- [31] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *TSE*, 27(4), 2001.
- [32] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.
- [33] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.
- [34] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *TSE*, 32(7), 2006.
- [35] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE*, 1996.
- [36] F. Swiderski and W. Snyder. *Threat modeling*. Microsoft Press, 2004.
- [37] P. Tonella and A. Potrich. *Reverse engineering of object-oriented code*. Springer, 2004.
- [38] P. Torr. Demystifying the threat-modeling process. *IEEE Security & Privacy*, 3(5), 2005.
- [39] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI*, 2008.