Enforcing Conformance between Security Architecture and Implementation

Marwan Abi-Antoun   Jeffrey M. Barnes

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Analysis at the level of a runtime architecture matches the way experts reason about security or privacy better than a purely code-based strategy. However, the architecture must still be correctly realized in the implementation.

We previously developed SCHOLIA to analyze, at compile time, communication integrity between arbitrary object-oriented code, and a rich, hierarchical intended runtime architecture, using typecheckable annotations. This paper applies SCHOLIA to security runtime architectures. Having established traceability between the target architecture and the code, we extend SCHOLIA to enforce structural architectural constraints. At the code level, annotations enforce local, modular constraints. At the architectural level, predicates enforce global constraints. We validate the end-to-end approach in practice using a real 3,000-line Java implementation, and enforce its conformance to a security architecture designed by an expert.

Contents

1 Introduction

Companies such as Boeing and Microsoft have been using *threat modeling* [1] as a lightweight approach to reason about security, to capture and reuse security expertise and to find security design flaws during development. During threat modeling, development teams construct security architectures that are later reviewed by security experts.

Although threat modeling often finds security design flaws, it suffers from the two known problems of *architectural extraction* and *conformance analysis*. 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 [2] or privacyLevel, and enumerate all possible communication between the more trusted and the less trusted components of the system. But if the architecture does not show all the communication that is present in the system, the results of an architectural-level analysis may be incorrect. While any architecture-based approach suffers from these problems, security architectures pose special challenges.

A security architecture[1] is a *runtime architecture* which shows runtime components and connectors, uses hierarchical decomposition, and partitions a system into tiers [3]. Unfortunately, the tools for extracting and analyzing the conformance of a runtime architecture are immature.

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 one or more program runs [4], may miss important objects or relations that arise only in other executions.

The *communication integrity* property [5] defines one notion of conformance as: "Each component in the implementation may only communicate directly with the components to which it is connected in the architecture" [6].

Abi-Antoun and Aldrich previously developed an approach, SCHOLIA, to analyze at compile time communication integrity between arbitrary object-oriented code, and a rich, hierarchical intended runtime architecture [7]. SCHOLIA uses typecheckable annotations and establishes traceability between the target architecture and the code.

This paper's contributions are the following:

- An application of SCHOLIA to analyze conformance between a Java implementation and a security runtime architecture, entirely statically and using annotations;
- An illustration of enforcing constraints at the code level and architecturally;
- A validation using a real 3,000-line Java implementation of a security architecture designed by an expert.

This paper is organized as follows. Section 2 discusses relating a security architecture to code. Section 3 describes enforcing architectural intent. Section 4 presents an evaluation of the approach on a real system. Finally, we discuss related work (Section 5) and conclude.

Our design intent-based analysis for relating security architectures and code has two main stages: the *conformance* stage and the *enforcement* stage. Each stage consists of several steps. The overall process is iterative, so the term does not imply following these steps in a strict sequence.

---

[1]Threat modeling typically uses Data Flow Diagrams (DFDs) with security-specific annotations to describe how data enters, leaves and traverses the system by showing data sources and destinations, the relevant processes that data goes through and the trust boundaries in the system [1]. This paper uses a slightly different architectural style: a security architecture shows points-to (not data flow) connectors, has no explicit data stores or external interactors, and uses more general boundaries that are tiers.

2 Conformance Stage

The object-oriented analogues to a *runtime architecture* and a *code architecture* would be a global *object diagram* and a *class diagram*, respectively. While in the class diagram a single node represents a class and summarizes the properties of all of its instances, an object diagram represents different instances as distinct nodes, with their own properties. Thus, an object diagram makes explicit the object structures that are only implicit in a class diagram [8]. One can generalize an object diagram into a runtime architecture which abstracts objects into components, and represents how those components interact. Usually, distinct component instances have different values for architectural properties such as trustLevel.

Architectural reasoning about security is best accomplished with a runtime architecture, not a code architecture. The appendix [9, §2] contains class diagrams extracted from CryptoDB, the secure database system we evaluate in Section 4. These class diagrams are not comparable to the security architecture drawn by its designer [9, §1].

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. Moreover, such a snapshot shows only one execution, meaning the developer may miss important objects or relations that show up 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. A flat object graph will again have a plethora of objects that is unreadable, even for relatively small programs, and will not convey sufficient architectural abstraction to be used for conformance analysis (See the appendix for flat object graphs for CryptoDB [9]).

A central difficulty is that architectural hierarchy is not readily observable in arbitrary code. Some language-based solutions, e.g., ArchJava [6], specify architectural hierarchy and instances directly in code. But ArchJava's breaking extensions restrict how objects are used and require re-engineering an existing Java system [10].

In contrast, SCHOLIA achieves hierarchy in an object graph by having a developer pick a top-level object as a starting point, then use local modular ownership annotations in the code [11, 12] to impose a conceptual hierarchy on objects, with architecturally significant objects near the top of the hierarchy and data structures further down. The annotations and object graph extraction are at the core of the approach, so we discuss these next.

2.1 Ownership domain annotations

A developer uses local, modular (one class at a time) annotations to specify, in code, object encapsulation, logical containment and architectural tiers, which are not explicit constructs in general-purpose programming languages.

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 [12]. Each object is assigned to a single ownership 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. For example, "DOM Type obj" declares a reference `obj` of type `Type` in a domain `DOM` (Fig. 1).

Domain names are arbitrary, except for a few special annotations we discuss later. Ideally, a domain name conveys architectural intent. We also use capital letters to distinguish domain names from other program identifiers. A typechecker validates the annotations and identifies inconsistencies between the annotations and the code.

The SCHOLIA tools use existing language support for Java 1.5 annotations, which tends to be verbose (Fig. 3) [11]. In this paper, we use a more readable syntax, focusing on a core Java language (Fig. 1). An

$$
\begin{aligned}
L \in ClassDecl \quad &::=\quad \texttt{class } C\texttt{<}\overline{\alpha}\texttt{>} \; [\texttt{extends } C'\texttt{<}\overline{\beta}\texttt{>}] \\
&\qquad\quad \texttt{assumes } \overline{\alpha'} \rightarrow \overline{\alpha''} \; \{\overline{L}; \overline{D}; \overline{F}; \ldots\} \\
L \in LinkDecl \quad &::=\quad \texttt{link } d \rightarrow d'; \\
D \in DomDecl \quad &::=\quad [\texttt{public}] \; \texttt{domain } d; \\
F \in FieldDecl \quad &::=\quad T \; f; \\
n \quad &::=\quad d \mid v \\
p \quad &::=\quad \alpha \mid n.d \mid \texttt{shared} \\
T \in Type \quad &::=\quad p_{owner} \; C\texttt{<}\overline{p_{params}}\texttt{>} \\
\alpha, \beta \in DomParam \quad &\qquad\quad C, C' \in ClassName
\end{aligned}
$$

Figure 1: Simplified annotation syntax [12].

```
1   class LocalKeyStore<KEYID> {
2     private domain OWNED, KEYDATA;
3     public domain KEYS;
4     link KEYS -> KEYID, KEYS -> KEYDATA, OWNED -> KEYS;
5     assume OWNER -> KEYID;
6     private OWNED List<KEYS LocalKey<KEYDATA,KEYID>> keys;
7
8     public unique List<KEYS LocalKey<...>> getKeys() {
9       unique List<KEYS LocalKey<...>> copy = copy(keys);
10      return copy;
11    }
12  }
13  class List<ELTS T> {
14    private domain OWNED; // Private domain
15    OWNED Object[] rep; // Private representation
16    ELTS T obj; // Virtual field declaration
17  }
18  class LocalKey<KEYID,KEYDATA> {
19    assume OWNER -> KEYID, OWNER -> KEYDATA;
20    private KEYDATA String keyData; // encrypted key
21    private KEYID String keyId; // encrypted key id
22    ...
23    private OWNER SecretKeySpec key; // Make peer to self
24  }
```

Figure 2: `LocalKeyStore` and `LocalKey` annotations.

overbar represents a sequence.

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

Logical containment   A public domain provides *logical containment* and makes an object conceptually "part of" another object. Having access to an object gives the ability to access objects inside all its public

```
1   @DomainParams({"KEYID", "KEYDATA"}) // Domain parameters
2   @DomainAssumes({"OWNER->KEYID", "OWNER->KEYDATA"})
3   class LocalKey {
4     private @Domain("KEYDATA") String keyData;
5     private @Domain("KEYID") String keyId;
6     ...
7   }
```

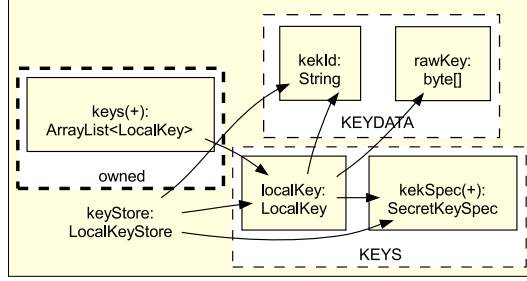Figure 3: Using concrete Java 1.5 annotations [11].

Figure 4: `LocalKeyStore` OOG.

domains. For example, in Fig. 2, `LocalKeyStore` declares a public domain, `KEYS`, to hold `LocalKey` objects (line 3).

Strict encapsulation A private domain provides *strict encapsulation*. For instance, a `public` method cannot return an alias to an object inside a private domain, even though the Java type system allows returning an alias to a field marked as `private`. For example, `LocalKeyStore` stores the list of `LocalKey` objects, keys, in a private domain, `OWNED` (line 6). As a result, the accessor `getKeys` must return a shallow copy of the list, and cannot return an alias (line 8).

Domain parameters `List` is part of the Java standard library. Library code is often parametric with respect to application components. For example, the `List` class is parametric in two ways (Fig. 2). First, `List` is parametric in the type of the element stored in the `List`, hence the `T` type parameter (line 13). `List` also takes a formal domain parameter, `ELTS`, that contains the elements stored in a `List` instance (this assumes an ownership model where all the objects referenced by a `List` object are in the same domain). Whenever a `List` is used, the formal domain parameter must be bound to another domain in scope, e.g., `KEYS`. The internal representation of the `List` is in a private domain. Because a `List` has virtual references to the elements it holds, the annotation system allows a *virtual field* declaration to simulate that (line 16).

Similarly, `LocalKey` takes the `KEYID` and `KEYDATA` domain parameters (line 18). In turn, `LocalKeyStore` takes a `KEYID` domain parameter (line 1). For example, `LocalKeyStore` binds its local domain `KEYDATA` to `LocalKey`'s `KEYDATA` parameter (line 6).

Special annotations There are additional special annotations that add expressiveness [12]: `unique` indicates an object to which there is only one reference, such as a newly created object, or an object that is passed linearly from one domain to another. One ownership domain can temporarily lend an object to another and ensure that the second domain does not create persistent references to the object by marking it `lent`. An object that is `shared` may be aliased globally but may not alias non-`shared` references, and little reasoning can be done about `shared` references.

2.2 Object graph extraction

SCHOLIA extracts a hierarchical object graph that provides architectural abstraction by ownership hierarchy and by types, the Ownership Object Graph (OOG).

The visualization uses box nesting to indicate containment of objects inside domains, and domains inside objects (Fig. 4). 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 for brevity, "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.
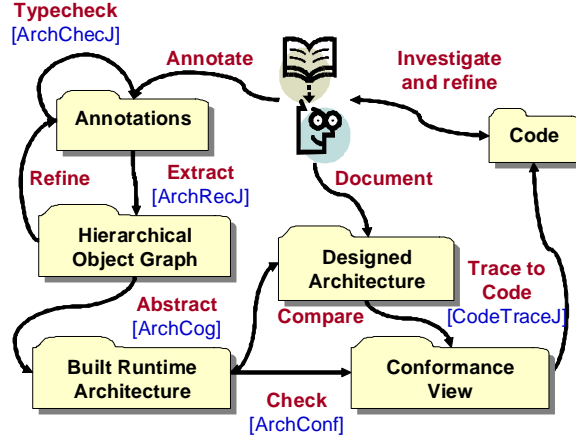
5

Figure 5: Overview of the SCHOLIA approach.

In a runtime architecture, it is common practice to represent multiple objects at runtime with one canonical component. Of course, at runtime, there are many `LocalKey` objects, but the OOG shows a single representative.

An extracted object graph is *sound* in two respects. First, it approximates all objects and all relations possibly created between those objects. Second, an object graph does not represent one runtime object as separate nodes. The latter, aliasing soundness, relies on the type system's guarantee that two objects in different domains cannot be assigned to each other, and thus can never alias (but two objects in the same domain may alias) [12]. Aliasing soundness is important for an architectural-level security analysis. For instance, if an architecture showed the same entity as two components, one could assign them different values for the key `trustLevel` property and potentially invalidate the analysis results.

In addition, ownership-parametric library code, such as `List` (Fig. 2), often creates interesting architectural relationships in application objects, when these formal parameters are bound to actual domains on specific objects created by the application. The static analysis resolves these parameters to ensure that the relevant object relations appear at the level of the global application object structures—hence the edge from `keys` to `localKey` (Fig. 4).

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.

2.3 Communication integrity and traceability

When reasoning about security, we want to ensure that the designed architecture is a conservative abstraction of all the objects in the implemented system and the relations between those objects at runtime. Thus, the goal is to show the worst case of possible communication between objects at runtime.

Of course, a static approach may generate false positives. An object graph obtained statically may show relations that may never exist at runtime, due to infeasible paths. However, an object graph extracted using a dynamic analysis can show the exact number of instances and the actual relations in a given program run, it may not reflect important objects or relations that show up only in other executions.

To analyze communication integrity, SCHOLIA follows the extract-abstract-check strategy [13], as follows (Fig. 5): (1) Document the designed runtime architecture; (2) Add annotations to the code and typecheck them (ARCHCHECKJ); (3) *Extract* an object graph (ARCHRECJ); (4) *Abstract* an object graph into a built architecture (ARCHCOG); (5) Structurally *compare* the built and the designed architectures; (6) *Check* and enforce *communication integrity* in the designed architecture (ARCHCONF).

A developer can perform any of the following: (a) Iteratively refine the annotations based on visualizing an extracted object graph, before abstracting it; (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 struc-

tural 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 considers that the implementation highlights an omission in the architecture.

In the terminology of Murphy [13], the analysis identifies:

- **Convergence:** a node or an edge that *is in both* the built and the designed architectures (shown as ✔);
- **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 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 the target architecture and the code, the enforcement stage can identify additional implementation-level violations of the architectural intent. At the code level, annotations can enforce local, modular constraints. In addition, architectural predicates can enforce global constraints.

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 somewhat. But a developer can still add communication paths by adding domain links, declaring additional domain parameters and passing additional domain arguments at object allocation sites. Code inspections could audit revisions that modify the domain link annotations more closely. However, the annotations enforce modular constraints, so it is still necessary to identify code modifications that impact the global architectural structure.

Extracting the up-to-date built architecture and analyzing its conformance to a target architecture makes it easier to trigger an architectural review. Various constraints can be enforced by a visual inspection of the conformance view. The structural constraints in the target architecture can always enforce these policies. Indeed, empirical evidence suggests that such policies are frequently needed during software evolution. For instance, a study using a well-designed framework (JHotDraw) 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 [14].

3.1 Code-level constraints

We use *domain link* annotations to specify explicit policies that govern how a domain can reference objects in other domains [12]. We illustrate them again by example.

A `LocalKey` *assumes* that its owning domain can access the `KEYID` and `KEYDATA` domain parameters. In turn, when a `LocalKeyStore` instantiates a `LocalKey`, and binds `KEYID` and `KEYDATA` to `KEYID` and `KEYS`, respectively, `LocalKeyStore` must satisfy those permissions. For the first one, it declares a *domain link* from `KEYS` to `KEYID` (line 4). For the second one, it links `KEYS` to `KEYDATA`.

3.2 Architectural constraints

Documenting an architecture in an architecture description language (ADL) enables various architectural-level analyses. We use Acme, a general-purpose ADL with mature tool support [15]. An ADL allows setting architectural types, properties and constraints to specify architectural intent.

Architectural types The built architecture does not usually have rich architectural types. In principle, one could add some to the built architecture, while abstracting an object graph, by mapping implementation

types to architectural types. However, this is only a first approximation, because different instances of the same implementation type such as `HashMap`, could correspond to architectural components of different types.

Relating the built and the designed architectures, and enriching the designed architecture, can uncover additional violations of the architectural intent in the code.

Architectural properties   In previous work, we defined element-level properties, such as `trustLevel`, to support an architectural-level analysis to identify spoofing or tampering [2].

Structural constraints   First-order logic predicates can enforce structural constraints [16], such as:
- Component instance c_1 never directly connects to Component instance c_2;
- A Component of type t_1 never directly connects to a Component of type t_2;
- No component in Group g_1 communicates directly with any component in Group g_2.

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

4   Evaluation

We validate the end-to-end approach using CryptoDB, a secure database system designed by security expert Kevin Kenan in his book [17]. 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 make CryptoDB an appropriate choice to demonstrate our approach.

During the evaluation, the coauthors played the roles of architect and developer. The architect controlled the target architecture, and the developer controlled the annotations and the code. In particular, the developer was not allowed to change the target architecture himself, but instead had to convince the architect that the proposed change was architecturally justified. Also, we forbade ourselves from making changes to the source code, except to annotate it.

4.1   Conformance stage

During this stage, we annotated the code, extracted OOGs, and iterated the annotations until the OOG had roughly similar tiers, a similar hierarchical decomposition, and a similar number of components in each tier, when visually compared to the target architecture. We then constructed the target architecture, analyzed its communication integrity, and established traceability to the code.

4.1.1   Gather available documentation

We studied the available architectural documentation, which consisted of various Data Flow Diagrams (DFDs) along with accompanying, explanatory text [17]. We used these materials only as a guide, because the implementation departed from this documentation in some respects (see Section 4.1.4 for an example). We mined the diagrams for the architecturally significant elements, the top-level tiers, and the hierarchical system decomposition.

It quickly became apparent that the documentation and the code used slightly different terminology. For example, the documentation referred to a "key manager," but the code had a `KeyTool`. In the following discussion, we use the names from the implementation. A mapping between the two terminologies is in the appendix [9].

4.1.2   Annotate the code

We organized instances of the core types into four top-level domains, as follows (Fig. 7):
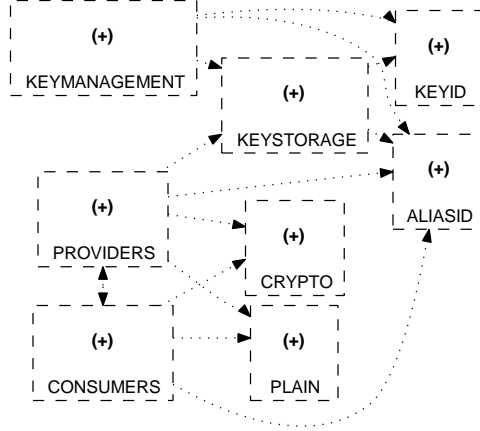- `CONSUMERS`: has `CustomerManager`, and `EncryptionRequest`s, such as `CustomerInfo` and `CreditCardInfo`;

Figure 6: CryptoDB OOG (Level 0) with Strings.

- PROVIDERS: has `Provider`, `EngineWrapper`;
- KEYSTORAGE: has `KeyAlias`es and `LocalKeyStore`;
- KEYMANAGEMENT: has a `KeyTool` object.

For several classes C_i, we also defined one or more nested domains D_i, which we refer to using the $C_i{::}D_i$ notation:

- `CustomerManager::RCPTS` has `CryptoReceipt`s;
- `LocalKeyStore::KEYS` has instances of `LocalKey`, `SecretKeySpec`, etc.; (Fig. 4)
- `Provider::RCPTMGR` has `CompoundReceipt` objects;

Refining the annotations. As part of applying the approach, we refined the annotations. One such refinement was related to reasoning about `String` objects. In many applications, `String` objects are uninteresting, and annotated with `shared`. Unless the user requests otherwise, an OOG purposely excludes objects that are `shared` since they often add uninteresting clutter.

When reasoning about security, `String` objects can be interesting. Indeed, in CryptoDB, much communication takes place through `String`s. To better understand this communication, we declared different domains for plain-text (`PLAIN`), encrypted (`CRYPTO`), alias identifier (`ALIASID`), and key identifier (`KEYID`) `String`s. In particular, the annotation typechecker checks that these `String`s are not assigned to each other, a perfectly valid operation in Java.

For example, Fig. 6 shows only the top-level domains and summarizes the field references between objects in those domains using dotted edges. However, when analyzing conformance later, we simplified the OOG by binding all the additional parameters for `PLAIN`, `CRYPTO`, etc., to the `shared` domain. This required changing only the binding of these domain parameters in the top-level class, and changing a few lines of annotations in the top-level class.

4.1.3   Extract object graphs

We then used ARCHRECJ to extract an OOG from the annotated code. An OOG illustrates some of the key differences between a code and a runtime architecture. For example, inside the `Provider`'s `RCPTMGR` domain, a `CompoundReceipt` encapsulates a `HashMap` that maps `String` to `CryptoReceipt` objects. Separately, each `EncryptionRequest` inside the `CONSUMERS` domain has a `HashMap` that maps `String`s to `String`s.

Abstraction by types. In addition to abstraction by ownership hierarchy, an OOG can provide abstraction by types, where a developer specifies which types are more architecturally significant than others [18]. Based on this optional input, an OOG merges closely related objects, in a given domain, based on their declared types. For example, with abstraction by types turned on, the CryptoDB OOG merges objects of type `CustomerInfo`, and `CreditCardInfo` in the `CONSUMERS` domain, because their classes implement the `EncryptionRequest` interface (Fig. 7).

```
interface EncryptionRequest<PLAIN> {
  unique Map<PLAIN String, PLAIN String> getPlaintexts();
}
class DecryptionResults<PLAIN>
     implements EncryptionRequest<PLAIN> {
 private domain OWNED;
 OWNED Map<PLAIN String, PLAIN String> plaintexts = new ...;
 unique Map<...> getPlaintexts() {
   return copy(plaintexts); // Return copy of field
 }
}
class CompoundReceipt<RCPTS,PLAIN,CRYPTO,ALIASID> {
 private domain OWNED;
 OWNED Map<PLAIN String,RCPTS CryptoReceipt> receipts = new ...;
}
class CryptoReceipt<CRYPTO,ALIASID> {
 CRYPTO String ciphertext;
 CRYPTO String iv;
 ALIASID String aliasId;
}
class Provider<RQSTS,PLAIN,CRYPTO,ALIASID,RCPTS...> {
 public domain RCPTMGR;
 public RCPTMGR CompoundReceipt<...> encrypt(RQSTS EncryptionRequest<PLAIN> rqst...) {...}
 public unique DecryptionResults<PLAIN> decrypt(RCPTMGR CompoundReceipt<...> wrapper) {...}
}
class CreditCardInfo<PLAIN>
     implements EncryptionRequest<PLAIN> {
 public unique Map<...> getPlaintexts() {
   unique Map<PLAIN String, PLAIN String> map = new ...;
   map.put(CustomerManager.CREDIT_CARD, creditCard);
   ...
   return map;
 }
}
class CustomerManager<CNSMRS,PRVDRS,PLAIN,CRYPTO,ALIASID...> {
 public domain RCPTS;
 PRVDRS Provider<CNSMRS,PLAIN,CRYPTO,ALIASID,RCPTS...> prov;
 void testEncrypt() {
   CNSMRS CreditCardInfo<PLAIN> cci = new CreditCardInfo();
   prov.RCPTMGR CompoundReceipt<...> cciRcpts = prov.encrypt(cci, "cci");
 }
 void testDecrypt() {
   prov.RCPTMGR CompoundReceipt<...> pii = new ...;
   RCPTS CryptoReceipt<CRYPTO,ALIASID> r1 = new ...;
   pii.addReceipt(FIRST_NAME, r1);
   CNSMRS DecryptionResults<PLAIN> piiPlaintexts = prov.decrypt(pii);
 }
}
class System {
 domain CNSMRS,PRVDRS,KMGT,KSTR...;
 KSTR LocalKeyStore<...> store = new LocalKeyStore();
 KMGT KeyTool<KSTR...> tool = new KeyTool(store);
 CNSMRS CustomerManager<...> mgr = new CustomerManager(store);
}
```

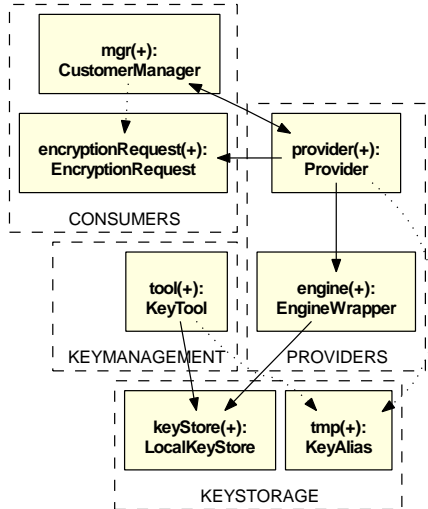Figure 7: Portions of CryptoDB with annotations.

Figure 8: CryptoDB OOG (Level 1), no Strings.

Hierarchy. Hierarchy allows both high-level and detailed understanding, by expanding or collapsing selected elements. Fig. 8 shows the top-level domains and the objects directly inside them, with their substructure collapsed, after binding all the domain parameters containing `String`s to `shared`. In Fig. 9, we manually expanded the substructures of `mgr`, `provider`, `engine`, etc. Here, we collapsed the substructure of `keyStore` (which appears in Fig. 4).

4.1.4 Model the target architecture

We designed a target architecture using Acme. We based this architecture largely on the available DFDs (Section 4.1.1). We represented the DFD processes and data stores using components. We used the Acme representation feature to include subarchitectures corresponding to second-level DFDs. We used Acme groups, depicted with dashed lines, to partition the architecture into broad areas of responsibility.

We added directional connectors based on the information in the textbook. In many cases, the points-to connectors were the reverse of the data flow connectors in the DFDs.

We went through a process of iteration to get the architecture right. 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 [17, §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 necessary.

This iteration was partly due to the mismatch between conceptual and implementation-level architectures. In Acme, a component is just a transparent view of a more detailed decomposition specified by the representation of that component [19]. In an OOG, and the resulting built architecture, a component collapses one or more objects that constitute its parts, according to their ownership and type structures.

In general, developers do not use hierarchical decomposition rigorously in DFDs. But in SCHOLIA, annotations can push almost any object underneath any other object in the ownership hierarchy (without creating cycles). A child object may or may not be encapsulated by its parent object: a child object can still be referenced from outside its owner if it is part of a public domain of its parent, or if a domain parameter is linked to a private domain [12]. This allows a developer to use annotations to control the
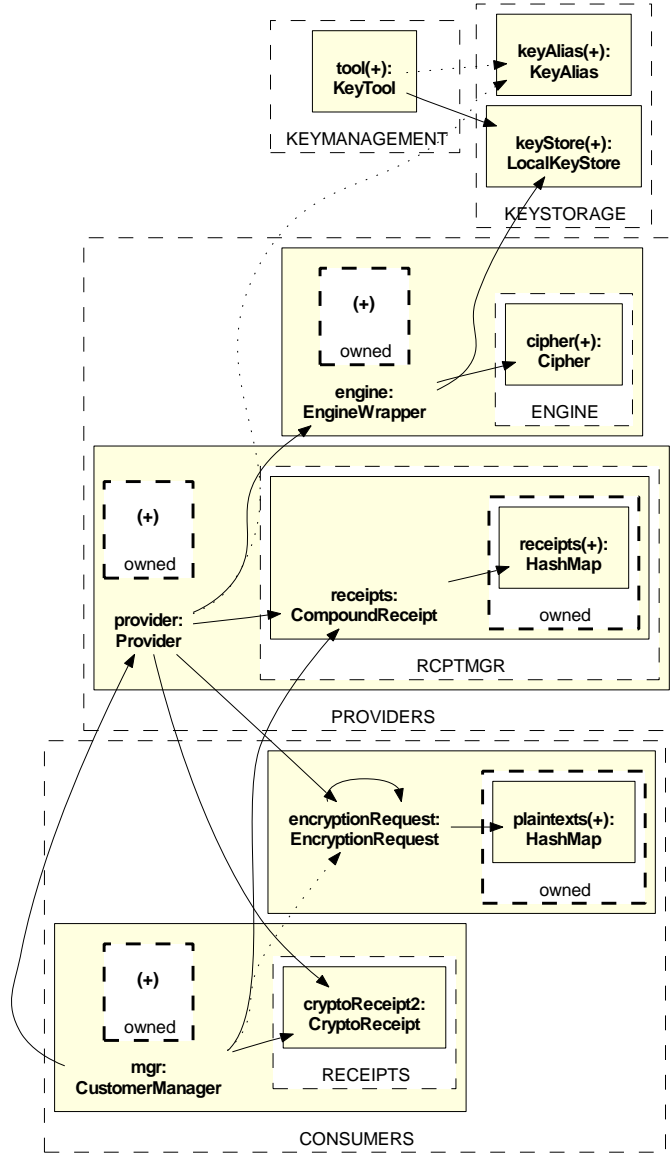
Figure 9: CryptoDB OOG (Level 2).

system decomposition in the OOG.

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) and so did not facilitate the analysis.

While iterating the annotations, we determined the similarity between the OOG and the target architecture by visual inspection.

4.1.5 Analyze communication integrity

Object graphs tend to expose low-level implementation details. In SCHOLIA, when internal state is placed in private domains, the OOG abstraction tool, ARCHCOG, can leverage the semantic distinction between private and public domains. For example, in `LocalKeyStore`, the private OWNED domain contains an `ArrayList` of `LocalKeys` (Fig. 4). In the OOG (Fig. 9), we made the private domains appear as OWNED(+).
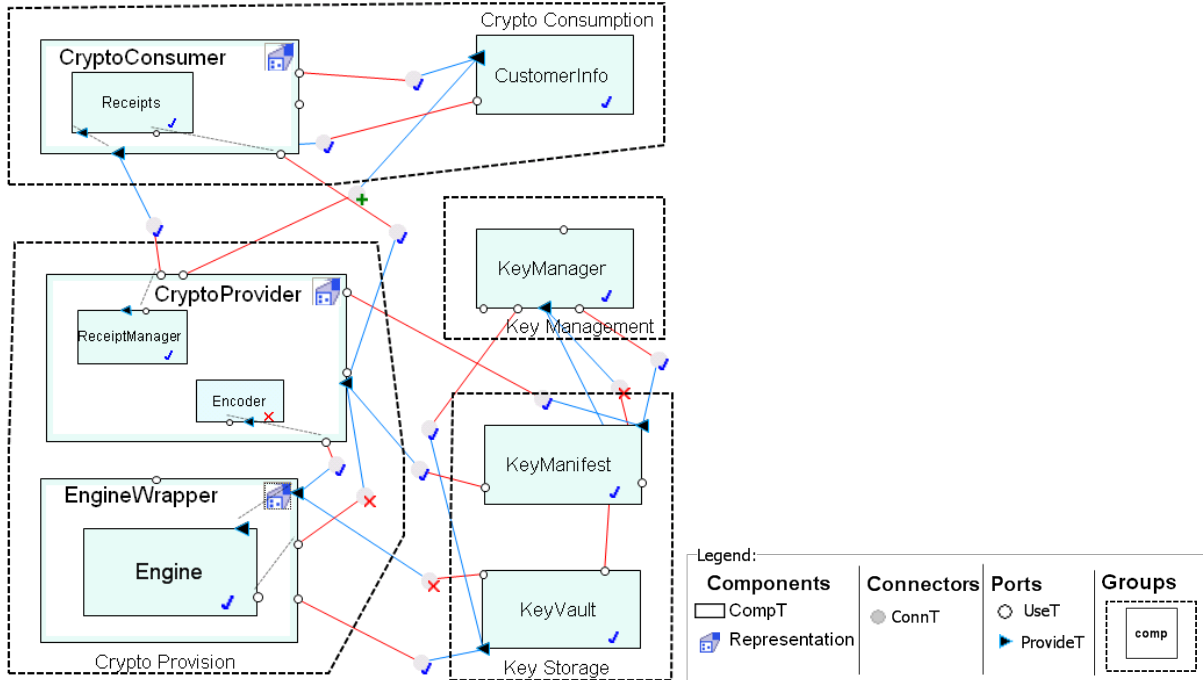
Figure 10: CryptoDB conformance view in Acme.

We then analyzed communication integrity. ARCHCONF creates a *conformance view* of the target architecture (Fig. 10), which shows convergences, divergences, and absences, and has traceability to the code.

4.2 Enforcement stage

Having analyzed conformance and established traceability between the designed architecture and the CryptoDB code, we now move to the enforcement stage.

4.2.1 Define code-level constraints

We defined domain links and assumptions, as discussed in Section 2.1. The resulting domain link declarations in the top-level class were largely expected. As can be seen in Fig. 6, there are bidirectional links between `PROVIDERS` and `CONSUMERS`. But the links are unidirectional from `PROVIDERS` and `KEYMANAGEMENT` to `KEYSTORAGE`. Of course, there are no links from `CONSUMERS` to `KEYSTORAGE`. Note that domain link permissions are not transitive.

4.2.2 Set architectural constraints

We wrote architectural constraints to express restrictions on the communication allowed in the architecture. Then, we formalized these 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 they are all roughly derived from commentary in Kenan's book [17]. For example, constraint 3 is an adaptation of the following remark: "Access to the key vault [. . .] should be granted to only security

```
class Provider<RQSTS,KSTR...> {
  assume OWNER->KSTR;
  KSTR LocalKeyStore<KEYID> keyStore; // (1)
  OWNER EngineWrapper<KSTR...> engine;

 Provider(KSTR LocalKeyStore<KEYID> store) {
   // Inject architectural violation
   this.keyStore = store; // (2)
   this.engine = new EngineWrapper(store);
 }
}
```

Figure 11: Injected architectural violation.

officers and the cryptographic engine" (p. 71). The key manager is the architectural agent that security officers use, hence we arrive at constraint 3.

Once we wrote these constraints, we formalized them using the Acme predicate language [16], as follows:

1. `forall c :  Component in KeyManagement.MEMBERS |`
        `!connected(c, EngineWrapper)`

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

The full Acme specification of the target architecture, including the architectural style and the definition of the `pointsTo` predicate above, is in the appendix [9].

4.3   Evaluation summary

SCHOLIA was able to successfully relate the security architecture and the implementation.

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

Conformance findings. Overall, the top-level components in the target architecture (based on a Level-1 DFD) and the implementation were mostly consistent, as indicated by the large number of convergences (Fig. 10).

Drilling down into the representations of the some of the top-level components revealed more interesting differences. For example, a Level-2 DFD (in the appendix [9]) 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's behavior.

In the process of modeling the target architecture, we confronted a number of 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. Naturally, distinguishing between deliberate departures from the architecture and genuine architecture violations requires careful judgment. However, we view it as a strength of our iterative approach that architects have the opportunity to exercise their judgment in this way to forestall uninteresting violation reports from the tool.

In other cases, we refined the annotations. For instance, we had initially modeled all instances of `CryptoReceipt` and `CompoundReceipt` in a `RECEIPTS` domain inside the `CustomerManager`. As a result, the analysis flagged the `ReceiptManager` inside the `CryptoProvider` as an absence. Then we looked more carefully at how the `Provider` and the `CustomerManager` exchanged these objects (Fig. 7). This led us to define a `RCPTMGR` domain inside `provider` for `CompoundReceipt`s, and left the `CryptoReceipt`s in the

`RECEIPTS` domain inside `mgr` (Fig. 9).

Constraint violations. Once we added the constraints to the target architecture, we used the AcmeStudio tool to verify them. Due to the traceability we established between the architecture and source code, we can have some confidence that the implementation meets these constraints.

To further validate our approach, we modified the CryptoDB code, injecting a manufactured architecture violation to confirm that our constraints would catch it. Specifically, we coupled the `Provider` and the `LocalKeyStore` as shown in Fig. 11. According to constraint 3 above, the `Provider` is not allowed to point to the `LocalKeyStore` in this way. In the architecture, access to the key vault is highly restricted due to the sensitivity of the contents.

When we modified the code in this way and ran our analysis, the predicate raised a warning about the architectural violation in the conformance view. It is true that enforcing predicates at the architectural level is not novel. But since our approach establishes traceability between the architecture and the code, enforcing constraints at the architectural level allows enforcing global constraints on the application structure in the code. In addition, the domain link checks alone would not have caught this violation. Both `engine` and `provider` are peers in the same `PROVIDERS` domain (Fig. 8). So, there must already be a domain link from `PROVIDERS` to `KEYSTORAGE` for `engine` to access the key vault.

5   Related Work

Architectural security analysis. Various architectural-level security analyses have been proposed [20, 21]. For example, UMLsec [22] extends UML with secrecy, integrity and authenticity, to allow analyzing 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 legacy code. One could use the approach in this paper to analyze an existing system, after the fact, by adding annotations to the code.

Conformance analysis. There are many approaches to analyze conformance to a code architecture (see Knodel and Popescu for a comparative analysis [23]). However, the tool support for analyzing, *statically*, communication integrity in a runtime architecture is much less mature. SCHOLIA is modeled after, and complements, Reflexion Models [13], which handles the code architecture only.

Language-based solutions. Like ArchJava [6], SCHOLIA integrates architectural intent into source code, but instead of extending Java with architectural components and ports, SCHOLIA uses language support for annotations. The evaluation in this paper did not require re-engineering a system to follow ArchJava's rules [10]. In this paper, we only added annotations to the code and typechecked them using a tool.

Abi-Antoun et al. also added ownership domain annotations to several subject systems [11, 24]. The study in this paper has novel aspects. We added domain links (they were part of the formal model, but previously not supported by the tools) and reasoned about `String`s instead of marking them `shared`. Moreover, the CryptoDB target architecture was drawn by a security expert instead of a professor [7], and has richer types, properties and constraints than the previous architectures that SCHOLIA analyzed, which increases the external validity of the result.

Code generation. SecureUML [25] recommends a model-driven approach in which security constraints are imposed on a model that is later elaborated into code. Of course, 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.

Code-level analyses. Architectural analysis matches the way experts reason about security or privacy better than a purely code-based strategy. Our approach 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 [26] can use this traceability to assign to its warnings more appropriate priorities based on a more holistic view of the system.

Security testing. Analysis offers substantial benefits beyond those of testing alone. Perhaps most significantly, since our approach 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 [27, 4].

Design enforcement. Many approaches can enforce local, modular, code-level constraints, e.g., [28]. Our approach is complementary, and can enforce structural constraints on the global runtime architectural structure.

6 Conclusion

We presented the first approach to relate, entirely statically, a security runtime architecture to a program written in a widely used object-oriented language, using annotations. Such an approach can increase the effectiveness of reasoning architecturally about the security of existing systems, because it ensures that the architecture is a faithful representation of the code, which is ultimately the most reliable and accurate description of the built system.

Acknowledgements

APPENDIX

A Documented Architectures

Fig. 12 is a Level-1 DFD.
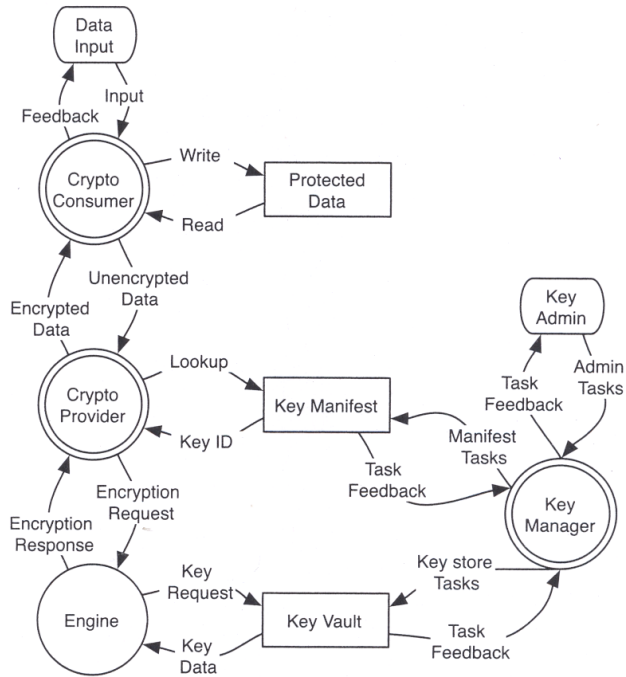
Figure 12: CryptoDB documented DFD (Level 1) [17, Fig. 9.1].

Fig. 13 is a Level-2 DFD which refines in place some of the components in the Level-1 DFD (Fig. 12).
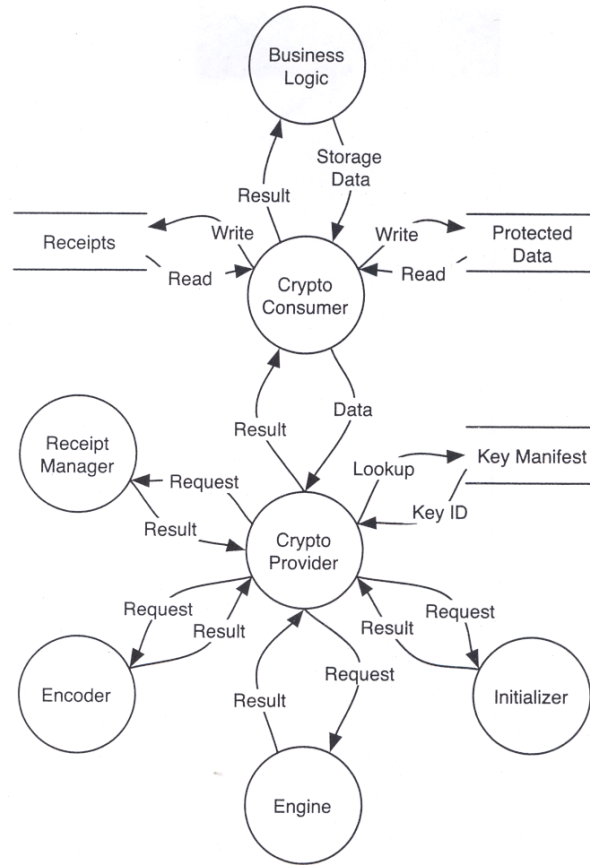
Figure 13: CryptoDB documented DFD (Level 2) [17, Fig. 6.1].

B Code Architecture

We used the Eclipse UML tool [29] to extract from the implementation various views of the code architecture. Fig. 14 shows the package structure. Fig. 15 shows the class diagram with a few selected core types.

Figure 14: CryptoDB layer diagram.

Figure 15: CryptoDB class diagram.

C Flat Object Graphs

Fig. 16 is a flat object graph obtained statically using PANGAEA [30].

Figure 16: CryptoDB flat object graph extracted using PANGAEA.

Figs. 17, 18 are flat object graphs obtained statically using WOMBLE [31].

Figure 17: CryptoDB flat object graph extracted using WOMBLE (rendered using GraphViz dot).

Figure 18: CryptoDB flat object graph extracted using WOMBLE (rendered using GraphViz fdp).

D   Acme Source Code for Designed Architecture

Here, we reproduce the entire architectural model, in Acme [15]. We provide both the family file, SyncFamily.acme, which defines the architectural family that supports SCHOLIA, and the target architecture itself, CryptoDBTarget.acme.

D.1   SyncFamily.acme

This file defines the architectural family SyncFamily. The properties defined here are used by SCHOLIA for conformance analysis.

import $AS_GLOBAL_PATH/families/TieredFam.acme;

Family SyncFamily **extends** TieredFam **with** {

    analysis isSrcComponent(d1 : SyncCompT, conn : SyncConnT) : **boolean** =
        connected(conn, d1) **and**
        exists src : SyncUserT **in** conn.**ROLES** | **exists** put : SyncUseT **in** d1.**PORTS** |
            declaresType(src, SyncUserT) **and declaresType**(put, SyncUseT)
            and attached(src, put);

    analysis isDstComponent(d2 : SyncCompT, conn : SyncConnT) : **boolean** =
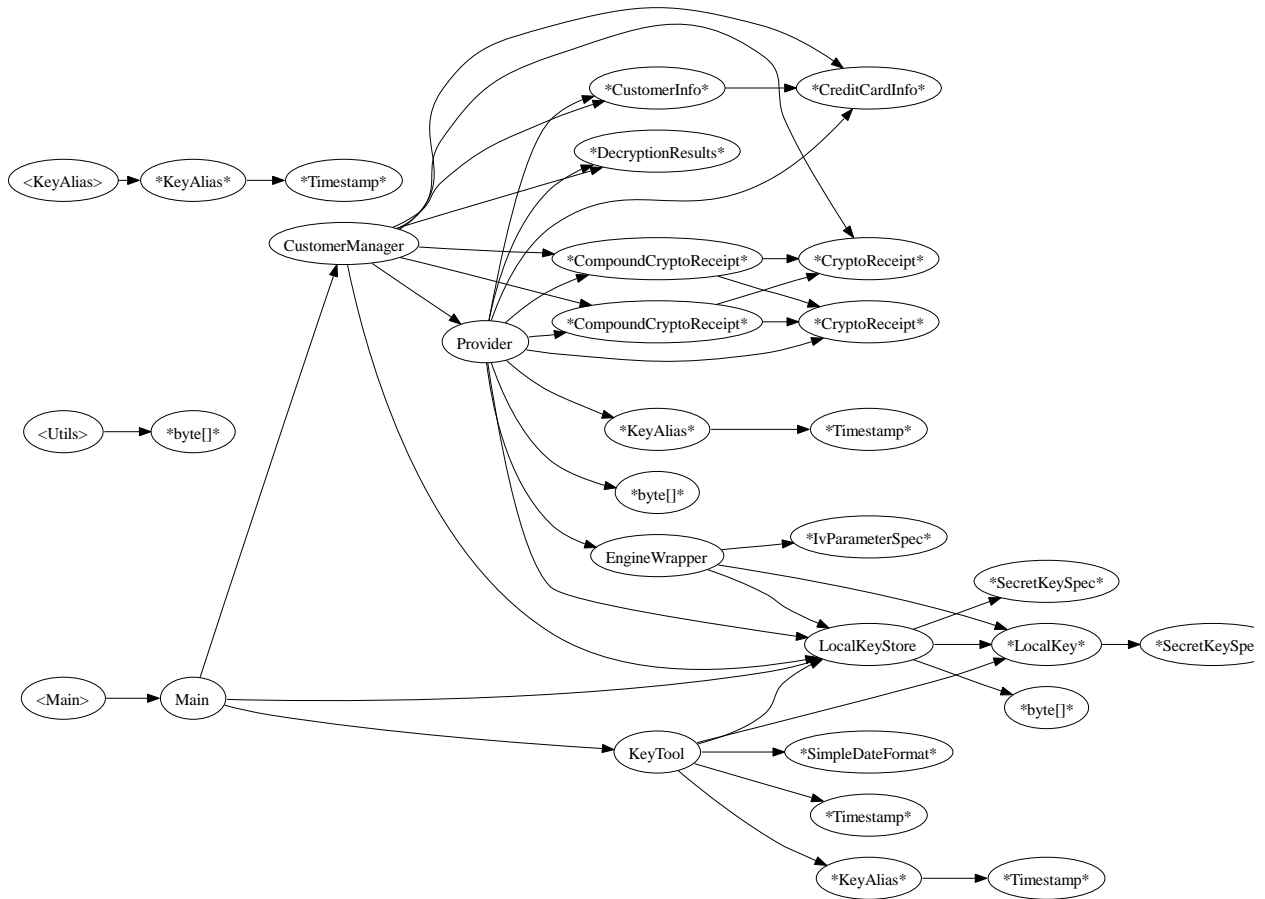   **connected**(conn, d2) **and**
        exists dst : SyncProviderT **in** conn.**ROLES** | **exists** get : SyncProvideT **in** d2.**PORTS** |
            declaresType(dst, SyncProviderT) **and declaresType**(get, SyncProvideT)
            and attached(dst, get);

    analysis pointsTo(d1 : SyncCompT, d2 : SyncCompT) : **boolean** =
        exists conn : SyncConnT **in** self.**CONNECTORS** |
            isSrcComponent(d1, conn) **and** isDstComponent(d2, conn);

    Role Type SyncUserT **extends** userT **with** {
        Property syncStatus : **int**;
    }
    Component Type SyncCompT **extends** TierNodeT **with** {
        Property syncStatus : **int**;
        Property label : **string**;
        Property hasDetail : **boolean**;
        Property detailStatus : **int**;
        Property traceability : **string**;
    }
    Connector Type SyncConnT **extends** CallReturnConnT **with** {
        Property syncStatus : **int**;
        Property label : **string**;
        Property traceability : **string**;
        Property summary : **int**;
    }
    Port Type SyncUseT **extends** useT **with** {
        Property syncStatus : **int**;
    }
    Port Type SyncProvideT **extends** provideT **with** {
        Property syncStatus : **int**;

```
        }
    Role Type SyncProviderT extends providerT with {
        Property syncStatus : int;
    }
}
```

D.2   CryptoTargetDB.acme

This file defines the target architecture itself, including the constraints we discussed in the paper.

import families/SyncFamily.acme;

System CryptoDBTarget : SyncFamily = **new** SyncFamily **extended with** {

```
    Component KeyVault : SyncCompT = new SyncCompT extended with {
        Port KeyVault : SyncProvideT = new SyncProvideT;
        Port KeyManager : SyncUseT = new SyncUseT;
        Port EngineWrapper : SyncUseT = new SyncUseT;

        Property label = "KeyVault";
    }
    Component CryptoProvider : SyncCompT = new SyncCompT extended with {
        Port KeyManifest : SyncUseT = new SyncUseT;
        Port CryptoProvider : SyncProvideT = new SyncProvideT;
        Port CustomerManager : SyncUseT = new SyncUseT;
        Port EngineWrapper : SyncUseT = new SyncUseT;

        Property label = "CryptoProvider";

        Representation CryptoProvider_Rep = {
            System CryptoProvider_Rep : SyncFamily = new SyncFamily extended with {
                Component ReceiptManager : SyncCompT = new SyncCompT extended with {
                    Port ReceiptManager : SyncProvideT = new SyncProvideT;
                    Port CryptoProvider : SyncUseT = new SyncUseT;

                    Property label = "ReceiptManager";
                }
                Component Encoder : SyncCompT = new SyncCompT extended with {
                    Port CryptoProvider : SyncUseT = new SyncUseT;
                    Port Encoder : SyncProvideT = new SyncProvideT;

                    Property label = "Encoder";
                }
            }
            Bindings {
                CustomerManager to ReceiptManager.CryptoProvider;
                EngineWrapper to Encoder.CryptoProvider;
            }
        }
    }
    Component KeyManager : SyncCompT = new SyncCompT extended with {
        Port KeyManifest : SyncUseT = new SyncUseT;
```

```
        Port KeyVault : SyncUseT = new SyncUseT;
        Port KeyManager : SyncProvideT = new SyncProvideT;

        Property label = "KeyManager";
}
Component KeyManifest : SyncCompT = new SyncCompT extended with {
        Port KeyManifest : SyncProvideT = new SyncProvideT;
        Port KeyManager : SyncUseT = new SyncUseT;
        Port CryptoProvider : SyncUseT = new SyncUseT;

        Property label = "KeyManifest";
}
Component EngineWrapper : SyncCompT = new SyncCompT extended with {
        Port EngineWrapper : SyncProvideT = new SyncProvideT;
        Port CryptoProvider : SyncUseT = new SyncUseT;
        Port KeyVault : SyncUseT = new SyncUseT;

        Property label = "EngineWrapper";

        Representation EngineWrapper_Rep = {
                System EngineWrapper_Rep : SyncFamily = new SyncFamily extended with {
                        Component Engine : SyncCompT = new SyncCompT extended with {
                                Port Engine : SyncProvideT = new SyncProvideT;
                                Port EngineWrapper : SyncUseT = new SyncUseT;

                                Property label = "Engine";
                        }
                }
                Bindings {
                        EngineWrapper to Engine.Engine;
                        CryptoProvider to Engine.EngineWrapper;
                }
        }
}
Component CustomerManager : SyncCompT = new SyncCompT extended with {
        Port CustomerManager : SyncProvideT = new SyncProvideT;
        Port CryptoProvider : SyncUseT = new SyncUseT;
        Port CustomerInfo : SyncUseT = new SyncUseT;

        Property label = "CustomerManager";

        Representation CustomerManager_Rep = {
                System CustomerManager_Rep : SyncFamily = new SyncFamily extended with {
                        Component Receipts : SyncCompT = new SyncCompT extended with {
                                Port Receipts : SyncProvideT = new SyncProvideT;
                                Port CustomerManager : SyncUseT = new SyncUseT;

                                Property label = "Receipts";
                        }
                }
                Bindings {
```

```
            CustomerManager to Receipts.Receipts;
            CryptoProvider to Receipts.CustomerManager;
        }
    }
}
Component CustomerInfo : SyncCompT = new SyncCompT extended with {
    Port CustomerManager : SyncUseT = new SyncUseT;
    Port CustomerInfo : SyncProvideT = new SyncProvideT;

    Property label = "CustomerInfo";
}
Connector CustomerInfo_CustomerManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CustomerManager_CustomerInfo : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CustomerManager_CryptoProvider : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CryptoProvider_CustomerManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector EngineWrapper_CryptoProvider : SyncConnT = new SyncConnT extended with {
    Role user : SyncUserT = new SyncUserT;
    Role provider : SyncProviderT = new SyncProviderT;
}
Connector CryptoProvider_EngineWrapper : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyVault_KeyManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManager_KeyVault : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManifest_KeyManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManager_KeyManifest : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
```

Connector KeyManifest_CryptoProvider : SyncConnT = **new** SyncConnT **extended with** {
    Role provider : SyncProviderT = **new** SyncProviderT;
    Role user : SyncUserT = **new** SyncUserT;
}
Connector CryptoProvider_KeyManifest : SyncConnT = **new** SyncConnT **extended with** {
    Role provider : SyncProviderT = **new** SyncProviderT;
    Role user : SyncUserT = **new** SyncUserT;
}
Connector KeyVault_EngineWrapper : SyncConnT = **new** SyncConnT **extended with** {
    Role provider : SyncProviderT = **new** SyncProviderT;
    Role user : SyncUserT = **new** SyncUserT;
}
Connector EngineWrapper_KeyVault : SyncConnT = **new** SyncConnT **extended with** {
    Role provider : SyncProviderT = **new** SyncProviderT;
    Role user : SyncUserT = **new** SyncUserT;
}
Attachment CryptoProvider.CustomerManager **to** CryptoProvider_CustomerManager.user;
Attachment CustomerManager.CustomerManager **to** CryptoProvider_CustomerManager.provider;
Attachment CustomerManager.CustomerManager **to** CustomerInfo_CustomerManager.provider;
Attachment CustomerInfo.CustomerInfo **to** CustomerManager_CustomerInfo.provider;
Attachment CustomerInfo.CustomerManager **to** CustomerInfo_CustomerManager.user;
Attachment KeyManifest.CryptoProvider **to** KeyManifest_CryptoProvider.user;
Attachment KeyVault.EngineWrapper **to** KeyVault_EngineWrapper.user;
Attachment EngineWrapper.EngineWrapper **to** CryptoProvider_EngineWrapper.provider;
Attachment EngineWrapper.CryptoProvider **to** EngineWrapper_CryptoProvider.user;
Attachment EngineWrapper.EngineWrapper **to** KeyVault_EngineWrapper.provider;
Attachment EngineWrapper.KeyVault **to** EngineWrapper_KeyVault.user;
Attachment CryptoProvider.EngineWrapper **to** CryptoProvider_EngineWrapper.user;
Attachment CryptoProvider.KeyManifest **to** CryptoProvider_KeyManifest.user;
Attachment KeyVault.KeyManager **to** KeyVault_KeyManager.user;
Attachment KeyManager.KeyVault **to** KeyManager_KeyVault.user;
Attachment KeyManifest.KeyManager **to** KeyManifest_KeyManager.user;
Attachment KeyManager.KeyManifest **to** KeyManager_KeyManifest.user;
Attachment CryptoProvider.CryptoProvider **to** CustomerManager_CryptoProvider.provider;
Attachment CryptoProvider.CryptoProvider **to** KeyManifest_CryptoProvider.provider;
Attachment CryptoProvider.CryptoProvider **to** EngineWrapper_CryptoProvider.provider;
Attachment KeyManifest.KeyManifest **to** CryptoProvider_KeyManifest.provider;
Attachment KeyManifest.KeyManifest **to** KeyManager_KeyManifest.provider;
Attachment KeyManager.KeyManager **to** KeyManifest_KeyManager.provider;
Attachment KeyManager.KeyManager **to** KeyVault_KeyManager.provider;
Attachment KeyVault.KeyVault **to** EngineWrapper_KeyVault.provider;
Attachment CustomerManager.CryptoProvider **to** CustomerManager_CryptoProvider.user;
Attachment CustomerManager.CustomerInfo **to** CustomerManager_CustomerInfo.user;
Attachment KeyVault.KeyVault **to** KeyManager_KeyVault.provider;
Group KeyManagement = {
    Members {KeyManager}
}
Group CryptoConsumption = {
    Members {CustomerManager, CustomerInfo,
        CustomerManager_CustomerInfo, CustomerInfo_CustomerManager}
}

```
    Group CryptoProvision = {
        Members {CryptoProvider, EngineWrapper,
            CryptoProvider_EngineWrapper, EngineWrapper_CryptoProvider}
    }
    Group KeyStorage = {
        Members {KeyManifest, KeyVault}
    }
    rule noVaultToManifest = invariant !pointsTo(KeyVault, KeyManifest);
    rule keyManagementAndEngineDisconnected = invariant
        forall c : Component in KeyManagement.MEMBERS | !connected(c, EngineWrapper);
    rule limitedVaultAccess = invariant forall c : SyncCompT in self.COMPONENTS |
        pointsTo(c, KeyVault) −> c.label == "KeyManager" OR c.label == "EngineWrapper";
}
```

E Mapping between Architectural Components and Code Elements

The names of the components in the target architecture do not always match up exactly to the names of code elements in the Java implementation. For example, some of the Java class names are implementation-specific (LocalKeyStore instead of KeyVault). Table 1 provides a mapping between the components in the target architecture and the corresponding Java classes.

Architectural Component	Java Class	Note
CustomerManager	cryptodb.test.CustomerManager	AKA "crypto consumer"
CustomerManager.Receipts	cryptodb.CryptoReceipt	Receipts the consumer holds onto
CustomerInfo	cryptodb.test.CustomerInfo	AKA "protected data"
CryptoProvider	cryptodb.core.Provider	
CryptoProvider.ReceiptManager	cryptodb.CompoundCryptoReceipt	Used by the provider to produce receipts
CryptoProvider.Encoder	cryptodb.Utils	
EngineWrapper	cryptodb.core.EngineWrapper	
EngineWrapper.Engine	javax.crypto.Cipher	
KeyManifest	cryptodb.KeyAlias	The key manifest contains key aliases
KeyVault	cryptodb.core.LocalKeyStore	The key vault contains keys (LocalKeys)
KeyManager	cryptodb.KeyTool	

Table 1: Mapping between architectural components and code elements.

F Additional diagrams

F.1 Target architecture

The CryptoDB target architecture is in Fig. 19.

F.2 Built architecture

The C&C view obtained from the abstracted object graph is in Fig. 20.

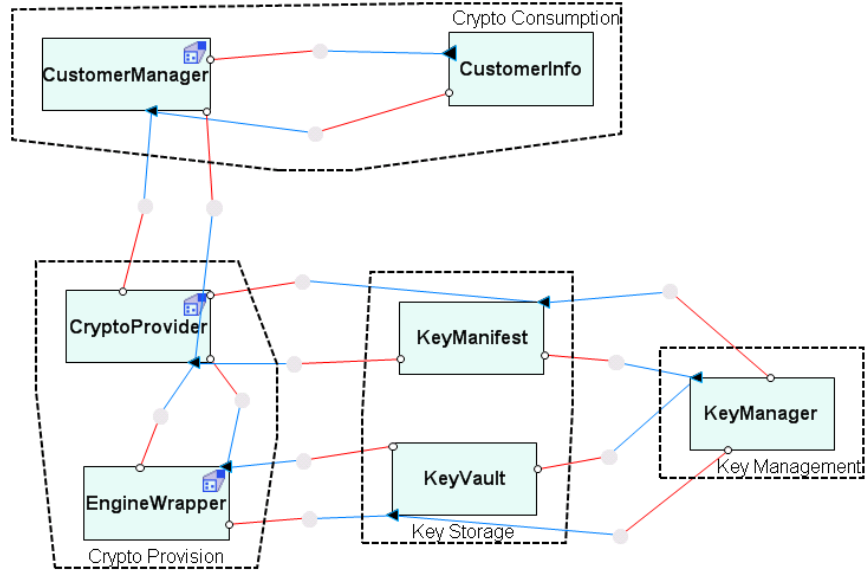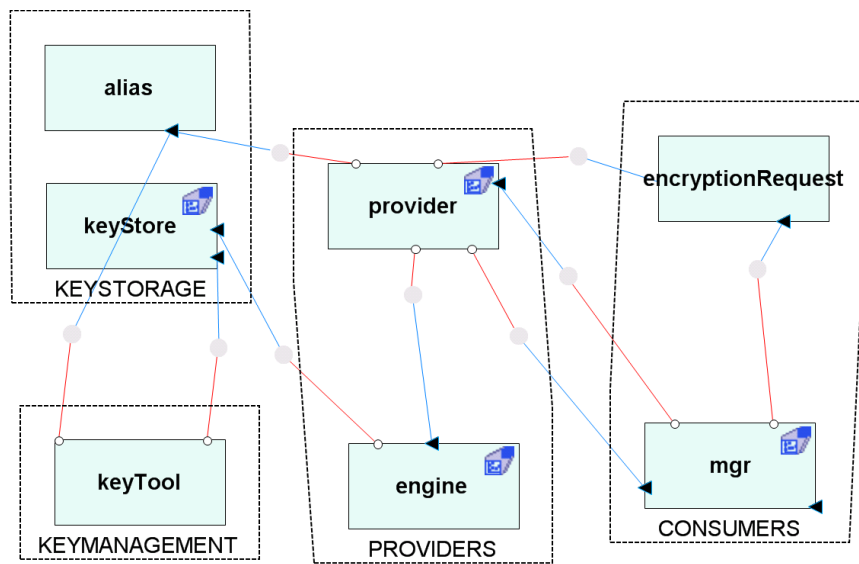Figure 19: CryptoDB target architecture in Acme.

Figure 20: CryptoDB built architecture in Acme.

F.3 Extracted object graphs

An object graph without abstraction by types shows separate `CustomerInfo` and `CreditCardInfo` (Fig. 21). With abstraction by types, these two are merged, because they both implement `EncryptionRequest`. An object graph showing explicit top-level domains for the different kinds of `String`s is in Fig. 22.
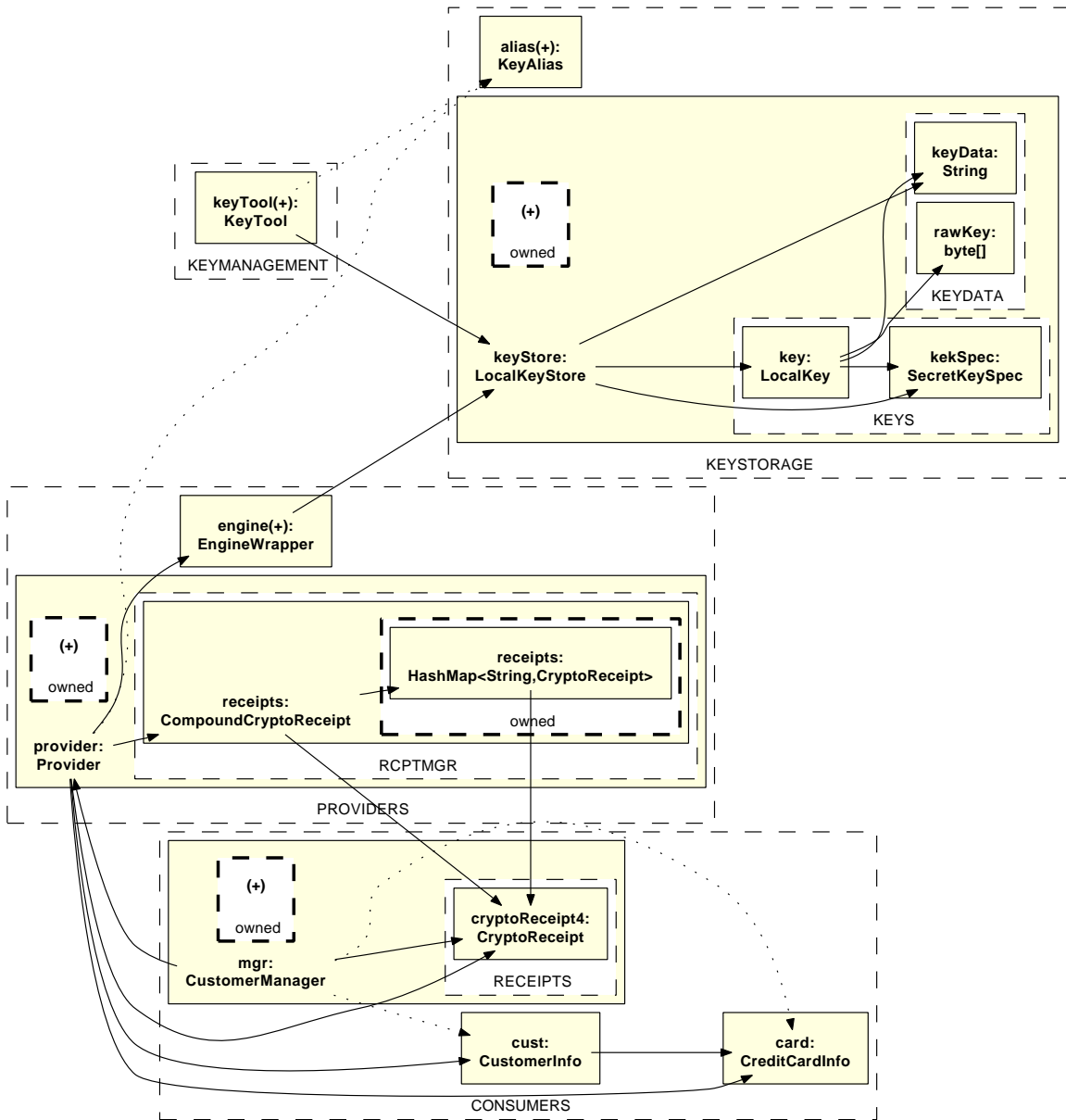
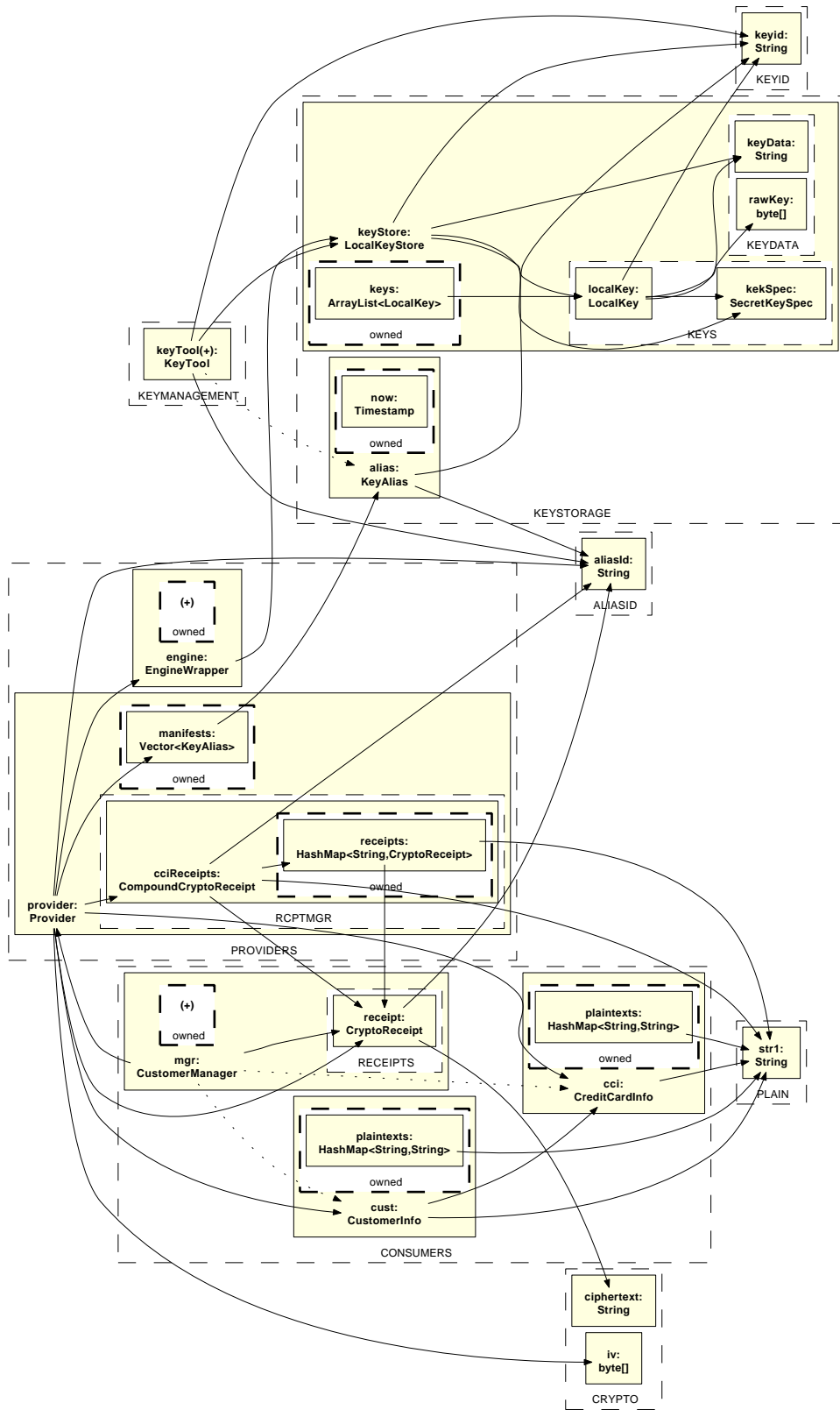Figure 21: CryptoDB OOG, binding top-level domains for String to shared.

Figure 22: CryptoDB OOG with Strings.

References

[1] P. Torr, "Demystifying the threat-modeling process," *IEEE Secur. & Priv.*, vol. 3, no. 5, 2005.

[2] M. Abi-Antoun, D. Wang, and P. Torr, "Checking threat modeling data flow diagrams for implementation conformance and security," in *ASE*, 2007.

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

[4] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE TSE*, vol. 32, no. 7, 2006.

[5] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE TSE*, vol. 21, no. 9, 1995.

[6] J. Aldrich, "Using types to enforce architectural structure," in *WICSA*, 2008.

[7] M. Abi-Antoun and J. Aldrich, "Static conformance checking of runtime architectural structure," Carnegie Mellon Univ., Tech. Rep. CMU-ISRI-08-132, 2008.

[8] P. Tonella and A. Potrich, *Reverse Engineering of Object Oriented Code.* Springer, 2004.

[9] M. Abi-Antoun and J. M. Barnes. (2009) Addendum. [Online]. Available: http://www.cs.cmu.edu/~mabianto/cryptodb/

[10] M. Abi-Antoun and W. Coelho, "A case study in incremental architecture-based re-engineering of a legacy application," in *WICSA*, 2005.

[11] M. Abi-Antoun and J. Aldrich, "Ownership domains in the real world," in *IWACO*, 2007.

[12] J. Aldrich and C. Chambers, "Ownership domains: Separating aliasing policy from mechanism," in *ECOOP*, 2004.

[13] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE TSE*, vol. 27, no. 4, 2001.

[14] D. Kirk, M. Roper, and M. Wood, "Identifying and addressing problems in object-oriented framework reuse," *Empir. Softw. Eng.*, vol. 12, no. 3, 2006.

[15] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems.* Cambridge Univ. Press, 2000.

[16] R. Monroe, "Capturing software architecture design expertise with Armani," Carnegie Mellon Univ., Tech. Rep., 2001.

[17] K. Kenan, *Cryptography in the Database.* Addison-Wesley, 2006, accompanying code at http://kevinkenan.blogs.com/downloads/cryptodb_code.zip.

[18] M. Abi-Antoun and J. Aldrich, "Static extraction of sound hierarchical runtime object graphs," in *TLDI*, 2009.

[19] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan, "Differencing and merging of architectural views," *Autom. Softw. Eng.*, vol. 15, no. 1, 2008.

[20] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong, "Secure software architectures," in *IEEE Secur. & Priv.*, 1997.

[21] Y. Deng, J. Wang, J. J. P. Tsai, and K. Beznosov, "An approach for modeling and analysis of security system architectures," *IEEE T. Knowl. & Data En.*, vol. 15, no. 5, 2003.

[22] J. Jürjens, *Secure Systems Development with UML.* Springer, 2004.

[23] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *WICSA*, 2007.

[24] M. Abi-Antoun, J. Aldrich, and W. Coelho, "A case study in re-engineering to enforce architectural control flow and data sharing," *J. Syst. & Softw.*, vol. 80, no. 2, 2007.

[25] T. Lodderstedt, D. A. Basin, and J. Doser, "SecureUML: A UML-based modeling language for model-driven security," in *UML*, 2002.

[26] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," in *ICSE*, 2006.

[27] M. Sefika, A. Sane, and R. H. Campbell, "Monitoring compliance of a software system with its high-level design models," in *ICSE*, 1996.

[28] H. J. Hoover and D. Hou, "Using SCL to specify and check design intent in source code," *IEEE TSE*, vol. 32, no. 6, 2006.

[29] Omondo, "EclipseUML," http://www.omondo.com/, 2009.

[30] A. Spiegel, "Automatic distribution of object-oriented programs," Ph.D. dissertation, FU Berlin, 2002.

[31] D. Jackson and A. Waingold, "Lightweight extraction of object models from bytecode," *IEEE TSE*, vol. 27, no. 2, 2001.