Analyzing Security Architectures

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

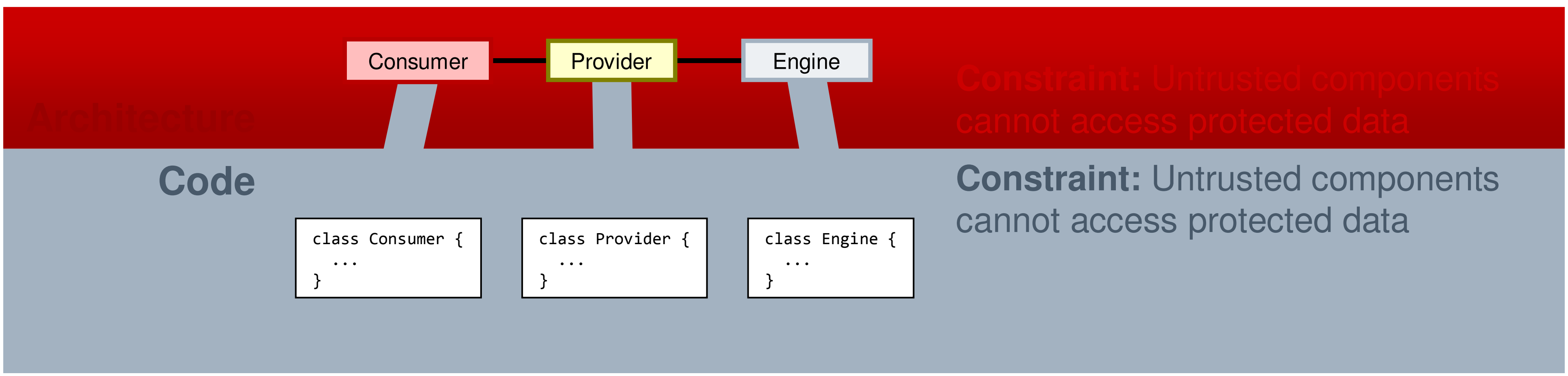
Jeffrey M. Barnes
Inst. for Software Research
Carnegie Mellon University
jmbarnes@cs.cmu.edu

Problem background

- Engineers use tools like **data flow diagrams** (DFDs) to analyze security properties of software systems
- Often these are constructed from developers' recollection of how a system works, with little automated support
- This architectural representation may fail to capture all communication present in the system

Architecture conformance

- In essence, this is a problem of **architecture conformance**
- Want to reason at an architectural level but relate it to code at the same time



Security architectures as runtime architectures

- A security architecture is an example of a **runtime architecture**
  - Shows runtime components such as objects and data stores
  - Shows runtime connectors such as communication links and points-to relations
  - May have many instances of a single component type
- Contrast with static code views such as class diagrams

The challenge of analyzing security architectures

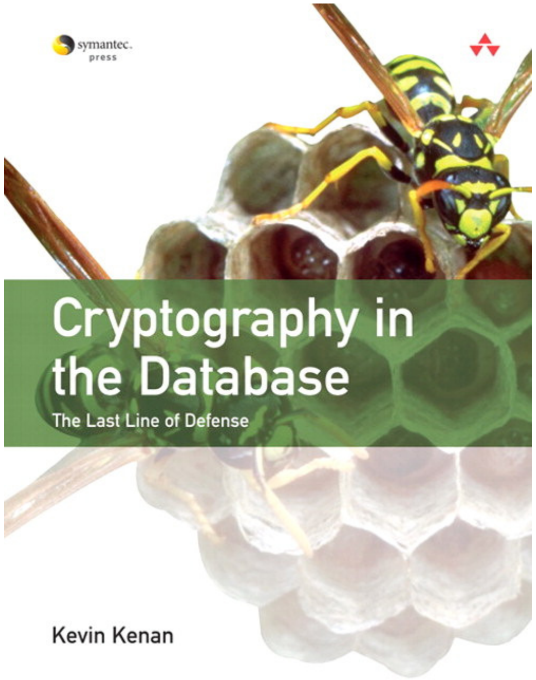
- Tools for analyzing **conformance of runtime architectures** are immature compared to those for code architectures
- A security analysis must consider the **worst case**, not the typical case, of possible component communication
  - Demands **static analysis**
  - Dynamic analysis can tell us about only a limited number of runs

Our contribution

- An architecture-centric approach, **SECORIA**, that enables reasoning at the level of a **security runtime architecture**, and relating it to code at the same time
- Can enforce both code-level and global architectural constraints

Evaluation

- Validated SECORIA on **CryptoDB**, a secure database system designed by a security expert
- Database architecture that provides cryptographic protections against unauthorized access
- Includes 3,000-line sample implementation in Java

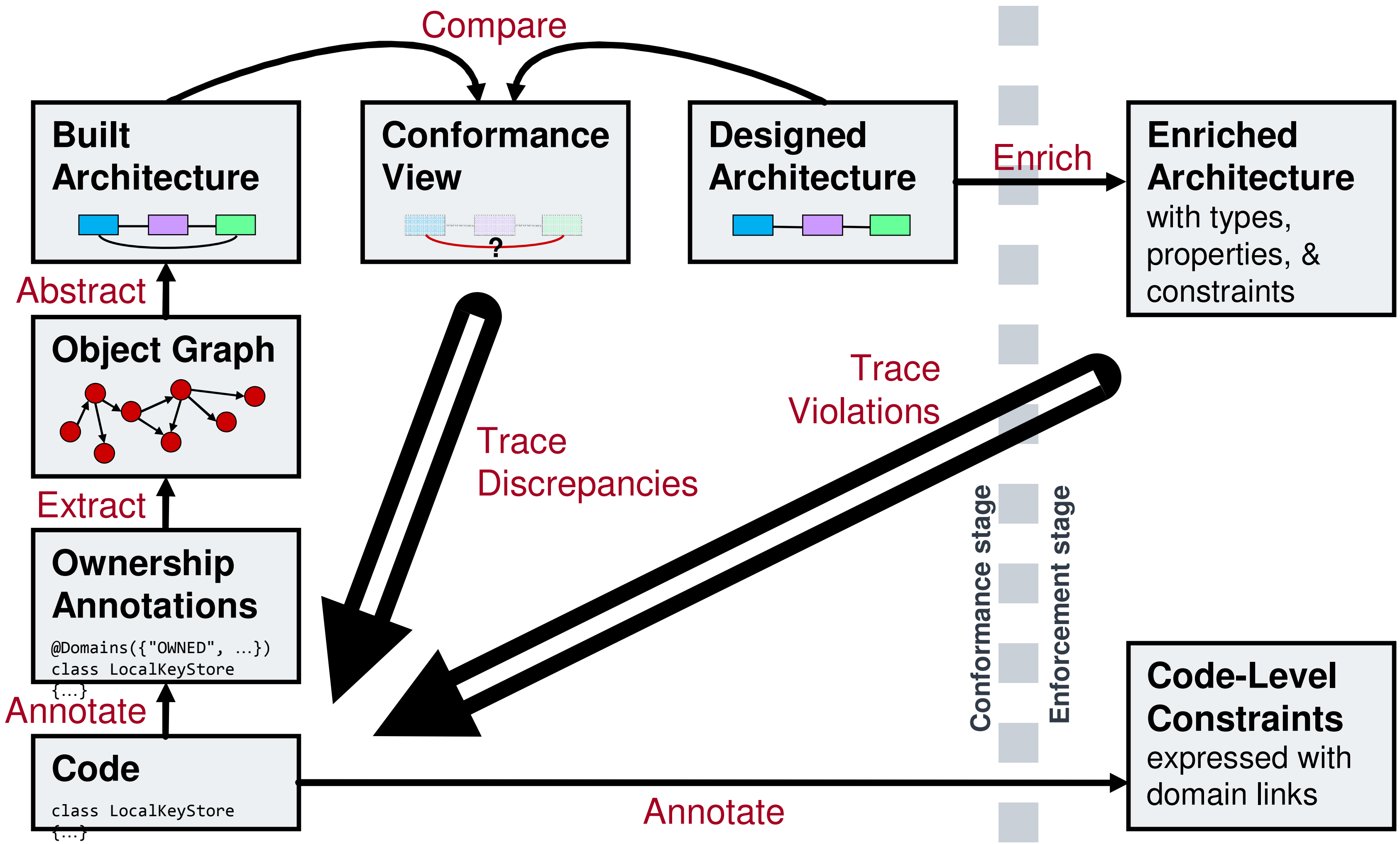


Approach

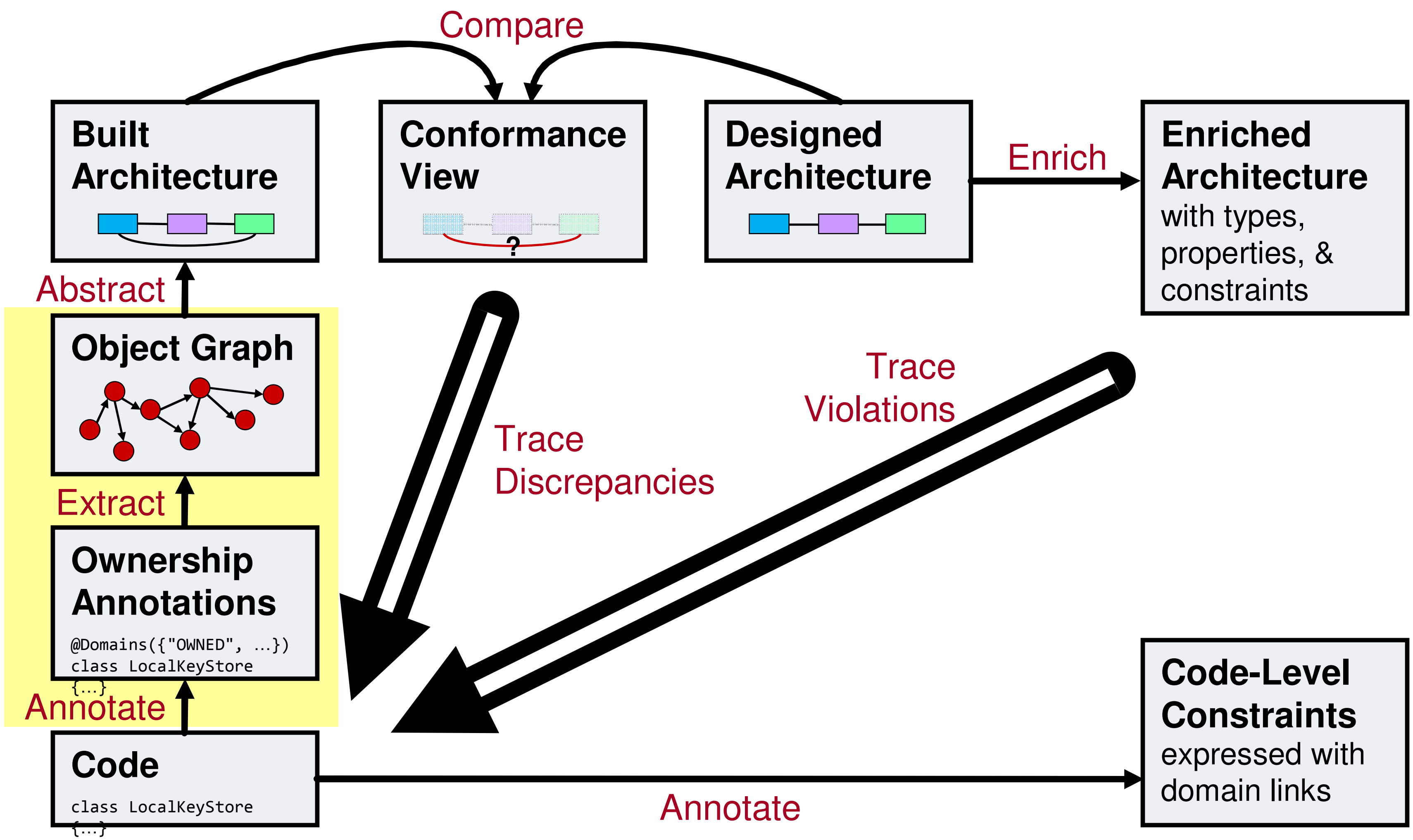
Overview of SECORIA

- Specialization of **SCHOLIA** [Abi-Antoun & Aldrich, OOPSLA'09], which analyzes conformance between object-oriented code and a hierarchical, target runtime architecture
- SECORIA is an iterative process with two main stages: **conformance** and **enforcement**

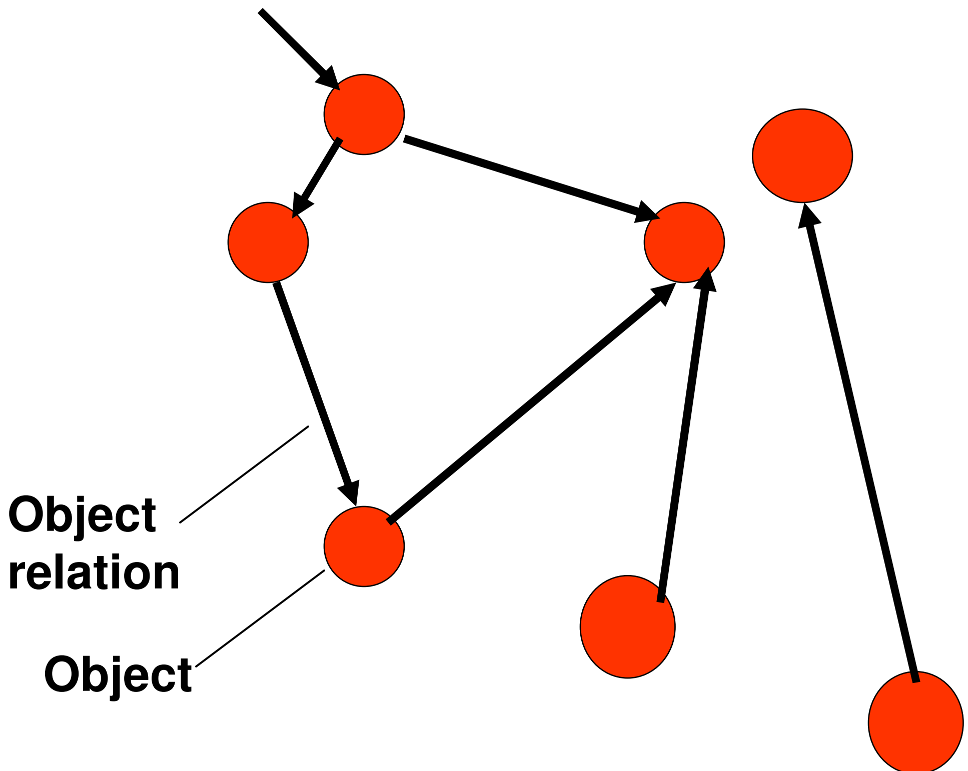
Overview of SECORIA



Conformance stage: annotate; extract object graph

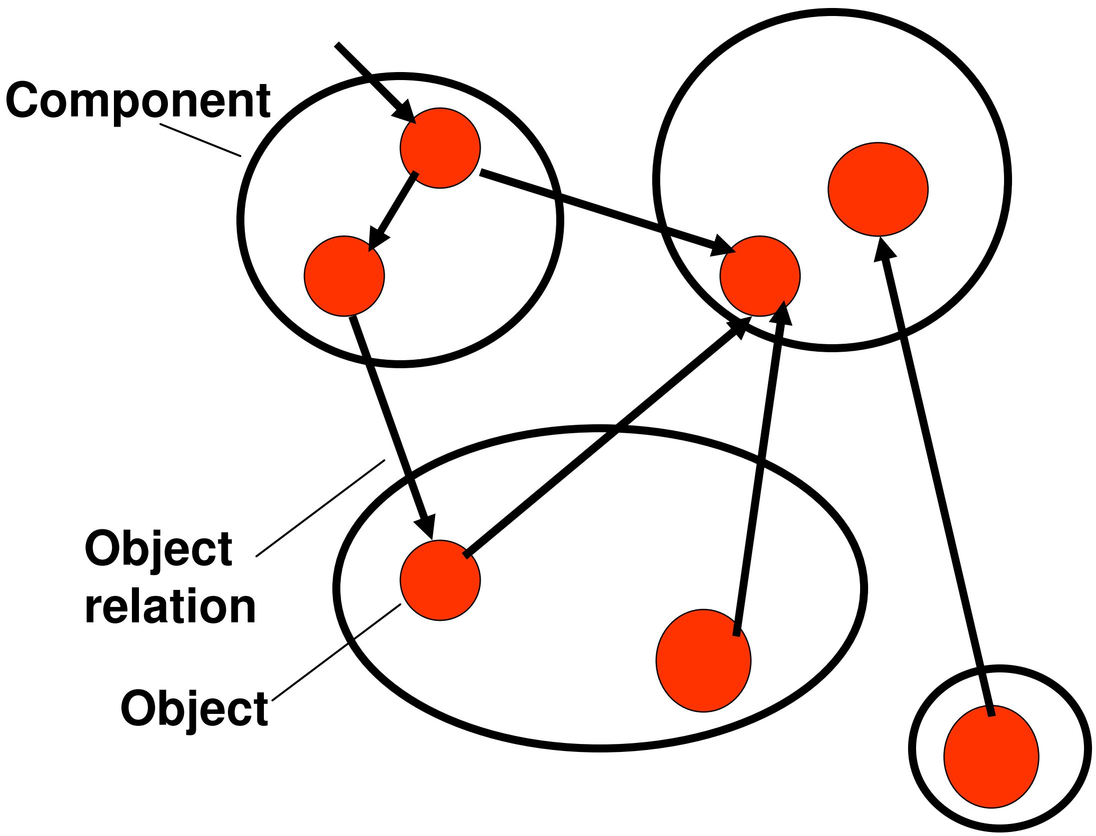


At runtime, an object-oriented system appears as a Runtime Object Graph (ROG)

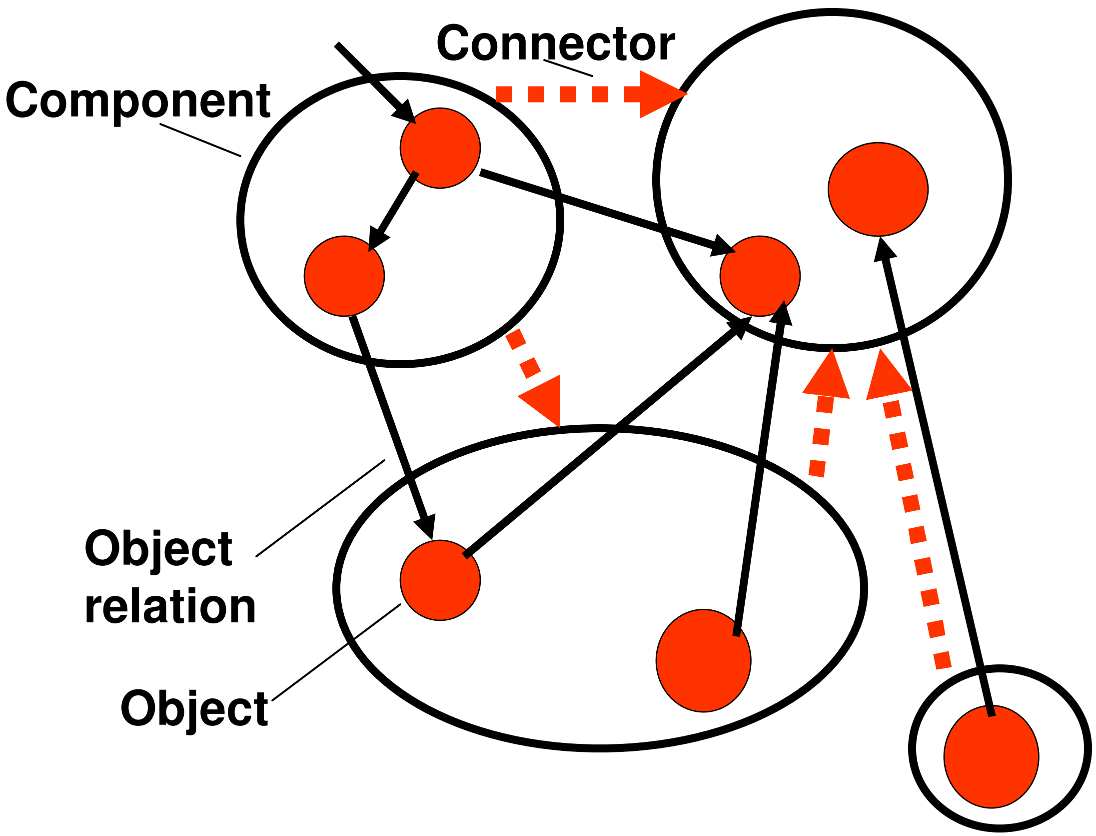


- A node represents a runtime object
- An edge represents a points-to relation

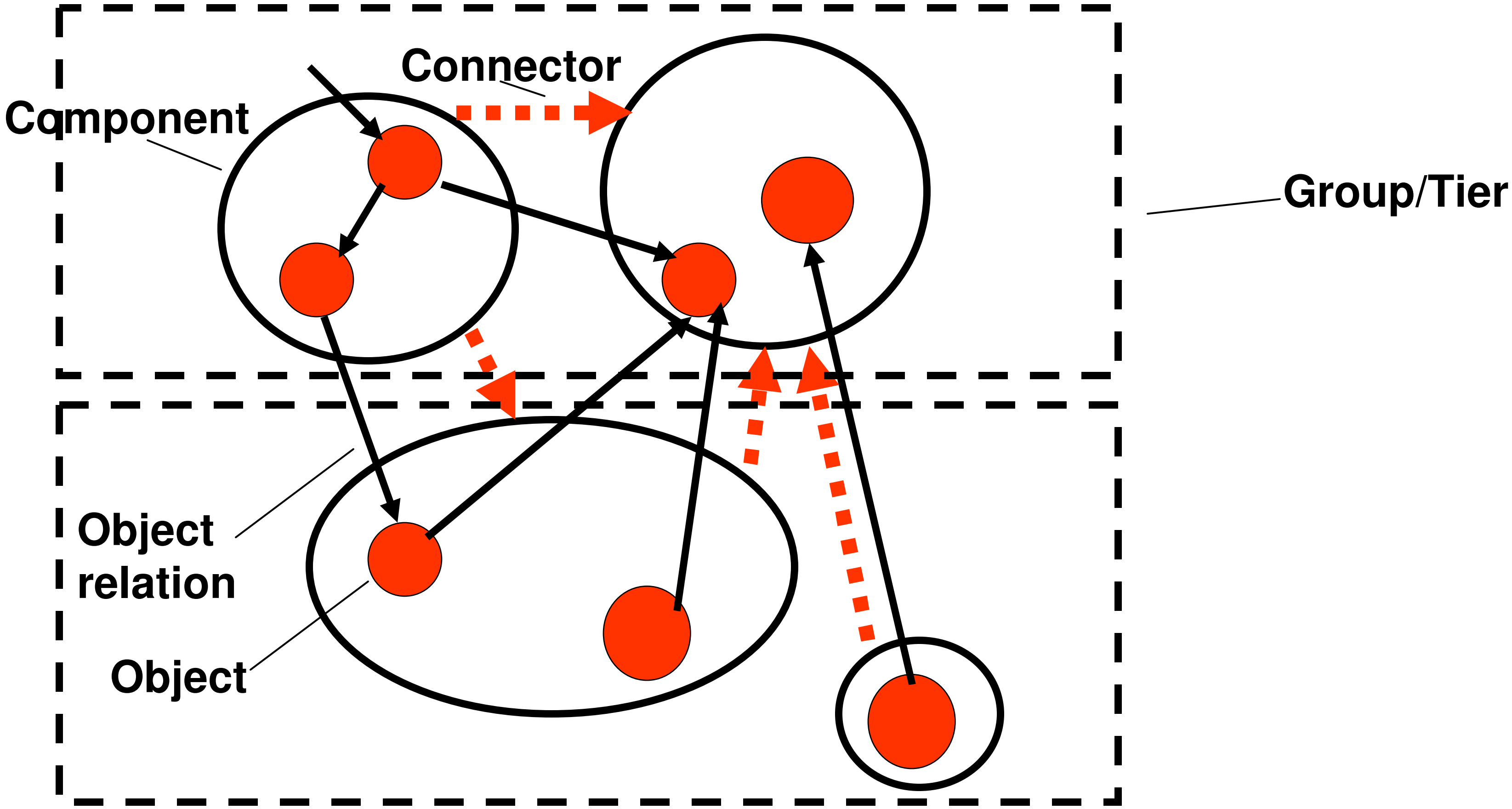
Abstract objects into “components”



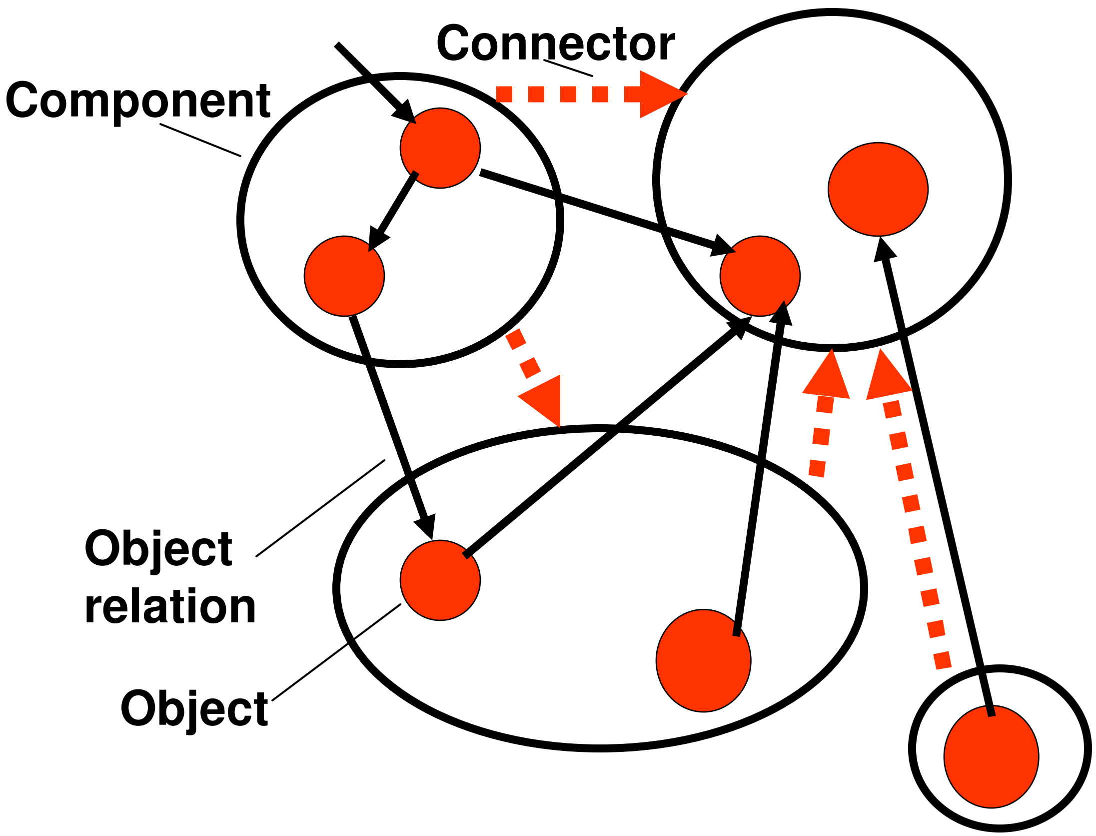
Abstract relations between components



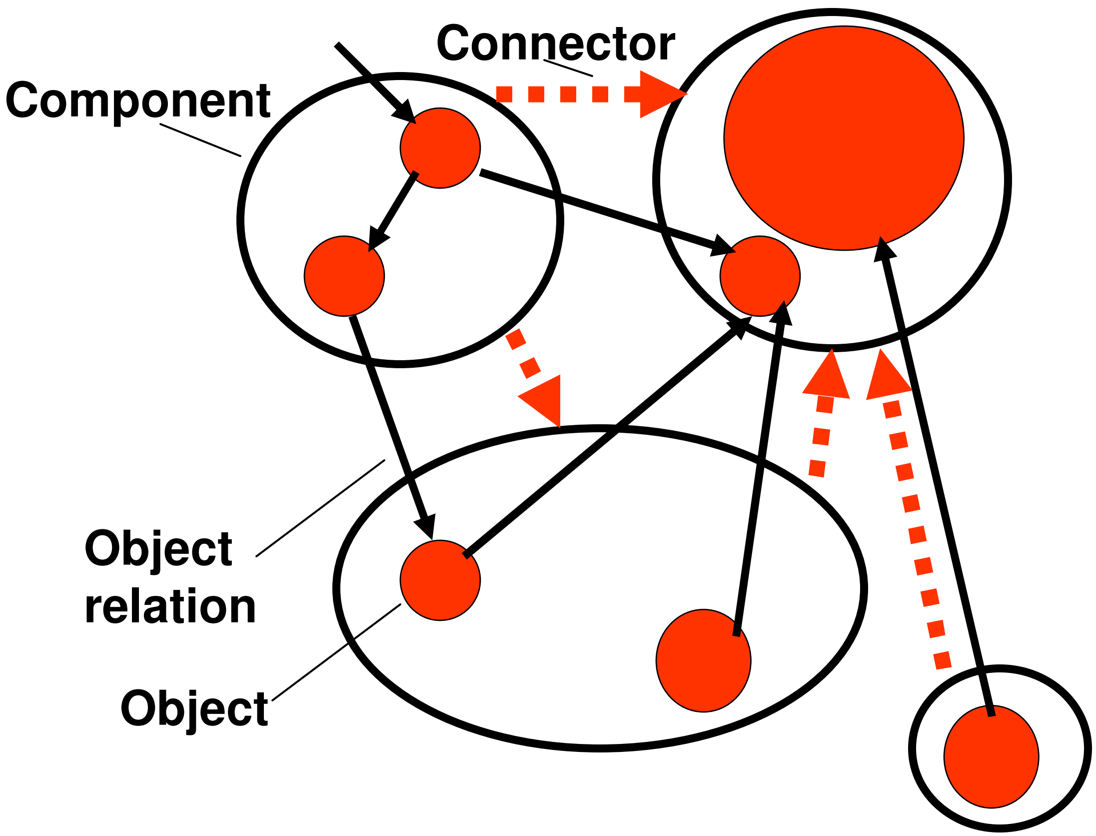
Organize components into groups/tiers



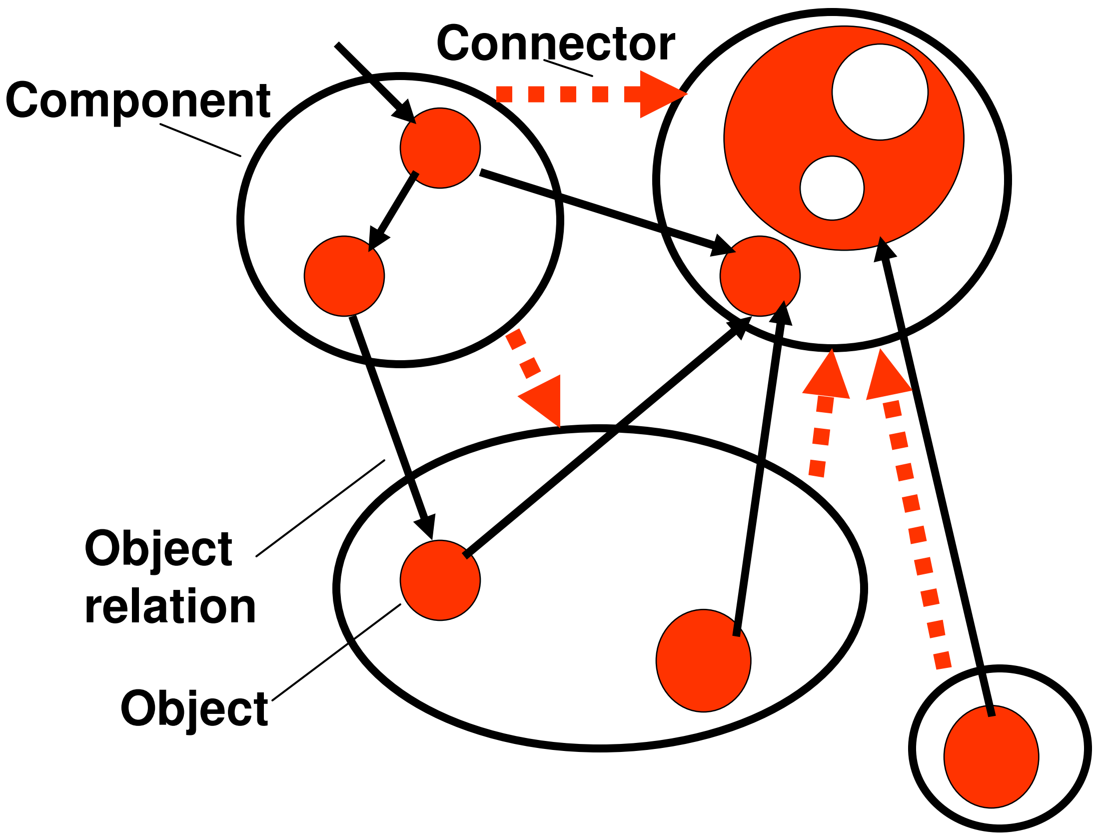
Make some components part of others



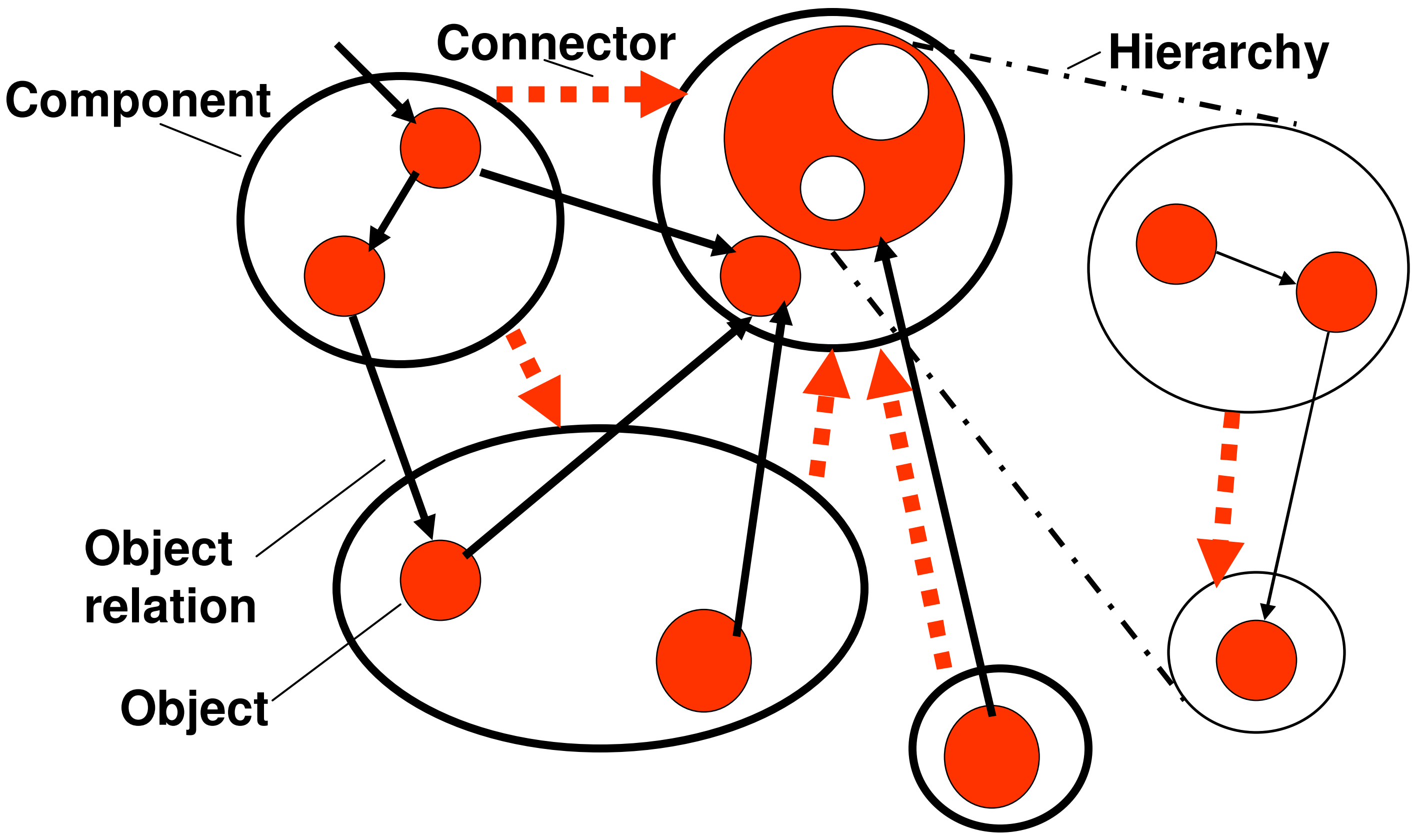
Make some components part of others



Make some components part of others



Make some components part of others

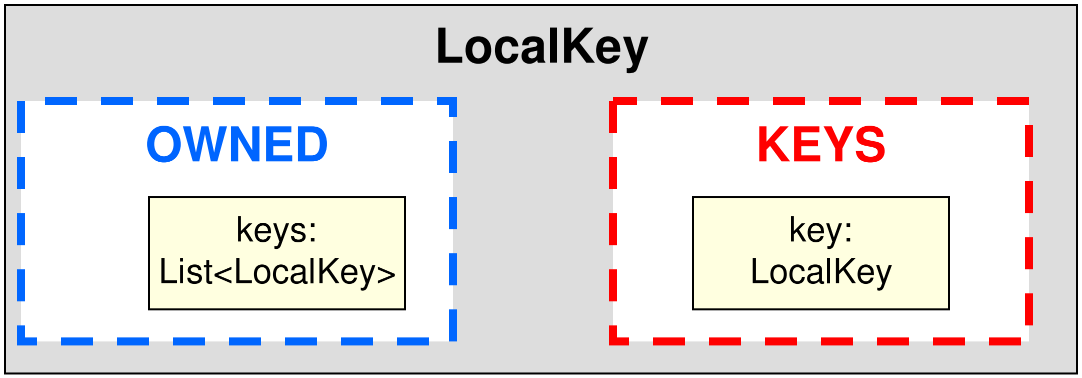


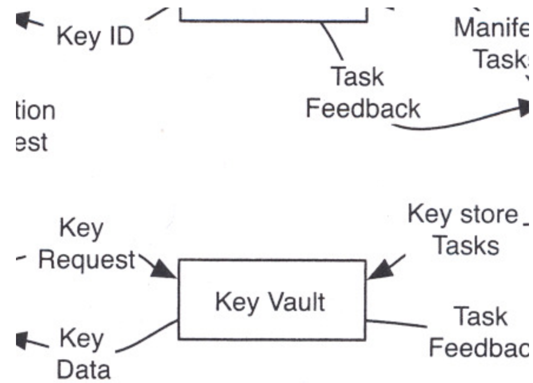
Annotate **code and extract** **object graph**

- **Problem:** Architectural **hierarchy** not readily observable in arbitrary code
- To solve this, we use **annotations** to capture architectural intent
- Developer picks top-level entry point
- Use annotations to impose an **ownership hierarchy** on objects
- Annotations are minimally invasive, modular, and statically typecheckable

Ownership domains are groups of objects

[Aldrich and Chambers, ECOOP'04] [Krishnaswami and Aldrich, PLDI'05]





object: Type Object

Type Type

```
@Domains({"OWNED", "KEYS"})
class LocalKeyStore {
  @Domain("OWNED") List<LocalKey> keys;
  @Domain("KEYS") LocalKey key;
  ...
}
```

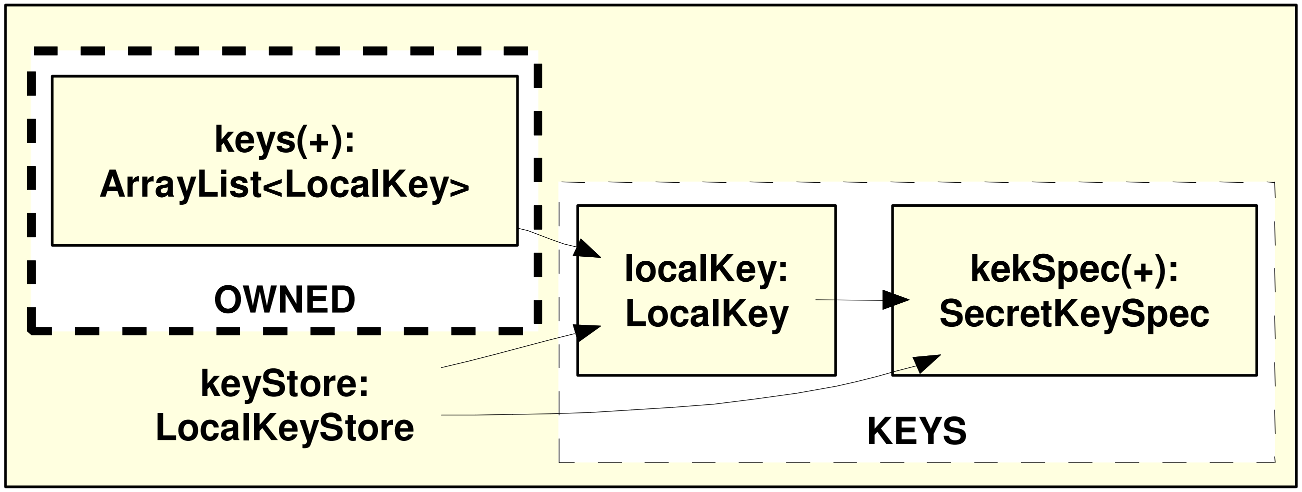
Declarations are simplified

- Ownership domain = conceptual group of objects
- Each object **in exactly one domain**

Annotations define two kinds of **object hierarchy**

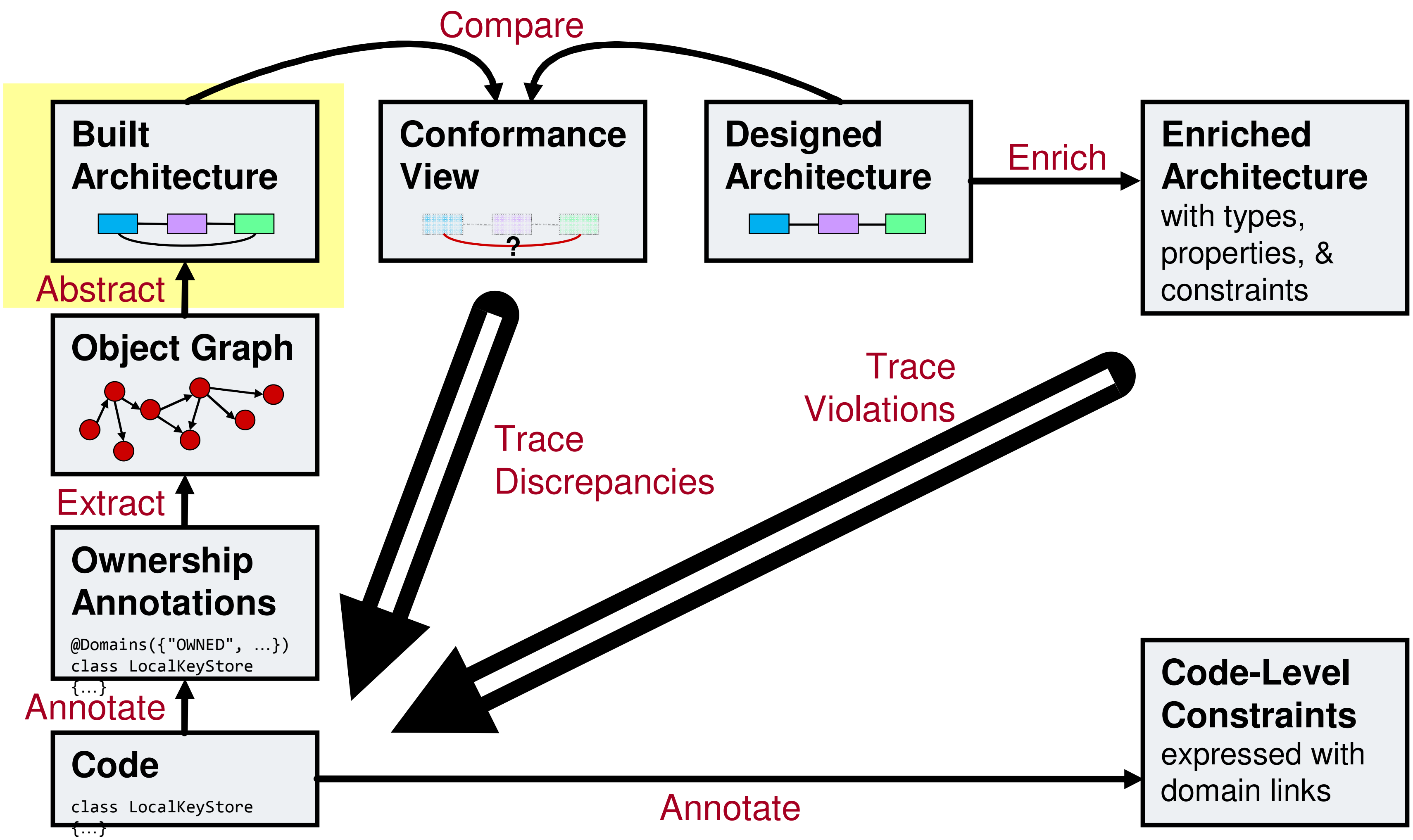
- A public domain provides **logical containment**: an object is conceptually "part of" another
  - Having access to an object also gives access to objects inside its public domains
- A private domain provides **strict encapsulation**
  - E.g., a `public` method cannot return an alias to an object in a private domain, even though Java allows returning an alias to a `private` field

Examples of object hierarchy



- `LocalKeyStore` has a **public domain** to hold `LocalKey` objects
- `LocalKeyStore` stores the `ArrayList` of `LocalKey` objects in a **private domain**

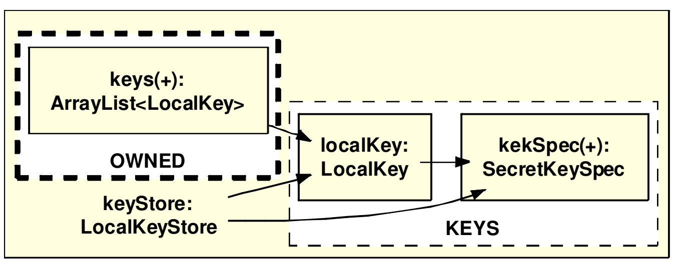
Conformance stage: Abstract object graph



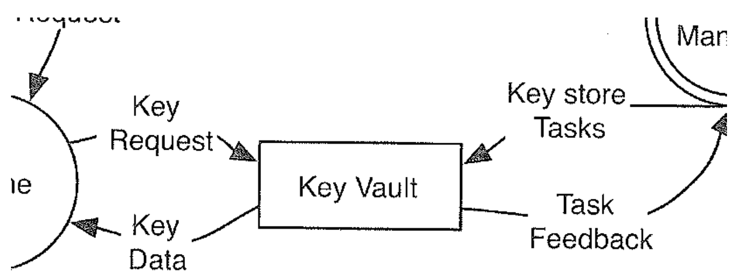
Object graph vs. target architecture

- Often, object graph **not isomorphic** to architect's intended architecture
- So **abstract** and represent in **component-and-connector** view

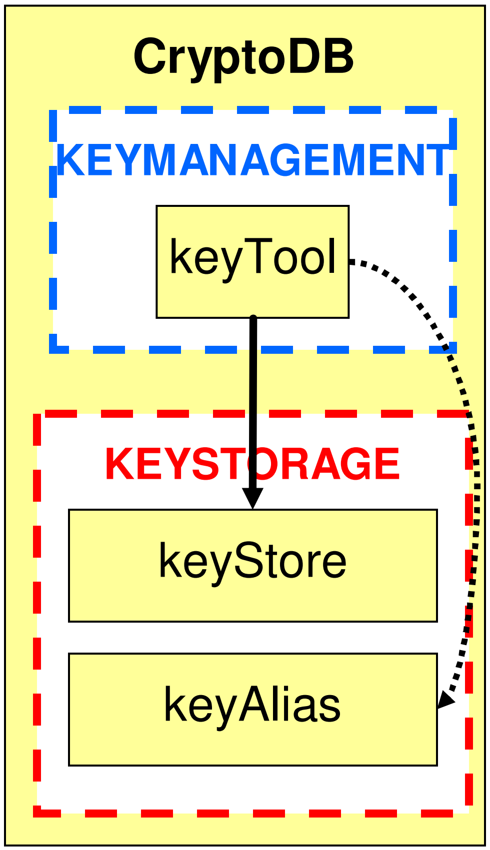
CryptoDB object graph



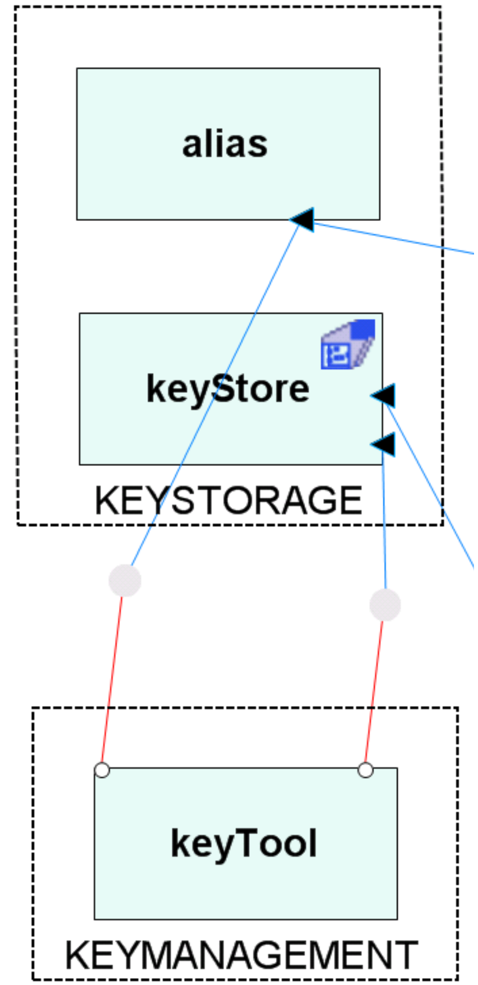
CryptoDB target architecture



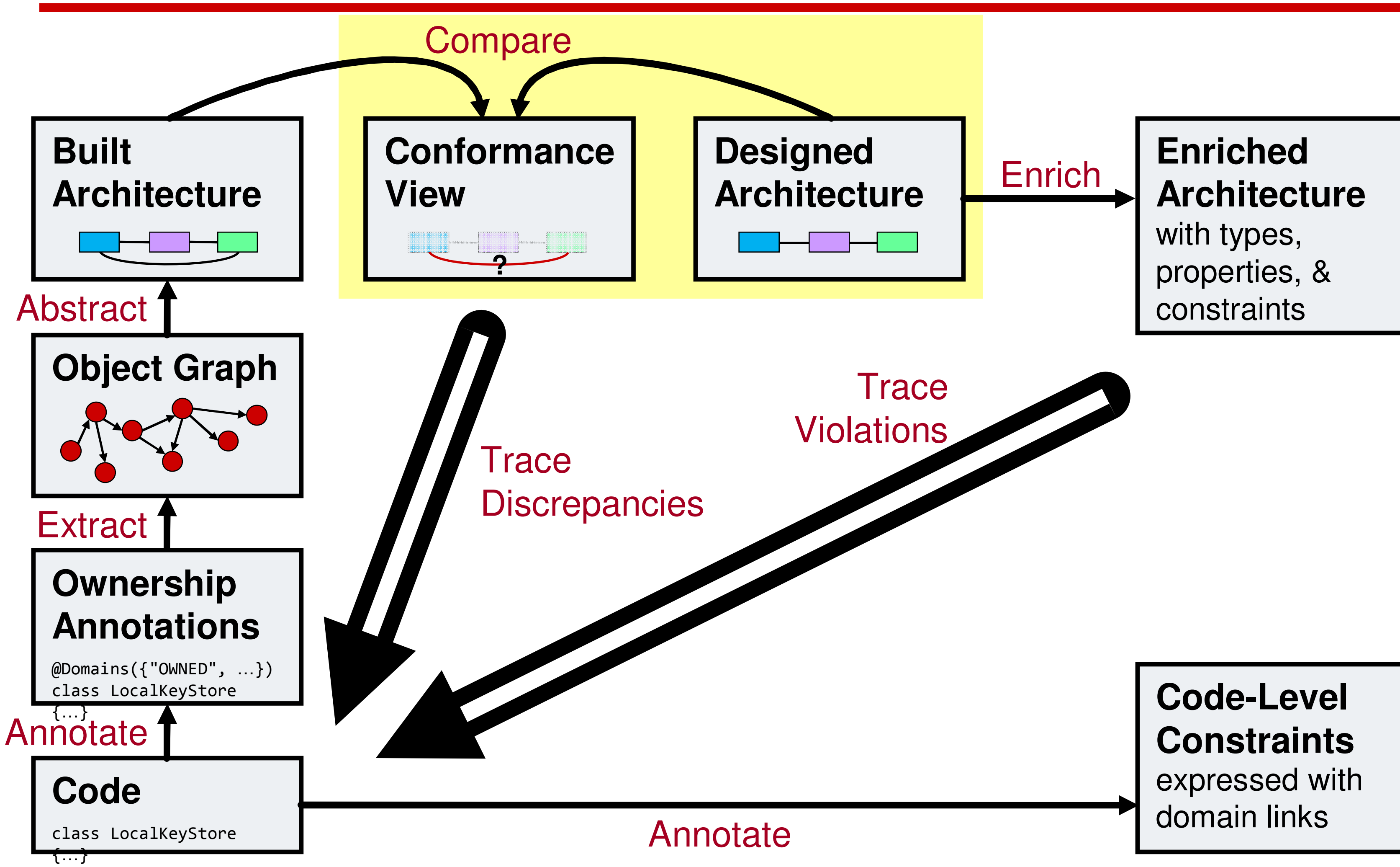
Represent abstracted object graph as component-and-connector (C&C) view



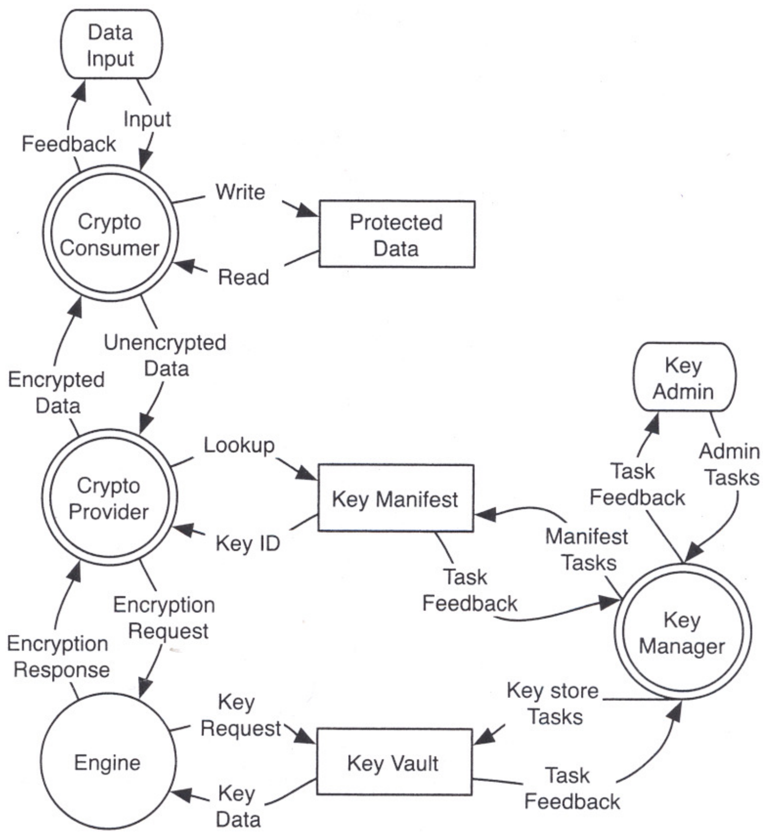
object graph	↔	**C&C view**
top-level object	↔	system
object	↔	component
domain	↔	group
interface	↔	provide port
field reference	↔	use port
object relation	↔	connector
substructure	↔	representation



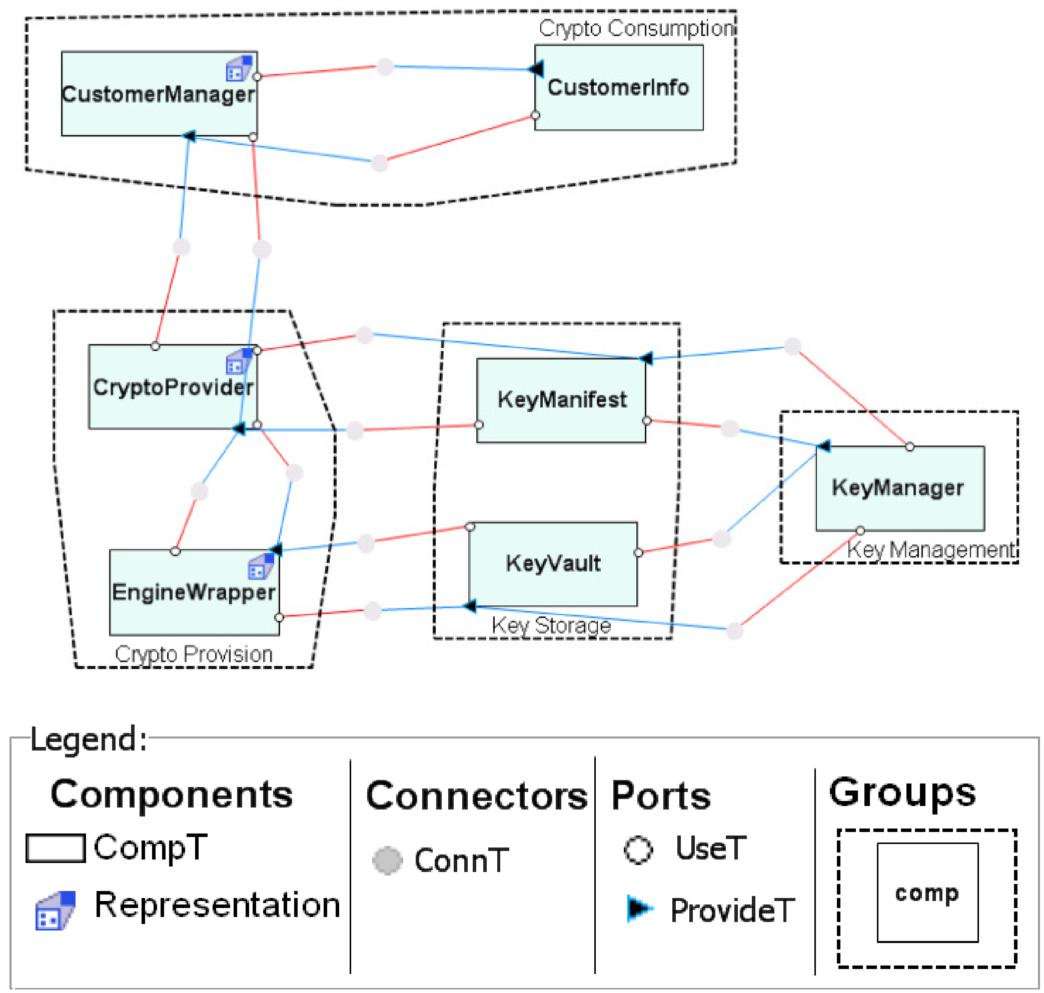
Conformance stage: Document target architecture; check conformance



CryptoDB: Document designed architecture



CryptoDB: Level-1 DFD
[Kenan, Fig. 3.2]

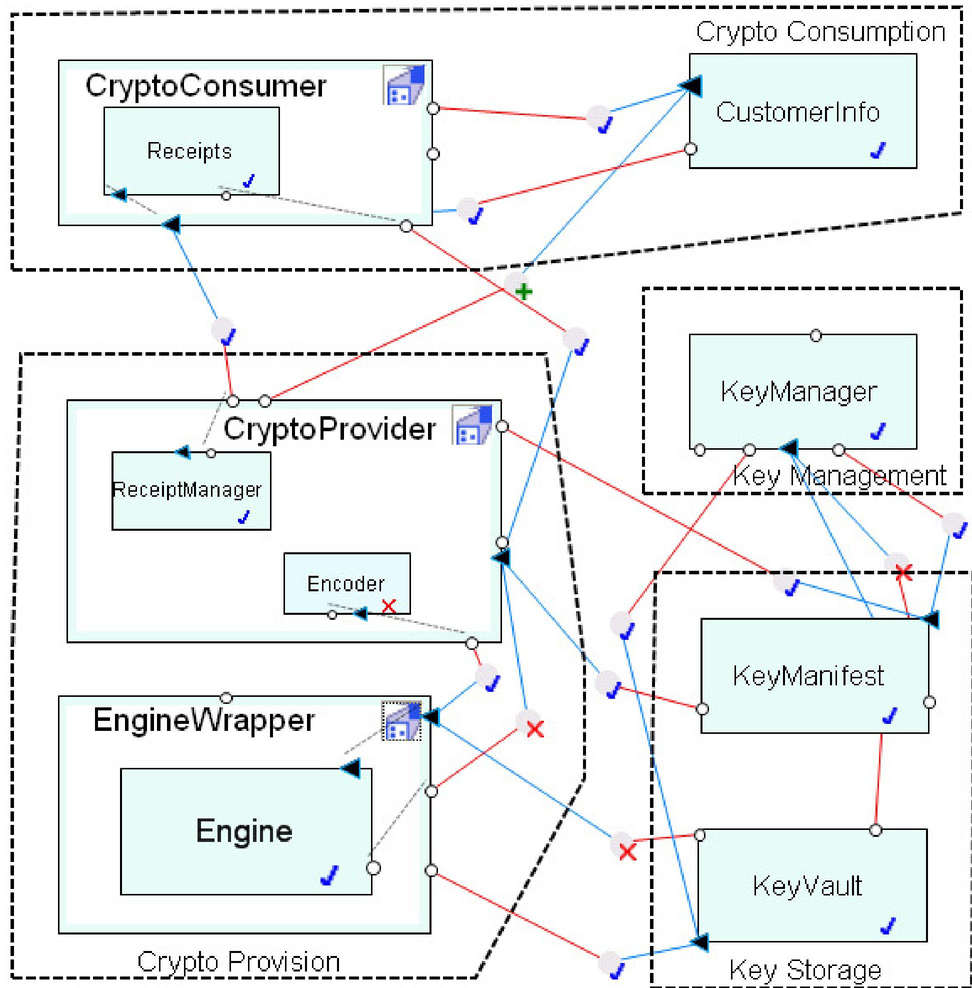


Analyzing conformance of system to target architecture

- Conformance analysis based on **communication integrity**
  [Luckham and Vera, TSE'95]
- Identifies following differences:
  - **Convergence**: node or edge **in both** built and in designed view ✓
  - **Divergence**: node or edge in built view, but **not in designed** view ✚
  - **Absence**: node or edge in designed view, but **not in built** view ✗

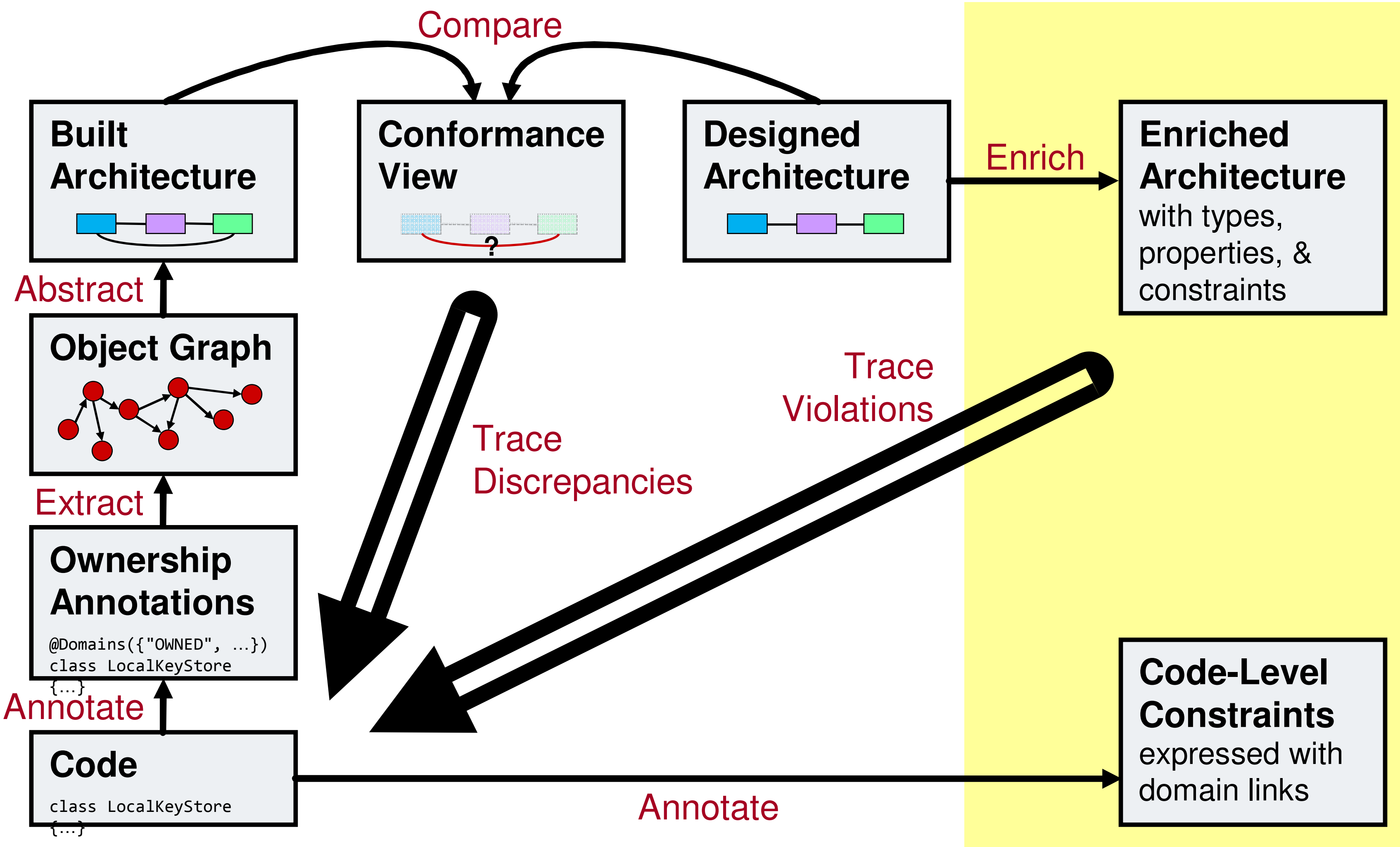
Developer investigates reported differences

- Study findings
- Trace to code



Convergence
Divergence
Absence

Enforcement stage



Architectural types

- Target architecture described in an **architecture description language** such as Acme
- Architectures described using **components**, **connectors**, and other elements
- These elements participate in a **type system**
  - E.g., many component instances may belong to a single component type

Architectural families

- Element types are used to build up **families**
  - Encapsulate types applicable to a broad class of software architectures
- We have a reusable **DFD family**
  - Component types like Process, DataStore, etc.
- A family can also define **architectural properties**
  - trustLevel, howFound, etc.

Architectural constraints

- **Security constraints**
  - Automatically applied when the security family is imported and types and properties are set
- **Application-specific constraints**
  - Can be introduced as constraints in the target architecture
  - Based on the specific security requirements of the system under study

Security constraints

- Common constraints defined by the DFD family
- **Well-formedness** constraints
  - E.g., two DataStores cannot be connected directly
- **Information flow** constraints
  - Based on **STRIDE** principles [Howard & LeBlanc, *Writing Secure Code*]
    - Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege
  - E.g., **information disclosure**: The trustLevel of a DataFlow's source should not be higher than its destination

Application-specific constraints

- Documentation of the target architecture:
  - "Access to the key vault […] should be granted to only security officers and the cryptographic engine" [Kenan, p. 71]
- Our interpretation:
  - Only `KeyManager` and `EngineWrapper` should have access to `KeyVault`
- Our formalization:
  - ```
    forall c : SyncCompT in self.COMPONENTS |
        pointsTo(c, KeyVault)
            -> c.label = "KeyManager"
            or c.label = "EngineWrapper"
    ```
    ```

Validation results; related work

CryptoDB: Summary of findings

- We successfully related the security architecture and implementation
- **Renames**: The structural comparison allowed us to analyze conformance despite naming discrepancies (e.g., `KeyManager` versus `KeyTool`)
- **Conformance findings**: Top-level components in the target architecture and implementation were mostly consistent

Defect prevention

- Manually injected manufactured architecture violation into code
  - Coupled `Provider` and `LocalKeyStore`
- Conformance view showed new divergence between `provider` and `keyVault`
- Predicate raised warning about violation

Related work

- **Architecture extraction** & **conformance**
  - Most work focuses on static extraction of a **code architecture** [Murphy et al., TSE'01]
- Approaches based on **dynamic analysis** or **testing**
  - Cannot check all runs
- **Threat modeling tools**
  [Swiderski & Snyder, *Threat Modeling*]
  - Provide architectural analysis of security, but do not relate architecture to code

Summary

- First approach, **SECORIA**, to analyze, entirely statically, a **security runtime architecture** for some **information flow vulnerabilities** and for **conformance** to an object-oriented implementation
- Evaluation shows we can detect code changes that introduce architectural violations
- **Architecture-based** analysis matches the way experts reason about security during threat modeling

Supplementary material

Download Acme specifications, our DFD security family, and other related material at:

http://www.cs.wayne.edu/~mabianto/cryptodb/