Ownership Object Graphs with Dataflow Edges

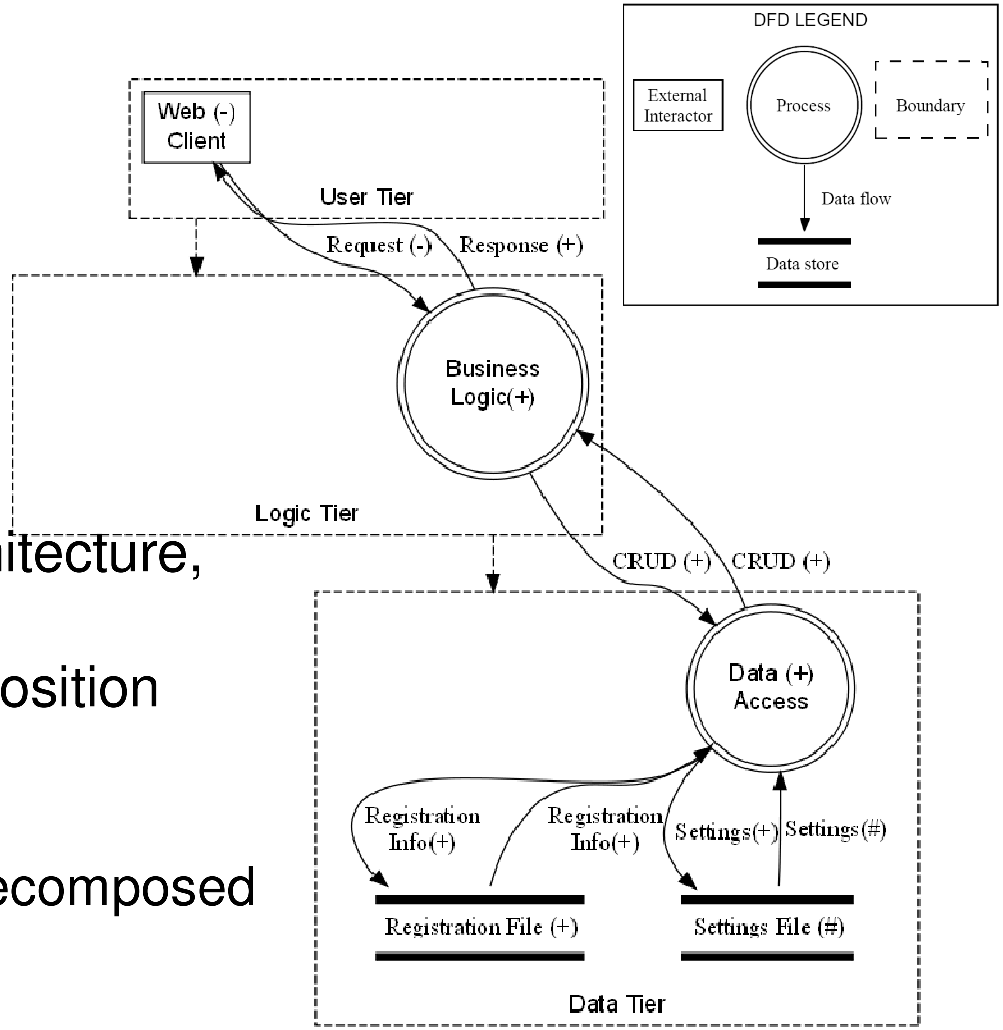
By Radu Vanciu and Marwan Abi-Antoun
Department of Computer Science
Wayne State University
Detroit, Michigan, USA

Architectural Risk Analysis helps build secure systems [Howard and Lipner, Microsoft Press'06]

- 50% of security flaws are architectural [McGraw, Addison-Wesley'05]
  - Example: illogical access over tiers
  - Other 50% are coding bugs (buffer overflow)
- Human experts use architectural diagrams:
  - Forest-level view of system (not reading code)
  - Diagram must represent runtime – not code – structure
- Limitations with today's approach:
  - Diagram may be missing
  - Diagram may not match the code
- Solution: extract architectural diagram from code

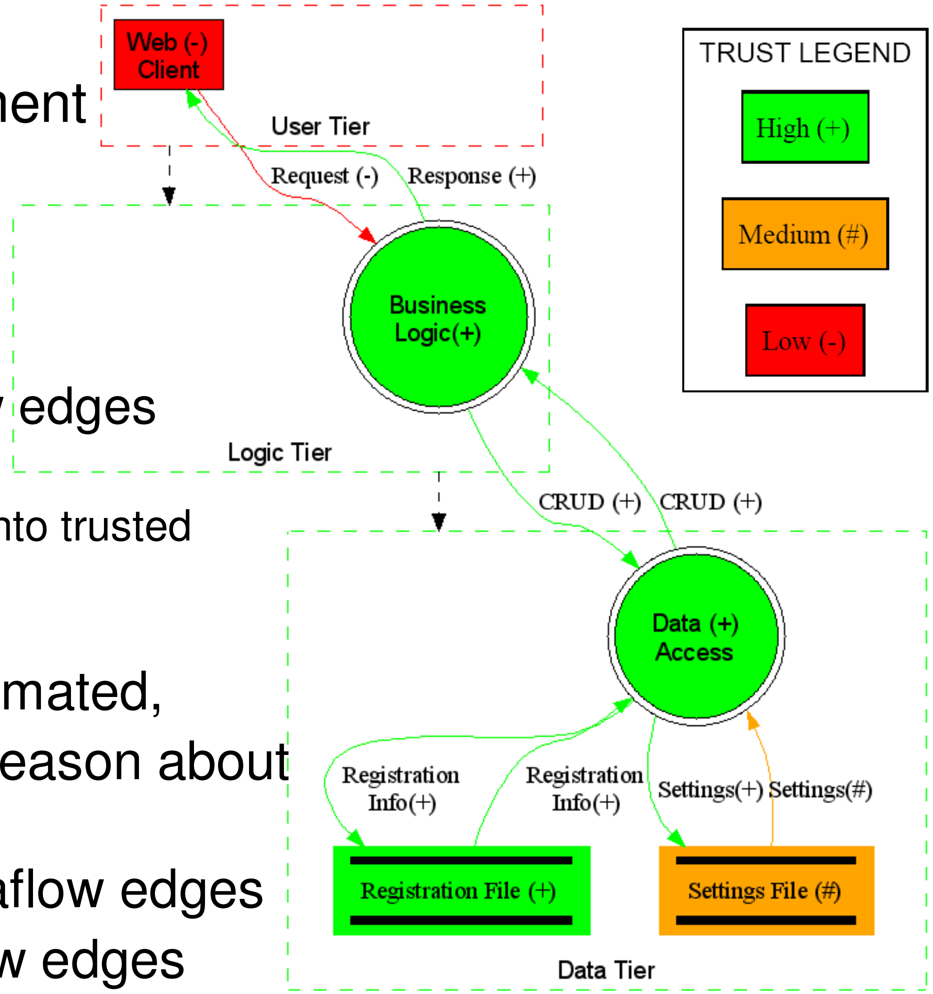
Architectural Risk Analysis uses Data-Flow Diagram (DFD)

- What is a DFD?
  - components
  - subcomponents
  - interactors
  - dataflow edges
  - tiers
- Show dataflow edges on architecture, and type of data
- **Hierarchical** system decomposition
  - Promotes both **high-level understanding** and **detail**
  - Example: Business Logic decomposed into separate DFD



Architectural Risk Analysis using DFD

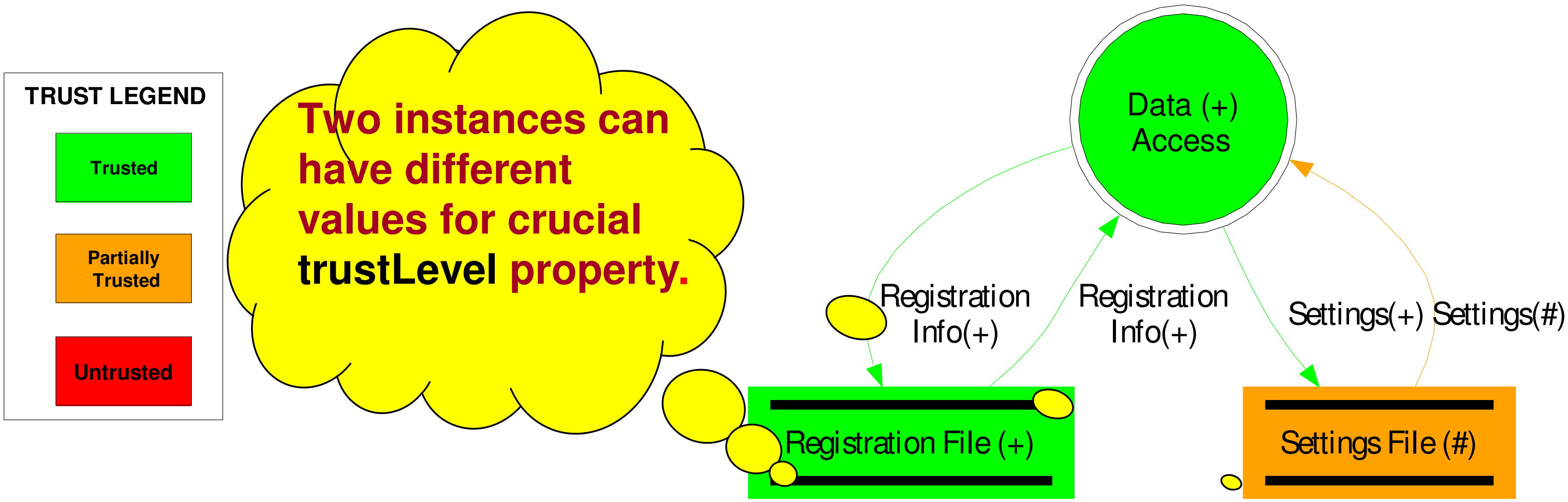
[Abi-Antoun, Wang and Torr, ASE'07] [Abi-Antoun and Barnes, ASE'10]



- Set of **properties** for each element
  - Example: **trustLevel**
    - **High(+)**
    - **Medium(#)**
    - **Low(-)**
  - Estimate risk based on dataflow edges
    - **Tampering**: **low(-) → high(+)**
      dataflow that crosses boundary into trusted component
    - Solution: ensure data is validated
- Some simple checks can be automated,
- Ultimately, human experts must reason about DFD
- Previous work approximated dataflow edges
- Our goal is to extract real dataflow edges

DFD is a diagram of runtime structure

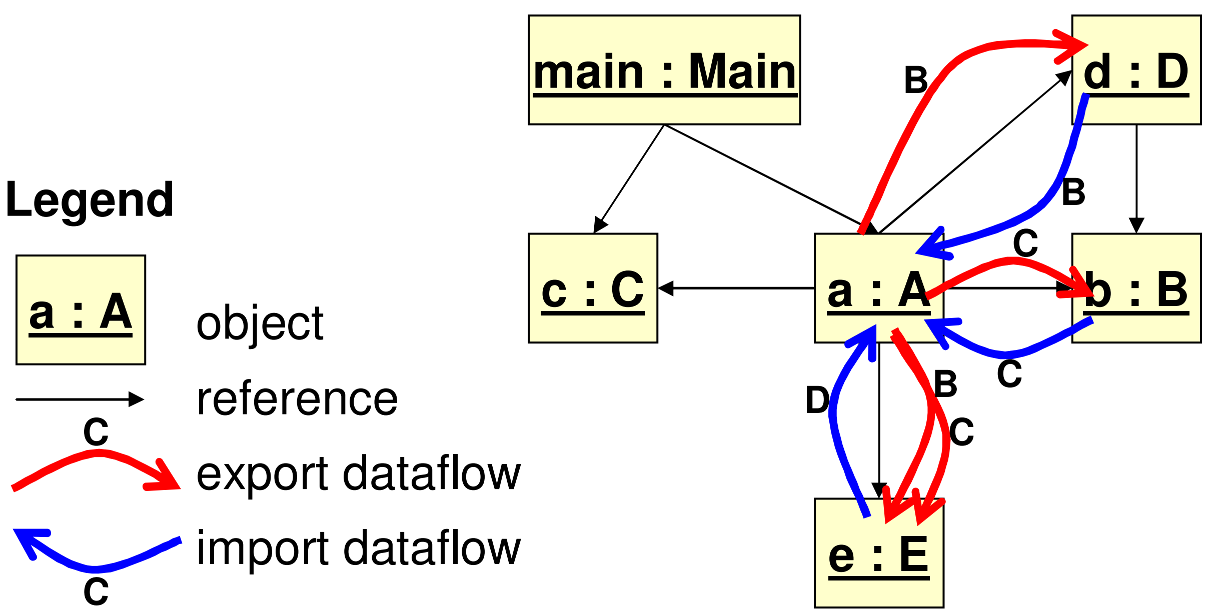
- A diagram of runtime structure distinguishes between different instances of the same class
- **Different instances** usually have different architectural **properties**
  - Here, **trustLevel** = **Full** vs. **Partial**
  - Usually, **one** java.io.File class in class diagram (code structure)



DFD is similar to object graphs

- Nodes represent instances of classes
- Edges represent dataflow:
  [Spiegel, Ph.D. Thesis'02] [Lienhard et al., COMLAN'09]
  - method invocation
  - field read
  - field write
- Label edges with type of flow

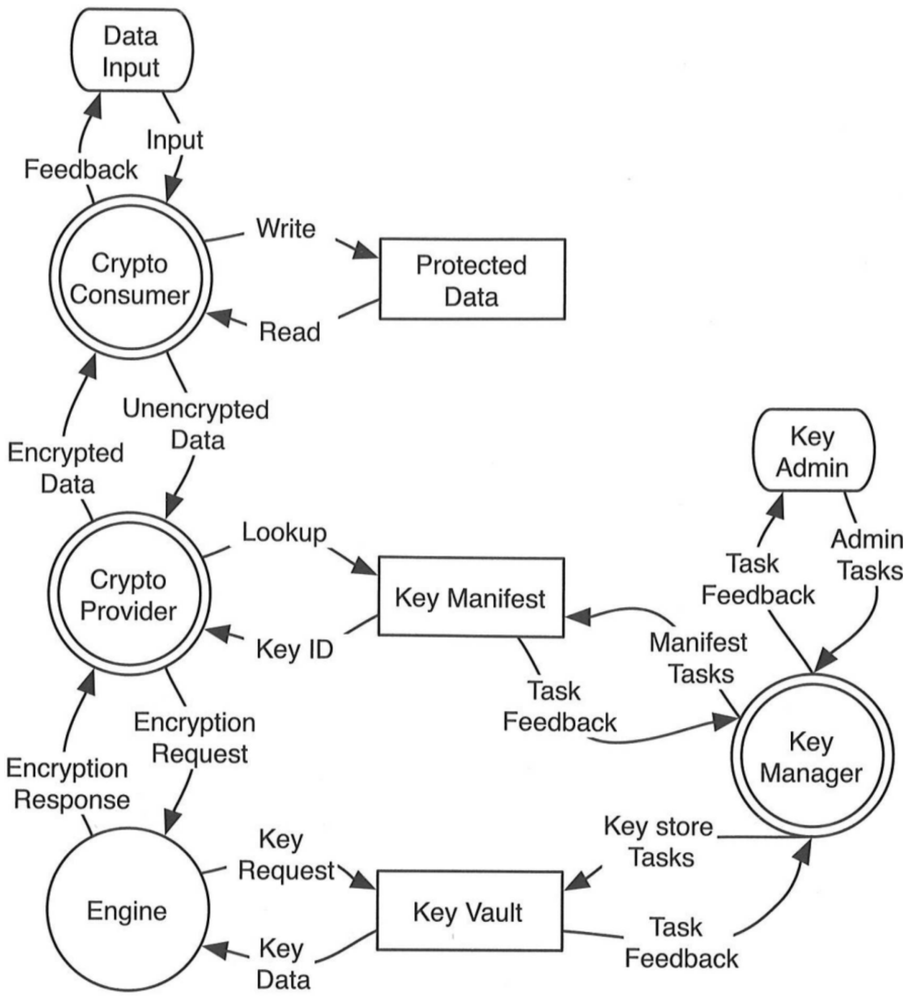
```
class A{
  B b; C c; D d; E e;
  void m1(){
    d.setB(b);
  }
  B m2(){
   return d.getB();
  }
  C m3(){
    return b.c;
  }
  void m4(){
    b.c = c;
  }
  D m5(){
      return e.me(b,c);
  }
}
```



Challenge: Static analysis that extract object graph with dataflow edges for security

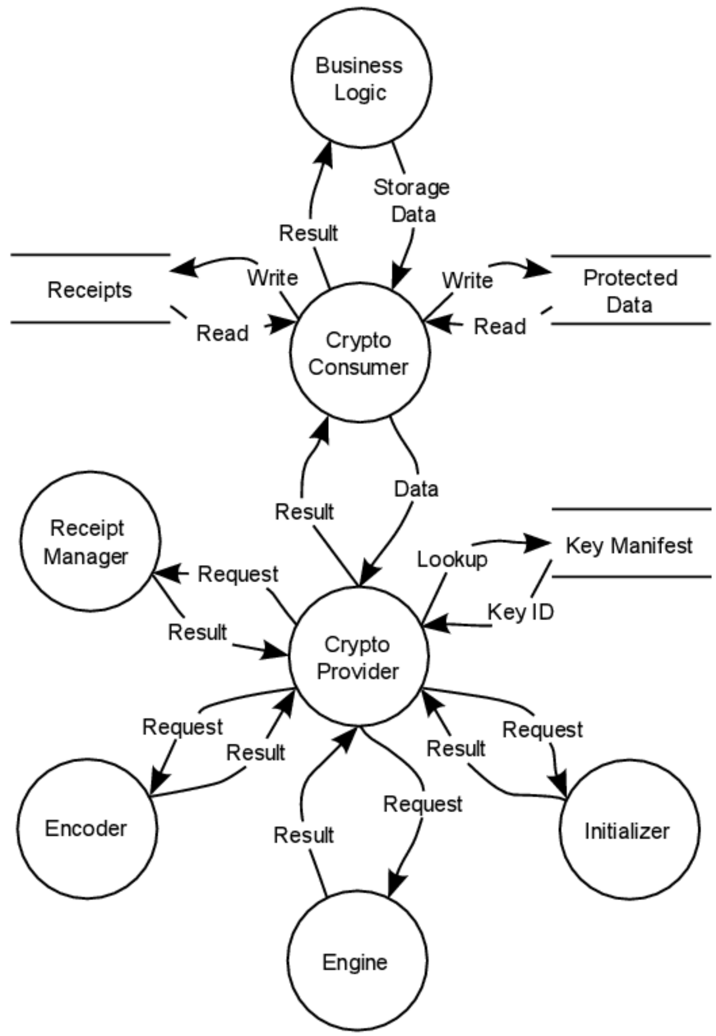
- Hierarchy
- Soundness
- Aliasing
- more…
  - Summarization
  - Precision
  - Traceability to code

Challenge: hierarchy allow high level understanding and details



DFD – Level 1

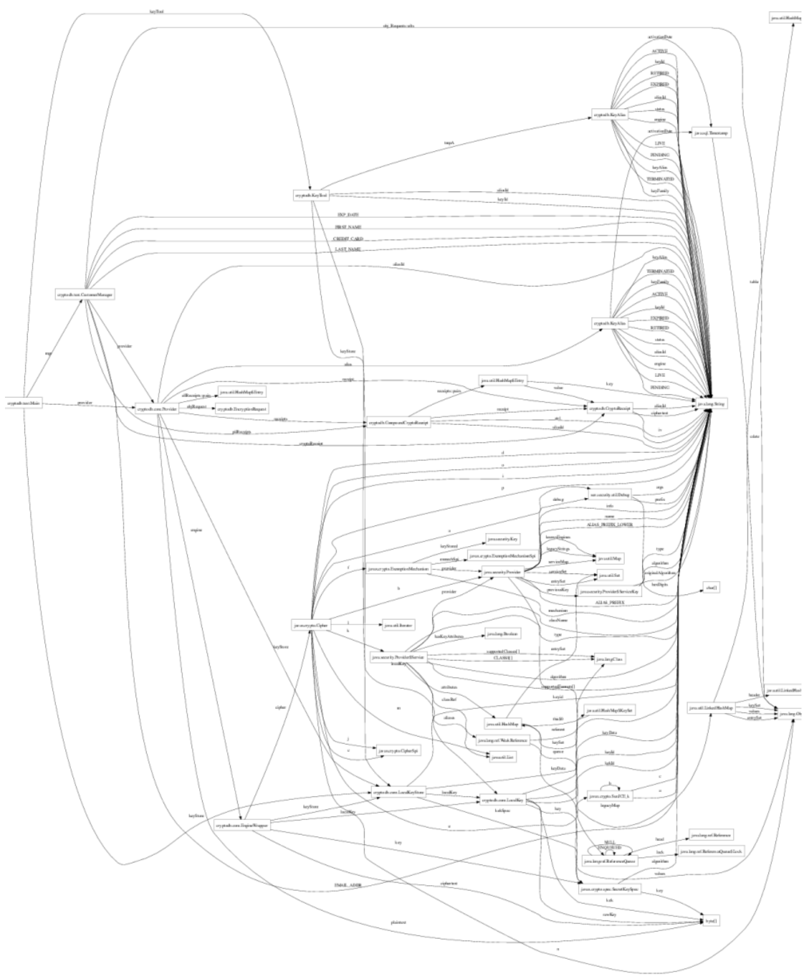
[Fig. 3.2., Kenan, Symantec Press'06]



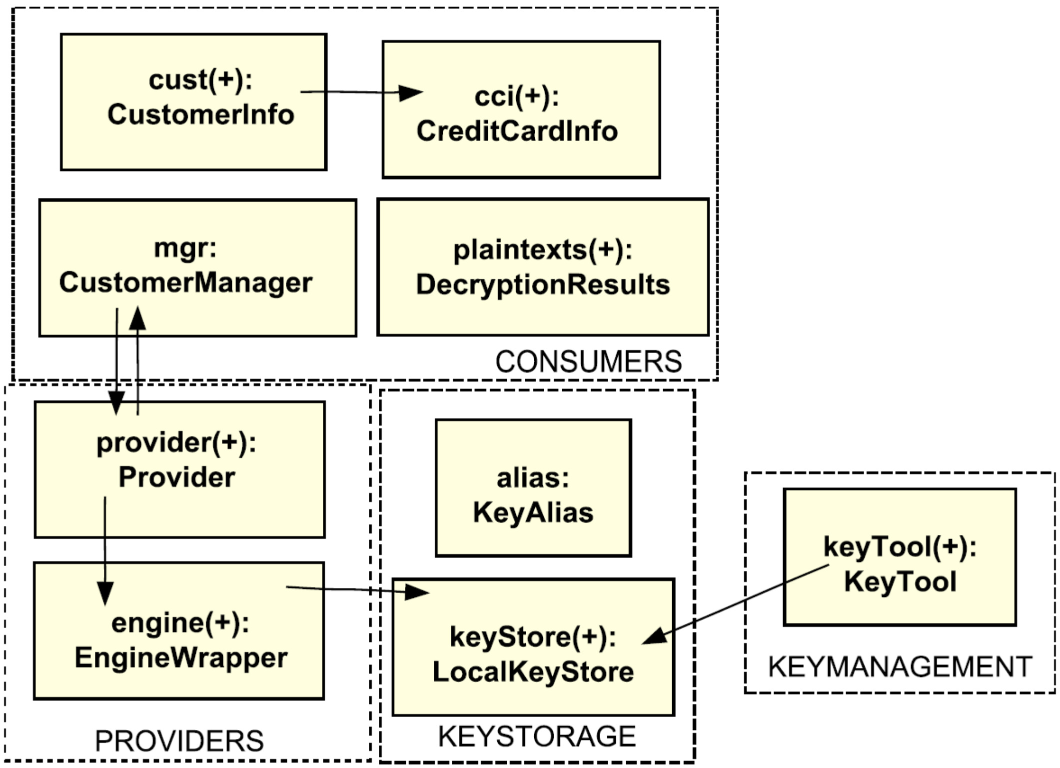
DFD – Level 2

[Fig 6.1, Kenan, Symantec Press'06]

Extracted object graphs



Flat object graph
[Jackson and Waingold, TSE'01]



Ownership Object Graph with points-to edges
[Abi-Antoun and Aldrich, OOPSLA'09]

Step 1. Use ownership domain annotations

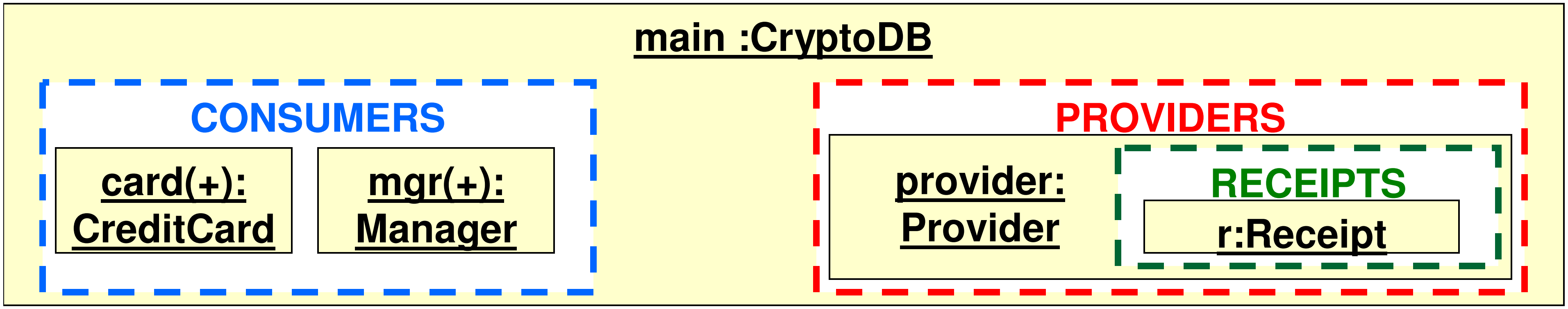
[Aldrich and Chambers, ECOOP'04]

- Assign each object to an **ownership domain**
  - **Domain:** *defn. a* ***conceptual group of objects***
  - Domain is similar to architectural runtime tier
- Hierarchical organization of objects
  - Not available in plain Java code
  - Allow abstraction by ownership hierarchy
    - Architecturally significant objects near top of hierarchy
    - Implementation details further down
- Typechecker ensures annotations and code **are consistent!**

Step 2. Run analysis

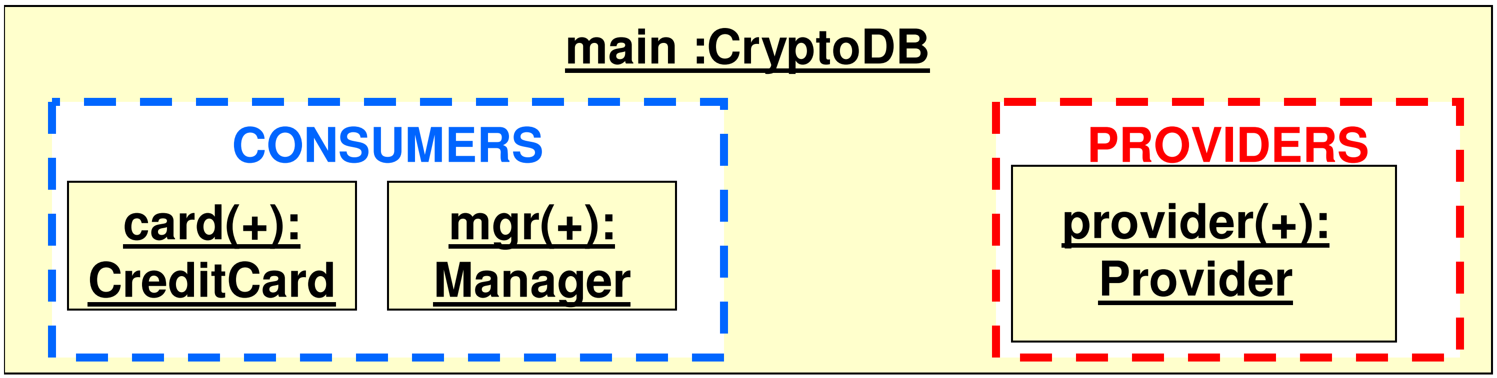
- Annotations:
  - Local/modular (checked one class at a time)
- Static analysis abstractly interprets program with annotations
  - Whole program analysis
  - Constructs a global, hierarchical Ownership Object Graph (OOG)
  - Object/domain hierarchy
  - Dataflow edges
- Analysis must address challenges

Abstraction by ownership hierarchy



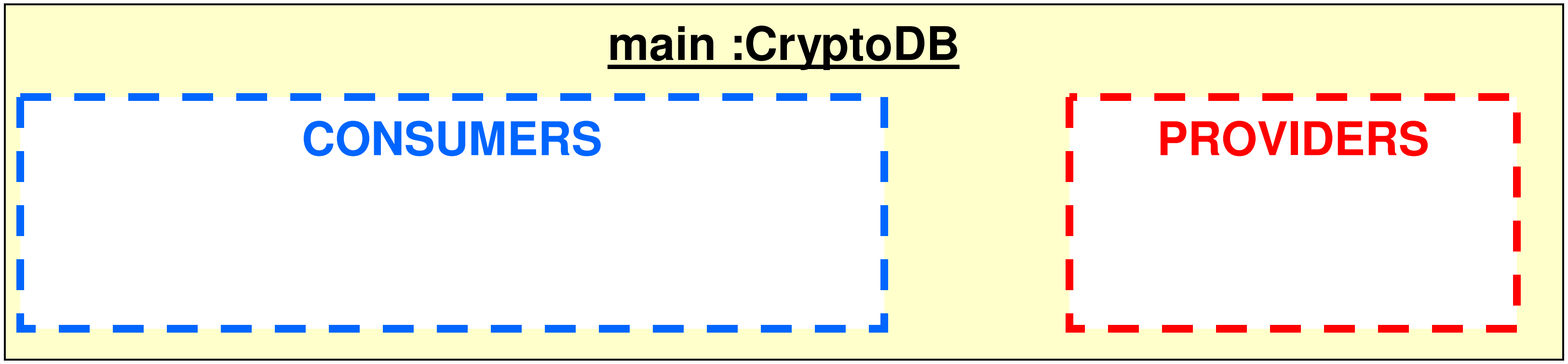

```
+-main: CryptoDB /
  +- PROVIDERS /
  | +- provider:Provider /
  |   +- RECEIPTS /
  |     +- r: Receipt
  +- CONSUMERS /
  | +- card: CreditCard
  | +- mgr: Manager
```

Abstraction by ownership hierarchy




```
+-main: CryptoDB /
  +- PROVIDERS /
  | +- provider:Provider
  +- CONSUMERS /
  | +- card: CreditCard
  | +- mgr: Manager
```

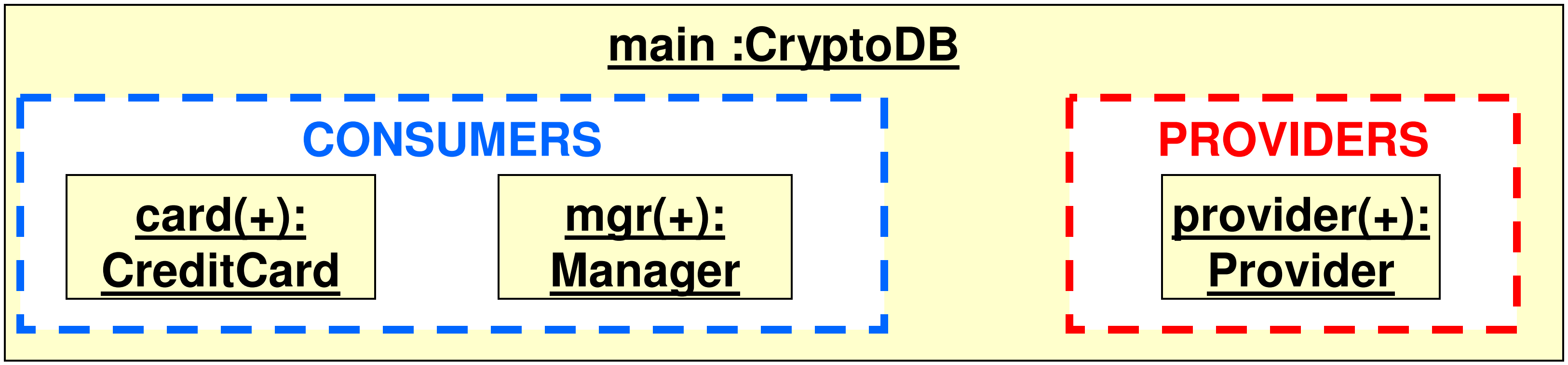
Analysis uses annotations to create tiers




```
@Domains({"CONSUMERS","PROVIDERS"})
class CryptoDB {


}
```

Analysis adds objects to domains

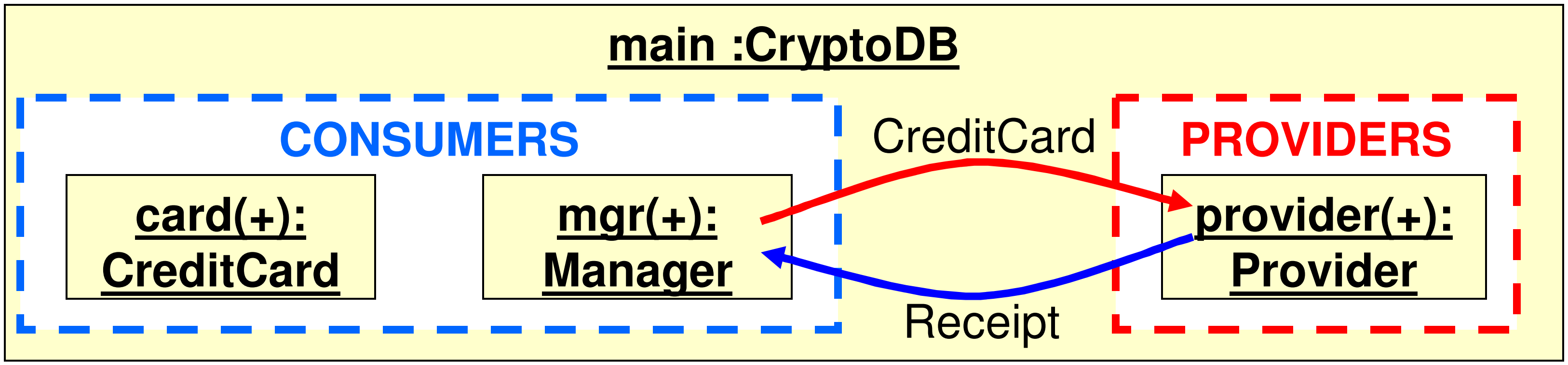



```
@Domains({“CONSUMERS”,“PROVIDERS”})
class CryptoDB {
   @Domain(“CONSUMERS”) Manager mgr = new Manager();
   @Domain(“PROVIDERS”) Provider provider = new Provider();
}

class Manager{
 ...
 @Domain(“CONSUMERS”) CreditCard card = new CreditCard();

}
```

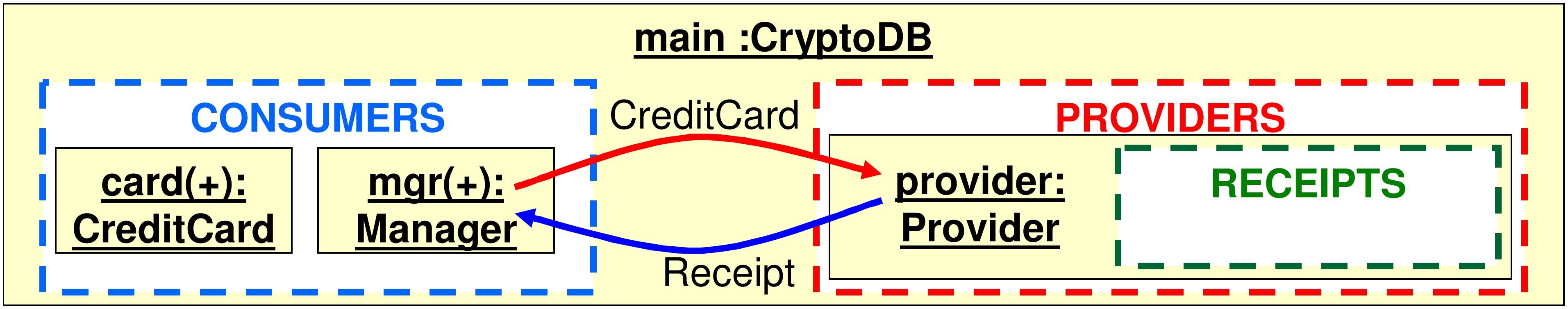
… and dataflow edges




```
@Domains({"CONSUMERS","PROVIDERS"})
class CryptoDB {
   @Domain("CONSUMERS") Manager mgr = new Manager();
   @Domain("PROVIDERS") Provider provider = new Provider();
}

class Manager{
 ...
 @Domain("CONSUMERS") CreditCard card = new CreditCard();
 Receipt r = provider.encrypt(card);
}
```

… creates domain in objects

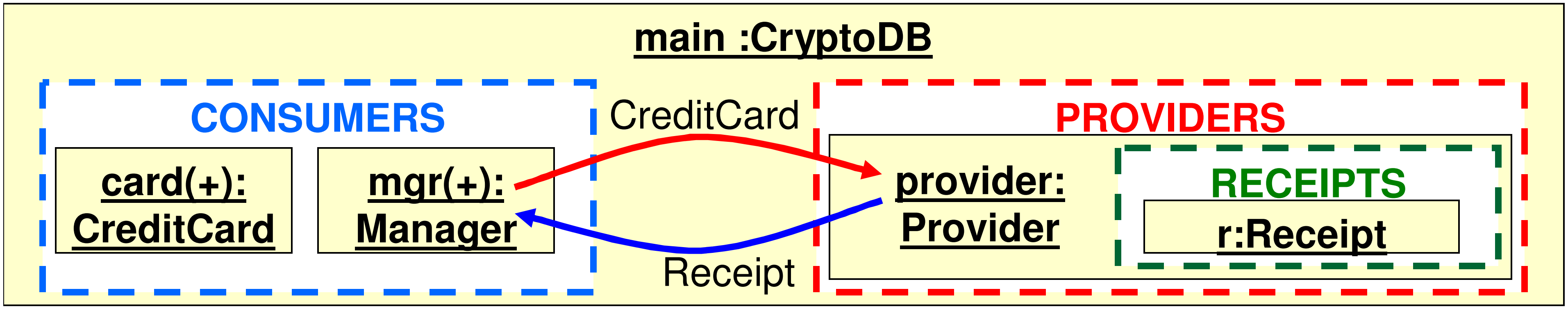



```
class Manager {
   ...
   @Domain("CONSUMERS") CreditCard card = new CreditCard();
   Receipt r = provider.encrypt(card);
}

@Domains({"RECEIPTS"})
class Provider {


}
```

… and objects in domains

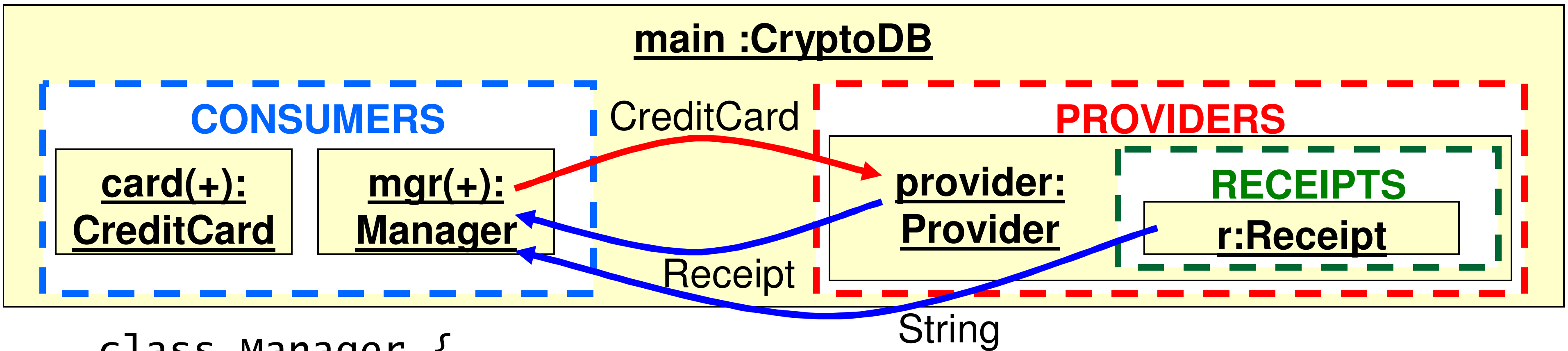



```
class Manager {
   ...
   @Domain("CONSUMERS") CreditCard card = new CreditCard();
   @Domain("provider.RECEIPTS")
    Receipt r = provider.encrypt(card);

}

@Domains({"RECEIPTS"})
class Provider {
   @Domain("RECEIPTS") Receipt encrypt( CreditCard c){
        return new Receipt();
   }
}
```

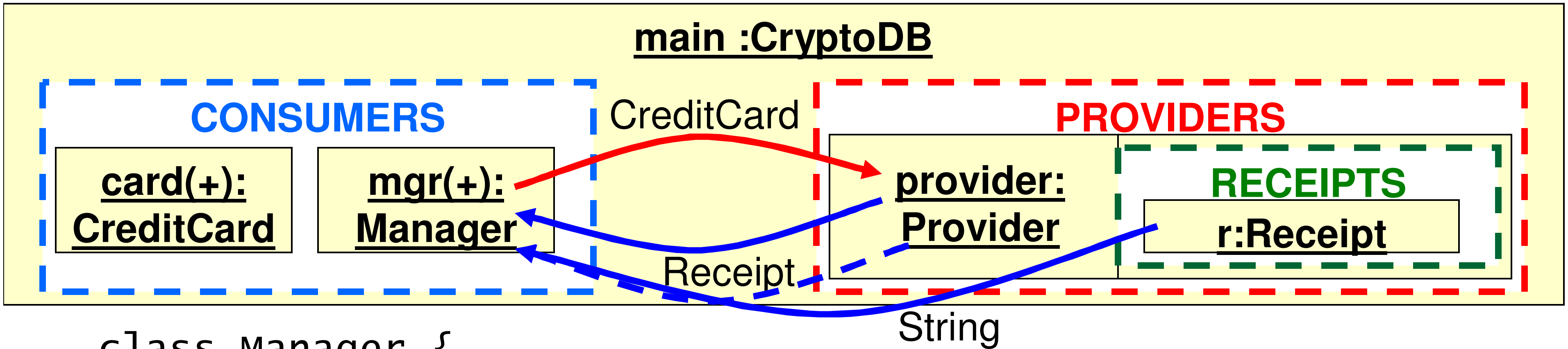
…and more edges between objects




```
class Manager {
   ...
    @Domain("CONSUMERS") CreditCard card = new CreditCard();
    @Domain("provider.RECEIPTS")
   Receipt r = provider.encrypt(card);
   String id = receipt.getId();
}

@Domains({"RECEIPTS"})
class Provider {
   @Domain("RECEIPTS") Receipt encrypt( CreditCard c){
        return new Receipt();
   }
}
```

OOG allows different levels of abstraction

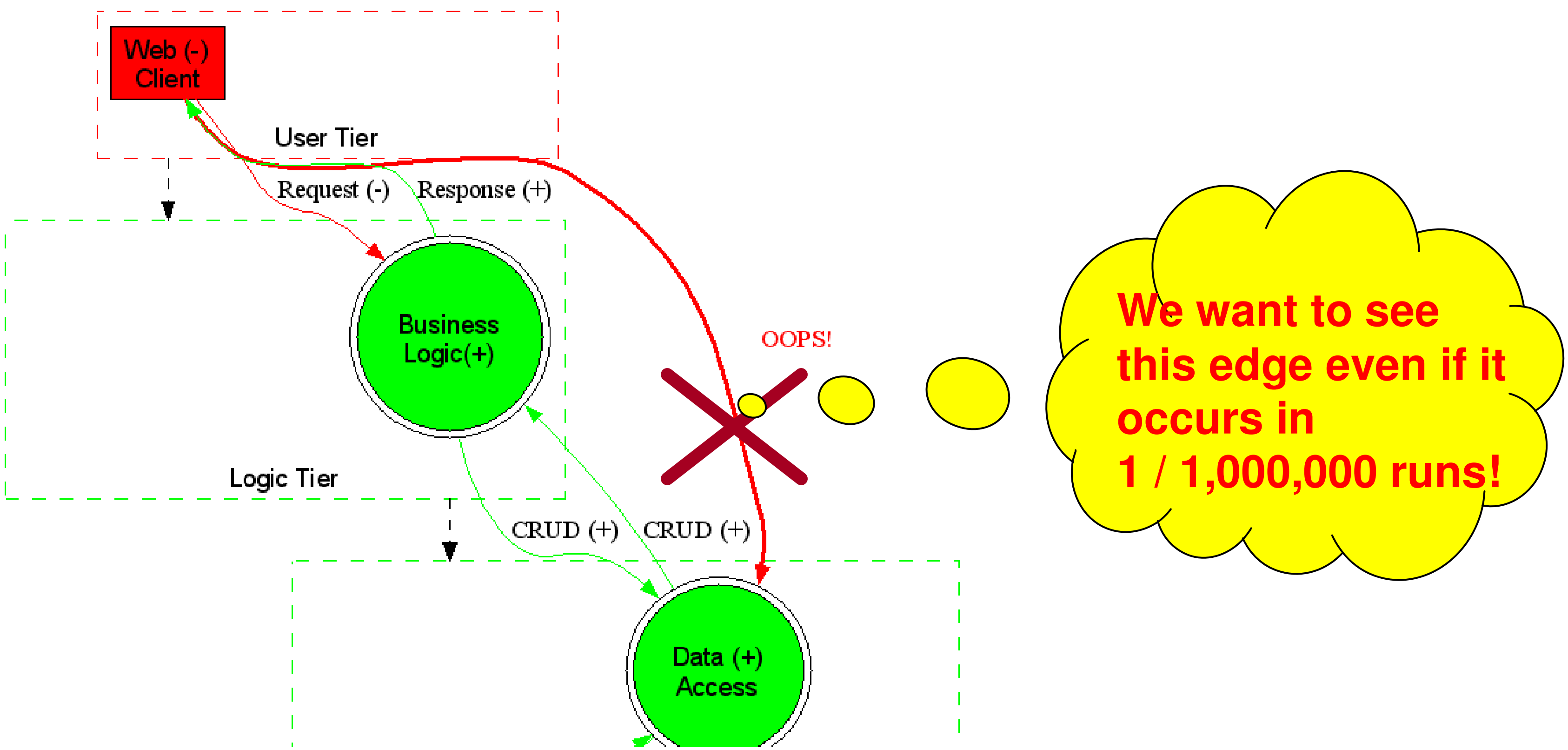



```
class Manager {
   ...
   @Domain("CONSUMERS") CreditCard card = new CreditCard();
   @Domain("provider.RECEIPTS")
   Receipt r = provider.encrypt(card);
   String id = receipt.getId();
}

@Domains({"RECEIPTS"})
class Provider {
   @Domain("RECEIPTS") Receipt encrypt( CreditCard c){
        return new Receipt();
   }
}
```

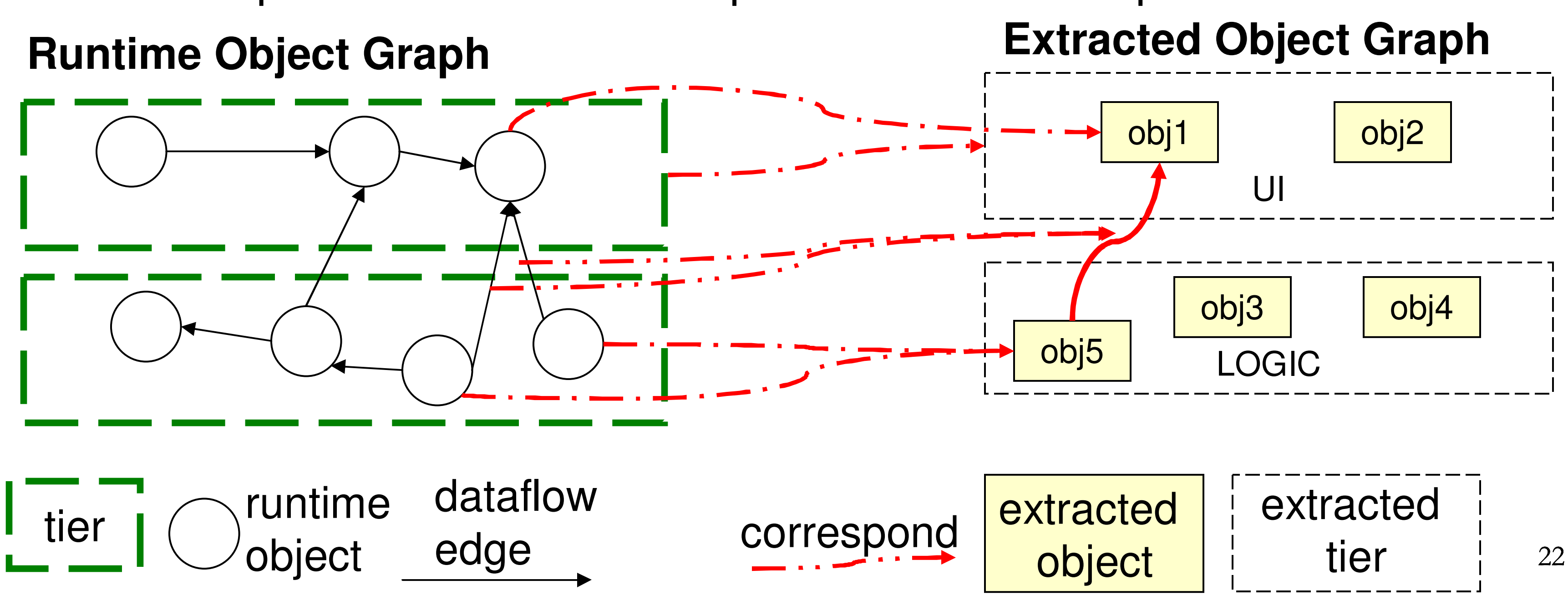
Challenge: soundness

- **Soundness**: **represent all** objects and relations that may exist at runtime



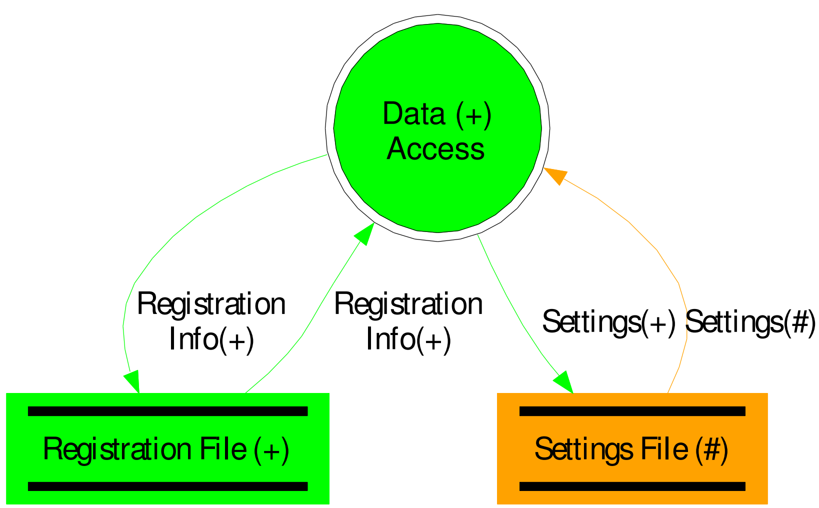
Soundness of extracted object graph

- Extracted sound object graph approximates any Runtime Object Graph
  - Each runtime object has **exactly one** representative in extracted object graph
  - Edge between two runtime objects must correspond to edge between representative of two objects
- Formal proof available in companion technical report

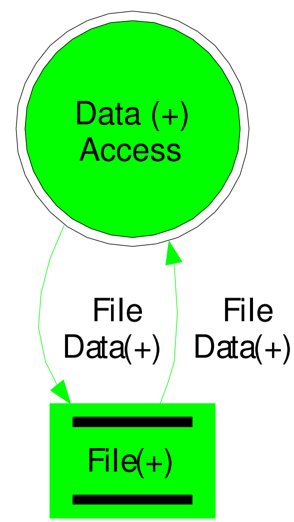


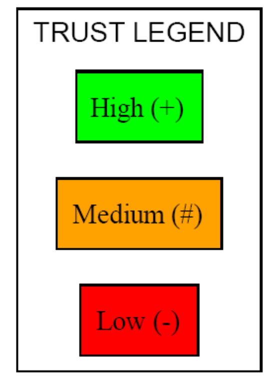
Aliasing challenge: no one runtime entity should appear as two "components"

- Impacts **architectural properties**
  - Settings File (**trustLevel** = **Medium**)
  - vs. Registration File (**trustLevel** = **Full**)
  - Combine these two instances into one?



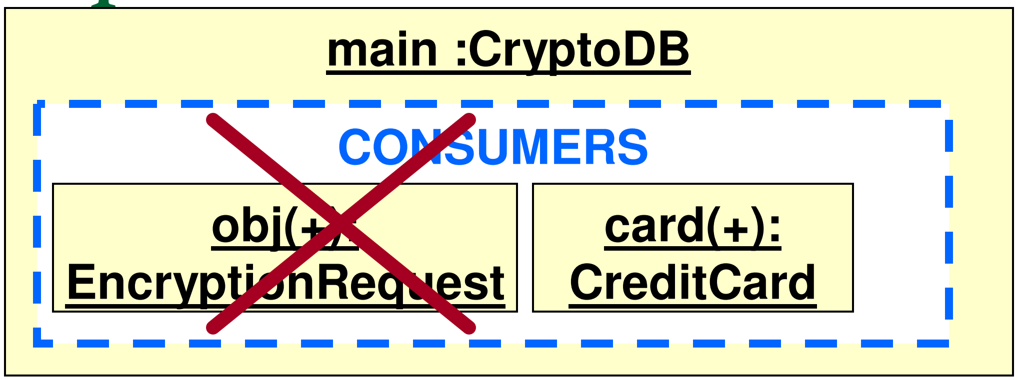
Assume 'Registration File' and 'Settings File' distinct, with different values for trustLevel.

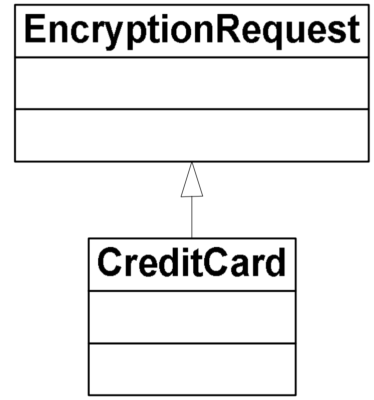




Assume one File DataStore, with one value for trustLevel.

No one runtime object has two representatives in OOG





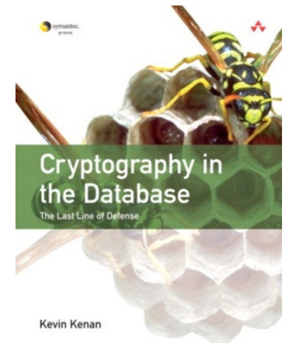

```
class Manager{
 ...
 @Domain("CONSUMERS") CreditCard card = new CreditCard();
 @Domain("CONSUMERS") EncryptionRequest obj = card;
}

class CreditCard extends EncryptionRequest{...}
```

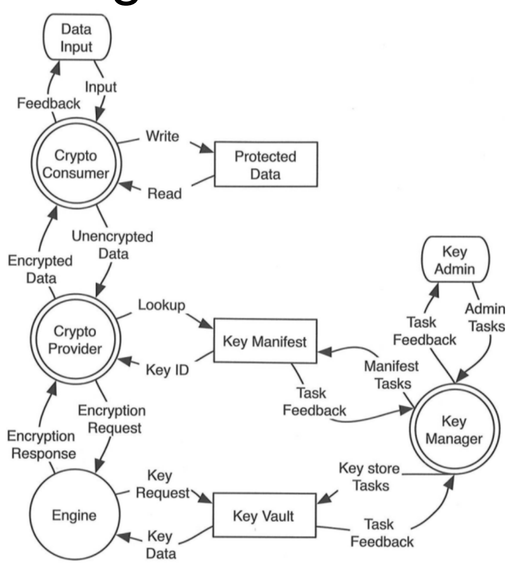
Evaluation: extended example

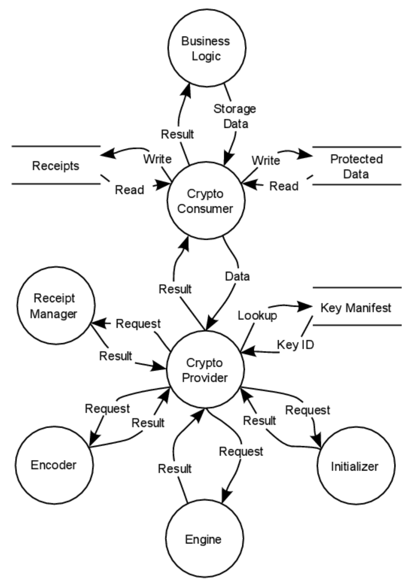
- ***Hypothesis:*** *Given legacy code to which we add ownership domain annotations, a* ***static analysis*** *can extract a* ***sound OOG*** *depicting* ***dataflow edges*** *that correspond to those manually drawn by a developer who is reasoning about the* ***runtime architecture*** *of a system and* ***dataflow communication****.*

Subject system

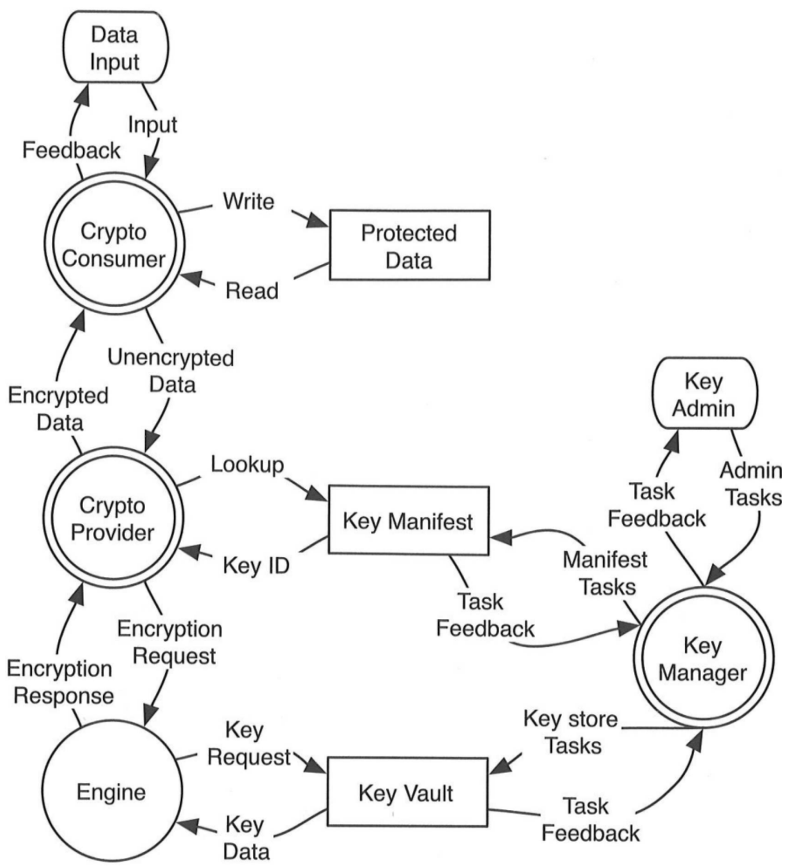


- Cryptographic database (3KLOC)
- As-designed DFDs available in book [Kenan, Symantec Press'06]
- Allows us to compare as-built OOG against as-designed DFD

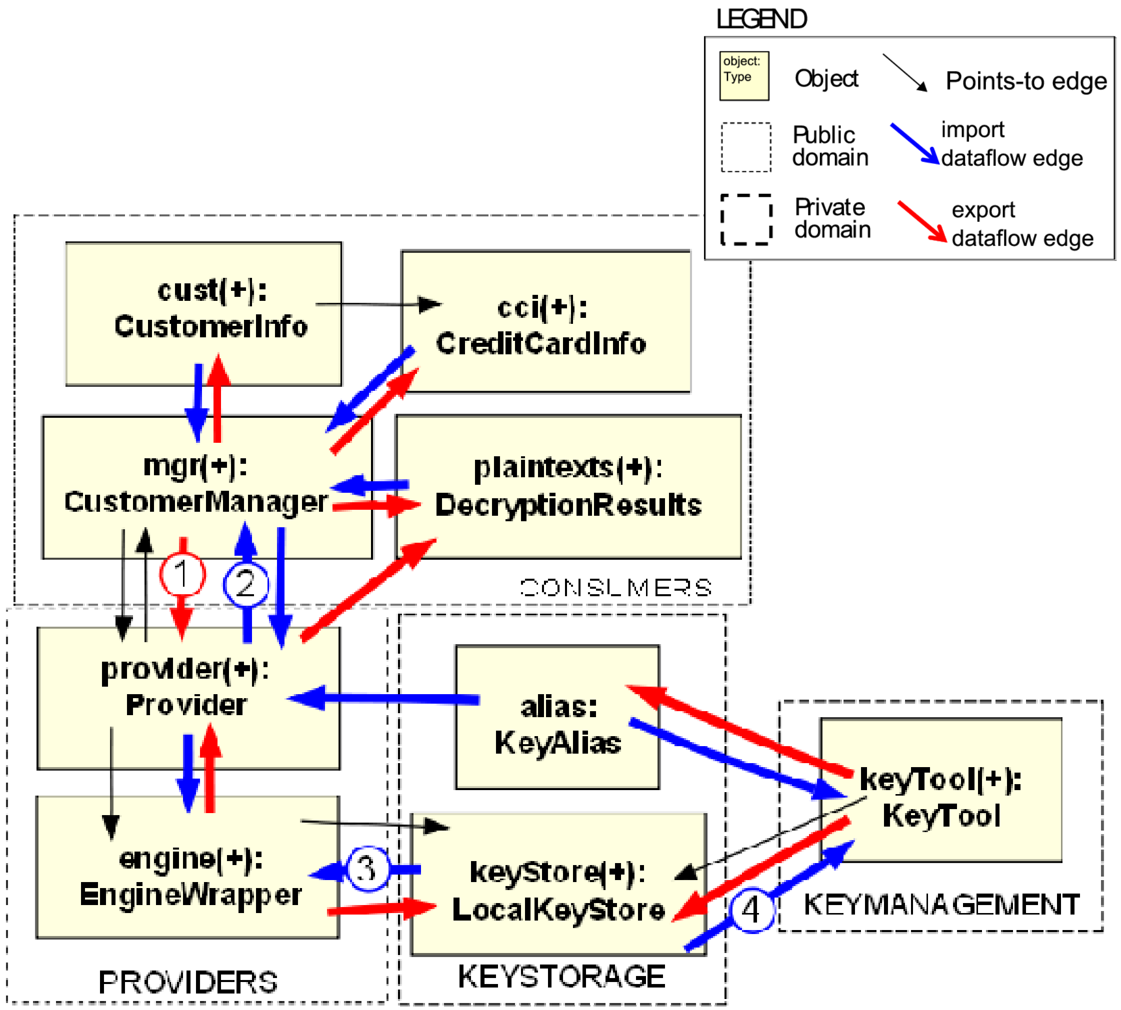




As-designed DFD vs. As-built OOG



As-designed DFD



As-built OOG

Evaluation is similar in style to Reflexion Models [Murhphy et al., TSE'01]

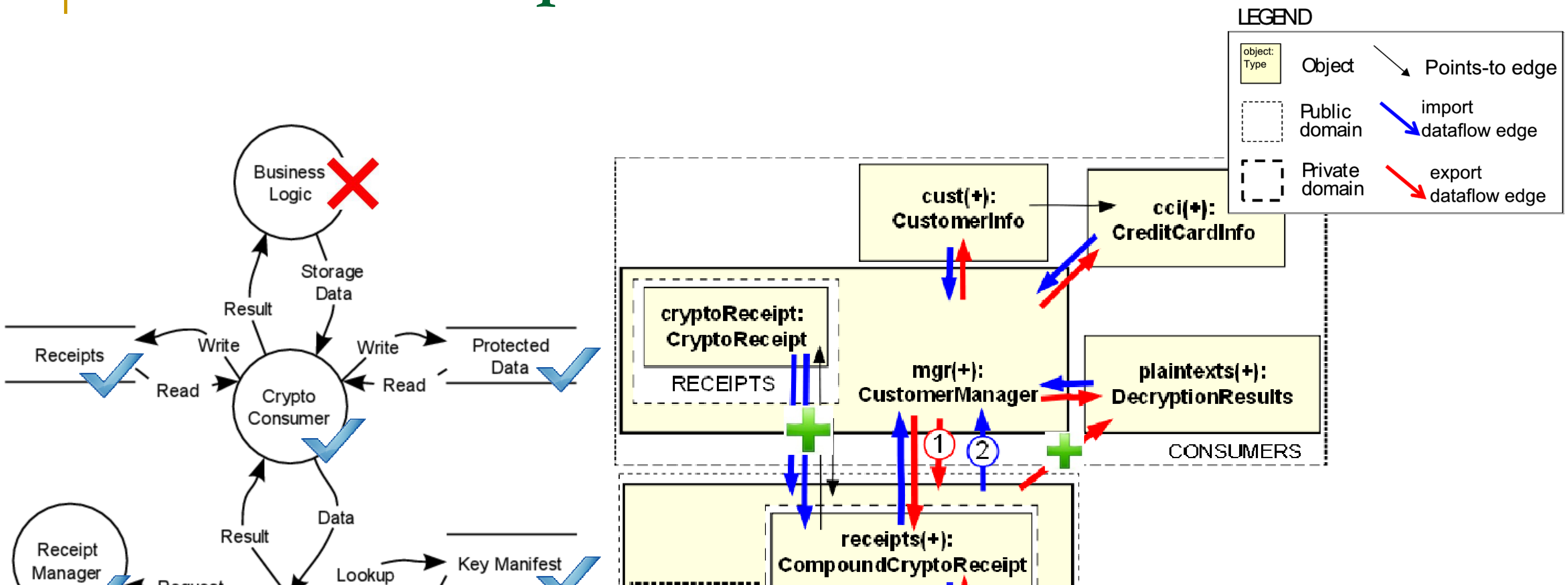
- Identify following differences:

Convergence: node or edge **in both** OOG and DFD

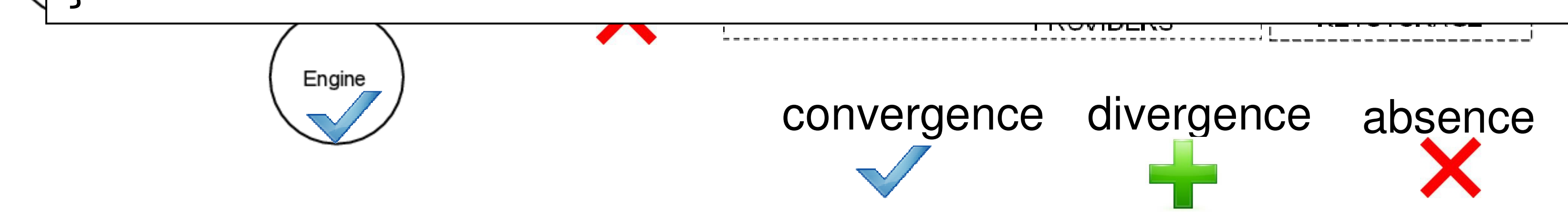
Divergence: node or edge in OOG, but **NOT in DFD**

Absence: node or edge **NOT in OOG**, but in DFD

Hierarchical representation has more details




```
class CompoundCryptoReceipt{
...
  public void addReceipt(…,@Domain("RECEIPTS") CryptoReceipt receipt) {
    @Domain("shared") String aliasIdReceipt = receipt.getAliasId();
  }
}
```



Limitations

- Expressiveness challenges in Ownership Domains
- Cost of adding annotations
  - Estimated to be 1 hour/KLOC [Abi-Antoun et al. QoSA ‘12]
- Known limitations of static analysis:
  - Dynamic code loading
  - Reflection

Future work

- Analyze more, larger systems
  - Added annotations to Apache FtpServer
  - Add security properties (trustlevel) to OOG
  - Analyze security constraints
- Ask security experts to use OOG with dataflow edges
- Use OOG with dataflow edges for program comprehension [Ammar and Abi-Antoun, WCRE'12]

Conclusion

- Designed static analysis to extract sound, hierarchical object graph with dataflow edges
- Formalized the analysis and proved its soundness (proof in technical report)
- Compared as-built object graph to as-designed Data-Flow Diagram
- Reverse-engineered dataflow edges are similar to edges drawn by architects who are reasoning about security