CSE 543: Computer Security
Module: Future Secure Programming

Prof. Syed Rafiul Hussain
Department of Computer Science and Engineering
The Pennsylvania State University

Programmer's Problem

- What does "program for security" mean?
- Have you ever "programmed for security"?
- When do you start consider security when you program?
- What do you try to do to make your code secure?
- When do you know you are done making your code secure?

Programmer's Problem

- Implement a program
  - Without creating vulnerabilities
- What is a vulnerability?

Software Vulnerabilities

- Vulnerability combines
  - A flaw
  - Accessible to an adversary
  - Who can exploit that flaw
- Which would you focus on to prevent vulnerabilities?

Buffer Overflow Detection

- For C code where
  - `char dest[LEN]; int n;`
  - `...`
  - `n = input();`
  - `...`
  - `strncpy(dest, src, n);`

- Can this code cause a buffer overflow?

Runtime Analysis

- One approach is to run the program to determine how it behaves
- Analysis Inputs
  - Input Values - command line arguments
  - Environment - state of file system, environment variables, etc.
- Question
  - Can any input value in any environment cause a vulnerability (e.g., exploit a buffer overflow)?
- What are the limitations of runtime analysis?

Fuzz Testing

- Dynamic software testing technique …
  - Run the software
- Where invalid, unlikely, and/or random inputs are provided to the program …
  - See what happens
- To detect crashes, exceptions, etc.
  - Which may be indicate of flaws that can be exploited
  - How would this detect a buffer overflow?
- Fuzz testing is "black-box testing" — do not need to examine the program code to run
- Research in grey/white-box testing, but industry uses fuzzing

Static Analysis

- Explore all possible executions of a program
  - All possible inputs
  - All possible states

Static Analysis

- Provides an approximation of behavior
- "Run in the aggregate"
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- "Run in non-standard way"
  - Run in fragments
  - Stitch them together to cover all paths
- Runtime testing is inherently incomplete, but static analysis can cover all paths

Static Analysis Example

- Descriptors represent the sign of a value
  - Positive, negative, zero, unknown
- For an expression, $c = a * b$
  - If a has a descriptor *pos*
  - And b has a descriptor *neg*
- What is the descriptor for c after that instruction?
- How might this help?

Descriptors

- Choose a set of descriptors that
  - Abstracts away details to make analysis tractable
  - Preserves enough information that key properties hold
    - Can determine interesting results
- Using *sign* as a descriptor
  - Abstracts away specific integer values (billions to four)
  - Guarantees when $a*b = 0$ it will be zero in all executions
- Choosing descriptors is one key step in static analysis

Buffer Overflow Static Analysis

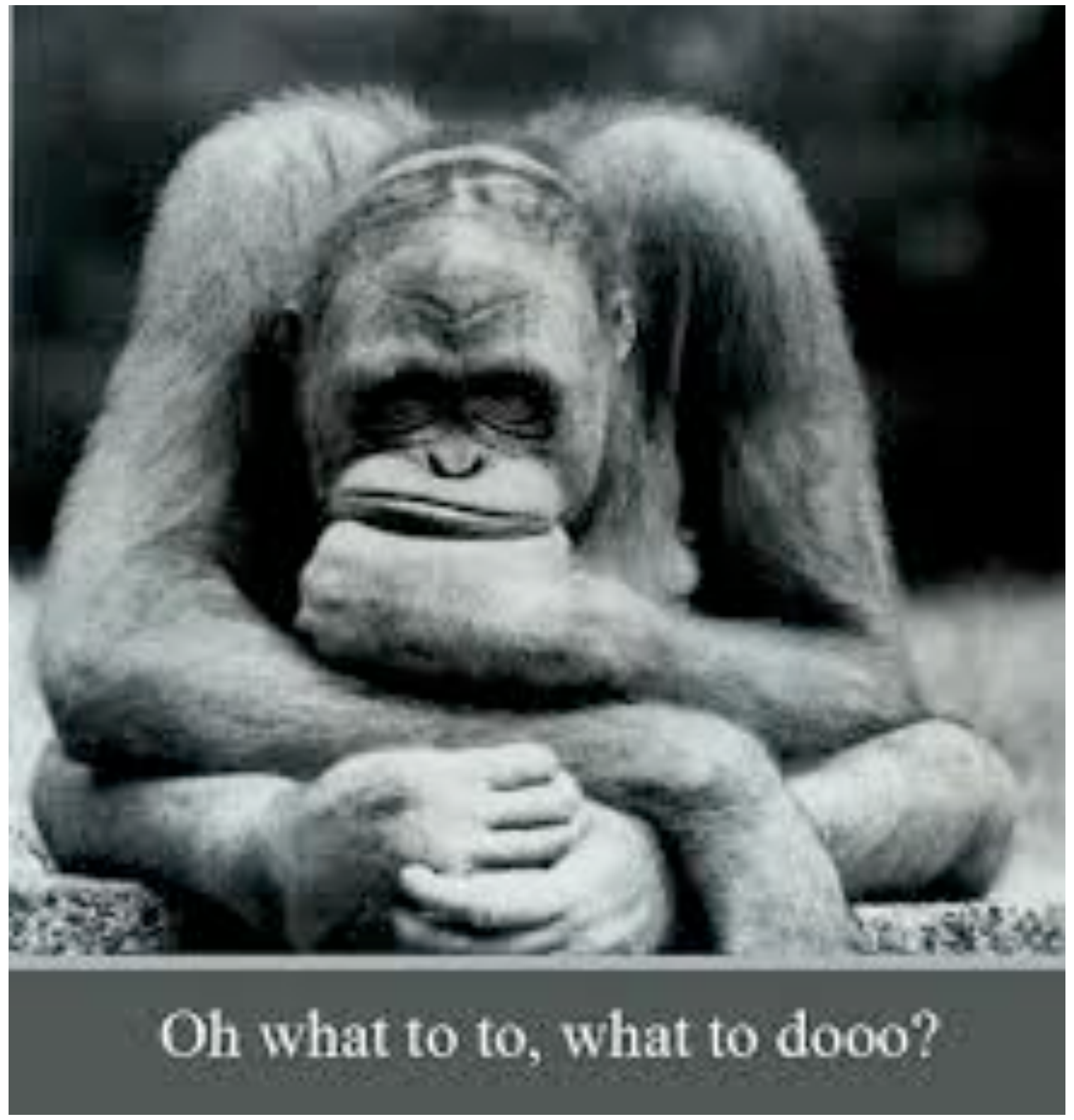
- For C code where
  - `char dest[LEN]; int n;`
  - `n = input();`
  - `strncpy(dest, src, n);`
- Static analysis will try all paths of the program that impact variable n and flow to strncpy
  - May be complex in general because
    - Paths: Exponential number of program paths
    - Interprocedural: n may be assigned in another function
    - Aliasing: n's memory may be accessed from many places
- What descriptor values do you care about for n?

Limitations of Static Analysis

- Scalability
  - Can be expensive to reason about all executions of complex programs
- False positives
  - Overapproximation means that executions that are not really possible may be found
- Accuracy
  - Alias analysis and other imprecision may lead to false negatives
  - Sound methods (no false negatives) can exacerbate scalability and false positives problems
- Bottom line: Static analysis often must be directed

Preventing Vulnerabilities

- What can the programmer do to secure their program?



Denning's Lattice Model

- Formalizes information flow models
  - FM = {N, P, SC, /, >}
- Shows that the information flow model instances form a lattice
  - N are objects, P are processes,
  - {SC, >} is a partial ordered set,
  - SC, the set of security classes is finite,
  - SC has a lower bound,
  - and / is a lub operator
- Implicit and explicit information flows
- Semantics for verifying that a configuration is secure
- Static and dynamic binding considered
- Biba and BLP are among the simplest models of this type

Implicit and explicit flows

- Explicit
  - Direct transfer to b from a (e.g., b = a)
- Implicit
  - Where value of b may depend on value of a indirectly (e.g., if a = 0, then b = c)
- Model covers all programs
  - Statement S
  - Sequence S1, S2
  - Conditional c: S1, …, Sm
- Implicit flows only occur in conditionals

Semantics

- Program is secure if:
  - Explicit flow from S is secure
  - Explicit flow of all statements in a sequence are secure (e.g., S1; S2)
  - Conditional c: S1, …, Sm is secure if:
    - The explicit flows of all statements *S1, …, Sm* are secure
    - The implicit flows between c and the objects in Si are secure

Build on Type Safety

- A type-safe language maintains the semantics of types.
  E.g., can't add int's to Object's.
- Type-safety is compositional.
  A function promises to maintain type safety.

Example 1

```
Object obj;
int i;
obj = obj + i;
```

X

Example 2

```
String proc_obj(Object o);
...
main()
{
  Object obj;
  String s = proc_obj(obj);
   ...
}
```

Labeling Types

Example 1

```
int{high} h1,h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
l = h2 + l;   ✗
```

- Key insight:
  label types with security levels
- Security-typing is compositional

Example 2

```
String{low}
proc_obj(Object{high} o);
...
main()
{
  Object{high} obj;
  String{low} s;
  s = proc_obj(obj);
  ...
}
```

Implicit Flows

<u>Static (virtual) tagging</u>

```
int_Low mydata = 0;
int_Low mydata2 = 0;
if (test_High)
    mydata = 1;
else
    mydata = 2;
mydata2 = 0;
print_Low(mydata2);
print_Low(mydata);
...
```

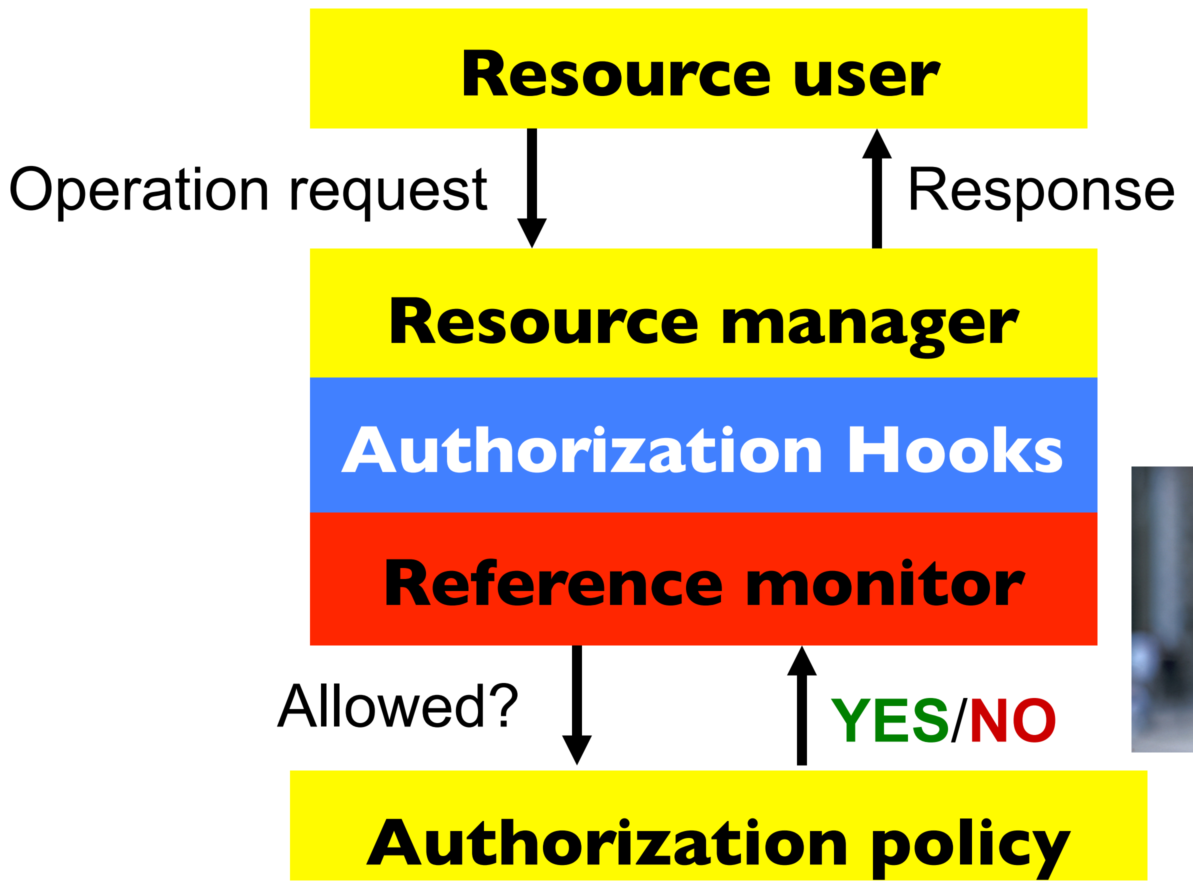
`mydata` **contains information about** `test` **so it can no longer be** `Low`**, but** `mydata2` **is outside the conditional, so it is untainted by** `test`

Causes type error at compile-time

Retrofitting for Security

- Take the code written in a language of the programmers' choice (for functionality) and retrofit with security code (mostly-automated)
- Consider authorization bypass vulnerabilities
  - In these vulnerabilities, programmers forget to add code to control access to program resources

What is authorization?



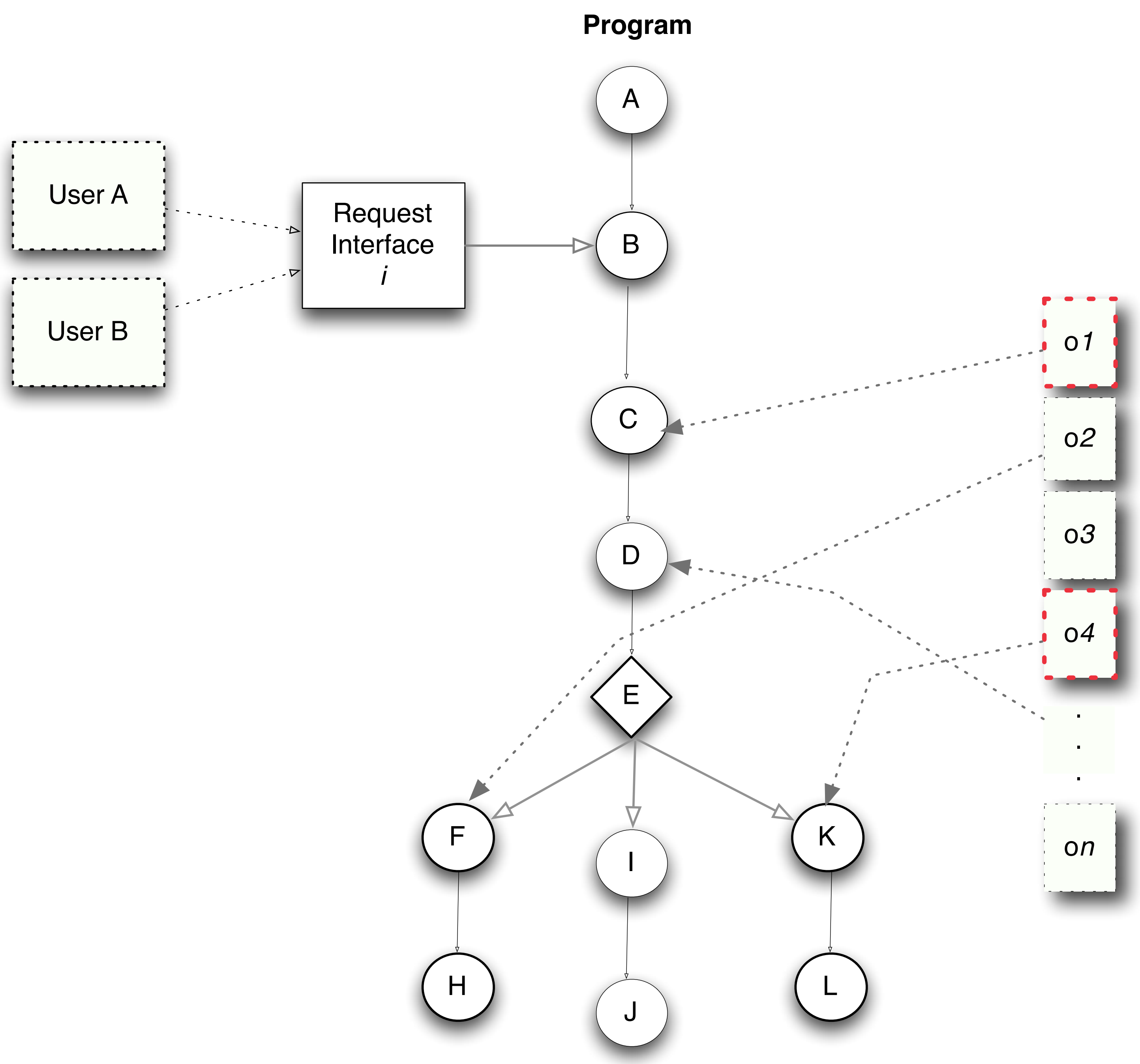
⟨Alice, /etc/passwd, ***File_Read***⟩

What Should a Programmer Do?

- How would you ensure that all accesses to all security-sensitive window objects in the X Server are authorized?

Inferring Sensitive Operations

Program



Challenges

A. Identify security-sensitive *resources*

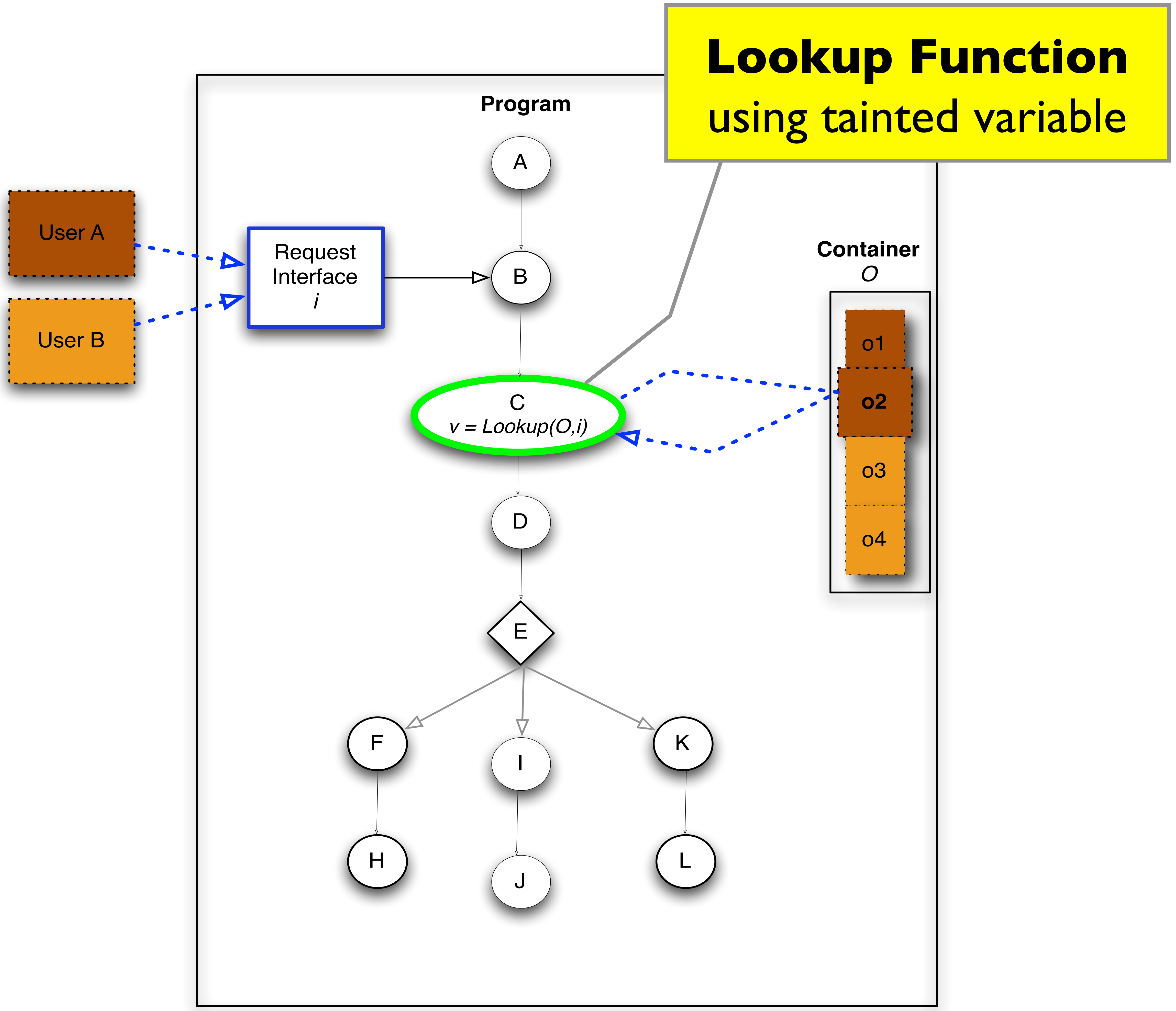
- Programs manipulate many variables
  - 7800 in X Server
  - Of over 400 structures
  - Many, many structure-member accesses

Solution

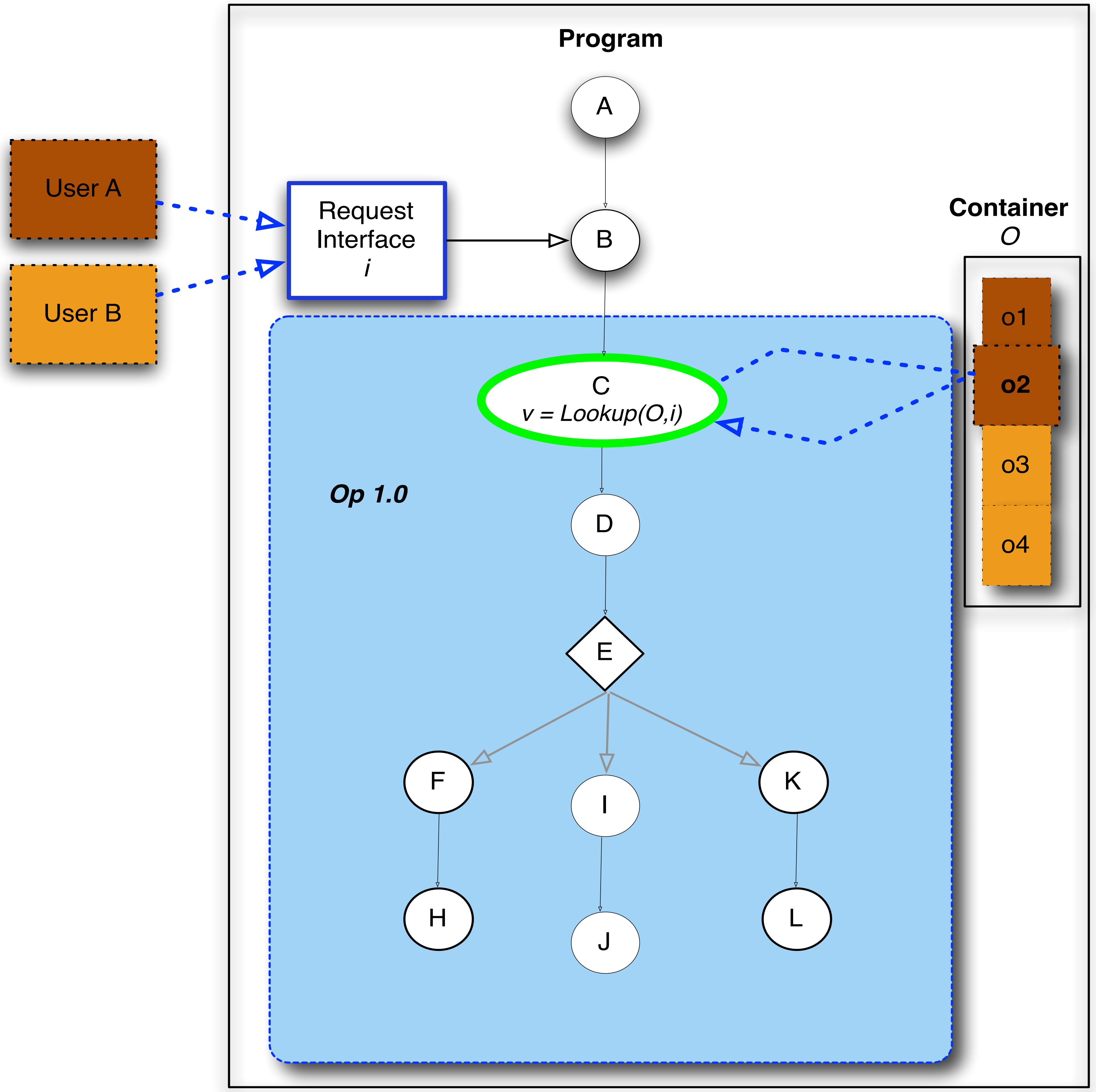
Requests make choices

- In servers, *client-request* determines *choices* that client subjects can make in the program
- "Choice":
  - **Resources**: Determine which *elements* are chosen from containers.
  - **Operations**: Determine which *program path* is selected for execution.

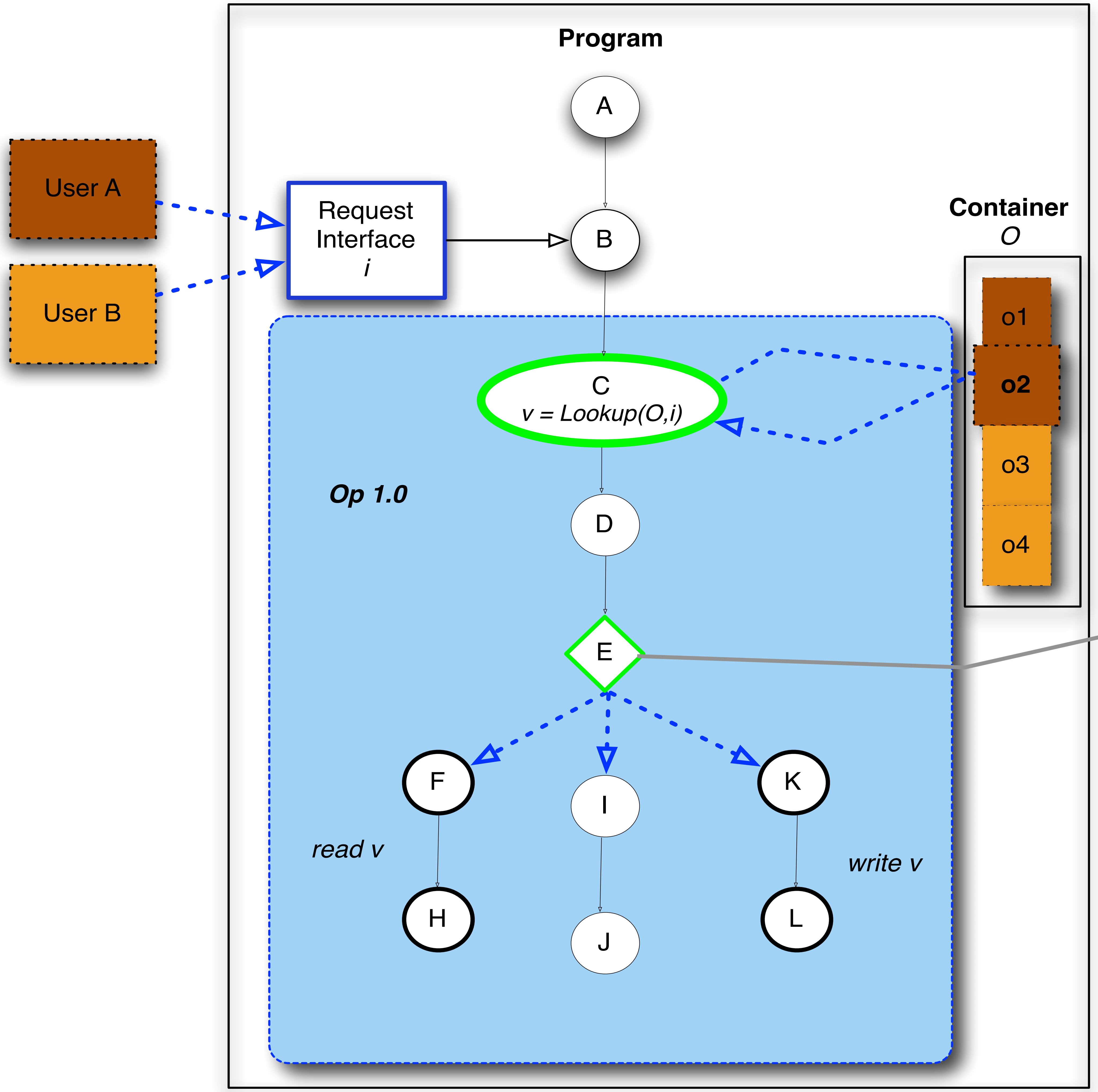
Idea: Request Choices



Idea: Request Choices

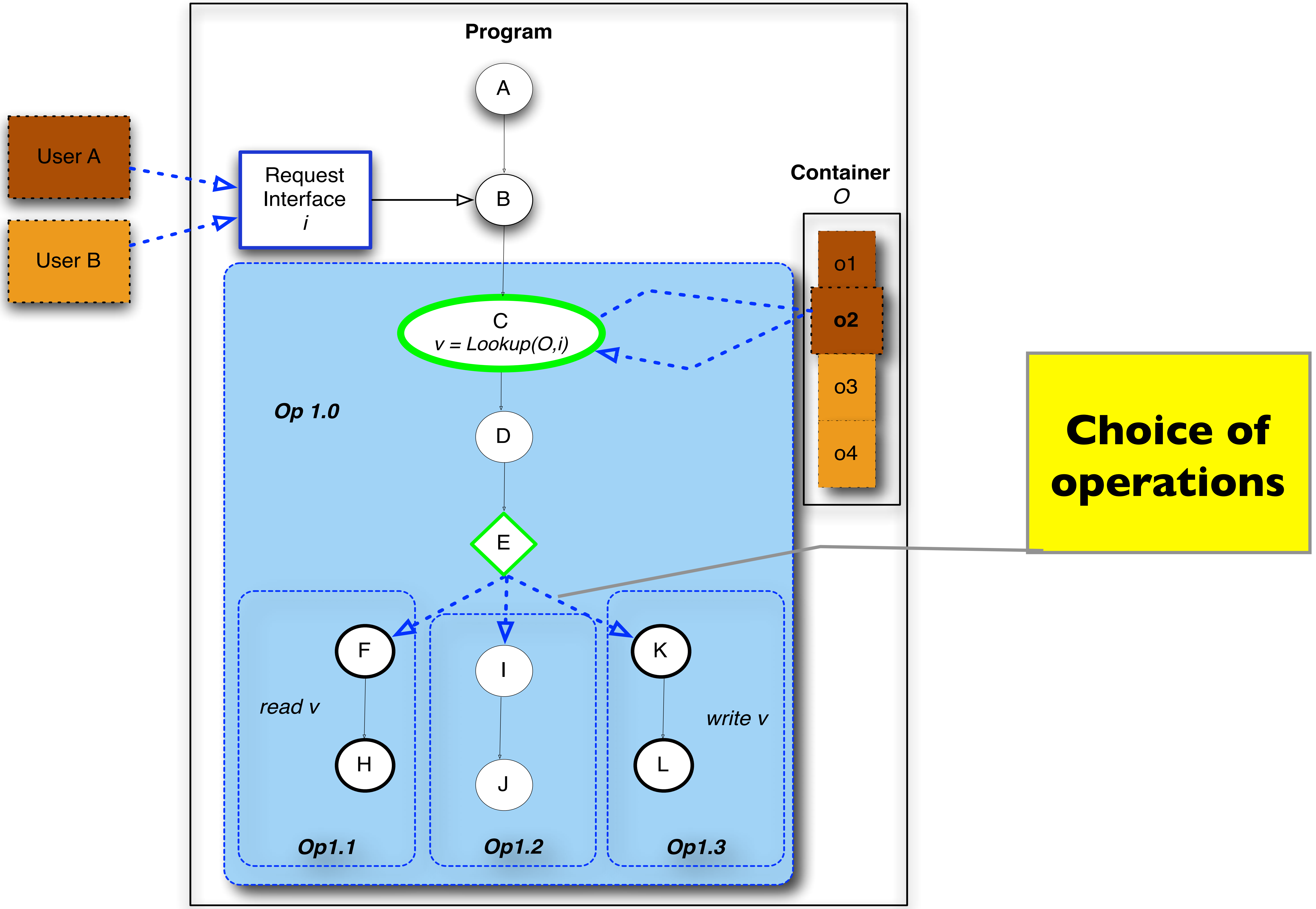


Idea: Request Choices

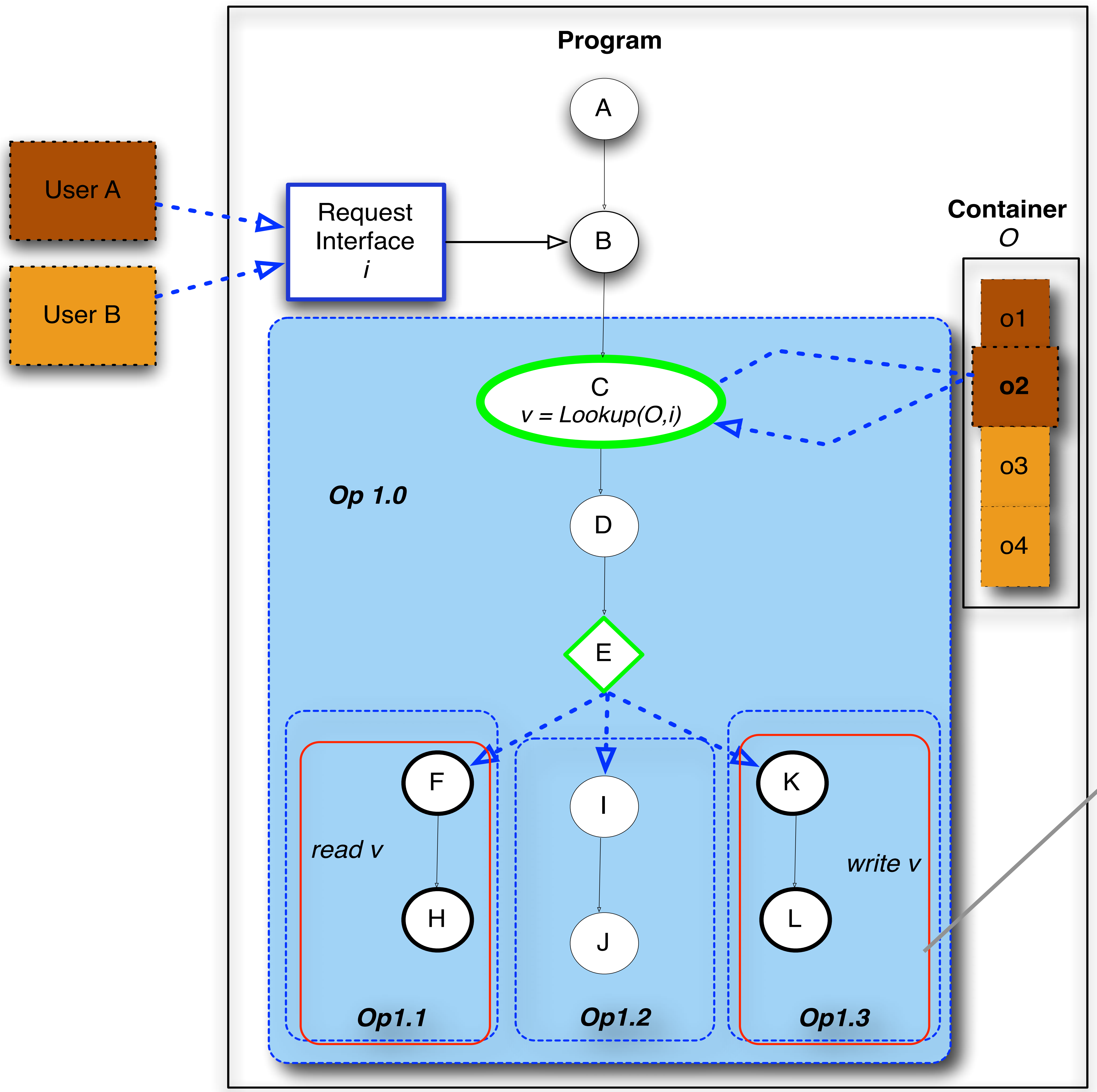


Control Statement Predicated on a tainted variable

Idea: Request Choices



Idea: Request Choices



Security sensitive operation

Mediate SSOs

- Where should we place authorization hooks?
- Mediate all **security-sensitive operations** found
  - Good: Enforce least privilege flexibly
  - Bad: Maximal number of hooks means…
- Ensure at least one hook per security-sensitive operation
  - Good: Minimal number of hooks
  - Bad: Must ensure that all authorized subjects pass…
- Idea: Determine if you have blocked enough
  - Suppose OP-1 dominates OP-2, then if policy for OP-1 blocks all the unauthorized subjects for OP-2…

Future of Secure Programming

- Write your program with functionality in mind
- Determine security policies to be enforced on the program
  - Semi-automated - e.g., use program analysis to find SSOs
- Use security policies to guide retrofitting of program with security code automatically
- Can it be done?
  - Caveat: Some security knowledge is application-specific
  - Caveat: Cannot retrofit for security from program code alone

Take Away

- Programming for security is difficult
  - Programmers create "flaws" that are often accessible and exploitable by adversaries (vulnerabilities)
- Program analysis can find some flaws
  - Static and dynamic, but limitations for each
- May need to fix program - security types and "choice"
- The future of secure programming may look very different
  - Now: use favorite language for achieving function and try to add security code without creating flaws
  - Future: use favorite language for achieving function and retrofit based on a "security program"