C12 Type Safety

1. Type Safety

- Definition depends on the definition of error and guarantees the absence of untrapped errors. A language is unsafe if an error which has occurred during execution is not made known

1.1 Memory and Type Safety

Memory Safety

- Each dereference can only access "intended target"
  - Spatial access errors (out of bounds array access)
  - Temporal access error (stale pointer dereference)

Type Safety

- Stronger than memory safety
- Operations on the object are always compatible with the object's type
- **Type safe languages**
  - Improves quality of software
  - Safety guaranteed by static checks and runtime checks
  - Dynamically typed languages (Python) does not require declaration of types and each operation on objects created is permitted but may be unimplemented (which throws an exception)
- **Limitations**
  - Enforcement is expensive since garbage collection is required to avoid temporal violations; Bounds and null pointer checks are present to avoid spatial violations; Hiding abstract types may inhibit optimisations
  - Does not deal with all security problems (race conditions possible in Java code; erasure of sensitive data not guaranteed)

2. Execution Integrity

2.1 Execution Integrity

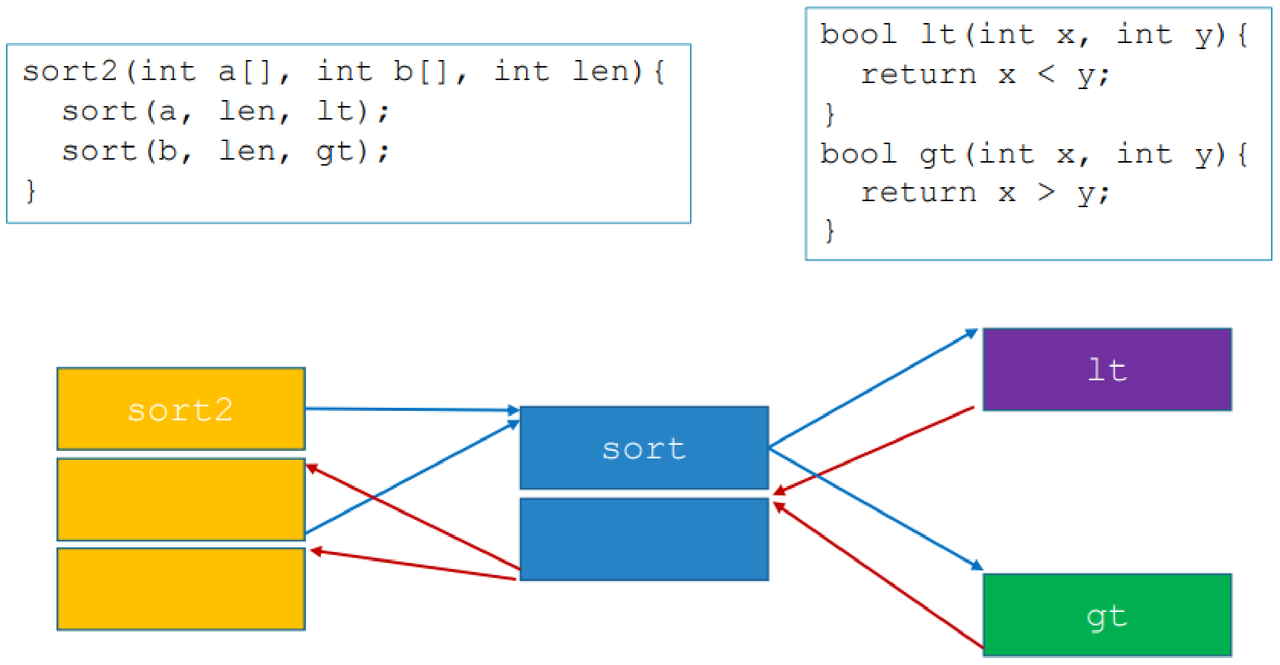
- Core of software security which includes
  - Separating data from control information
  - Enforcing system-environment integrity where certain aspects of the processor cannot be changed

    - Modules can only access certain memory regions & calls from a module can only be done to predefined routines and returns only at defined entry points

2.2 Control Flow Integrity

- Control-flow integrity means execution must follow a control-flow graph (CFG) determined in advance.
- A CFG can be defined by analysis, by execution profiling or by an explicit security policy; can be checked during runtime using machine-code rewriting to dynamically ensure that control flow adheres to a given CFG

CFG Example

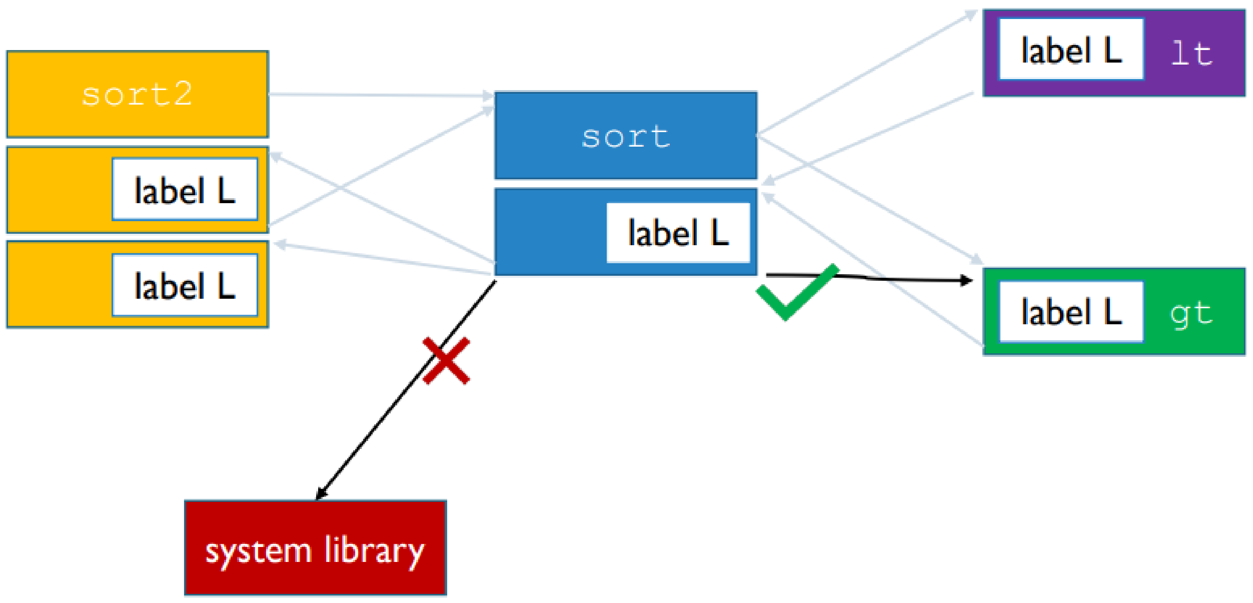


```
sort2(int a[], int b[], int len){
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y){
  return x < y;
}
bool gt(int x, int y){
  return x > y;
}
```

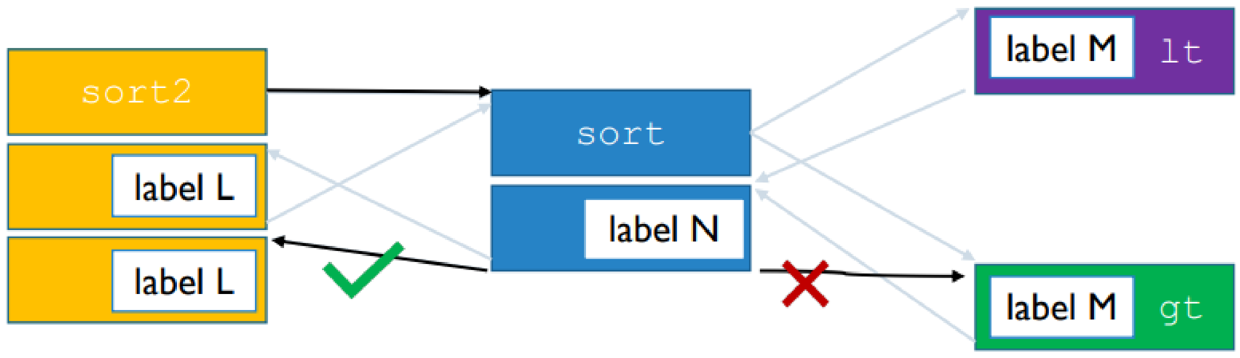
Break into basic blocks
Distinguish calls from return

- Direct calls need not be monitored since target address cannot be changed (assuming code is immutable) $\longrightarrow$ only need to monitor indirect calls (e.g. `jmp`, `call`, `ret` via registers)
- Forward-edge CFI is used for indirect calls and jump instructions
- Backward-edge CFI is used for return instructions
    - Shadow stack for - separate isolated stack for storing return address
    - Enforces natural semantics of call / return - return is supposed to transfer the control flow to the next instruction after the calling instruction that invoked the current function
- During execution each machine instruction that transfers control must target a constant destination determined by the CFG
    - Instructions that target a constant destination can be checked statically
    - At each destination an ID is inserted which identifies an equivalence class of destination which is validated with an ID-check inserted before each source



Equivalence class 1: use the same label at all targets

- Attacker who does not know the IDs has to be lucky when deviating from the CFG to reach a destination with a valid ID. While an attacker who knows the IDs may try to construct an attack that avoids detection



Equivalence class 2:

- Return sites from calls to `sort` must share a label (L)
- Call targets `gt` and `lt` must share a label (M)
- Remaining label unconstrainted (N)

Still permits call from site A to return to site B

- **Backward-edge CFI & Exceptions**
    - Call / Return semantic can be broken in some benign cases (exceptions does not need to be caught by the calling function and can be caught by another function in the call hierarchy / exception handler)
        - Called function (current function) returns to the function in the call stack that implements the exception handler (not the instruction that called the current function)

3 Type Confusion

- **Static Type Checking** checks all possible executions of a program to see whether a type violation could occur
- **Dynamic Type Checking** checks the class tag when access is requested
- Type confusion attack exploits a vulnerability in type checking procedure to create two pointers to the same object with incompatible type tags.
  - Attack code passes static type checking but actual access is to object of the wrong type
- In Java, all objects has a class and is labelled with a class tag in memory; when being referenced the pointer referencing the object can be thought of as tagged with a label that says what kind of object the pointer is pointing to.

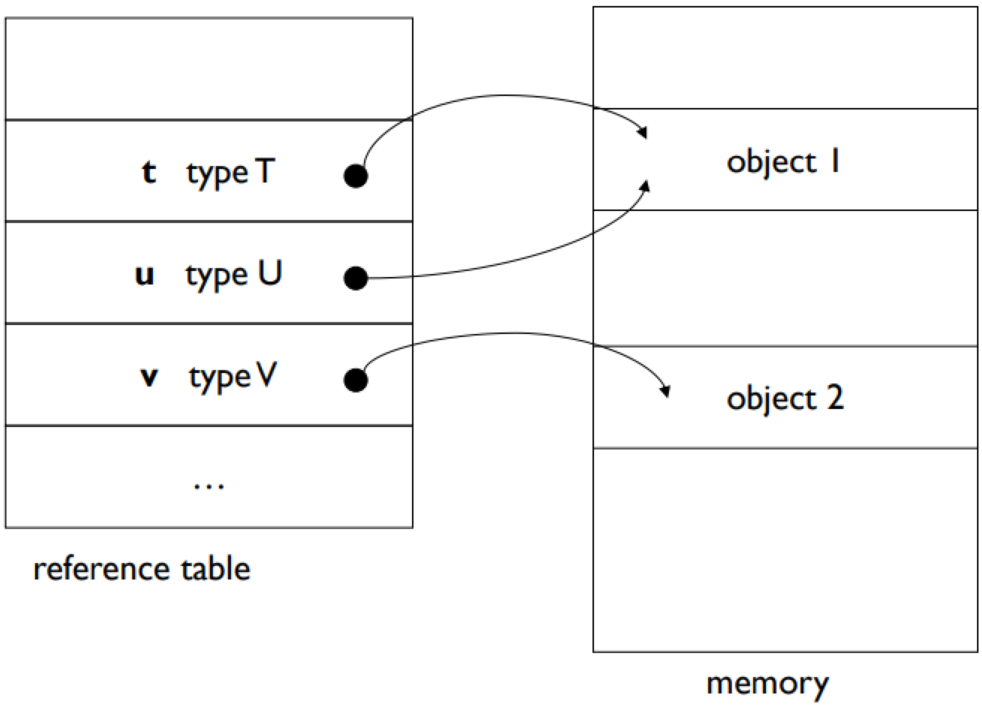
```
class T {
   SecurityManager x;
}

class U {
   MyObject x;
}
```

class definitions

```
T t = the pointer tagged T;
U u = the pointer tagged U;
t.x = System.getSecurity();
MyObject m = u.x;
```

malign applet



  - The SecurityManager field can then be manipulated from MyObject

Netscape vulnerability

- In Java, a program that uses type `T` can also use type `array of T` with array names beginning with `[` which cannot be used as a start of a classname.
- Java **bytecode file** (executable) could declare its own name to be a special array types name, redefining Java's array type. Loading this file would generate error but Java VM would install the name in internal table anyway

SUN Security Bulletin

- Caused by a flaw in the checking of type casts
- Applets could exploit flaw to increase privileges and get out of sandbox; if user loads such applets, attackers can execute arbitrary code on local machine with user's privileges.

Summary

- Type confusion attacks are an example for attacks below the level that performs access control and often result in complete system penetration.
- Not easy to spot and not easy to prove that a system is type safe even during first round of design.