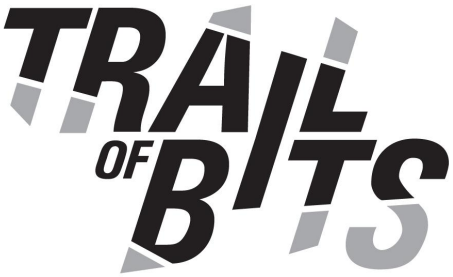


PointsTo

Static Use-After-Free Detector for C/C++

Presented by: Peter Goodman

PointsTo detects use-after-free bugs

- Static, whole-program analyzer that finds use-after-free bugs in C/C++ code

- Implemented in C++ as an LLVM plugin that analyzes LLVM bitcode

- Scales to large programs
  - Starts producing reports on Mozilla Spidermonkey after ~2 days

We use PointsTo, you should too

- Use-after-free bugs are subtle, don't always lead to crashes
  - Corruption: mutating new object that occupies the same memory as free'd object
  - Unintended control-flow: invoking virtual method of an unexpected class
- C++ exacerbates these problems
  - Type confusion: call virtual method of a freed object recently reallocated
  - Hidden stuff: operator overloading

But.. But… Smart pointers?!


I'M NOT SAYING IT'S LEGACY CODE
BUT IT'S LEGACY CODE
HISTORY.COM
memegenerator.net

Not all C++ code is smart

- Consistent, correct, and comprehensive use of smart pointers can eliminate UAFs
  - Most code doesn’t use smart pointers
- Often stuck with legacy codebases using plain old pointers

Using PointsTo is easy

- Compile your code to bitcode with Clang
  - Use whole-program-llvm to make this easier!
  - `./env.sh make all`
- Run PointsTo on your program's bitcode
  - `./run.sh program.bc`
- Inspect reports in your IDE
  - One report per warning (path from a `free` to a deref)
  - `file:line function symbol`

Three stages of PointsTo

- Inlines functions to get **context-sensitivity**
  - Generic, improves accuracy
- Then does a **flow-insensitive** (Anderson style) points-to analysis
  - Conservative, easy to scale
  - Doesn’t have a notion of “after”
- Uses flow-insensitive analysis results to improve accuracy of a **flow-sensitive** analysis
  - Precise, hard to scale

Context-sensitive analysis

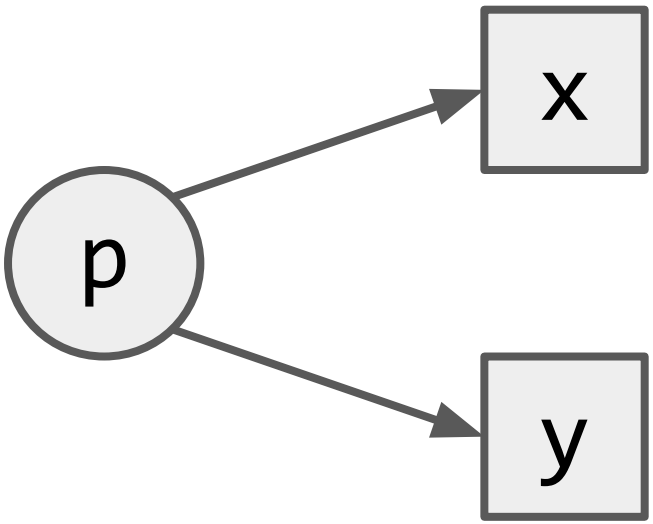
- Want to analyze each function with respect to its caller
  - Eliminates some code paths, e.g. constant propagation
  - Elides some pointer operations, e.g. address of a local

- **Key insight**: inline a function into its caller
  - More aggressive inlining = more context sensitivity

Flow-insensitive points-to analysis (1)

- Ignores control-flow

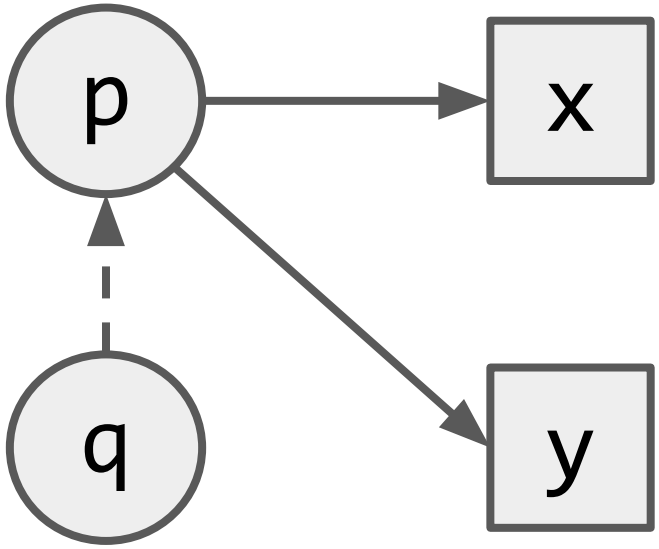
```
int x;
int y;
int *p = &x;
…
p = &y;
```



Flow-insensitive points-to analysis (2)

- Assignments imply subset inclusion: $q \supseteq p$

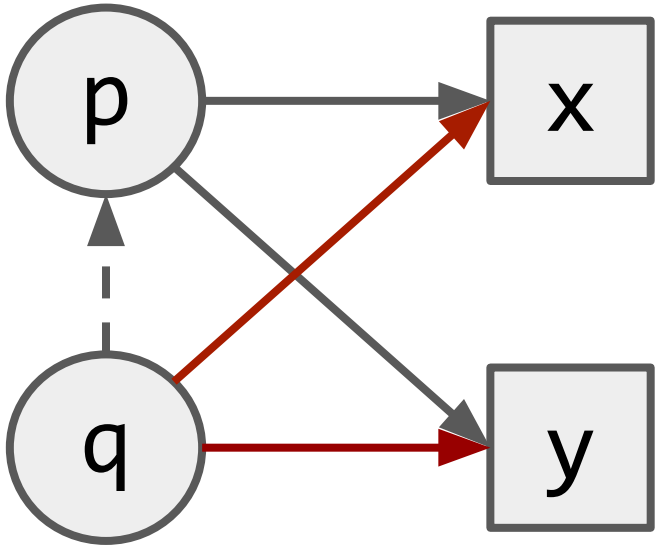
```
int x;
int y;
int *p = &x;
int *q = p;
p = &y;
```



Flow-insensitive points-to analysis (3)

- Iterate until we reach a fixed point

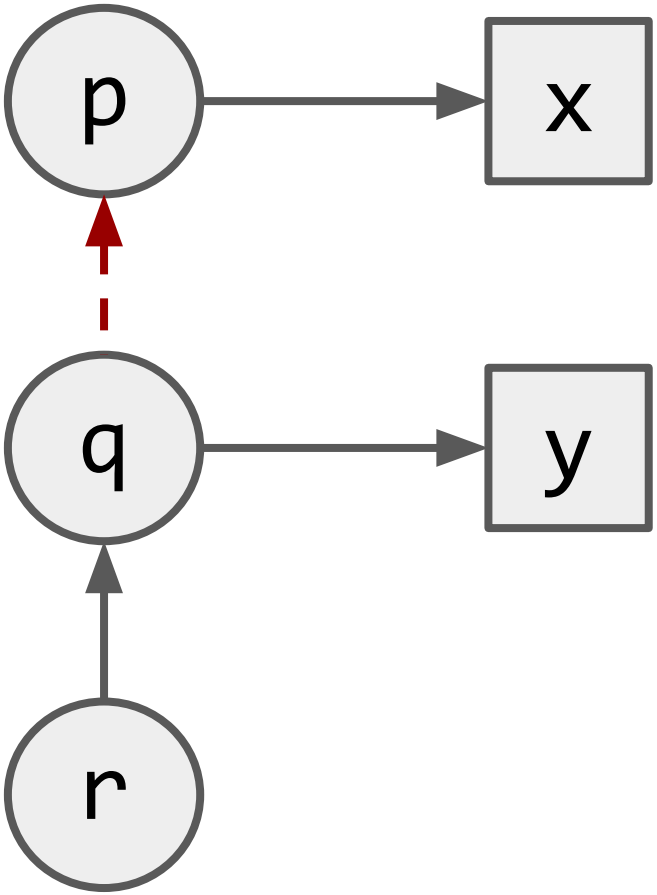
```
int x;
int y;
int *p = &x;
int *q = p;
p = &y;
```



Flow-insensitive points-to analysis (4)

- Dereferences introduce new inclusions

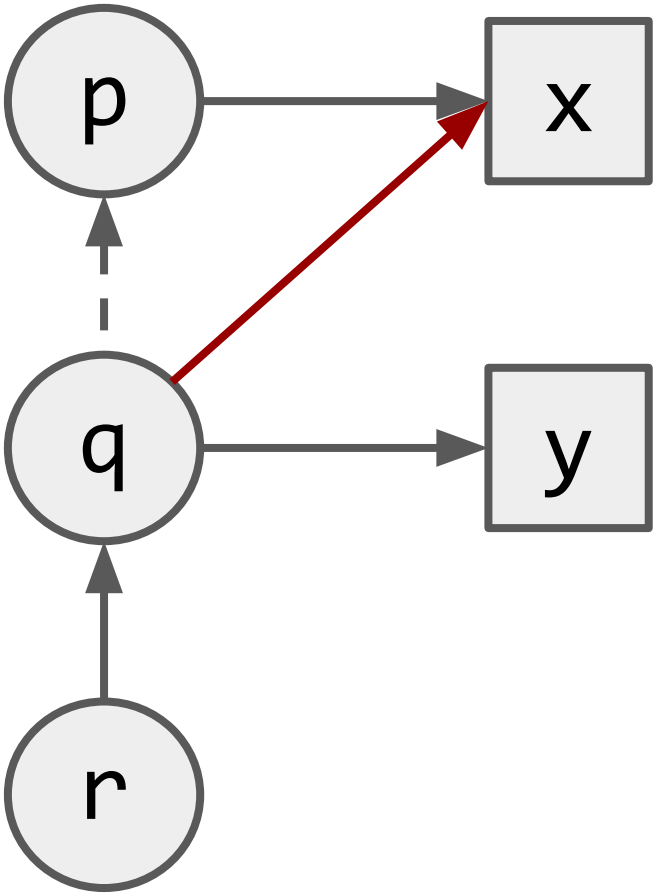
```
int x;
int y;
int *p = &x;
int *q = &y;
int **r = &q;
*r = p;
```



Flow-insensitive points-to analysis (5)

- Dereferences introduce new inclusions

```
int x;
int y;
int *p = &x;
int *q = &y;
int **r = &q;
*r = p;
```

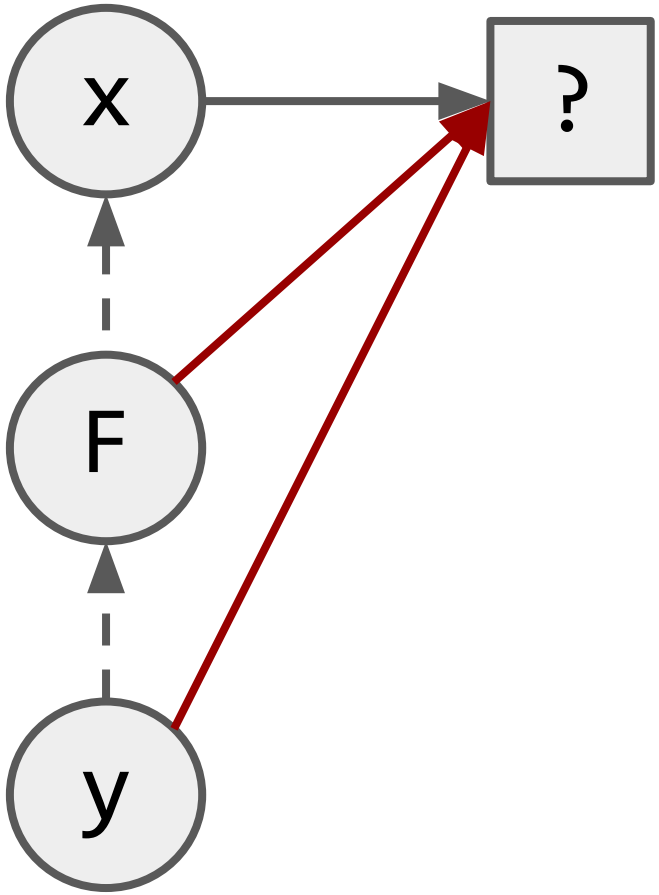


Flow-insensitive points-to analysis (6)

- Functions are treated like variables

```
int *F() {
   ...
   return x;
}

int *y = F();
```

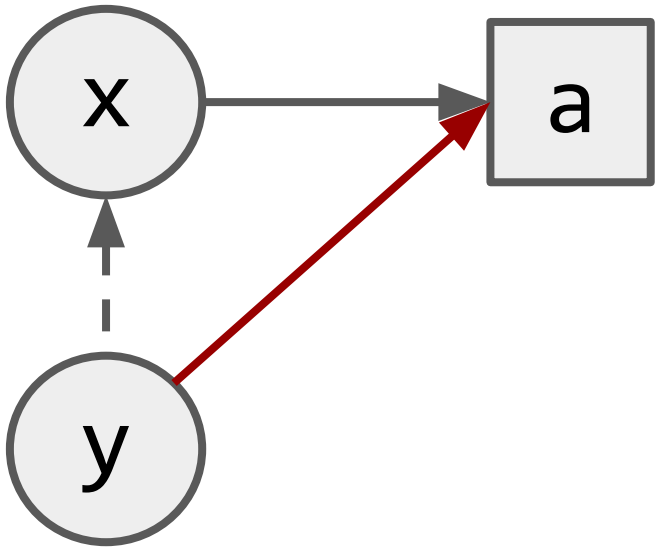


Flow-insensitive points-to analysis (7)

- Arguments are similarly treated

```
int F(int *y){
   ...
}

int a;
int *x = &a;
F(x);
```



Flow-insensitive points-to analysis (8)

- Each call to `malloc`/`new` is treated as an object
  - Imagine every call to an allocator is a unique local variable that is a pointer

- As many “objects” as there are calls to allocators

Flow-insensitive points-to analysis (9)

- Flow-insensitive analyses scale but are imprecise
  - Oblivious of `frees`/`deletes`
  - No paths = less helpful for diagnosing a bug

- Flow-insensitive analyses are conservative
  - Over-approximate points-to sets
  - Good for compilers where correctness is required
  - Useful for building call graphs with function pointers

Flow-sensitive points-to analysis (1)

- Create a big (SSA-like) use-def graph
  - Uses: pointer dereference, `free`/`delete`
  - Defs: allocator calls, address-of, PHI, assignments

- Similar to tracking the set of reaching definitions at every program point
  - A use (dereference) points-to the set of all defs (allocations) that are reachable in the graph
  - Use a binary decision diagram to efficiently represent points-to sets

Flow-sensitive points-to analysis (2)

- Example use-after-free

```
A *a = new A;
if (...) {
  delete a;
}
a->x = 10;
```

Flow-sensitive points-to analysis (3)

- Convert frees into new definitions

```
A *a = new A;
if (...) {
  delete a;
  a = ERROR;
}
a->x = 10;
```

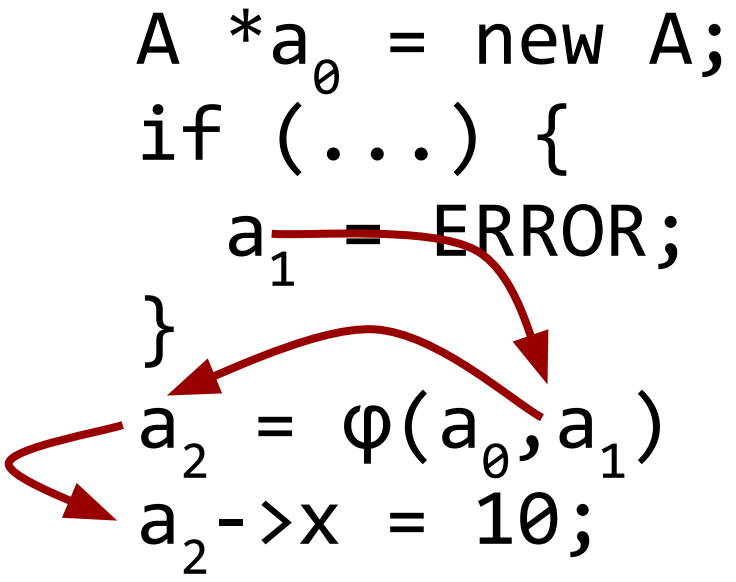
Flow-sensitive points-to analysis (4)

- Convert to SSA

```
A *a0 = new A;
if (...) {
  a1 = ERROR;
}
a2 = φ(a0,a1)
a2->x = 10;
```

Flow-sensitive points-to analysis (5)

- Find a path from an error definition to a use



Flow-sensitive points-to analysis (6)

- More precise than the flow-insensitive analysis
  - Tells us accurate points-to information at every program point
- Hard to scale
  - Must propagate information through the def-use graph
    - Iterate until a fixed-point is reached
  - Graph is proportional to program size
  - Graph changes over time as we discover more information about function pointer targets :-(

Can we stop the graph from changing while we analyze?

Combining analyses

- Problem: Need call graph for flow-sensitive analysis
  - Function pointers complicate things
- Solution: Use points-to information from flow-insensitive analysis to build call graph
- Result: Slightly less-precise analysis, but more scalable
  - We're prepared to accept false positives

Summary

- Different types of analyses have different trade-offs
  - Flow insensitive: scalable but imprecise
  - Flow sensitive: hard to scale but precise
- Combining analyses is feasible and useful
- Finding a use-after-free necessarily requires flow sensitivity: hint “after”