CS165 – Computer Security

Static Analysis (Testing)

November 6, 2024

Our Goal

- We want to develop techniques to detect vulnerabilities automatically before they are exploited
  - What's a vulnerability?
  - How to find them?

Without Running the Program

- Can we find vulnerabilities without running the program?
  - *Why would that be beneficial?*

Limitations of Dynamic Testing

- Time consuming
  - Have to run the program many, many times
- Some configuration
  - Need to choose the inputs to initiate mutation
  - Some setup
- Under-approximate
  - While a crash/hang found is real,…
  - Cannot test all paths and inputs in practice
    - I.e., dynamic testing is intractable

Static Analysis

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

Forms of Testing

- Static analysis is an alternative to dynamic testing
- Dynamic
  - Select concrete inputs
  - Find results of the execution given those inputs
  - Apply many concrete inputs (i.e., run many tests)
- Static
  - Select abstract inputs (set of inputs)
  - Find impact created by executing all abstract inputs
  - One “run”

Static Analysis

- Provides an approximation of program behavior
- "Run in the aggregate"
  - Rather than executing concretely one run at a time
  - Finite-sized descriptors representing a collection of values
- "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 Vulnerability Tools

- Quite a few commercial tools for vulnerability detection – for different languages, types of flaws, etc.
  - Checkmarx
  - SonarQube
  - Veracode
  - Snyk
  - Acunetix
  - Fortify
  - Coverity
- And compiler tools (LLVM Static Checker) and research tools galore

Static Analysis Example

- Can we find a use-after-free flaw with static analysis?

```
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```

Static Analysis Example

- Pointers have 2 values: (malloc, free)

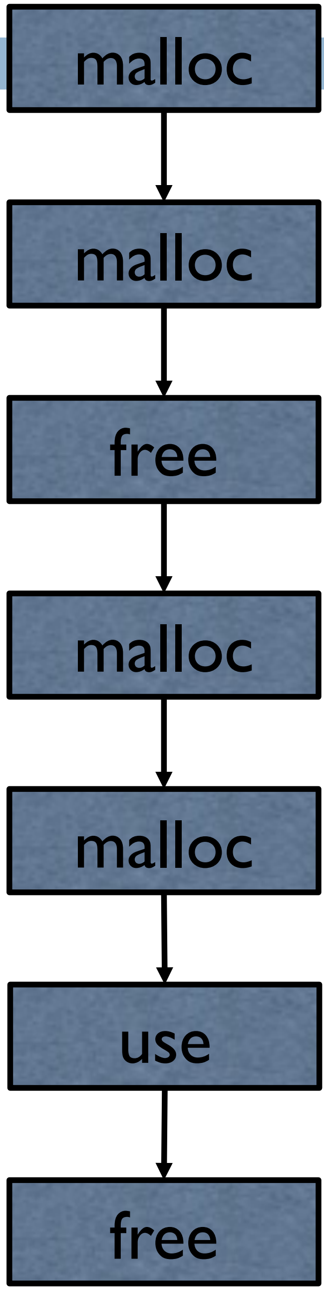
```
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```



Static Analysis Example

- Pointers have 3 ops: (malloc, use, free)

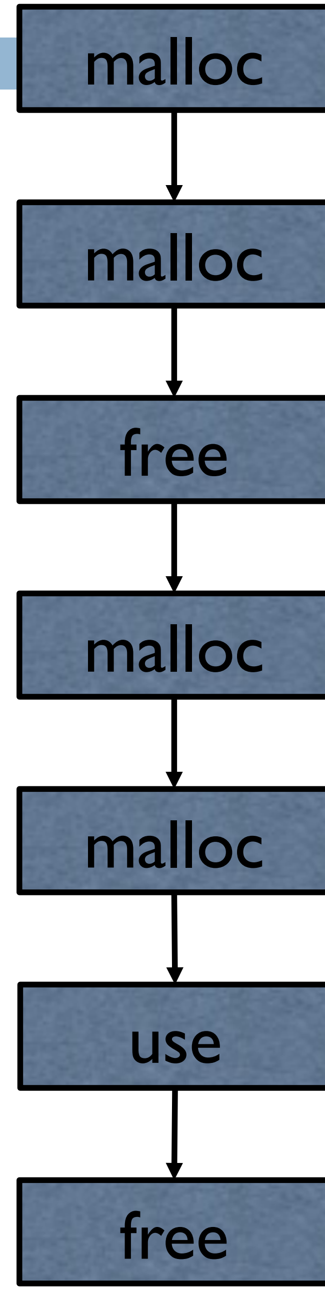
```
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```



Static Analysis Example

- A free pointer cannot have a use op

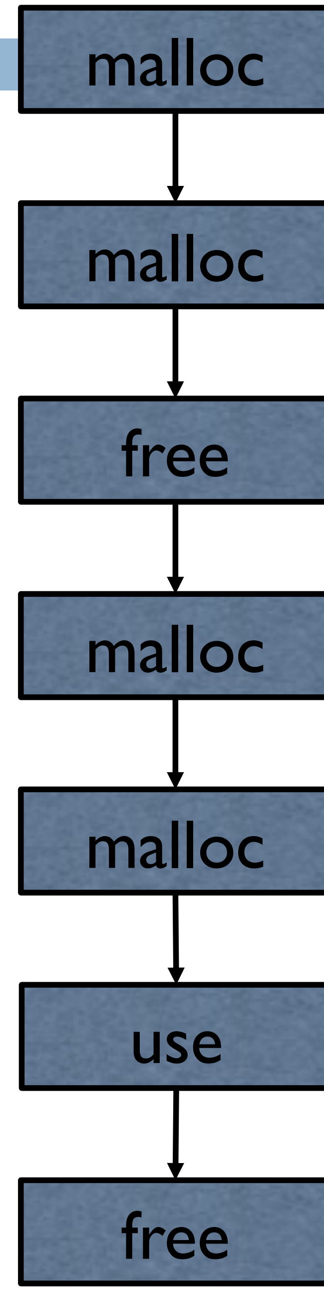
```
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```



Static Analysis Example

- A free pointer cannot have a use op

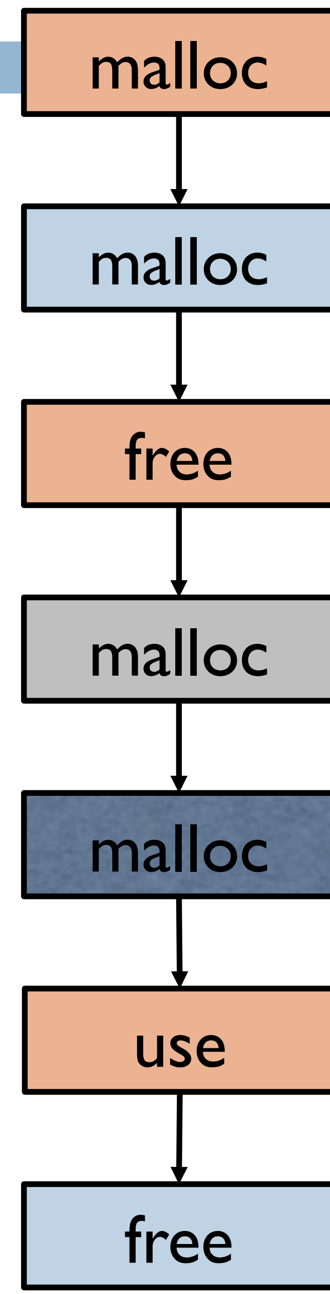
```
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```



Static Analysis Example Summary

- Provides an approximation of program behavior
- Approximate values based on analysis goals
  - We only care about whether pointers are assigned to objects (malloc) or not (free) – not specific pointer addresses
- Consider operations in terms of those abstract values
  - Malloc → changes pointer value to "malloc"
  - Free → changes pointer value to "free"
  - Use does not change pointer value
- Then, check all executions against a rule
  - Cannot have a "use" op on a pointer whose value is "free"
- Only one execution path here, but may be many

Static Analysis

- There, you completed your first static analysis!

What Do We Want to Find?

- Often whether one value affects another (taint)
- Secrecy – Does a secret value *x* affect a public value *y*
  - Can we leak *x*?
- Integrity – Does an adversary-controlled value *x* affect a critical (i.e., high integrity) value *y*
  - Can an adversary attack *y?*
- Problem statement:
  - Given an *x* we care about…
  - Is there any execution path in the program where the data assigned to *x* may be assigned to *y* (x taints y)

One Type of Leak

- Consider the simple program below, where x is a secret value

```
y = x;
output( y );
```

- If x is a secret value, is its value leaked?

Another Type of Leak

- Consider the simple program below

```
if ( x == 0 )
    b = 1;
else
    b = 2;
output( b );
```

- If x is a secret value, is its value leaked?

Information Flows

- What is going on here?
  - Dorothy Denning captured the essence in 1976

Implicit and Explicit Flows

- Explicit Flow
  - Direct assignment from a to b (e.g., b=a)
- Implicit Flow
  - Indirect assignment where value of b may depend on a indirectly (via a conditional)

Implicit and Explicit Flows

- Explicit Flow
  - Direct assignment from a to b (e.g., b=a)
- Implicit Flow
  - Indirect assignment where value of b may depend on a indirectly (via a conditional)
- Compute these flows and determine whether they leak a value
  - Could be a combination of explicit and implicit flows

Information Flow Secuity

- So, what does "secure" mean in information flow?
  - Cannot create an "information flow" from secret to public
  - Explicit flow: No assignment "x=y" where x is public and y is secret
  - Implicit flow: No conditional based on secret data that impacts the value of a public variable

Flow Checking

- Program is secure iff:
  - Explicit flow from *S* is secure
  - Explicit flow from all statements in a sequence are secure (e.g., *S1, …, Sm*)
  - Conditional *c: B1, …., Bn* is secure if:
    - The explicit flows of all branches: *Bi: S1, …, Sm* are secure
    - The implicit flows between *c* and *B1, …, Bn* are secure
- These are all forms of taint flows

Static Analysis Approach

- Possible values of a variable: secret, public, none
  - We don't care what the specific value is
- Operations
  - Explicit flow (`b = a`) or Implicit flow if (`if(a) b = any`)
    - If *b*'s value is none, then assign *b* to the value of *a*
    - If *b*'s value is public and *a*'s value is public or none, leave *b*'s value unchanged as public
    - If *b*'s value is public and *a*'s value is secret, report an error
- Do this for the entire program
  - All execution paths, although can optimize

Example

```
void fn( char *buf, int len )
{
    char public[SIZE];
    char secret[SIZE];

    copy( public, “MSG”, SIZE);
    if ( len < SIZE )
        copy( secret, buf, len );
    else
        send( public );     // a public sink
}
```

Example

```
void fn( char *buf, int len )
{
    char public[SIZE];
    char secret[SIZE];

    copy( public, "MSG", SIZE);
    if ( len < SIZE )
        copy( secret, buf, len );
    else
        send( public );     // a public sink
}
```

secret

public

Flow of Public

```
void fn( char *buf, int len )
{
    char public[SIZE];
    char secret[SIZE];

    copy( public, “MSG”, SIZE);
    if ( len < SIZE )
        copy( secret, buf, len );
    else
        send( public );      // a public sink
}
```

secret

public

Flow of Secret

```
void fn( char *buf, int len )
{
    char public[SIZE];
    char secret[SIZE];

    copy( public, "MSG", SIZE);
    if ( len < SIZE )
        copy( secret, buf, len );
    else
        send( public );     // error here
}
```

secret

public

Is Static Analysis a Miracle?

- Limitation: If we try to find all vulnerabilities via static analysis (i.e., over-approximate the program operations), then there will likely be false positives
  - False positive: violate an analysis rule in the approximation, but not in the real program execution
- Is every explicit/implicit flow a leak?
  - Should we be able to send an error message?
  - Also, more subtle cases due to the complexity of programs

What Can We Do with Static Analysis?

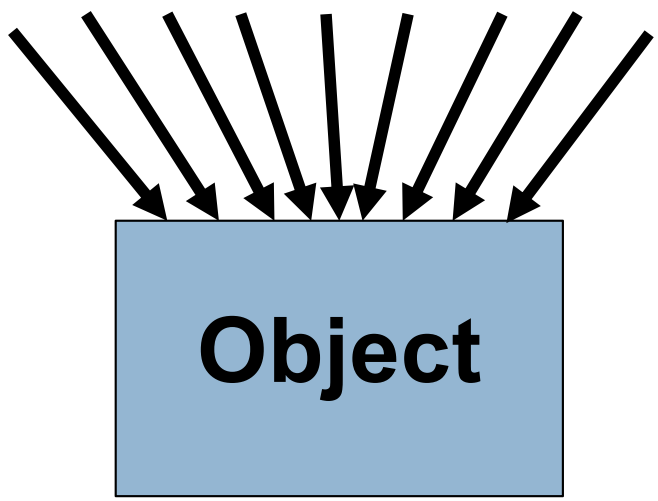
- Used to detect many types of vulnerabilities
- And show the absence of vulnerabilities in some cases

Memory Safety Validation

- Identify the objects whose accesses must all satisfy spatial, type, and temporal safety (all classes of memory safety)
  - Why care?
  - Protect those objects from accesses that may cause memory errors
  - If we identify these statically, we can protect without runtime checks

Memory Safety Validation

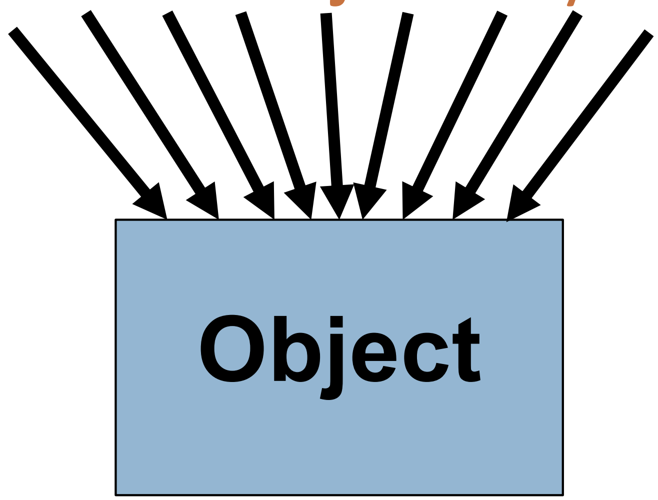
- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object



Memory Safety Validation

- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object

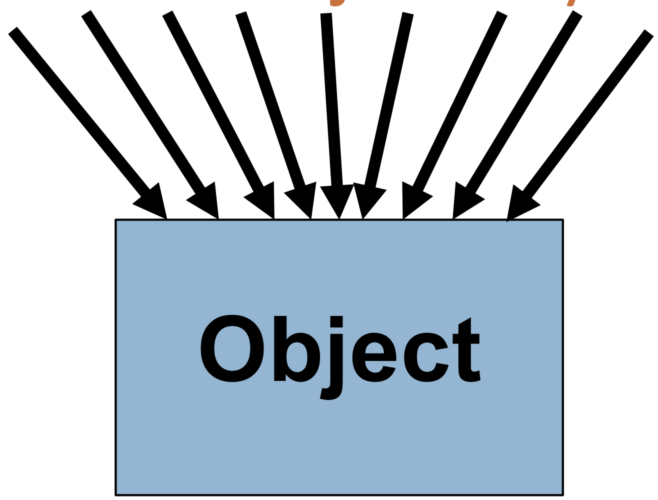
Suppose all aliases of **Object** only make safe accesses



Memory Safety Validation

- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object

Suppose all aliases of **Object** only make safe accesses

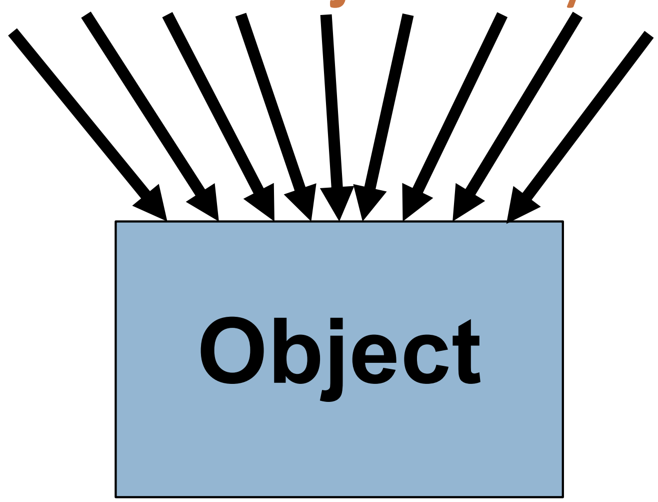


Is **Object** safe from exploitation via memory errors?

Memory Safety Validation

- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object

Suppose all aliases of **Object** only make safe accesses

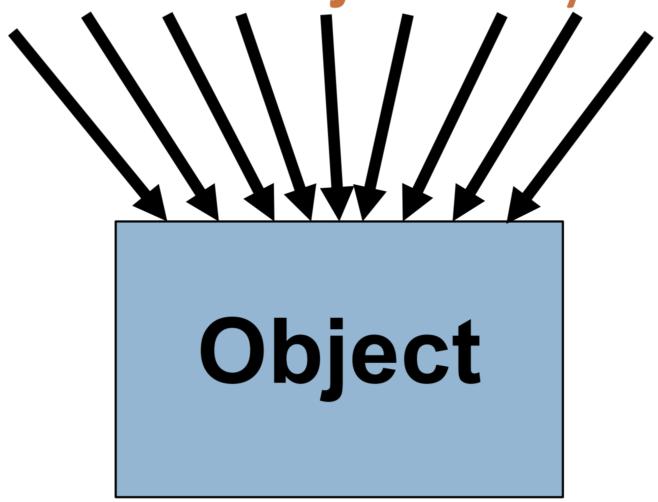


Is **Object** safe from exploitation via memory errors? No

Memory Safety Validation

- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object

Suppose all aliases of **Object** only make safe accesses

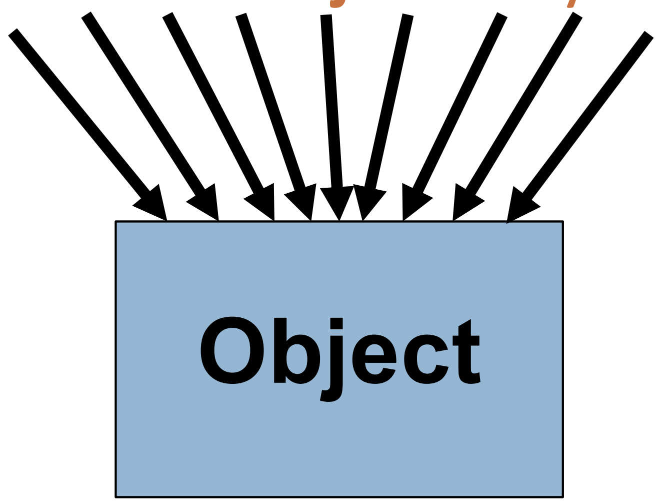


Suppose **Object** is isolated from all accesses to unsafe objects: Is **Object** safe from exploitation?

Memory Safety Validation

- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object

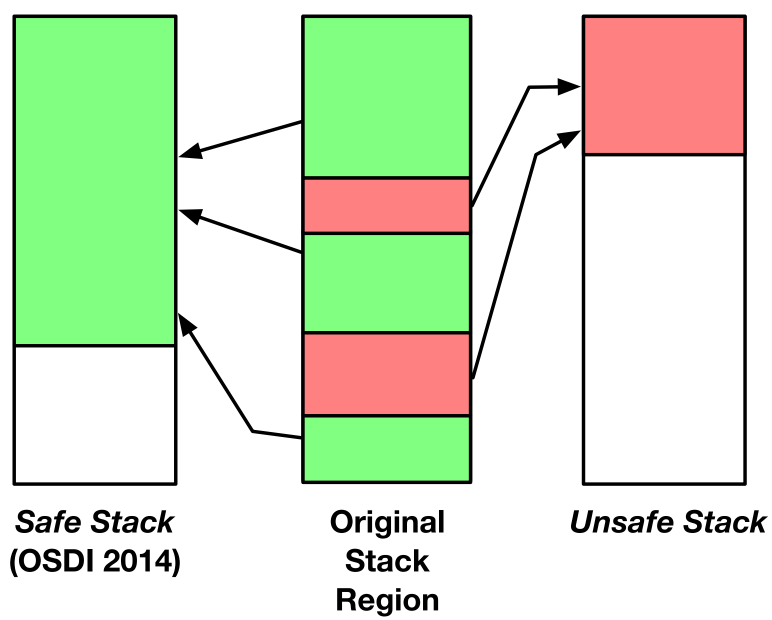
Suppose all aliases of **Object** only make safe accesses



Suppose **Object** is isolated from accesses to unsafe objects: Is **Object** safe from exploitation? YES

Isolate Safe Stack Objects

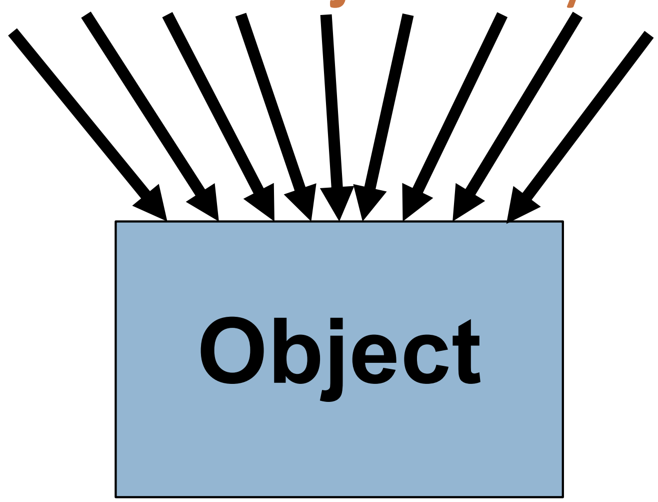
- Compute all possible aliases of an object, and if all uses satisfy spatial, type, and temporal safety
  - Isolate those objects from accesses to unsafe objects



Memory Safety Validation

- Objects may have many aliases
  - Alias: pointer that may be assigned (defined) to the object

Suppose all aliases of **Object** only make safe accesses



So, it is worth determining which objects must be safe from all memory errors and isolating them from unsafe

Impact of Memory Safety Validation

	CCured-default	*CCured-min*	*Safe Stack-default*	*Safe Stack-min*	*DataGuard*	*Total*
nginx	14,573 (79.52%)	14,496 (79.10%)	13,047 (71.20%)	12,375 (67.53%)	16,684 (91.05%)	18,324
httpd	61,915 (73.06%)	60,526 (71.42%)	49,523 (58.44%)	46,833 (55.27%)	78,266 (92.36%)	84,741
proftpd	14,521 (81.66%)	14,189 (79.79%)	12,837 (72.19%)	12.513 (70.37%)	16,190 (91.04%)	17,782
openvpn	48,379 (76.58%)	47,662 (75.45%)	40,627 (64.31%)	39,145 (61.97%)	57,693 (91.33%)	63,171
opensshd	20,238 (79.45%)	20,062 (78.75%)	18,176 (71.35%)	17,712 (69.53%)	23,871 (93.71%)	25,474
perlbench	52,738 (91.61%)	51,165 (88.57%)	42,398 (73.65%)	42,014 (72.98%)	52,324 (90.89%)	57,567
bzip2	1,293 (92.29%)	1,162 (82.94%)	1,057 (75.44%)	1,049 (74.87%)	1,238 (88.39%)	1,401
gcc	123,427 (73.34%)	120,856 (71.82%)	96,796 (57.52%)	91,344 (54.28%)	152,452 (90.59%)	168,283
mcf	580 (90.34%)	569 (88.63%)	441 (68.69%)	436 (67.91%)	602 (93.77%)	642
gobmk	34,376 (85.53%)	33,969 (84.52%)	26,229 (65.26%)	26,013 (64.72%)	38,552 (95.92%)	40,191
hmmer	20,133 (75.84%)	19,874 (74.87%)	13,873 (52.26%)	13,629 (51.34%)	25,674 (96.71%)	26,546
sjeng	3,461 (85.62%)	3,415 (84.49%)	2,798 (69.22%)	2,712 (67.10%)	3,741 (92.55%)	4,042
libquantum	2,576 (66.80%)	2,521 (65.38%)	2,036 (52.80%)	1,878 (48.70%)	3,214 (83.35%)	3,856
h264ref	19,525 (87.70%)	19,283 (86.61%)	14,418 (64.76%)	14,339 (64.40%)	20,177 (90.63%)	22,264
lbm	448 (82.96%)	442 (81.85%)	376 (69.63%)	369 (68.33%)	506 (93.70%)	540
sphinx3	2,744 (72.90%)	2,713 (72.10%)	2,058 (54.67%)	1,962 (52.13%)	3,398 (90.28%)	3,764
milc	4,325 (81.50%)	4,233 (79.76%)	3,887 (73.24%)	3,794 (71.49%)	4,680 (88.19%)	5,307
omnetpp	20,572 (83.44%)	20,264 (82.19%)	16,967 (68.82%)	16,283 (66.04%)	22,091 (89.60%)	24,655
soplex	14,253 (72.80%)	14,072 (71.87%)	11,044 (56.41%)	9,513 (50.12%)	16,368 (83.60%)	19,579
namd	21,676 (85.17%)	21,352 (83.90%)	18,389 (72.26%)	18,213 (78.34%)	23,249 (91.36%)	25,448
astar	4,016 (87.36%)	3,977 (86.51%)	3,606 (78.44%)	3,524 (76.66%)	4,206 (91.49%)	4,597

- 91.45% of stack objects are shown to be safe w.r.t. spatial, type, and temporal errors.
- 50% and 70% unsafe stack objects to CCured and Safe Stack are found safe by DataGuard.
- 3% and 6.3% safe stack objects by CCured and Safe Stack are not provably safe in DataGuard

Conclusions

- Memory safety validation is a typical static analysis
  - Assign values to pointers (e.g., in bounds)
  - Find operations that change those values
  - Detect violations of analysis rules
- Static analysis is a common technique to detect vulnerabilities and prove their absence
- Can find all vulnerabilities in a program, although may have false positives

Questions