Static Analysis

… because we can …
… because it helps …
… and it's easy to do …

Announcement

- L3: Static Analysis Lab Thursday
- Needs prep – 2 tool installations for each student (may require troubleshooting)
- See Canvas

Motivation

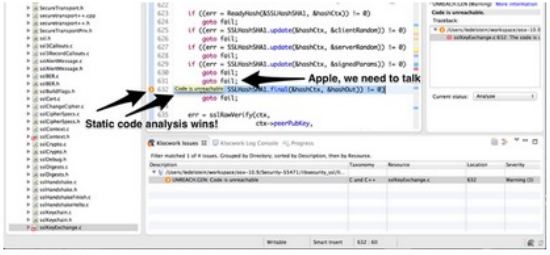
```
goto fail;
```

Klocwork: Our source code analyzer caught Apple's 'gotofail' bug

If Apple had used a third-party source code analyzer on its encryption library, it could have avoided the "gotofail" bug.

by Declan McCullagh @declanm / February 28, 2014 1:13 PM PST

25 / 0 / 2 / 0 / / more +



Klocwork's Larry Edelstein sent us this screen snapshot, complete with the arrows, showing how the company's product would have nabbed the "goto fail" bug.

Klocwork

It was a single repeated line of code -- "goto fail" -- that left millions of Apple users vulnerable to Internet attacks until the company finally fixed it Tuesday.

That OS X security vulnerability, which also affected iOS users, arose out of Apple's custom implementation of a security standard known as SSL/TLS. By including the

THIS WEEK'S MUST READS /

1. Klocwork: Our source code analyzer caught Apple's 'gotofail' bug
   Security
2. Google's Android Wear software will let you leave your phone at home (if the Wi-Fi)
   Mobile
3. In Adobe's new Lightroom, multiple photos can now m into one
   Photography
4. Yahoo's new search pact v Microsoft includes opt-out clause
   Internet
5. Microsoft rolling out 34 unscheduled patches for Windows today
   Operating Systems

http://www.cnet.com/news/klocwork-our-source-code-analyzer-caught-apples-gotofail-bug/

Is there a bug here?


```
1. static OSStatus
2. SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
3.                 SSLBuffer signedParams,
4.                  uint8_t *signature,
5.                 UInt16 signatureLen) {
6.       OSStatus err;
7.       …
8.       if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9.           goto fail;
10.      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11.          goto fail;
12.          goto fail;
13.      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14.          goto shutdown;
15.      …
16.      fail:
17.          SSLFreeBuffer(&signedHashes);
18.          SSLFreeBuffer(&hashCtx);
19.          return err;
20.      shutdown:
21.          PrematureShutdown(); return err;
20. }
```

Another example

```
1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh, int b_size) {
4.    struct buffer_head *bh;
5.    unsigned long flags;
6.    save_flags(flags); // save interrupt flags
7.    cli(); // disables interrupts
8.    if ((bh = sh->buffer_pool) == NULL)
9.       return NULL;
10.   sh->buffer_pool = bh -> b_next;
11.   bh->b_size = b_size;
12.   restore_flags(flags); // re-enable interrupts
13.   return bh;
14. }
```

D. Engler et al., "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," OSDI 2000

Another example

```
1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh, int b_size) {
4.    struct buffer_head *bh;
5.    unsigned long flags;
6.    save_flags(flags); // save interrupt flags
7.    cli(); // disables interrupts
8.    if ((bh = sh->buffer_pool) == NULL)
9.       return NULL;
10.   sh->buffer_pool = bh -> b_next;
11.   bh->b_size = b_size;
12.   restore_flags(flags); // re-enable interrupts
13.   return bh;
14. }
```

Function returns with interrupts disabled

How often could this occur? What are the consequences?


```
1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh, int b_size) {
4.    struct buffer_head *bh;
5.    unsigned long flags;
6.    save_flags(flags); // save interrupt flags
7.    cli(); // disables interrupts
8.    if ((bh = sh->buffer_pool) == NULL)
9.       return NULL;
10.   sh->buffer_pool = bh -> b_next;
11.   bh->b_size = b_size;
12.   restore_flags(flags); // re-enable interrupts
13.   return bh;
14. }
```

How easy is it to spot these kinds of bugs?

- in a 10 LOC method?
- in a 500 LOC file?
- in a module with 50K LOC?
- in the Linux Kernel?

Lesson: *Some serious faults are hard to find by non-brute-force methods (testing, code review), but they could be found with automated brute-force techniques*

Which brings us to… Static Analysis

- *Static*: without executing
- *Analysis*: studying an artifact or phenomenon by decomposing it into its constituent parts
  - *property of the part → property of the whole*

- What static analysis techniques/tools do you know of?

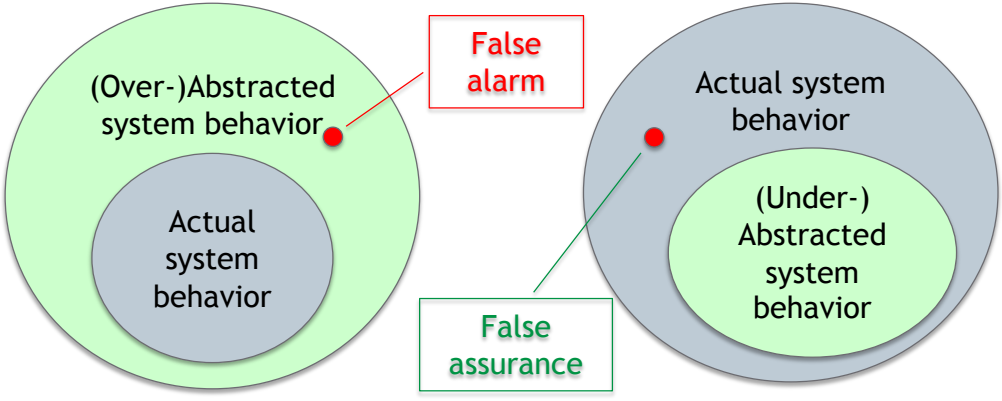
We use models to simplify static analysis

- Create a model of the code (or use an existing one) and verify properties of the model instead of the actual program
  - Models are representations that omit some detail to prove (or disprove) that a program exhibits a particular property

The models we use for the analysis are thus **abstractions** of the program

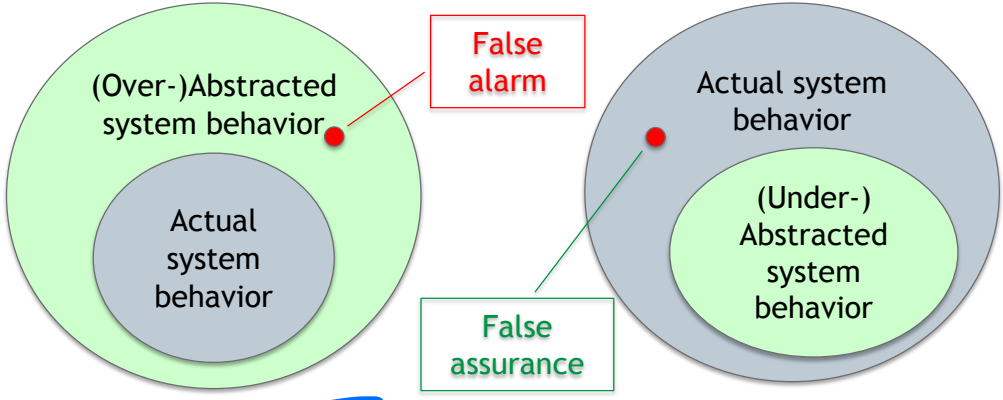
Static analysis often produces false alarms

- Because analysis is often based on a rough approximation (abstraction)
- But can check all of the code -- and totally automatable



Abstraction is an example of the Approximation principle

In theory…



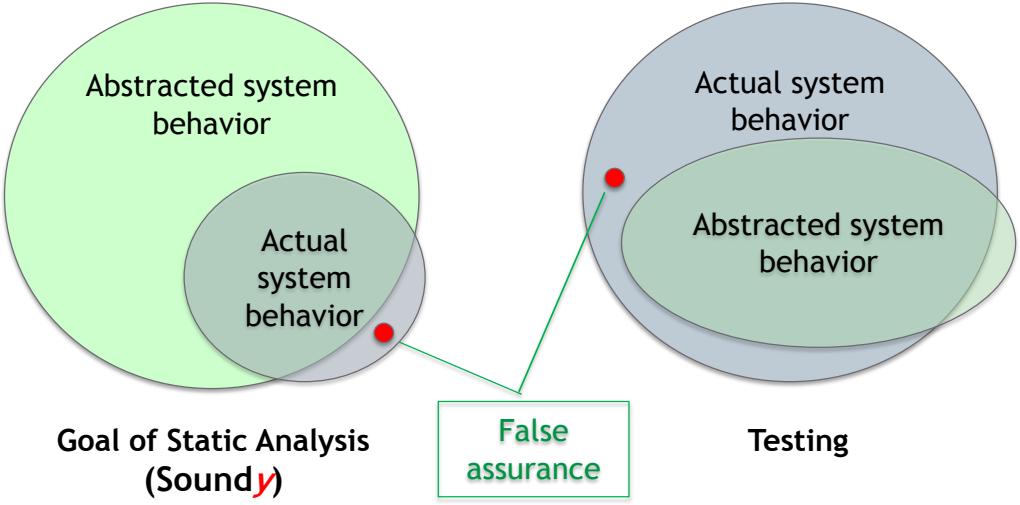
Goal of Static Analysis (Sound)

No false assurances

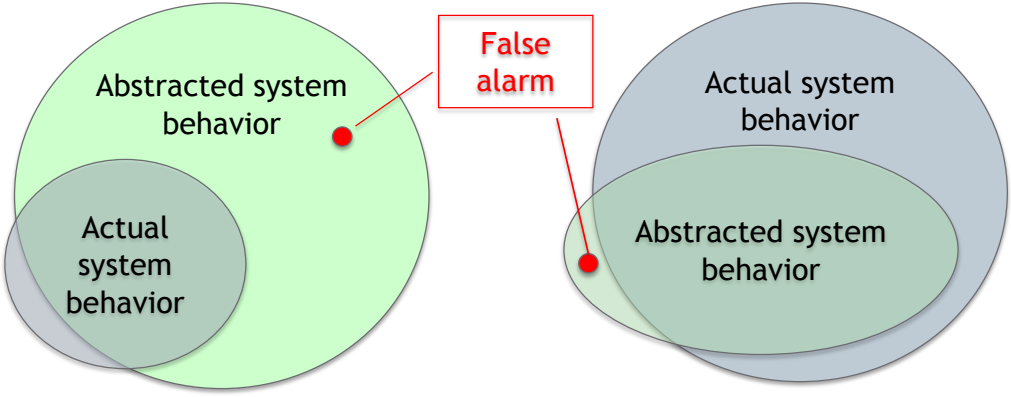
vs. Testing (Complete)

No false alarms

In reality…



In reality…



Goal of Static Analysis (Soundy)

Testing

Static analysis concepts: Results

- **True Positive**
  - Real issue
  - The error condition can occur at runtime, for some program input

  → **Must fix it!**

- **False Positive (False Alarm)**
  - The error condition cannot occur at runtime, no matter what the program input is

  → **Ignore**

- **False Negative (False Assurance)**
  - The error condition can occur at runtime, but the tool does not warn of it

  → **Cannot fix**, since we don't even know about it!

- **True Positive, *but don't care***
  - Why?

  → Decide not to fix it!

Faults that static analysis may catch are numerous!

Faults that may result from violating certain style and design rules

- **Security**: possible buffer overruns, non-validated input
- **Memory** coherence: null dereference, uninitialized data
- **Resource leaks**: access to illegal memory & OS resources, out of bounds errors
- **Violated API protocols**: illegal usage patterns in device drivers, real-time libraries, frameworks, *stateful* services
- **Exceptions**: illegal arithmetic, division by zero
- **Encapsulation violations**: accessing internal/forbidden data, calling forbidden functions (e.g., using reflection)
- **Potential Race conditions**: multiple threads accessing shared resources without synchronization, thread-synchronization violations (e.g., *notify* without *wait* in Java)

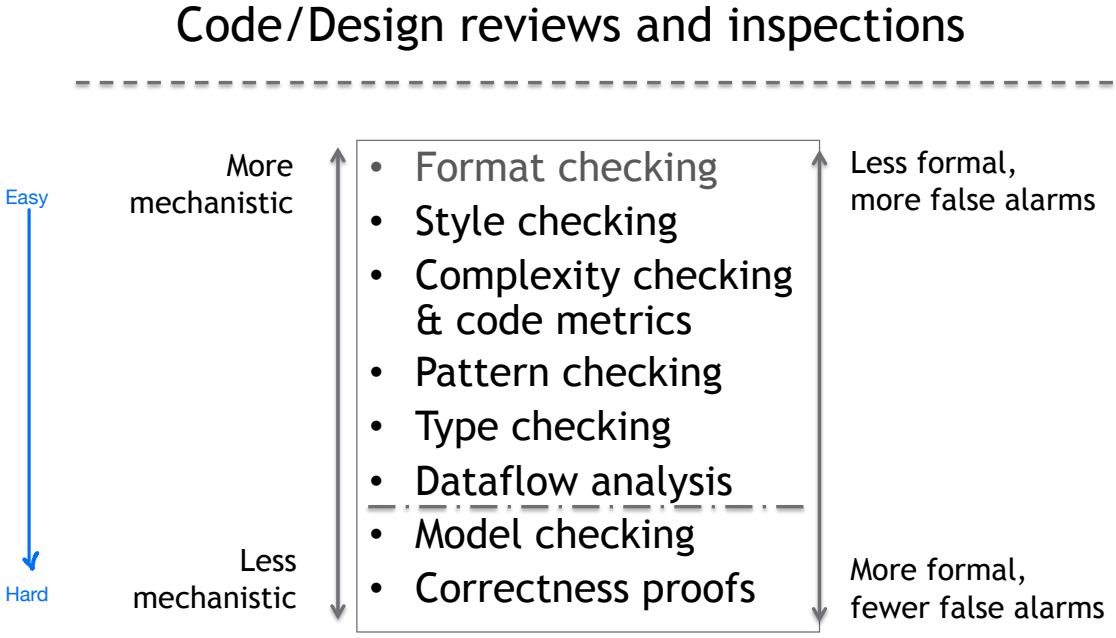
… incidentally, faults that static analysis is good at finding tend to be those that are difficult to find by testing and may cause critical failures!

Key insight: if such faults could result from non-compliance with well-known design & style rules, just check for compliance with them!

Disadvantage:

may have to deal with many false alarms!

There are many of variations of static analysis



Type checking is one of the most common

- Ensure variables are assigned to proper types
- Typically happens at compile time

Compiler is the static analyzer and can do a lot of type checking

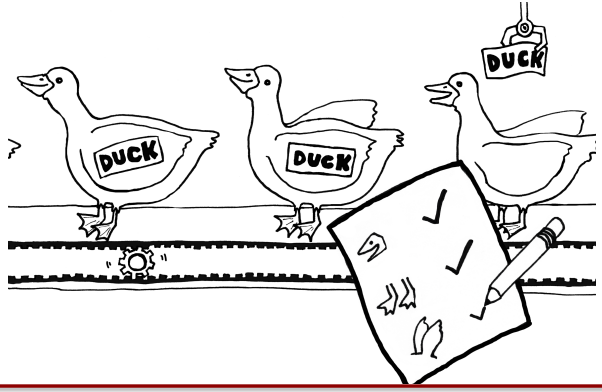
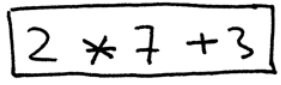


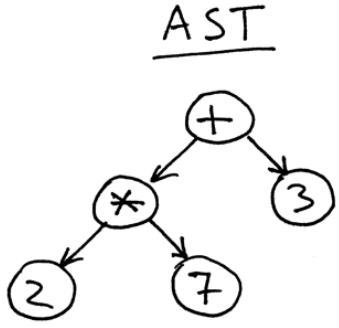
Image: http://www.smashcompany.com/technology/functional-programming-is-not-the-same-as-static-data-type-checking

Static type checking requires the AST

- Based on construction and traversal of **Abstract Syntax Tree**
  - AST: an internal tree representation of a program, used by compilers and static analyzers
  - Simpler than a CFG (CFG is typically cyclic, but AST is not)
  - Purely syntactic

AST is the model (an over-abstraction)



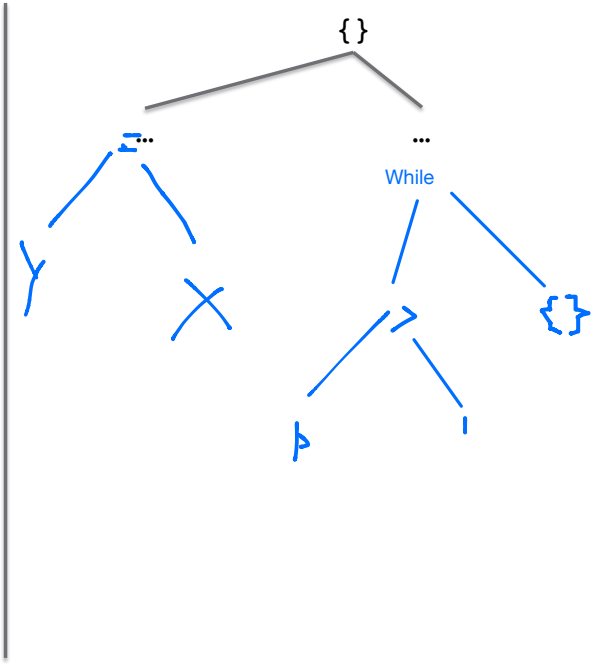


Compilers already produce and need AST!

Image: https://ruslanspivak.com/lsbasi-part7/

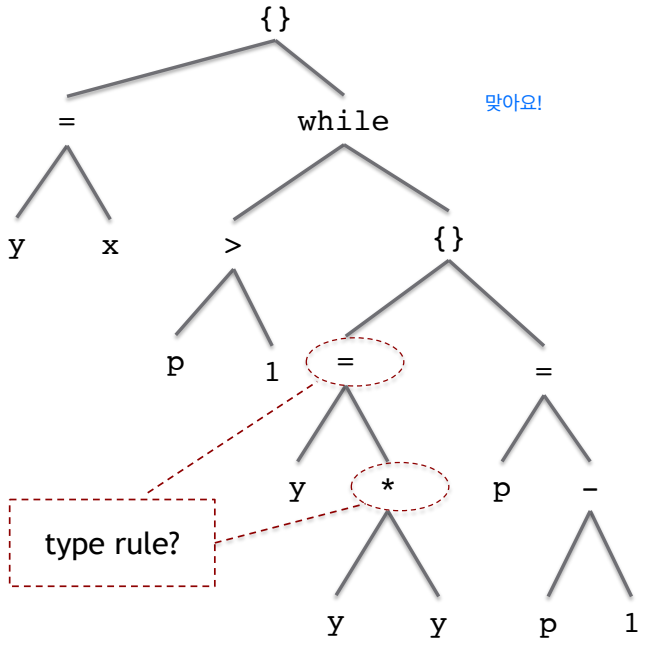
Create the AST of this program (Java) (5 min)

```
{
    y = x;
    while (p > 1) {
        y = y * y;
        p = p - 1;
    }
}
```



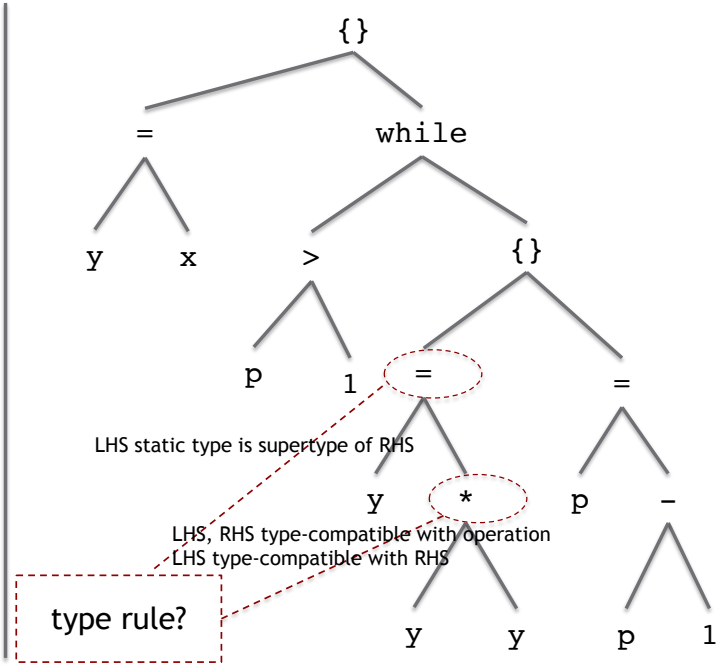
AST Example (Java)

```
{
    y = x;
    while (p > 1) {
        y = y * y;
        p = p - 1;
    }
}
```



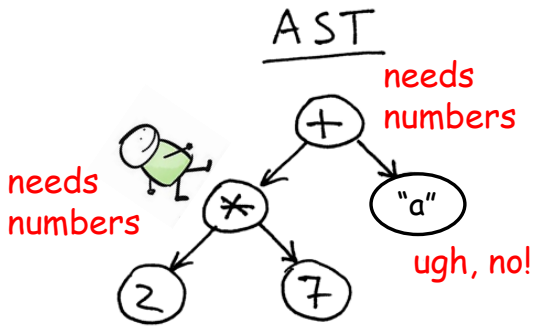
AST Example (Java)

```
{
    y = x;
    while (p > 1) {
        y = y * y;
        p = p - 1;
    }
}
```



Pattern checking generalizes type checking

- Walk through AST, collect information, check for patterns that represent
  - unsafe use
  - violation of design or style rules
  - violation of type rules



Dataflow analysis tracks value flow

Simplest form is a kind of reachability analysis based on identifying ***def-use pairs*** and ***data dependencies*** from the AST or control flow of a program

def-use pair:

- *def:* where a variable is defined/declared, or modified (initialized or assigned a value)
- *use:* where the variable is used without modifying it

```
int x; // def
…
x = 1; // def
…
y = x + 1; // use of x, def of y
…
foo(x); // use of x if x passed by value
```

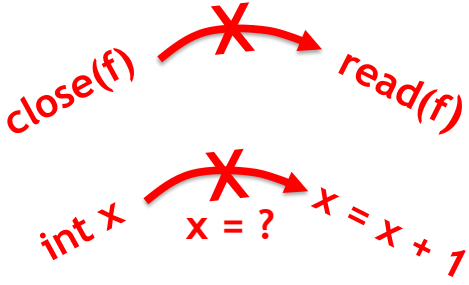
Pointers and object references (aliasing) need special attention!

Data flow analysis has many useful applications

- Ability to check propagation of values via variable assignments

"Is it possible for this object reference to become null at any point?"

- Ability to check constraints on ordering of data operations

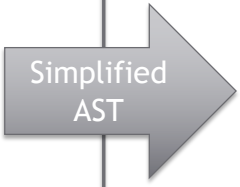


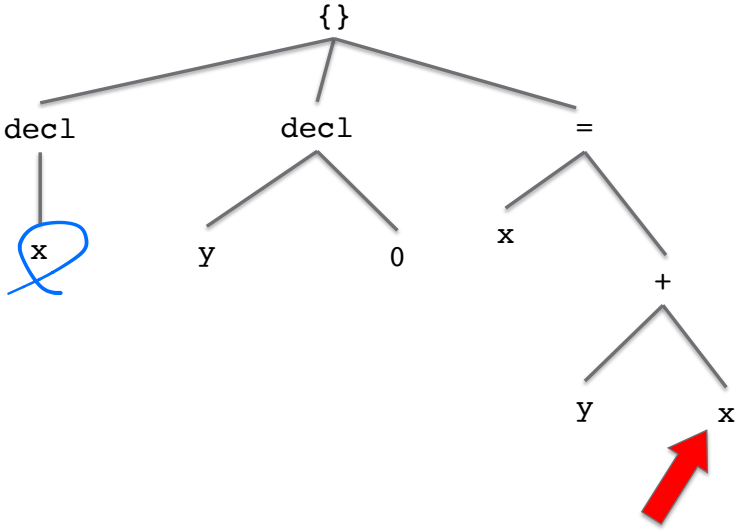
"No file reads (*use*) after file close (*undef*)"

"No using a variable (*use*) before initialization (*def*)"

Team up: Uninitialized variable (5 mins)

```
{
    int x;
    int y = 0;
    x = y + x;
}
```





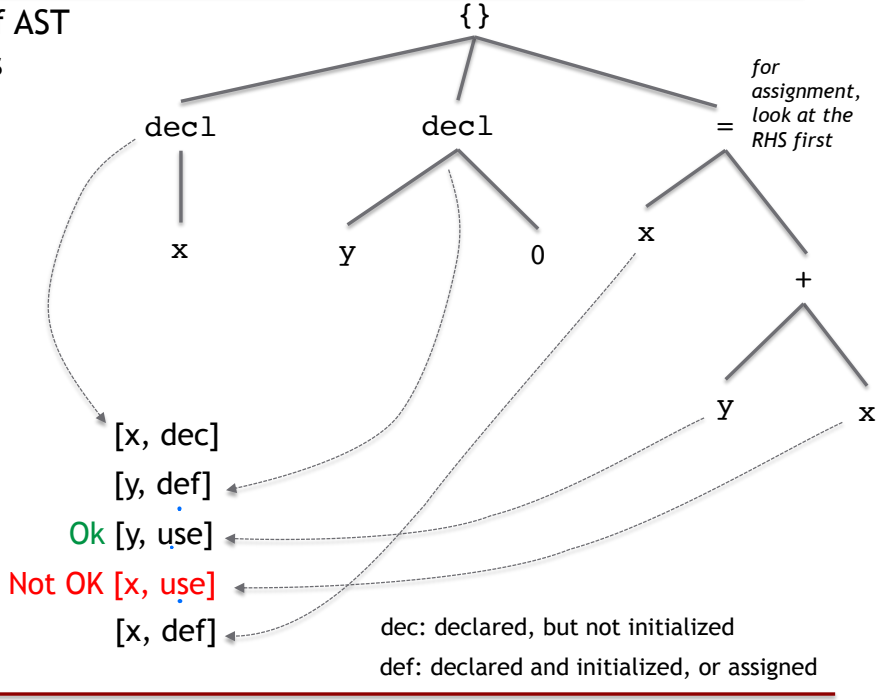
Define an algorithm that will detect that x is uninitialized!

Uninitialized variable


```
{
    int x;
    int y = 0;
    x = y + x;
}
```

Depth-first search of AST

- Log defs and uses
- Process RHS of assignments first



Dataflow analysis is often more than just walking through the AST

- Complex analysis needs CFG (another model, also over-abstraction)
  - can analyze loops to detect undesirable behavior
  - tracking dataflow through CFG allows checking more interesting properties
- May rely on other abstractions
  - Call Graph, Data Dependency Chain
  - Data abstraction

→ analysis become increasingly more “dynamic”

Dataflow analysis often relies on data abstraction

Abstraction depends on the particular property being checked

In the previous example, we abstracted a variable as having three states: { dec, def, use }

- Could a variable become null? { Null, NonNull }
- Is the file still open at an execution point? { Open, Closed }
- Could a variable's value become negative? { Neg, NonNeg }

To generalize dataflow analysis, we can define program behavior for abstract values and "pretend-execute" the program according to the new semantics (AST may be enough, or may need CFG)

- Define abstract type for int:

  int = {NEG, NON-NEG, UNKOWN}

- Define an abstract operator for each concrete operator applicable to the data type

  NEG + NEG ➔ NEG

- Pretend to "execute" the program with abstract values
- Loops: iterate until a fixed-point* or an error state is reached

*fixpoint: program behavior stabilizes and indefinitely cycles through the same states

Example: Division-by-Zero (DZ) analysis

Could the program execute a statement involving division by zero?

- Could a variable become zero?
- Is there a statement that contains division by a variable that can become zero?

Image: https://me.me/i/oh-crap-who-divided-by-zero-old-school-meme-thread-258c780f38514c27adfdc70371dc225d

Data abstraction in DZ analysis

- Concrete data type: `int`
- Abstract data type: `{ zero (Z), positive (P), negative (N), maybe-zero (M), error (E), warning (W) }`
- Abstraction function for `int` literals (primitive values):

  α(0) = Z;

  α(i) = P, if i > 0;

  α(i) = N, if i < 0;

What does this set look like?

Abstraction is a partition of the data type, *except…*

Blocks can:

- be **uncertain** (differentiate between possible and definite states)
- represent **error** states

Team up (2-3): Define the following abstract operations for the abstract domain… (10 mins)

```
{ zero (Z), positive (P), negative (N),
    maybe-zero (M),
    error (definite div by 0) (E),
    warning (possible div by 0) (W) }
```

Pick either:

unary – and one binary op among +, /, -, *

DZ analysis: all applicable operators are defined for the abstract data type

Sign Rules (+, –)	Sign Rules (*, /)	Error Rules
P + P = P	P * N = N	?? / Z = E
P – P = M	P * P = P	?? / M = W
P + N = M	Z * ? = Z	?? ± E = E
? ± Z = ?	…	?? * E = E
Z ± Z = Z	M * P = M	?? ± W = W
…	M * N = M	?? * W = W
–P = N	…	–E = E
–N = P	P / N = N	–W = W
–Z = Z	P / P = P	…
	…	

?: any value except E, W

??: any value

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
   x = x/y;
   y = y – 1;
   z = z + 1;
}
```

Use DZ analysis with abstraction to determine if this program can ever throw a division-by-zero exception

DZ analysis example

Program State

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

x:P

We'll mock-execute the program with abstract values: need CFG for this purpose

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

```
x:P
x:P, y:P
```

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

```
x:P
x:P, y:P
x:P, y:P, z:Z
```

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

```
x:P
x:P, y:P
x:P, y:P, z:Z
x:P, y:P, z:Z
```

We ignore loop condition (assume it's always true)

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

```
x:P
x:P, y:P
x:P, y:P, z:Z
x:P, y:P, z:Z
x:P, y:P, z:Z
```

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

```
x:P
x:P, y:P
x:P, y:P, z:Z
x:P, y:P, z:Z
x:P, y:P, z:Z
x:P, y:M, z:Z
```

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

```
x:P
x:P, y:P
x:P, y:P, z:Z
x:P, y:P, z:Z
x:P, y:P, z:Z
x:P, y:M, z:Z
x:P, y:M, z:P

Iteration 1
```

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

x:P	
x:P, y:P	
x:P, y:P, z:Z	
x:P, y:P, z:Z	x:P, y:M, z:P
x:P, y:P, z:Z	
x:P, y:M, z:Z	
x:P, y:M, z:P	
Iteration 1	**Iteration 2**

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

	Iteration 1	Iteration 2
	x:P	
	x:P, y:P	
	x:P, y:P, z:Z	
	x:P, y:P, z:Z	x:P, y:M, z:P
	x:P, y:P, z:Z	x:W, y:M, z:P
	x:P, y:M, z:Z	
	x:P, y:M, z:P	

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

x:P		
x:P, y:P		
x:P, y:P, z:Z		
x:P, y:P, z:Z	x:P, y:M, z:P	
x:P, y:P, z:Z	x:W, y:M, z:P	
x:P, y:M, z:Z	x:W, y:M, z:P	
x:P, y:M, z:P	x:W, y:M, z:P	
Iteration 1	**Iteration 2**	

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

	Iteration 1	Iteration 2	Iteration 3
	x:P		
	x:P, y:P		
	x:P, y:P, z:Z		
	x:P, y:P, z:Z	x:P, y:M, z:P	x:W, y:M, z:P
	x:P, y:P, z:Z	x:W, y:M, z:P	x:W, y:M, z:P
	x:P, y:M, z:Z	x:W, y:M, z:P	x:W, y:M, z:P
	x:P, y:M, z:P	x:W, y:M, z:P	x:W, y:M, z:P

Nothing would change from this point on, so a fixpoint has been reached: abstract values won't change no matter how many times the loop is iterated!

No change in while's loop condition: STOP!

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

	Iteration 2	Iteration 3
	x:P, y:M, z:P	x:W, y:M, z:P
	X:W, y:M, z:P	x:W, y:M, z:P
	X:W, y:M, z:P	x:W, y:M, z:P
	X:W, y:M, z:P	x:W, y:M, z:P

W: Warning
E: Error

Result: x:W, y:M, z:P => Warning (x may be divided by zero)

DZ analysis example

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

x:P, y:M, z:P
X:W, y:M, z:P
X:W, y:M, z:P

x:W, y:M, z:P
x:W, y:M, z:P

This kind of analysis is a (simplified) form of a program analysis technique called **Abstract Interpretation!**

W: Warning
E: Error

Result: x:W, y:M, z:P => Warning (x may be divided by zero)

Abstraction at work

- Analysis insensitive to initial value of loop condition variable y (no matter how large y is, we don't need to execute more than 3 iterations of the loop)
- If concrete values were used, max number of program states to explore could potentially be huge
  - with 3 × 32-bit variables, max $2^{(3\times32)} = 2^{96}$ states, depending on the initial value of y
- With abstract values, state space is considerably reduced
  - max $6^3 = 108$ states (not all are reachable)

But we can produce false alarms?

Why does this produce a false alarm?

```
x = -1;
x++;
if (x > 0)
  print(1/x);
```

Why does this produce a false alarm?

There is no bug here, but our DZ analysis tells us there could be: it's a false positive!

```
int x = -1;          // N
x++;                 // N++ → M
if (x > 0)           // (true) N
print(1/x);          // 1/M → W
```

저기사거리에서 오른쪽가면바로수산시장있습니다.

Back to pattern checking… one approach uses state machines to define undesirable programming patterns

- Define undesirable patterns syntactically (e.g., as a state machine)
- Check them while walking through the AST

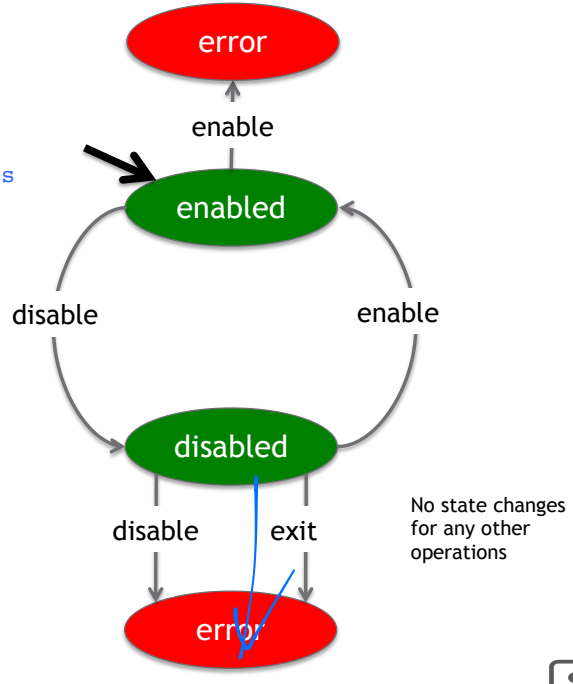
- Simple: AST is finite, no loops
- Syntactic approximation, no execution semantics
- Can be integrated into compilation if compiler is extensible with extra checks

Interrupt example revisited:

a compiler extension for checking unsafe interrupt enable-disable patterns

A DSL for defining unsafe patterns…

```
1. sm check_interrupts {
2.   // variables; used in patterns
3.   decl { unsigned } flags;
4.   // statements corresponding to enable/disable transitions
5.   pat enable = { sti() ; }
6.        | { restore_flags(flags); };
7.   pat disable = { cli() ; }
8.   // states; first state is initial
9.   enabled : disable -> disabled
10.      | enable -> { err("double enable"); };
12. disabled : enable -> enabled
13.      | disable -> { err("double disable"); }
14. //special pattern that matches when
15. // end of path is reached in this state
16.      | $end_of_path$ -> // exit
17.       { err("exiting with intr disabled!"); };
19. }
```



D. Engler et al., "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," OSDI 2000

Interrupt example revisited: searching AST for bad patterns

```
1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh, int b_size) {
4.  struct buffer_head *bh;
5.  unsigned long flags;
6.  save_flags(flags); // save interrupt flags
7.  cli(); // disables interrupts
8.  if ((bh = sh->buffer_pool) == NULL)
9.      return NULL;
10. sh->buffer_pool = bh -> b_next;
11. bh->b_size = b_size;
12. restore_flags(flags); // re-enable interrupts
13. return bh;
14. }
```

init state = **enabled**

move to **disabled**

move to **error**

move to **enabled**

final state = **enabled**

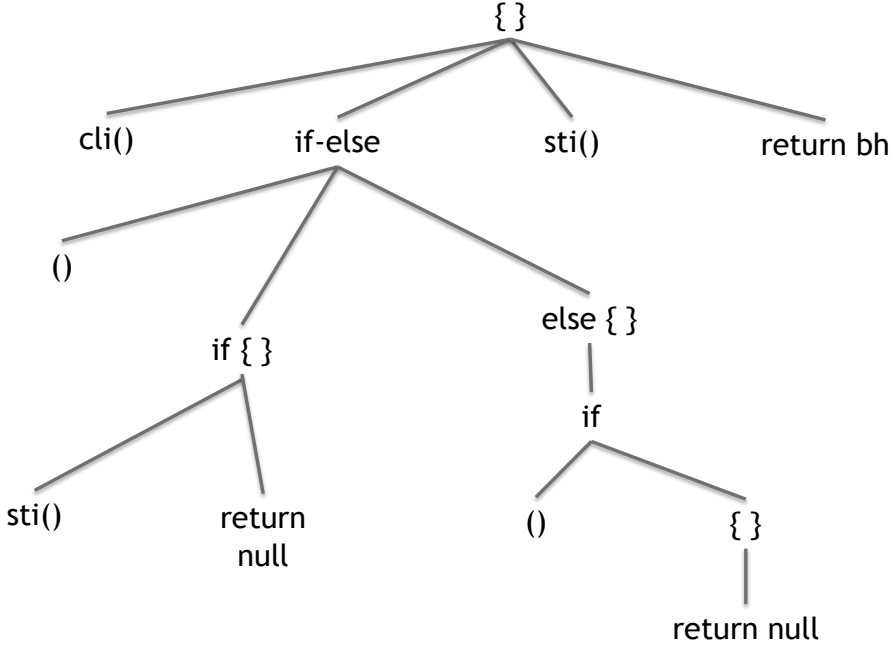
Simple pattern checking while searching the AST

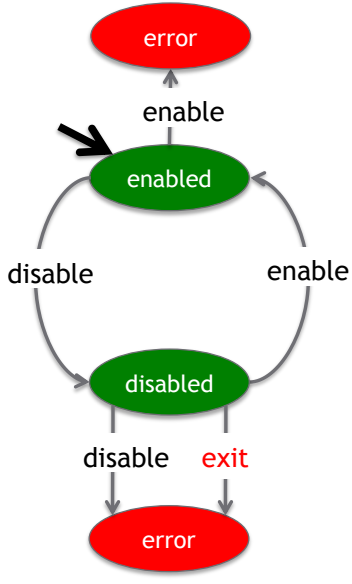
D. Engler et al., "Checking System Rules Using System-Specific,. Programmer-Written Compiler Extensions," OSDI 2000

Using AST, detect if interrupt pattern is violated

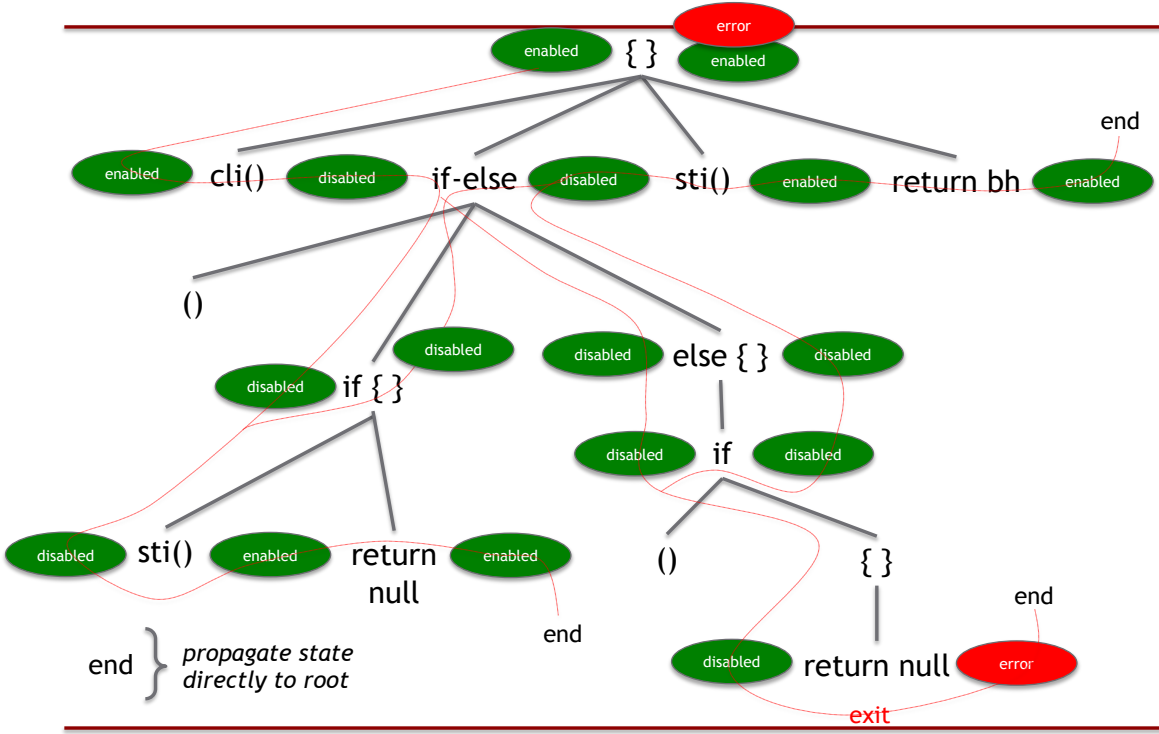
```
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh, int b_size) {
4.     struct buffer_head *bh;
7. cli(); // disables interrupts
8. if ((bh = sh->buffer_pool) == NULL) {
9.    sti(); return NULL;
10.} else {
11.   if (!validate(sh)) return NULL;
12.}
10. sh->buffer_pool = bh -> b_next;
11. bh->b_size = b_size;
12. sti(); // re-enable interrupts
13. return bh;
14. }
```

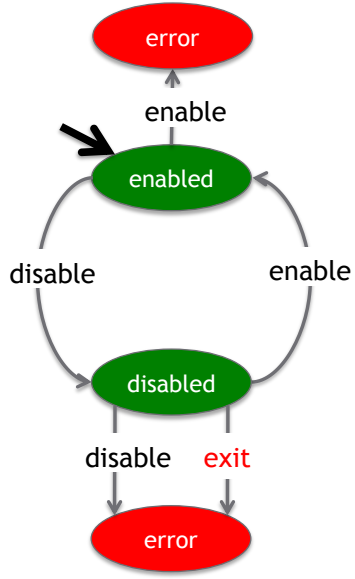
Simplified AST and FSM





Solution - error: exit with intr disabled





Tools: Compilers provide the first level of defense

- Type checking, proper initialization, API compliance

Program	Compiler output

`int add(int x,int y) {`
`  return x+y;`
`}`
 	
`void main() {`
`  add(2);`
`}`
	`$> error: too few arguments to function 'int add(int, int)'`

- Compile at a high warning level
  → static analysis with more rules activated

  $ gcc –Wall

Tools: Linters provide the second level of defense

- Lint was the original static checker for C code
  - Flagged suspicious and non-portable constructs
  - Performs stronger checking than compiler

- Splint (Secure Programming with Lint) is modern, extensible version of Lint

  “Lint-like”or “Lint” tool now refers to any static analyzer that flags suspicious or badly styled code

Splint example

Code (ex.c)	Splint output

`int main() {`
`  char c;`
`  while (c != 'x');`
`  {`
`  c = getchar();`
`  if (c = 'x')`
`  return 1;`
`  }`
`  return 0;`
`}`
	$> splint ex.c
	
	Splint 3.1.1 --- 19 Jul 2006
	
	ex.c:3:10: Variable c used before definition. An rvalue is used that may not be initialized to a value on some execution path. (Use -usedef to inhibit warning)
	
	ex.c:3:10: Suspected infinite loop. No value used in loop test (c) is modified by test or loop body. This appears to be an infinite loop. Nothing in the body of the loop or the loop test modifies the value of the loop test. Perhaps the specification of a function called in the loop body is missing a modification. (Use -infloops to inhibit warning)
	
	ex.c:5:5: Assignment of int to char: c = getchar()
	To make char and int types equivalent, use +charint.
	...

Splint is extensible: how to extend it to detect *taintedness*

- Tainting marks data as untrusted
  - Tainted data originates from user/external environment (inputs)
  - Mark source data as tainted and analyze program's data flow to determine how it's used
- We can extend Splint to detect possible *taintedness*
- Based on data-flow analysis

In Security, this is called taint analysis

What kinds of problems can taint analysis reveal?

What kinds of problems can taint analysis reveal?

```
#!/usr/bin/perl

# Get the name from the browser
my $name = $cgi->param("name");
...

# Execute a SQL query
$dbh->execute("SELECT * FROM Users WHERE name = '$name';");
```

What happens if `$name` = “`Hakan' OR '1' = '1`”?

What kinds of problems can taint analysis reveal?

```
#!/usr/bin/perl

# Get the name from the browser
my $name = $cgi->param("name");
...

# Execute a SQL query
$dbh->execute("SELECT * FROM Users WHERE name = '$name';");
```

What happens if `$name` = "`Hakan' OR '1' = '1`"?

```
"SELECT * FROM Users WHERE name = 'Hakan' OR '1' = '1';"
```

Extending Splint to detect taintedness

Specification in Splint DSL

Telling Splint which abstractions to use in data flow analysis…

```
attribute taintedness
  context reference char *
  oneof untainted, tainted
  annotations
    tainted reference ==> tainted
    untainted reference ==> untainted
  transfers
    tainted as untainted ==> error
      "Possibly tainted storage!"
  merge
    tainted + untainted ==> tainted
  defaults
    reference ==> tainted
    literal ==> untainted
    null ==> untainted
end
```

taintedness property is associated with references that are char pointers

these references can be: tainted or untainted (an abstraction of “char *”)

defines annotations that can be used to indicate taintedness of references of this type in code; e.g., “tainted” annotation indicates a tainted reference

it’s an error to pass a tainted value in a parameter that is required to be untainted (abstract operation)

all ops merging tainted and untainted objects produce a tainted object (abstract operation)

literals and null values are by default untainted; unannotated references of type “char *” are by default tainted (abstraction function for literals)

http://www.splint.org/manual/html/sec10.html

Extending Splint to detect taintedness

Usage with code annotations

```
int printf(/*@ untainted @*/ char *fmt, ...);
```

Pre-condition to be checked: this parameter must be untainted (otherwise produces a warning)

```
char *fgets(char *s, int n,   FILE *stream)
    /*@ ensures tainted s @*/ ;
```

Automatic post-condition: after fgets returns, this reference is always tainted

```
char *strcat(/*@ returned @*/ char *s1,  char *s2)
   /*@ ensures s1:taintedness =
             s1:taintedness | s2:taintedness @*/
```

Automatic post-condition after strcat returns, s1 is tainted if either s1 or s2 was tainted before the call

http://www.splint.org/manual/html/sec10.html

This custom-pattern checking approach can be used with Java as well using the Checker Framework

```
// return value
@InternedString intern() { ... }

// parameter
int compareTo(@NonNullString other) { ... }

// receiver ("this" parameter)
String toString(@TaintedMyClass this) { ... }

// generics: non-null list of interned Strings
@NonNullList<@InternedString> messages;

// arrays:  non-null array of interned Strings
@InternedString @NonNull[] messages;

// cast
myDate = (@InitializedDate) beingConstructed;
```

See documentation for more information:

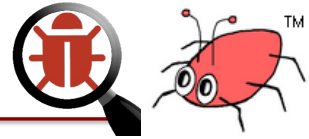
- https://checkerframework.org/manual
- https://checkerframework.org/tutorial

Based on the traversal of the AST

uses user-defined type rules and Java type annotations

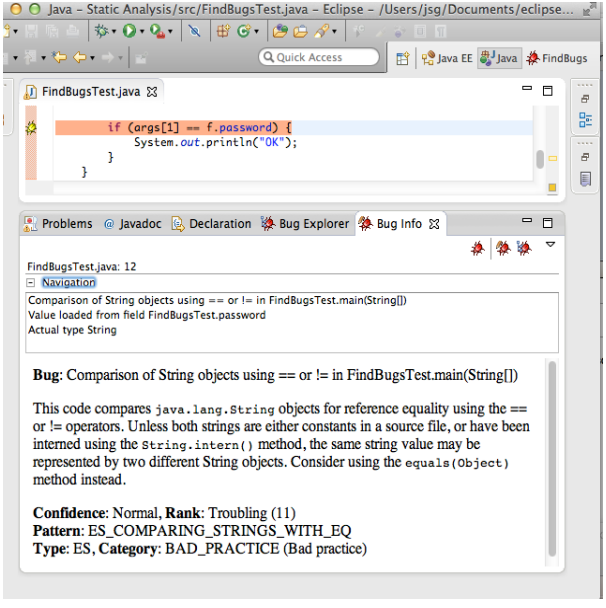
piggybacks on the Java compiler

Tools: SpotBugs (formerly FindBugs)



Detects Java bug patterns

- Categorized patterns (detectors)
  - security, correctness, concurrency, …
  - over 300 issues of different severity
- Extensible
- Uses pattern checking, dataflow analysis
- Plugins for popular IDEs



spotbugs.github.io

Some SpotBugs “Bugs"

- Improper use of equality comparison
- Unclosed streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation violations
- Inconsistent synchronization
- Excess string copying
- Dead store to variable (unused variable)

Tools: PMD

- Popular OS static analyzer for Java
  - Parses Java code into an AST representation
  - Checks are performed by traversing AST while looking for problematic patterns
  - Can perform simple dataflow analysis by adding extra information to AST during traversal
  - Extensible via rules that contain XPath expressions for navigating the AST

Tools: Checkstyle

checkstyle

- Style checker for Java
- Syntactic checks for adherence to coding conventions and formatting styles
- Complexity checks and code (OO) metrics
  - Size violations: large classes, long methods
  - Complex methods (based on cyclomatic complexity)
  - Complex boolean expressions

github.com/checkstyle/

Tools: Infer (by Facebook)

- For C, C++, Objective-C, and Java…
- Research-based, sound analyzer that relies on abstract interpretation
- Few false alarms
- Detects:
  - null pointer exceptions
  - resource leaks
  - missing lock guards
  - concurrency race conditions
- Open source

Used by *FB, Spotify, Uber, Mozilla, AWS, WhatsApp*

https://fbinfer.com/

ESLint

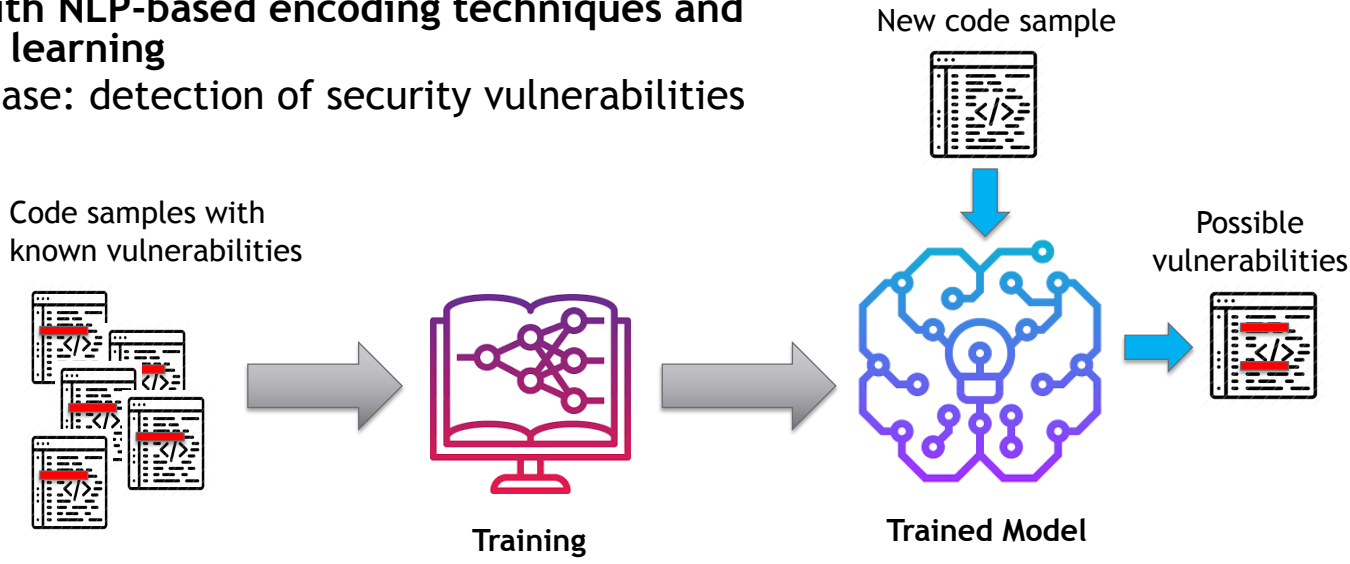
Cloud-based

support for multiple programming languages

New frontiers in static analysis explore the power of AI

... with NLP-based encoding techniques and deep learning

Use case: detection of security vulnerabilities



Static Analysis Takeaways

- Low-hanging fruit: easy, fully automated, push-button
- Can easily be integrated into build process
- May produce too many spurious warnings, but most tools can be fine-tuned
- May detect easy to miss errors that may be hard to detect with testing, but have severe consequences

Just do it: because you can, it's easy, and it may pay off very well

Credits

Many thanks to Claire Le Goues & Chris Timperley for sharing 17-654 lecture notes