ORACLE®

What is a Secure Programming Language?

Programming Languages Implementation Summer School 2019

Cristina Cifuentes
Research Director and Architect
Oracle Labs
23rd – 24th May 2019

ORACLE

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

ORACLE


1999
The University Of Queensland Australia
1996
2005
1994
UTAS
The World Physical
Pacific Centred


~11,300 km
The World Physical

Agenda

1. Thursday 23rd May
   - What is a Secure Programming Language?
   - Quick Intro to GraalVM and Simple Language (SL)
2. Friday 24th May
   - Hands-on Session: Let's add a TaintString to SL

What is a Secure Programming Language?

Cristina Cifuentes and Gavin Bierman, “What is a Secure Programming Language?”, 3rd Summit on Advances in Programming Languages (SNAPL), 16-17 May 2019.

ORACLE

5899

exploited vulnerabilities due to **buffer errors** (2013-2017)

National Vulnerability Database, http://nvd.nist.gov

ORACLE

5851

exploited vulnerabilities due to **injection errors** (2013-2017)

National Vulnerability Database, http://nvd.nist.gov

3106

exploited vulnerabilities due to **information leak** (2013-2017)

National Vulnerability Database, http://nvd.nist.gov

ORACLE

53% (labeled) exploited vulnerabilities in NVD were buffer errors, injections and information leak (2013-2017)

National Vulnerability Database, http://nvd.nist.gov

53%

(labeled ... nerabilities in NVD ... errors, injections ... tion leak (2013-2017)

All of these are issues are within the realm of Programming Language design

National Vulnerability Database, http://nvd.nist.gov

ORACLE

- Buffer overflow used in the Morris worm

1988

- Cross-site scripting exploits

1990s

- SQL injection explained in the literature

1998

Examples of the Three Vulnerability Categories

Buffer Errors

```
void host_lookup (char *user_supplied_addr){
 struct hostent *hp;
 in_addr_t *addr;
 char hostname[64];
 in_addr_t inet_addr(const char *cp);

 /* routine that ensures user_supplied_addr is in the right format for
 conversion */

 validate_addr_form(user_supplied_addr);
 addr = inet_addr(user_supplied_addr);
 hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);
 strcpy(hostname, hp->h_name);
}
```

https://cwe.mitre.org/data/definitions/121.html

ORACLE

Buffer Errors

```c
void host_lookup (char *user_supplied_addr){
 struct hostent *hp;
 in_addr_t *addr;
 char hostname[64];
 in_addr_t inet_addr(const char *cp);

 /* routine that ensures user_supplied_addr is in the right format for
 conversion */

 validate_addr_form(user_supplied_addr);
 addr = inet_addr(user_supplied_addr);
 hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);
 strcpy(hostname, hp->h_name);
}
```

https://cwe.mitre.org/data/definitions/121.html

Buffer Errors

```
# define BUFSIZE 256
int main (int argc, char **argv) {
 char *buf;
 buf = (char *)malloc(sizeof(char)*BUFSIZE);
 strcpy(buf, argv[1]);
}
```

https://cwe.mitre.org/data/definitions/122.html

Buffer Errors

```
# define BUFSIZE 256
int main (int argc, char **argv) {
 char *buf;
 buf = (char *)malloc(sizeof(char)*BUFSIZE);
 strcpy(buf, argv[1]);
}
```

https://cwe.mitre.org/data/definitions/122.html

ORACLE

Cross-Site Scripting

```
<% String eid = request.getParameter("eid "); %>
...
Employee ID: <%= eid %>
```

https://cwe.mitre.org/data/definitions/79.html

Cross-Site Scripting

```
<% String eid = request.getParameter("eid "); %>
...
Employee ID: <%= eid %>
```

https://cwe.mitre.org/data/definitions/79.html

ORACLE

Cross-Site Scripting

```
<% Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery (“select * from emp where id="+eid);
if (rs != null) {
   rs.next();
   String name = rs.getString (“name");
}%>

Employee Name: <%= name %>
```

https://cwe.mitre.org/data/definitions/79.html

Cross-Site Scripting

```
<% Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery (“select * from emp where id="+eid);
if (rs != null) {
   rs.next();
   String name = rs.getString (“name");
}%>

Employee Name: <%= name %>
```

https://cwe.mitre.org/data/definitions/79.html

ORACLE

SQL Injection

```
...
string userName = ctx.getAuthenticatedUserName();
string query = “SELECT * FROM items WHERE owner = ’" + userName +
               "’ AND itemname = ’" + ItemName.Text + "’";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

https://cwe.mitre.org/data/definitions/89.html

ORACLE

SQL Injection

```
...
string userName = ctx.getAuthenticatedUserName();
string query = “SELECT * FROM items WHERE owner = ’" + userName +
               "’ AND itemname = ’" + ItemName.Text + "’";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

https://cwe.mitre.org/data/definitions/89.html

ORACLE

Information Leak

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());
...
catch (Exception e) {
 AlertDialog.Builder builder = new AlertDialog.Builder(this);
 builder.setMessage ("Sorry, this app has experienced an error.");
 AlertDialog alert = builder.create();
 alert.show();
 Log.e(“ExampleActivity", “Caught exception: " + e + ” While on User:"
      + User.toString());
}
```

https://cwe.mitre.org/data/definitions/532.html

Information Leak

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());
...
catch (Exception e) {
 AlertDialog.Builder builder = new AlertDialog.Builder(this);
 builder.setMessage ("Sorry, this app has experienced an error.");
 AlertDialog alert = builder.create();
 alert.show();
 Log.e(“ExampleActivity", “Caught exception: " + e + ” While on User:"
      + User.toString());
}
```

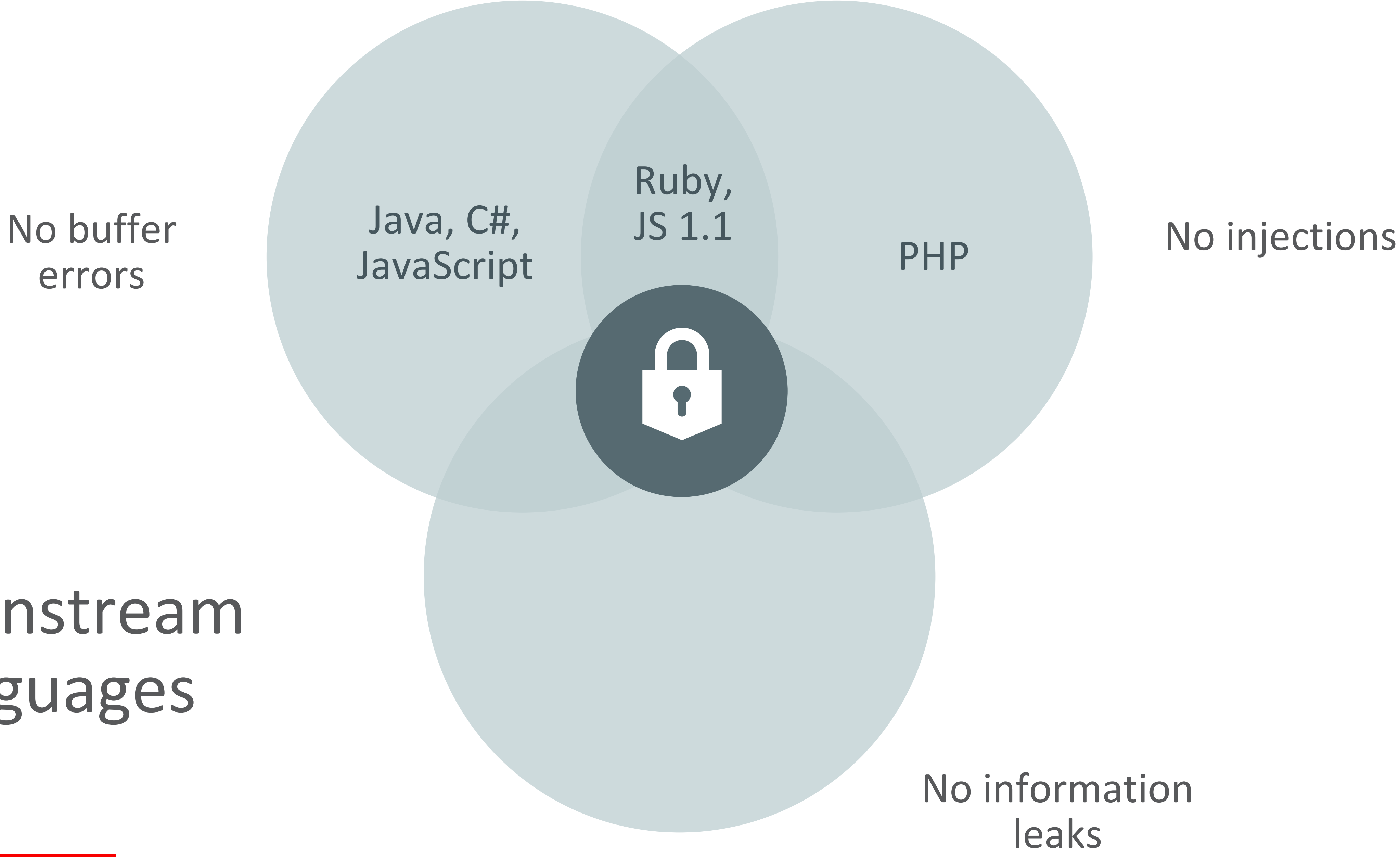
https://cwe.mitre.org/data/definitions/532.html

Mainstream Languages and Vulnerabilities

Top Mainstream Languages Over the Past 10 Years

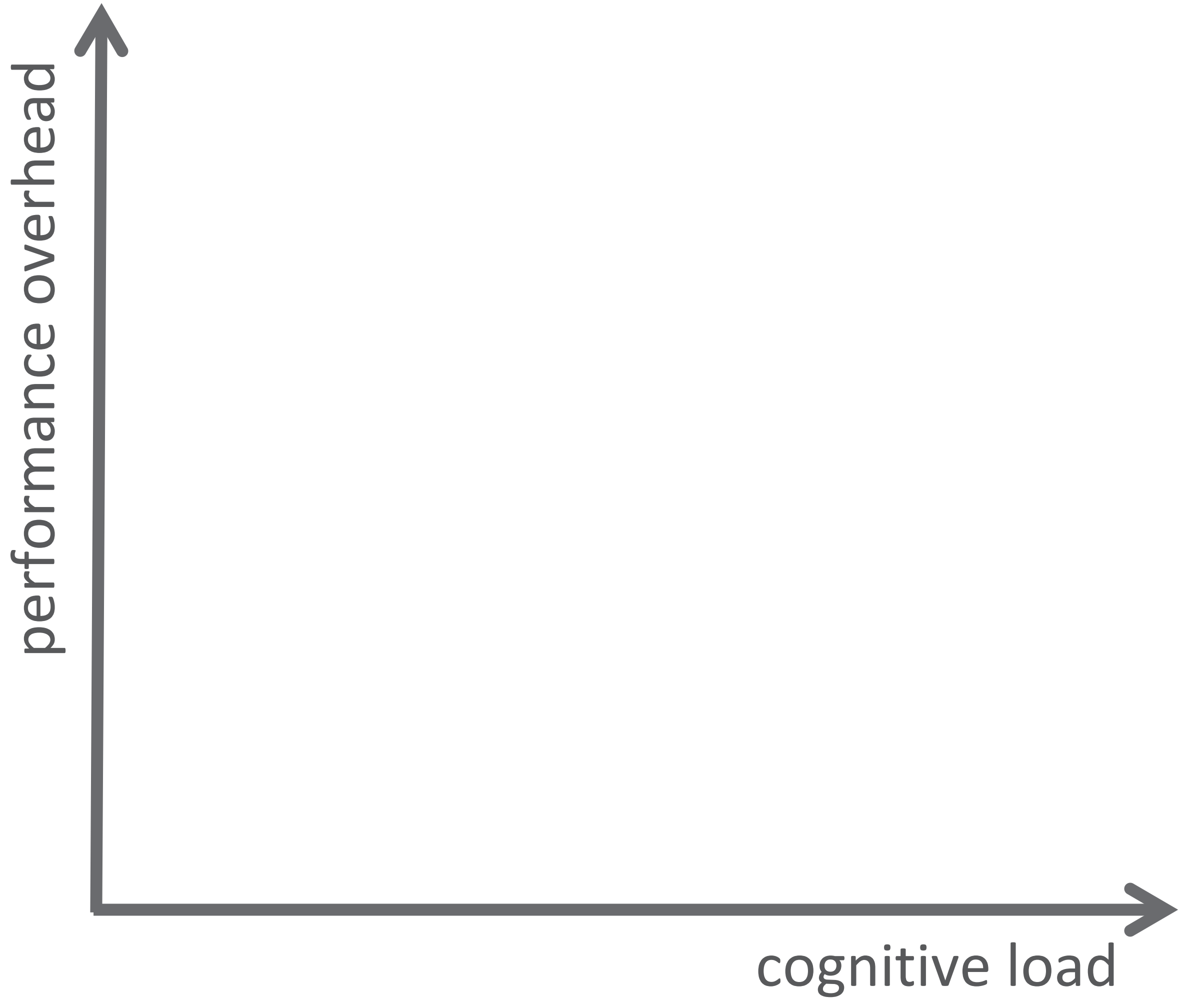
Based on TIOBE index as of January 2019
Java
C
C++
Python
C#
PHP
JavaScript
Ruby

Mainstream Languages

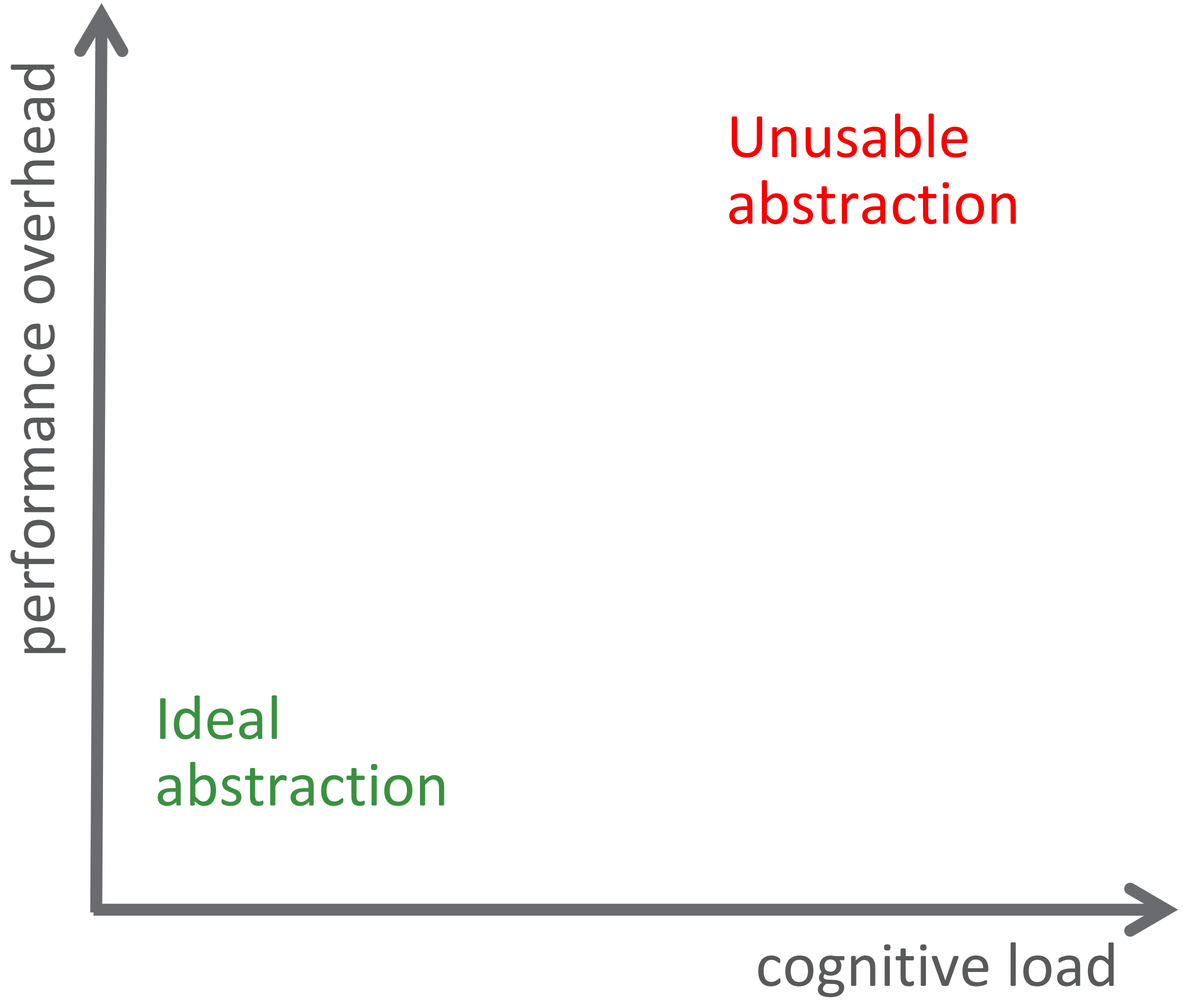


A Secure Language is One that Provides First-class Support for These Three Categories

What to Consider when Talking about Abstractions

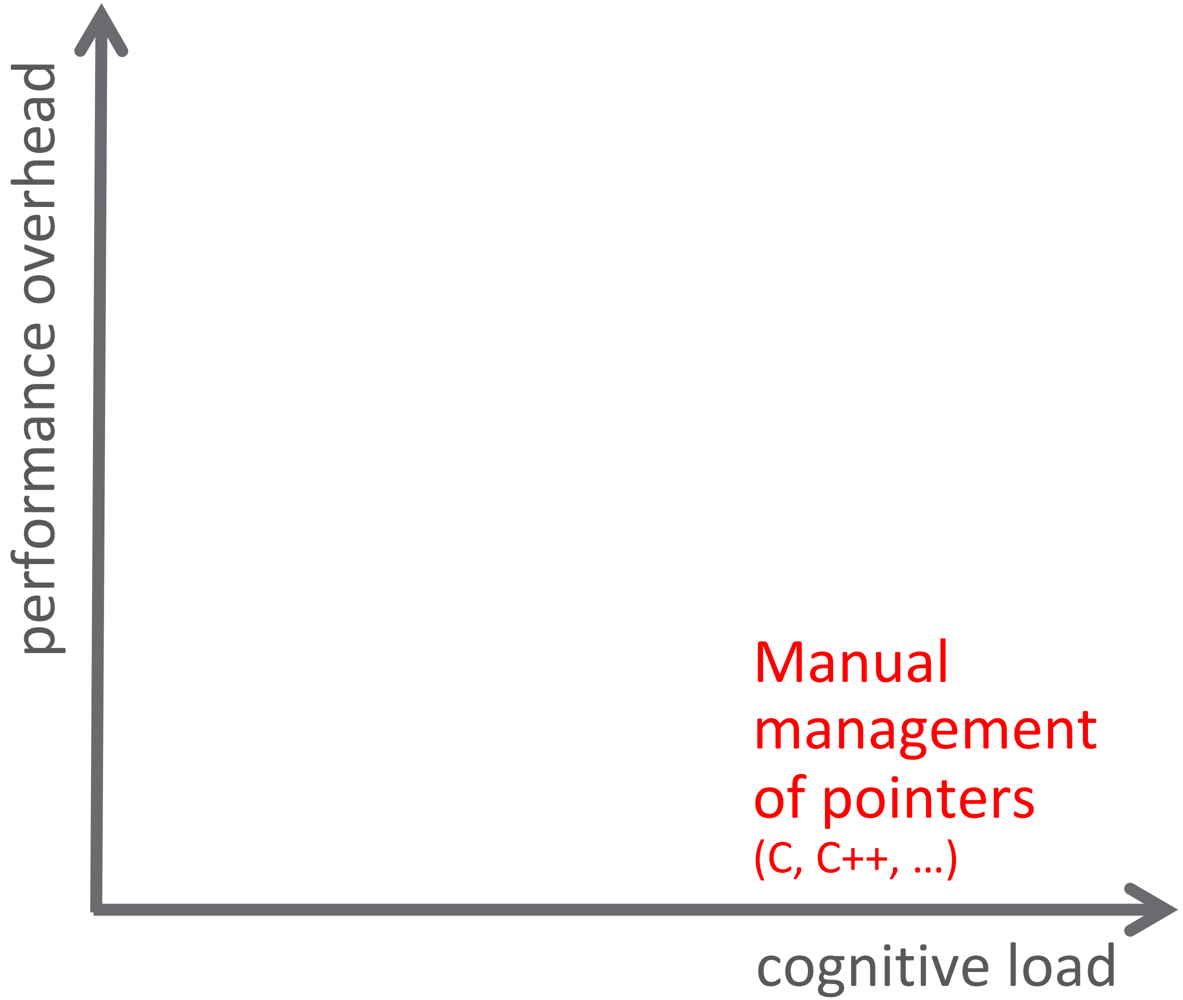


ORACLE

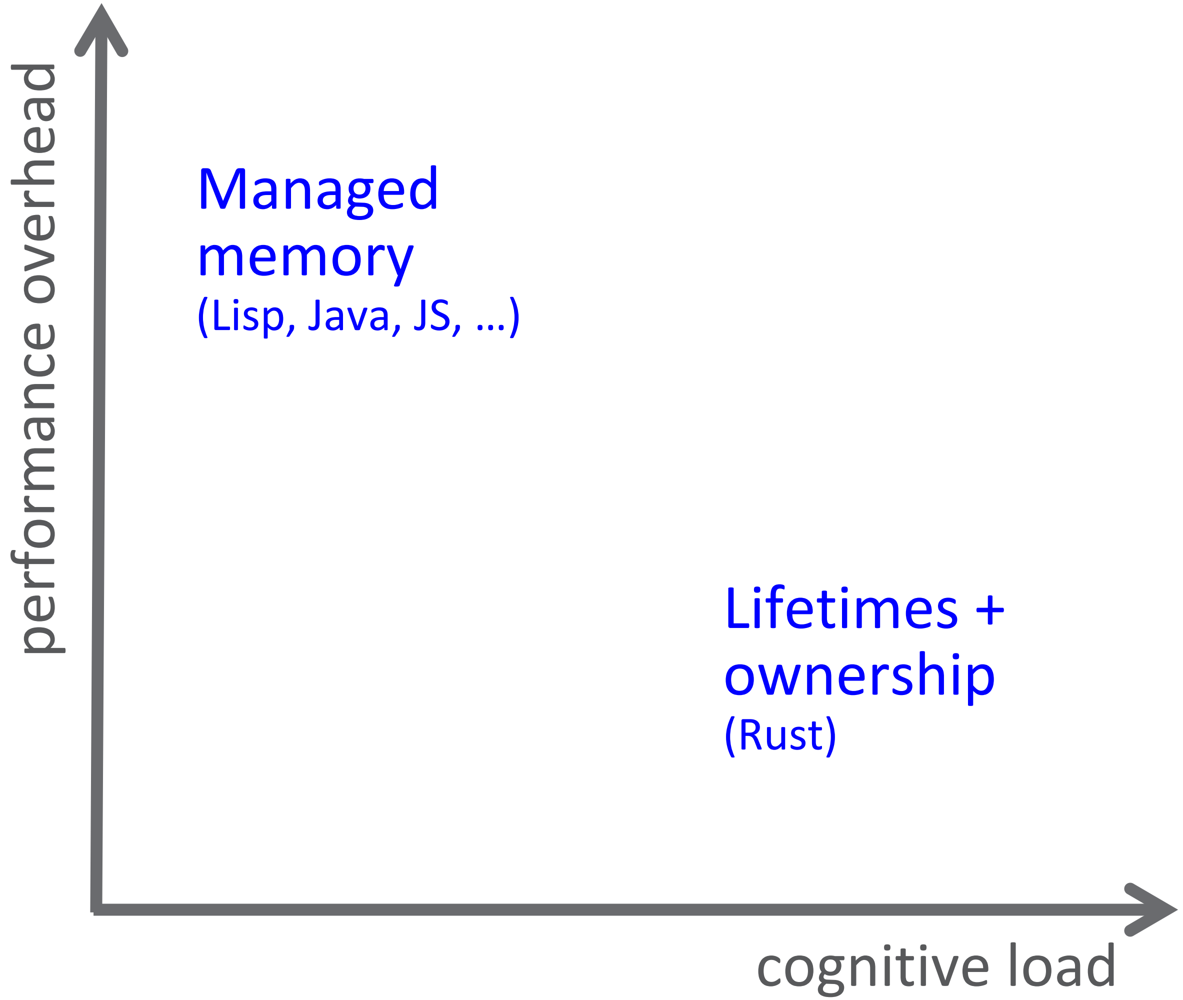

performance overhead
Unusable
abstraction
Ideal
abstraction
cognitive load

Language Support Addressing Buffer Errors

Buffer Errors – The Problem: Unsafe Abstraction



Buffer Errors – Solutions: Safe Abstractions



Avoid Buffer Errors Dynamically

LISP

John McCarthy, 1958

- Managed memory
  - Garbage collection was first introduced in LISP in 1958
- Now in
  - OO languages: Smalltalk, Java, C#, JavaScript, Go
  - Functional languages: ML, Haskell, APL
  - Dynamic languages: Ruby, Perl, PHP

Avoid Buffer Errors Statically

RUST

Graydon Hoare, 2009

- Guarantees* memory safety through new type system concepts
  - Ownership
  - Borrowing:
    shared borrow (&T)
    mutable borrow (&mut T)

* Formal guarantee proofs missing

ORACLE

Ownership With RAII (Resource Acquisition is Initialization)

```
fn main() {
    let x = 5u32;     // stack-allocated integer

    // *Copy* `x` into `y` - no resources are moved
    let y = x;
    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);

    // `a` is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);
    println!("a contains: {}", a);

    // *Move* `a` into `b`
    let b = a;
    // The pointer address of `a` is copied (not the data) into
    // `b`.  Both are now pointers to the same heap allocated
    // data, but `b` now owns it.
    println!("a contains: {}", a);    // Error
}
```

- **Resources can only have one owner**
- Not all variables own resources (e.g., references)
- Ownership of a resource is transferred (i.e., move'd) through assignments or passing arguments by value

http://rustbyexample.com/scope/move.html

ORACLE

Lifetimes

shared borrow

```
fn main() {
    let mut i = 3; // Lifetime for `i` starts.
    {
        let borrow1 = &i;   // `borrow1` lifetime starts.
        println!("borrow1: {}", borrow1); //
    } // `borrow1` ends.

    {
        let borrow2 = &mut i; // `borrow2` lifetime starts.
        *borrow2 = 5;         //
    } // `borrow2` ends.

}   // lifetime ends.
```

mutable borrow

- Rust compiler checks lifetimes are valid to ensure variables are used safely
- Borrows allow data to be used elsewhere, without giving up ownership
- There can be at most 1 mutable reference to a resource

http://rustbyexample.com/scope/lifetime.html

Lifetimes

```
fn main() {
    {
        let mut borrow3 = &mut i;
        *borrow3 += 1;
        println!("borrow3: {}", borrow3);
        let borrow4 = &i;               // error[E0502]: cannot borrow `i` as immutable
                                        // because it is also borrowed as mutable
        println!("borrow4: {}", borrow4);
    }
}
```

Rust Memory Safety Guarantees

- No buffer overflows
- No null pointer dereference
- No double freeing memory
- No stale pointers
- No data races
- No arithmetic overflows
- Warns about uninitialised memory and variables

ORACLE

Rust’s Unsafe Features

Must opt-in to use them

- Calling foreign code
- Calling unsafe code
- Dereferencing a raw pointer

ORACLE

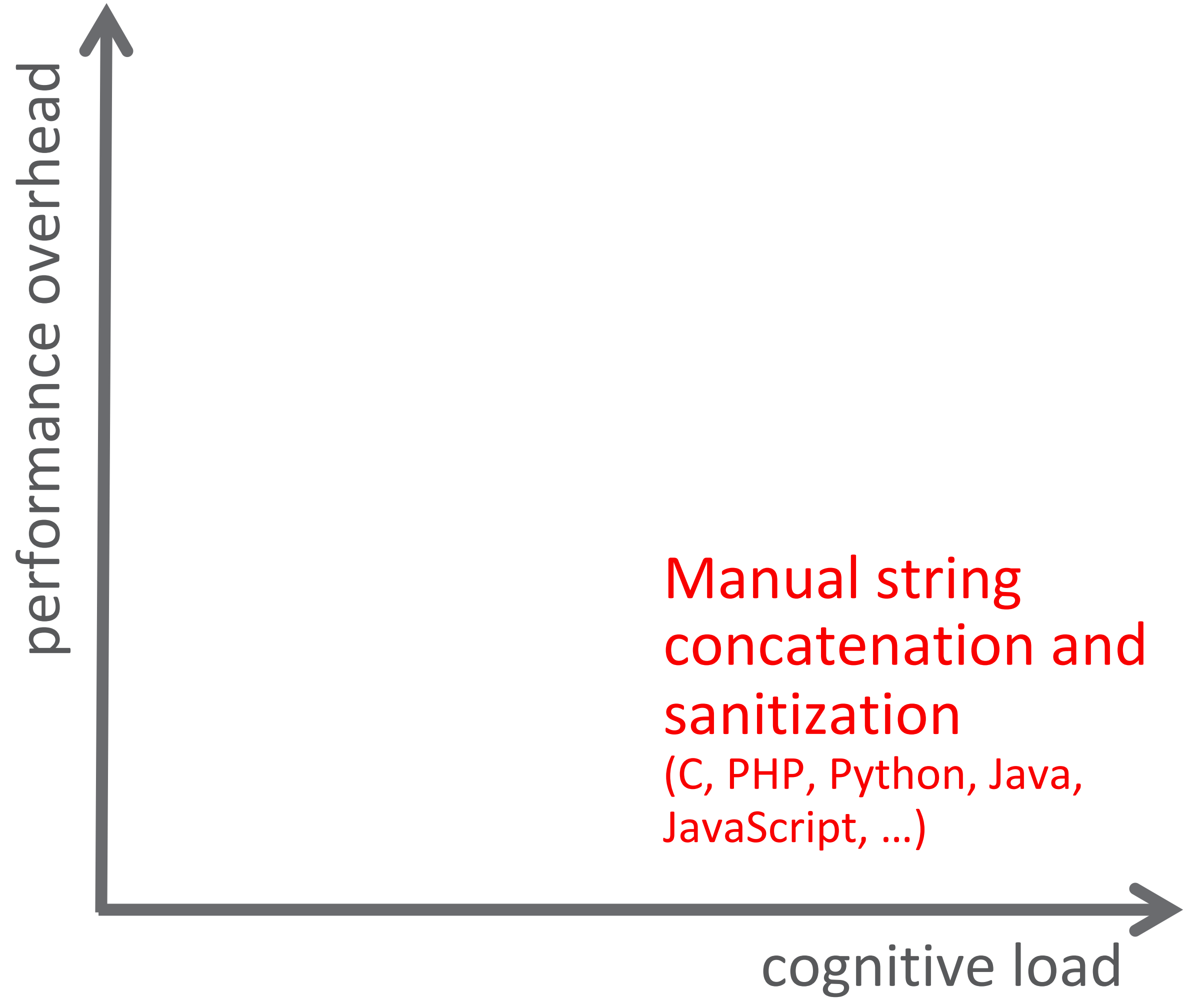
Rust

- Ownership and lifetimes allow for memory safety guarantees
  – No buffer overflows, no null pointer dereferences, no double freeing memory, no stale pointers, no data races, no arithmetic overflows
- Unsafe code
  – Needed to interface with native C code
  – To implement low-level libraries (e.g., Rust's own libraries, a user's library)
  – Unsafe code can void memory safety guarantees

ORACLE

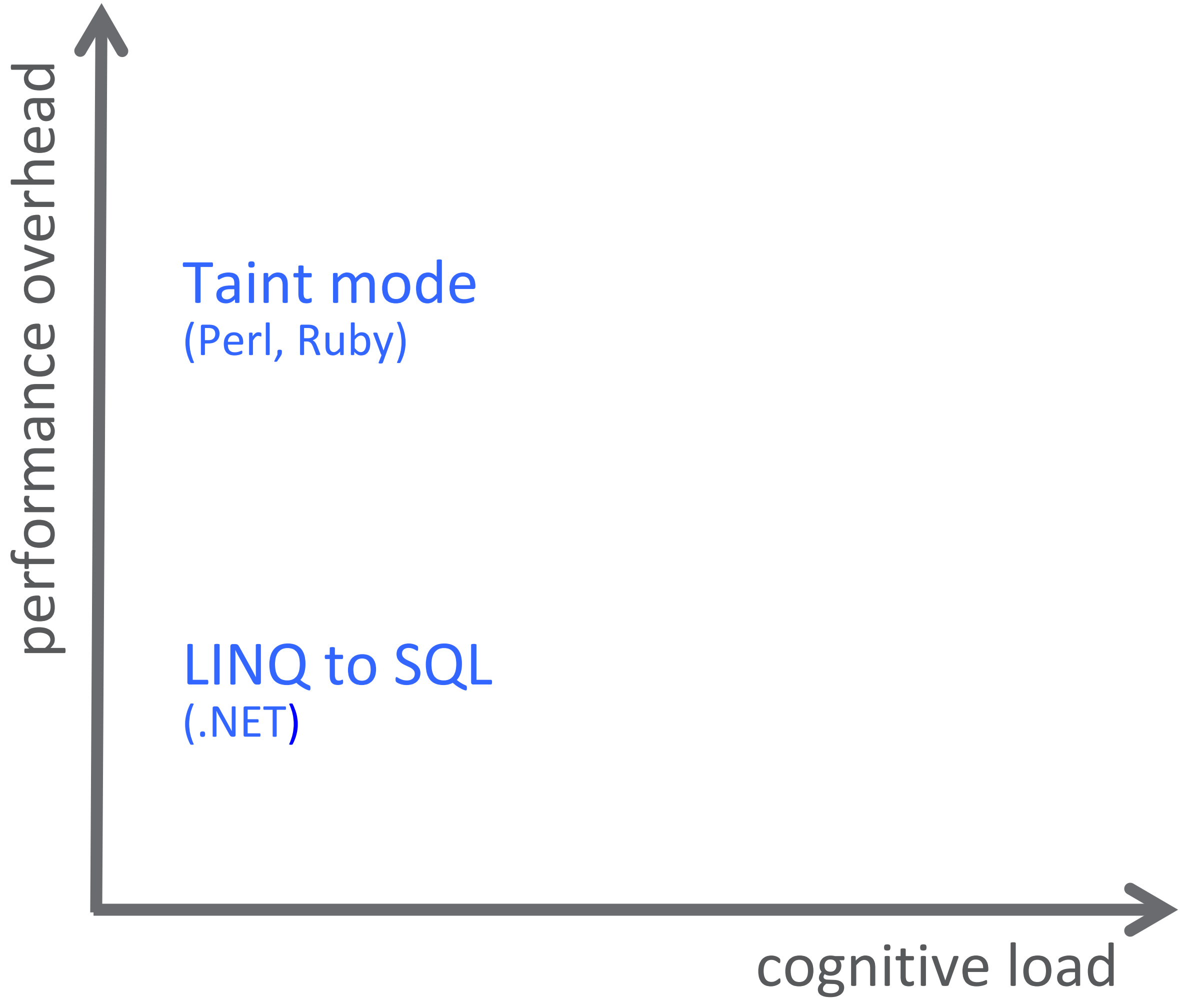
Language Support Addressing Injection Errors

Injections – The Problem: Unsafe Abstraction



ORACLE

Injections – Solutions: Safe Abstractions



Avoid SQL Injections Statically

Microsoft, 2007

- .NET’s Language INtegrated Query framework
- LINQ to SQL manages relational data as objects without losing the ability to query
  - Statically-typed
  - Not 100% compatible
- Avoids SQL injections by passing all data to `PreparedStatement` using SQL parameters
  - Not strings or string concatenation

Avoid Injection Errors Dynamically

PERL

Larry Wall, 1987

- Taint mode
  - Perl 3, 1989
  - Automatic checks when program running with different real and effective user or group IDs
  - -T flag to turn it on
- Similar ideas in
  - Ruby

Taint Mode Perl 3, 4, 5

- Default tainted values
  - All command-line arguments, environment variables, locale information, results of some system calls (`readdir()`, `readlink()`), the variable of `shmread()`, the messages returned by `msgrcv()`, the password, gcos, and shell fields returned by the `getpwxxx()` calls, and all file inputs
- Tainted data may not be used directly or indirectly in
  - any command that invokes a sub-shell, nor in
  - any command that modifies files, directories, or processes; except for
    - Arguments to `print` and `syswrite`
    - Symbolic methods and symbolic subreferences
    - Hash keys are never tainted

ORACLE

Taint Mode Perl 3, 4, 5

```
$arg = shift;                    # $arg is tainted
$hid = $arg . ‘bar’;             # $hid is also tainted

$line = <>;                      # tainted
$line = <STDIN>;                 # also tainted

open FOO, “/home/me/bar” or die $!;
$line = <FOO>;                   # still tainted

$path = $ENV{‘PATH’};            # tainted
$data = ‘abc’;                   # not tainted

system “echo $arg”;              # insecure
system “echo $data”;             # insecure until PATH set

exec “echo $arg”;                # insecure
exec “sh”, ‘-c’, $arg;           # very insecure
```

- Any value that is retrieved from an external source to the script is **tainted**
- Applies to individual scalar values
- One tainted value taints the whole expression
  - Except when using the ternary conditional operator

```
$result =
        $tainted_value ?
        "Untainted" :
        "Also untainted";
```

https://perldoc.perl.org/perlsec.html#Taint-mode

ORACLE

Taint Mode Perl 3, 4, 5

- Two modes
  1. Automatic – when running a script with different setuid and setgid
  2. Manual – activate with `-T` cmdline option to the Perl interpreter

- Untainting is done automatically
  - Using a tainted value as key in a hash
  - Regexp match on a tainted value

```
if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;                  # $data now untainted
}
```

Ruby

https://www.ruby-lang.org

- Expands Perl's taint mode – 4 safe levels
  - 0: no safety
  - 1: disallows use of tainted data by potentially dangerous operations
    default on Unix systems when Ruby script running as setuid
  - 2: prohibits loading of program files from globally-writable locations
  - 3: all newly created objects are considered tainted

ORACLE

Sample Vulnerable Code Due to Tainted Input

```
require ‘cgi’
cgi = CGI::new(“html4”)

# Assume input is an arithmetic expression
# Fetch the value of the form field “expression”
expr = cgi[“expression”].to_s

begin
result = eval(expr)
rescue Exception => detail
# handle bad expressions
end

# display result of arithmetic expression back to user
```

- External data is **tainted**
- User can type into the form `system(“rm *”)`

http://phrogz.net/ProgrammingRuby/taint.html

SAFE Level and Untaint Example

```
require ‘cgi’
$SAFE = 1
cgi = CGI::new(“html4”)

# Assume input is an arithmetic expression
# Fetch the value of the form field “expression”
expr = cgi[“expression”].to_s

if expr =~ %r{^[-+*/\d\seE.()]*$}
   expr.untaint
   result = eval(expr)
   # display result of arithmetic expression back to user
else
   # display error message
```

- Run CGI script at a safe level of 1
  - Raises exception if program passes the form data to eval
- Simple sanity check performed on the form data to untaint if the data looked innocuous

http://phrogz.net/ProgrammingRuby/taint.html

SAFE Level and XSS Example

```
require ‘cgi’
$SAFE = 1

cgi = CGI::new(“html4”)
expr = cgi[“expression”].to_s

if expr =~ %r{^[-+*/\d\seE.()]*$}
   expr.untaint
   result = eval(expr)
end
print “#{expr}:#{result}\n”
```

- External data is tainted
- Tainted data is sanitized
- Taint is not tracked to `print`

Modification of http://phrogz.net/ProgrammingRuby/taint.html

Perl and Ruby's Taint Mode

Perl

- Runtime tracks tainted data not to be used in subshell commands, or commands that modify files, directories, or processes (with some exceptions)

Ruby

- Extends Perl's taint mode to track direct data flows through SAFE modes 1-3
- Programmatic taint/untaint methods

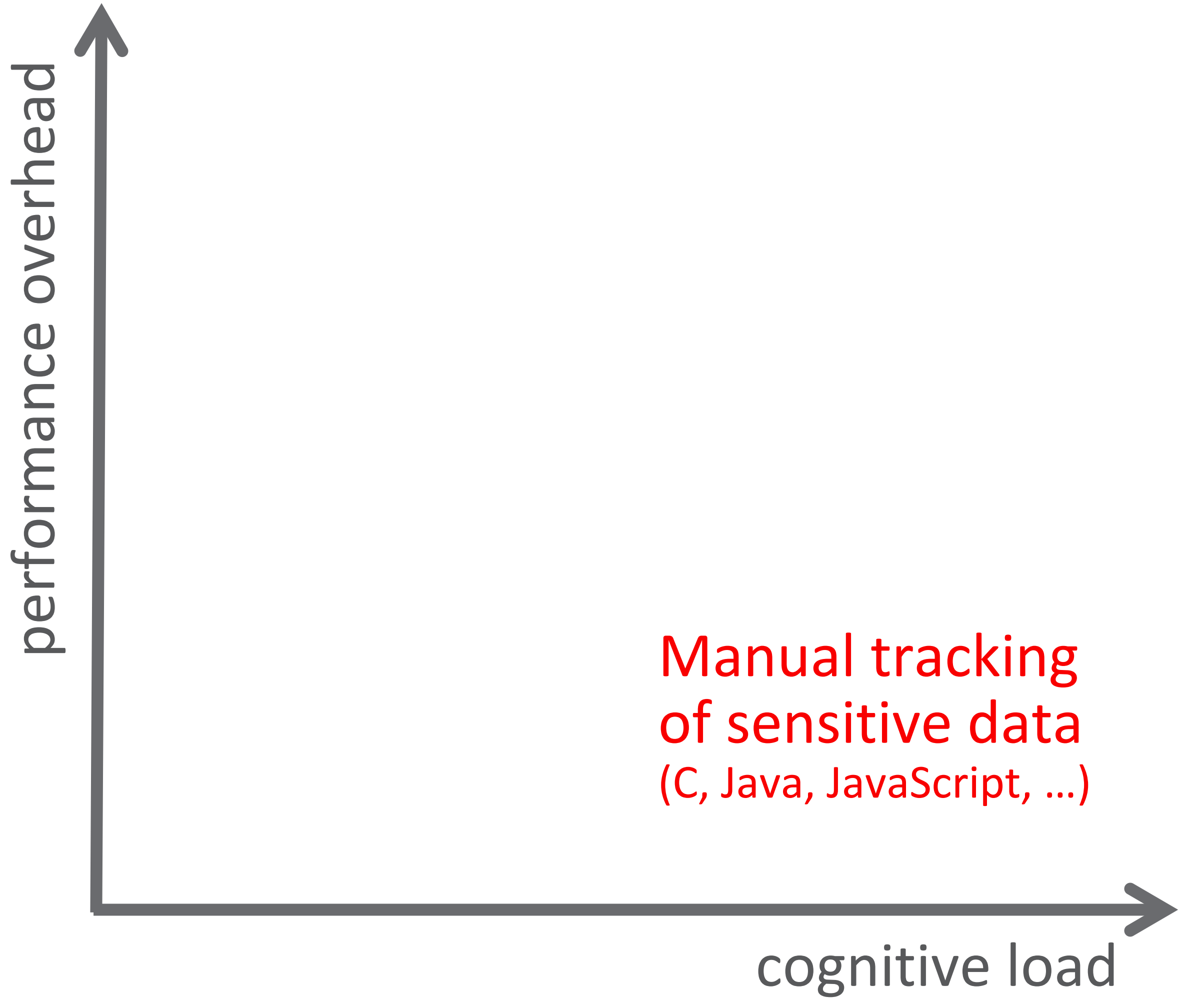
Cannot track XSS as do not track taint to `print` and `syswrite`

Do not track indirect/implicit data flows

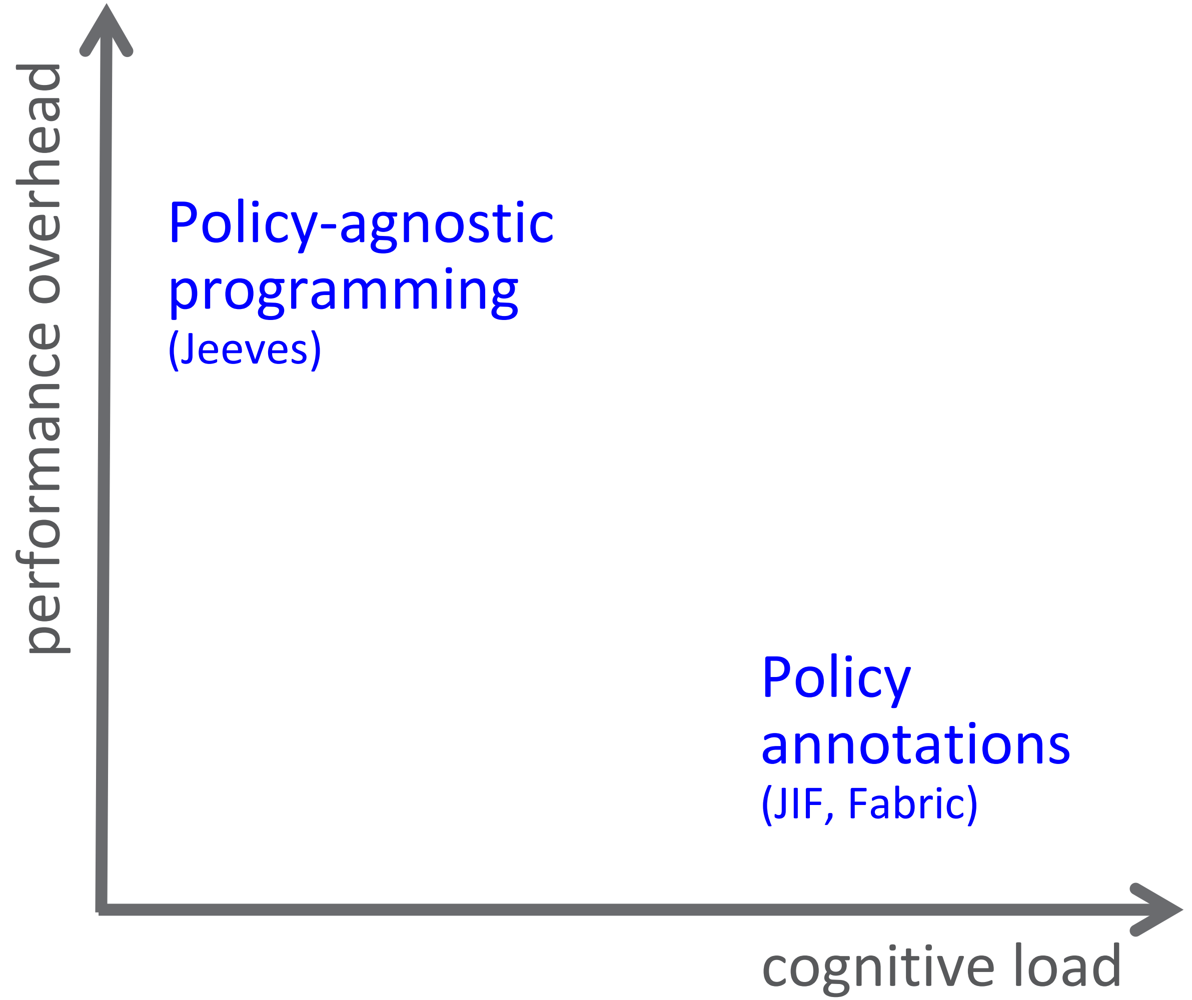
ORACLE

Language Support Addressing Information Leak Errors

Information Leaks – The Problem: Unsafe Abstraction



Information Leaks – Solutions: Safe Abstractions



Avoid Information Leaks and Injections Statically

Jif

Andrew Myers, 2002+

- Extends Java with information flow and access control, enforced at compile time and run time
  - Integrity and confidentiality
  - Can prevent covert information leaks
- Security policies are expressed as label annotations restricting how the information may be used

Avoid Information Leaks Dynamically

Policy-Agnostic Programming

Jean Yang, 2013+

- Faceted values: a policy guarding both, the security-sensitive and non-sensitive values
  - The runtime keeps track of policies associated with conditionals
  - Faceted database saves faceted values
- Sample web applications yield reasonable (< 2x) overheads

ORACLE

Faceted Values

- Faceted values
  - Used for sensitive values
  - Policy guards secret and non-secret value, i.e.,

    `<s | ns>(p)`

    equivalent to: `if (p) <s> else <ns>;`
- Developer specifies policies outside the code
- Language runtime enforces policy

- Faceted records in the DB
  - Faceted record `(p ? s : ns)`
  - Stored as two faceted rows of non-faceted relational records

id	val	fid	fpolicy
1	s	1	p==True
2	ns	1	p==False

  - Allows for faceted queries using `WHERE` and `JOIN` clauses

ORACLE

Example: Social Calendar App

- Alice wants to plan a surprise party for Bob at 7pm next Tuesday. She should be able to create an event such that information is visible only to guests. Bob should see that he has an event 7pm next Tuesday, but not that it is a party. Everyone else may see that there is a private event, but not event details.

Person ID	Event name	Faceted ID	Policy
1	'Surprise party'	1	'p=True'
2	'Private event'	1	'p=False'

```
class Event(Model):
    name = CharField(max_length=256)
    time = DayTimeField()
    ...

    # public value for name field
    def jacqueline_get_public_name(event):
        return "Private event"

    # policy for name field
    @label_for('name')
    def jacqueline_restrict_event(event, ctxt):
        return(EventGuest.objects.get(
               event=self, guest=ctxt) != None)

class EventGuest(Model):
    event = ForeignKey(Event)
    guest = ForeignKey(UserProfile)
```

http://www.cs.cmu.edu/~jyang2/papers/p631-yang.pdf

ORACLE

Example: Social Calendar App Query

- Without faceted records; policy not enforced at the query level

```
SELECT EventGuest.event,
       EventGuest.guest
FROM   EventGuest
JOIN   UserProfile
ON     EventGuest.guest_id =
       UserProfile.id
WHERE  UserProfile.name = 'Alice';
```

- Automatically-generated code with faceted records*; policy enforced at query time

```
SELECT EventGuest.event,
       EventGuest.guest,
       EventGuest.fid,
       EventGuest.fpolicy,
       UserProfile.fpolicy
FROM   EventGuest
JOIN   UserProfile
ON     EventGuest.guest_id =
       UserProfile.fid
WHERE  UserProfile.name = 'Alice';
```

SQL API used by developer, facets introduced by the system

ORACLE

Status – Results

- Applications
  - Conference management system
  - Health record manager
  - Course manager
- Reduced lines of code
  - Policy code: 106 LOC central vs 130 LOC spread out in the code
  - Auditing policy code: 200 LOC vs 575 LOC => 65% reduced size of application-specific trusted code base

- Performance
  - 1.75x overhead on stress tests
  - At par viewing profiles for a single user
  - Faster viewing profiles for a single paper in conference mgmt system (as policies resolved once)

ORACLE

Policy-Agnostic Programming

- New paradigm that centralises policy code outside of the main application and tracks information flows relevant to information leak at runtime
- Main benefits
  - Application and database code do not need to be trusted
  - Policies are localised
  - The size of the policy is smaller due to automatic policy enforcement
- Status
  - Academic prototype

ORACLE

Concluding Remarks

53% (labeled) exploited vulnerabilities in NVD were buffer errors, injections and information leak (2013-2017)

National Vulnerability Database, http://nvd.nist.gov

53%

(labeled ... nerabilities in NVD ... errors, injections ... tion leak (2013-2017)

All of these are issues are within the realm of Programming Language design

National Vulnerability Database, http://nvd.nist.gov

ORACLE

Top Mainstream Languages Over the Past 10 Years

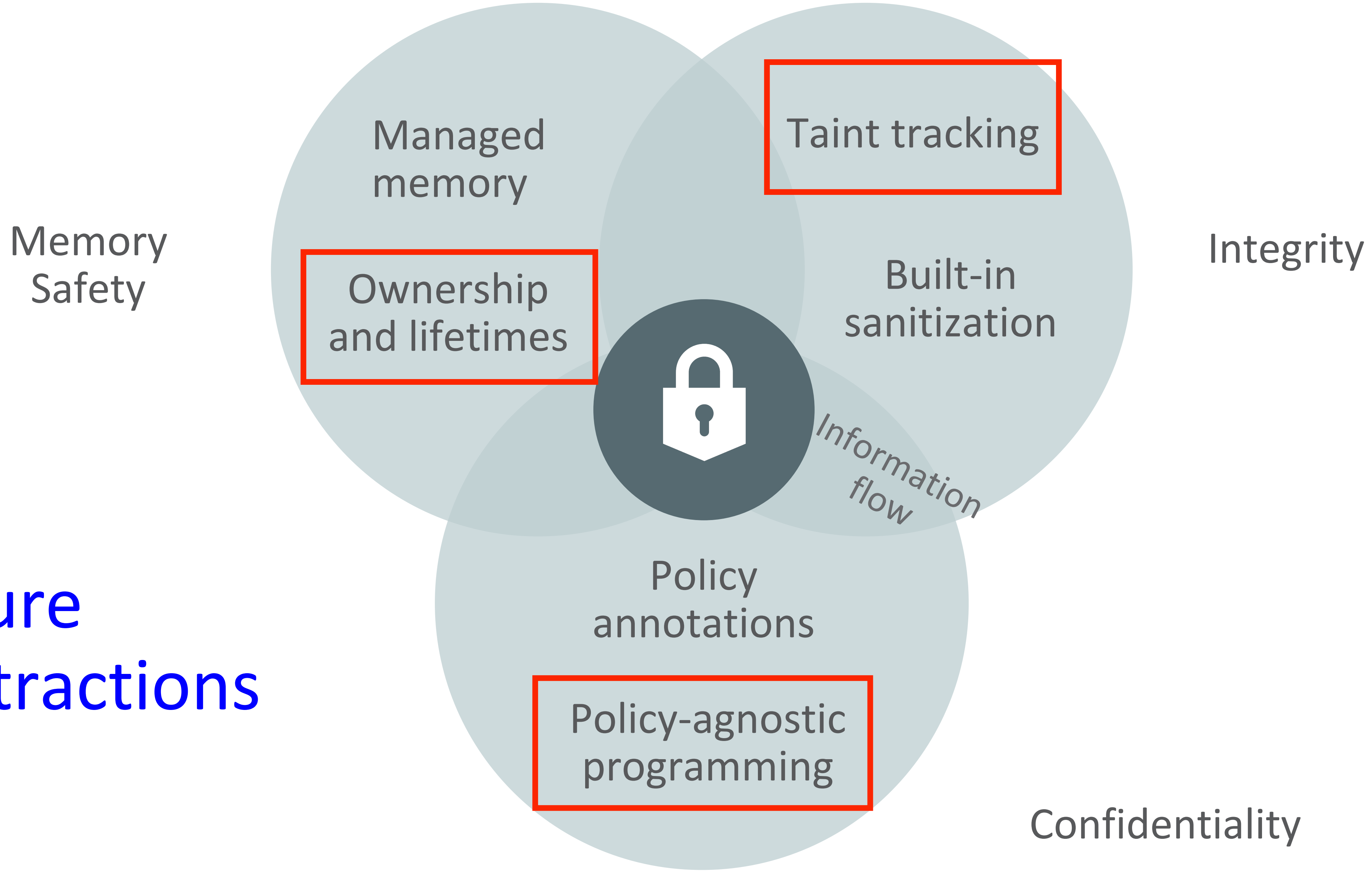
Based on TIOBE index as of January 2019
Java
C
C++
Python
C#
PHP
JavaScript
Ruby

A Secure Language is One that Provides First-class Support for These Three Categories

Today's mainstream languages do not support our developers in writing secure code that is free of buffer errors, injections, or information leaks.

Our mainstream languages are not secure languages.

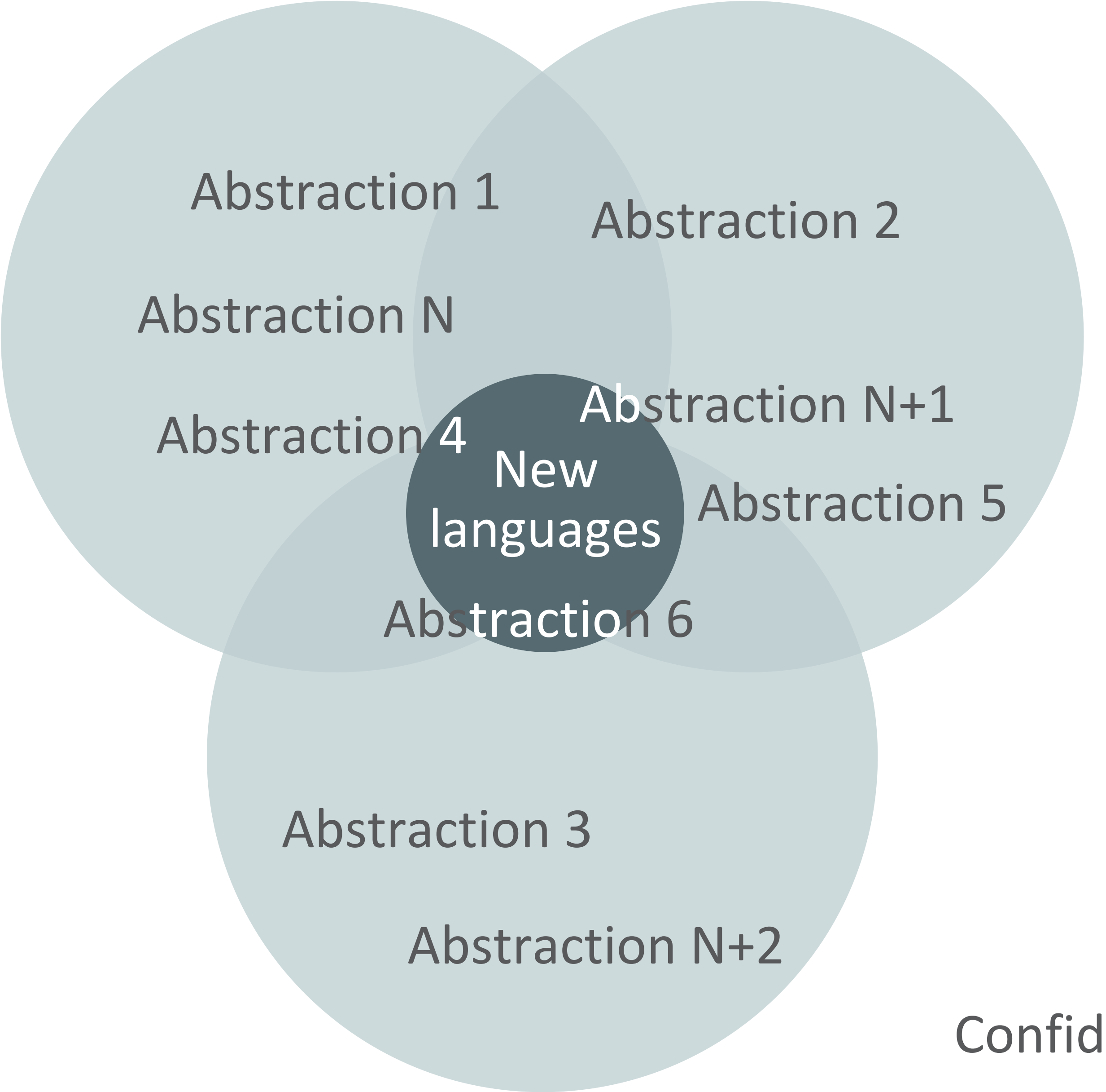
Secure Abstractions



Future

Memory Safety

Integrity



Confidentiality

ORACLE

Some Practical Issues to Consider

Issue

- Interoperability/Foreign Function Interface and properties provided by each language
- Complexity of modifying a VM

Approaches explored in the research community

- Multi-lingual compilers and runtimes, and linking types
- Compilation that preserves security properties via translations that are fully abstract

ORACLE

What If We Solved These Three Issues? What's Next?

- Other types of vulnerabilities would become prevalent, or other types of vulnerabilities are prevalent in your domain
  - Security features
    - Permissions, privileges and access control
    - Cryptographic issues
  - Poor code quality
    - Resource management
  - Time and state
    - Race conditions

ORACLE

What If We Solved These Three Issues? What's Next?

- Other types of vulnerabilities would become prevalent, or other types of vulnerabilities are prevalent in your domain
  - Security features
    - Permissions, privileges and access control
    - Cryptographic issues
  - Poor code quality
    - Resource management
  - Time and state
    - Race conditions

Rust and Pony prevent data race issues by design

ORACLE

What If We Solved These Three Issues? What's Next?

- Other types of vulnerabilities would become prevalent, or other types of vulnerabilities are prevalent in your domain
  - Security features
    - Permissions, privileges and access control
    - Cryptographic issues
  - Poor code quality
    - Resource management
  - Time and state
    - Race conditions

Rust and Pony prevent data race issues by design

- New paradigms may develop new types of issues
  - E.g., microservices – vulnerabilities or security at the edge?

ORACLE

18.5

million software developers worldwide (11M professional, 7.5M hobbyist)

http://www.idc.com, 2014 Worldwide Software Developer and ICT-Skilled Worker Estimations

ORACLE

Security is not just for expert developers


KEEP
CALM
AND
CARRY ON
PROGRAMMING

It’s time to introduce security abstractions into our language design

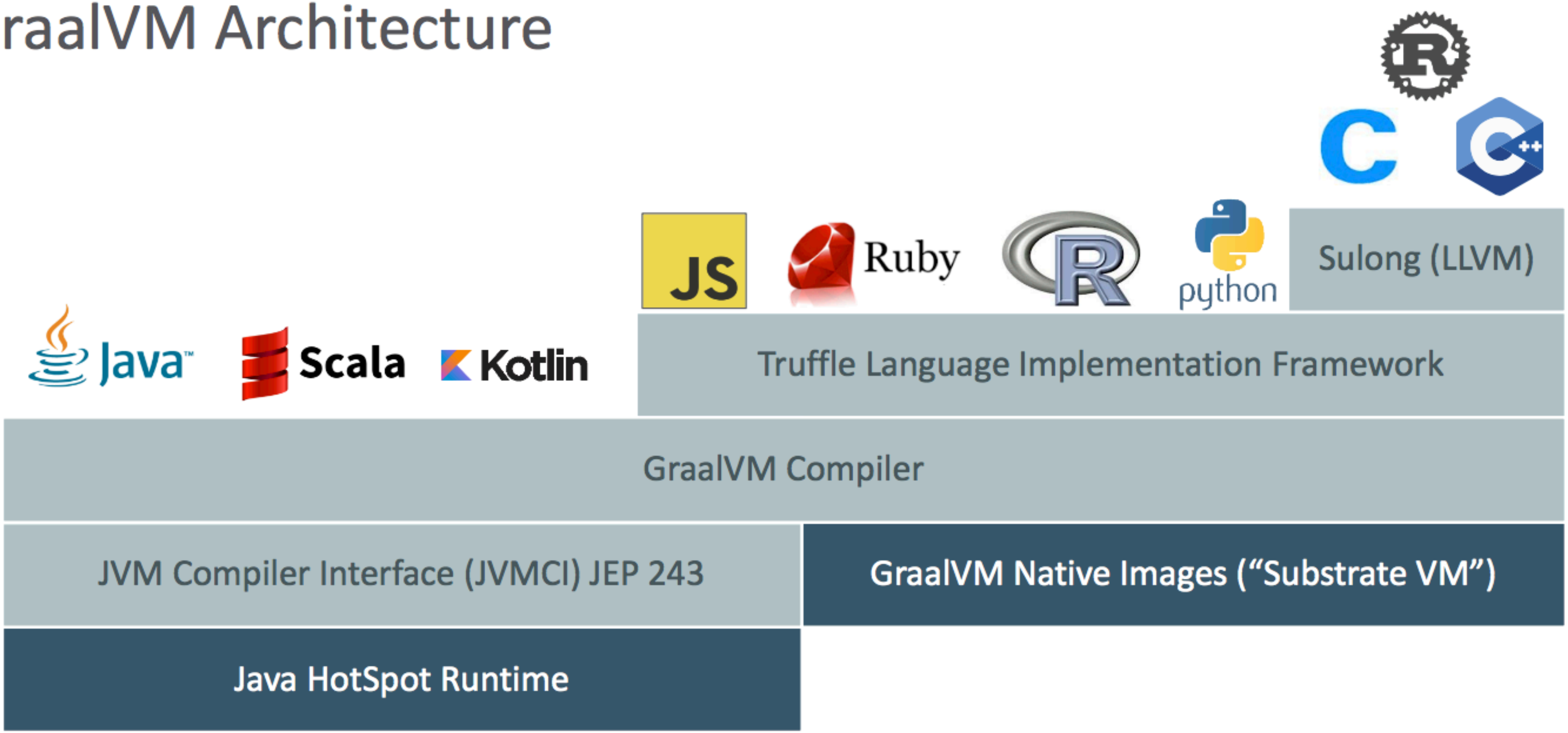
cristina.cifuentes@oracle.com
gavin.bierman@oracle.com

http://labs.oracle.com
@criscifuentes
@GavinBierman

ORACLE

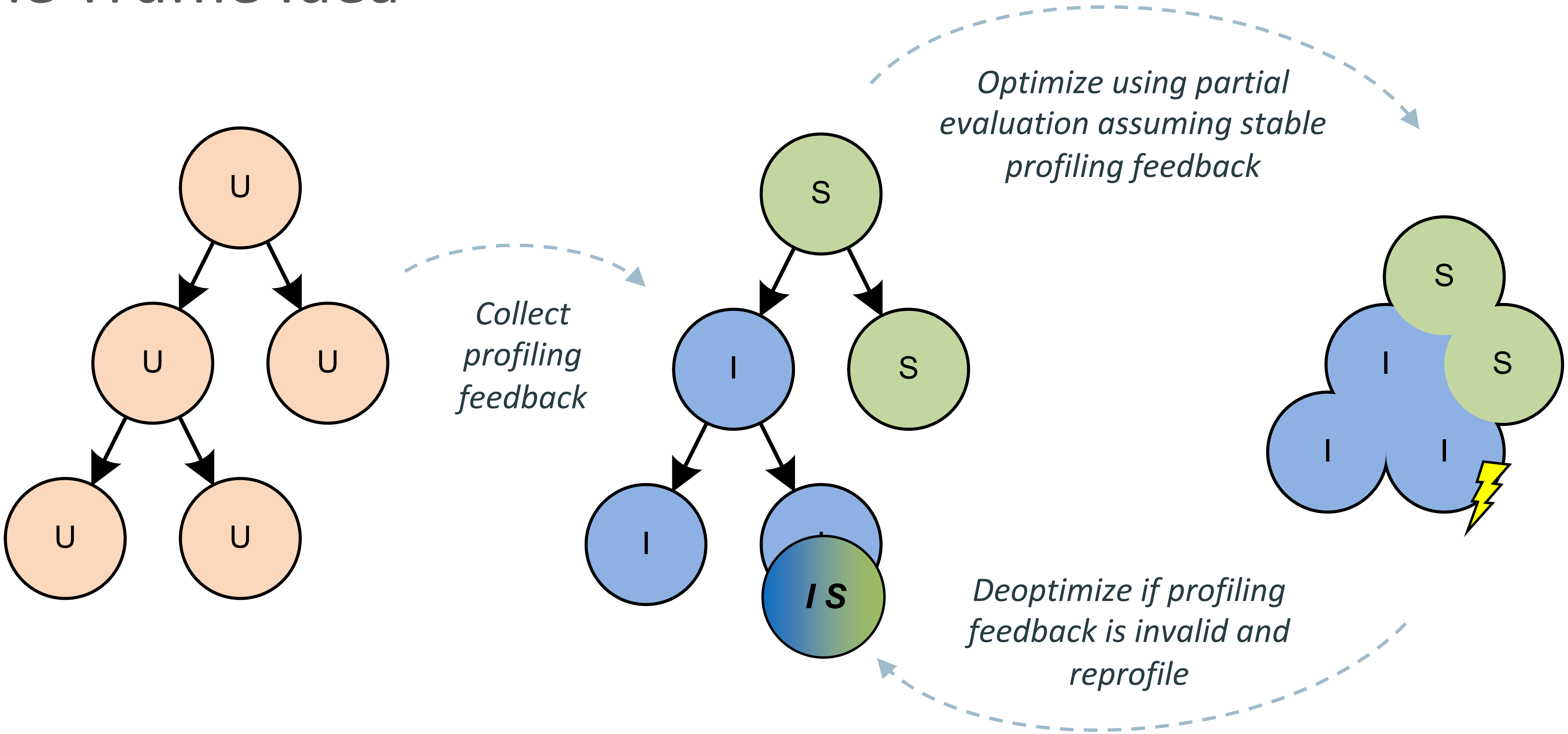
A Quick Introduction to GraalVM and Simple Language (SL)

GraalVM Architecture



ORACLE

The Truffle Idea



One VM to Rule Them All, Thomas Würthinger et al, Onward! 2013

ORACLE

SL: A Simple Language

- Language to demonstrate and showcase features of Truffle
  - Simple and clean implementation
  - Not the language for your next implementation project
- Language highlights
  - Dynamically typed
  - Strongly typed
  - Arbitrary precision integer numbers
  - First class functions
  - Dynamic function redefinition
  - Objects are key-value stores
    - Key and value can have any type, but typically the key is a String

About 2.5k lines of code

ORACLE

Types

SL Type	Values	Java Type in Implementation
Number	Arbitrary precision integer numbers	`long` for values that fit within 64 bits `java.lang.BigInteger` on overflow
Boolean	true, false	`boolean`
String	Unicode characters	`java.lang.String`
Function	Reference to a function	`SLFunction`
Object	key-value store	`DynamicObject`
Null	null	`SLNull.SINGLETON`

Null is its own type; could also be called "Undefined"

ORACLE

Syntax

- C-like syntax for control flow
  - `if`, `while`, `break`, `continue`, `return`
- Operators
  - +, -, *, /, ==, !=, <, <=, >, >=, &&, ||, ( )
  - + is defined on `String`, performs `String` concatenation
  - && and || have short-circuit semantics
  - . or [] for property access
- Literals
  - `Number`, `String`, `Function`
- Builtin functions
  - `println`, `readln`: Standard I/O
  - `nanoTime`: to allow time measurements
  - `defineFunction`: dynamic function redefinition
  - `stacktrace`, `helloEqualsWorld`: stack walking and stack frame manipulation
  - `new`: Allocate a new object without properties

SL Examples

Hello World:

```
function main() {
  println("Hello World!");
}
```

```
Hello World!
```

Strings:

```
function f(a, b) {
  return a + " < " + b + ": " + (a < b);
}

function main() {
  println(f(2, 4));
  println(f(2, "4"));
}
```

```
2 < 4: true
Type error
```

Objects:

```
function main() {
  obj = new();
  obj.prop = "Hello World!";
  println(obj["pr" + "op"]);
}
```

```
Hello World!
```

Simple loop:

```
function main() {
  i = 0;
  sum = 0;
  while (i <= 10000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

```
50005000
```

First class functions:

```
function add(a, b) { return a + b; }
function sub(a, b) { return a - b; }

function foo(f) {
  println(f(40, 2));
}

function main() {
  foo(add);
  foo(sub);
}
```

```
42
38
```

Function definition and redefinition:

```
function foo() { println(f(40, 2)); }

function main() {
  defineFunction("function f(a, b) { return a + b; }");
  foo();

  defineFunction("function f(a, b) { return a - b; }");
  foo();
}
```

```
42
38
```

ORACLE

Getting Started

- Download GraalVM Community Edition 19.0.0
  - https://github.com/oracle/graal/releases

GraalVM version used in this tutorial: graalvm-ce-19.0.0

- Install GraalVM
  - `tar -xvf graalvm-ce.tar`

- Download, install and verify Simple Language (steps 1-5 + load Maven project into your favourite IDE)
  - https://www.graalvm.org/docs/graalvm-as-a-platform/implement-language/

SL version used in this tutorial: 323876b.
git checkout 323876b

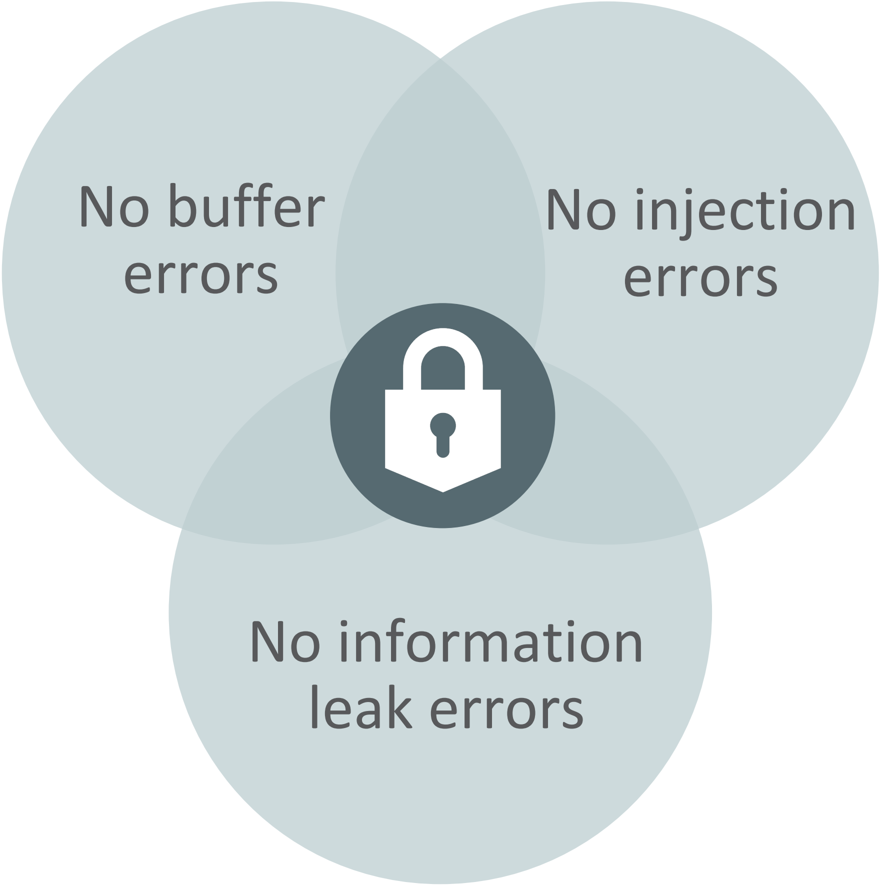
ORACLE

Program Agenda

1. Thursday 23rd May
   - What is a Secure Programming Language?
   - Quick Intro to GraalVM and Simple Language (SL)
2. Friday 24th May
   - Hands-on: Let’s add a TaintString to SL

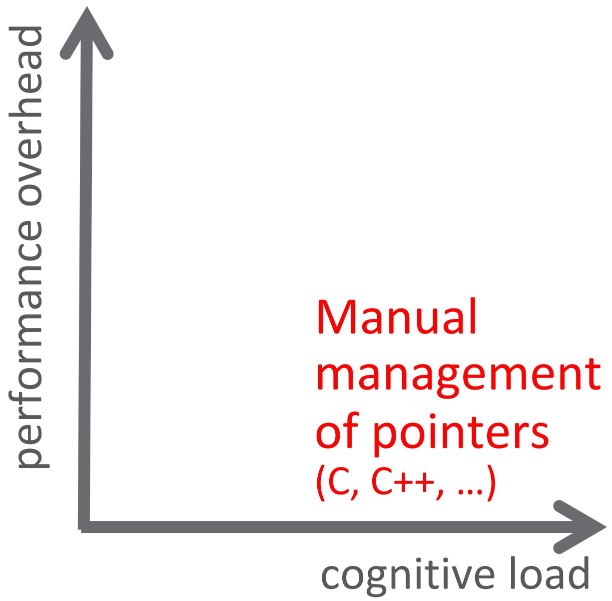
Recap: What is a Secure Language?

- One that addresses today's most common types of vulnerabilities, namely, buffer errors, injection errors, and information leak errors.

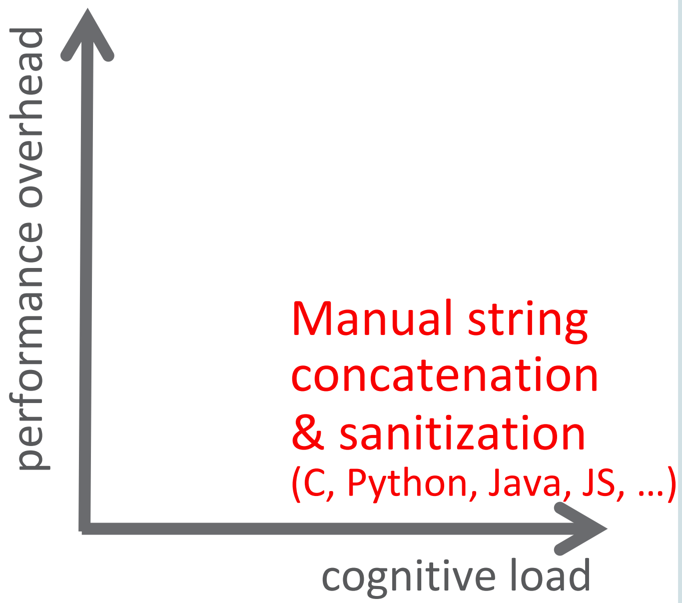


Recap: The Problem: Unsafe Abstractions

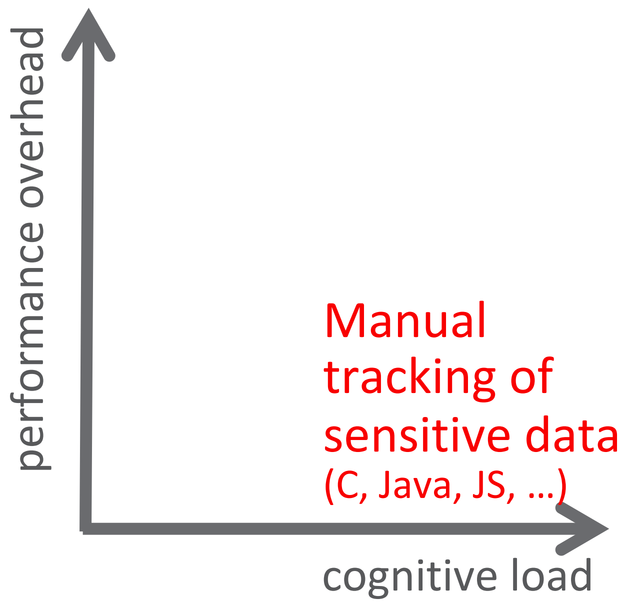
- Buffer errors



- Injections

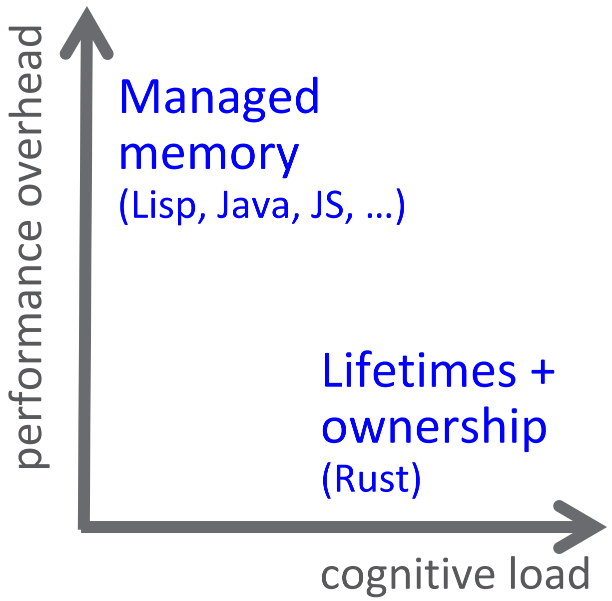


- Information leaks

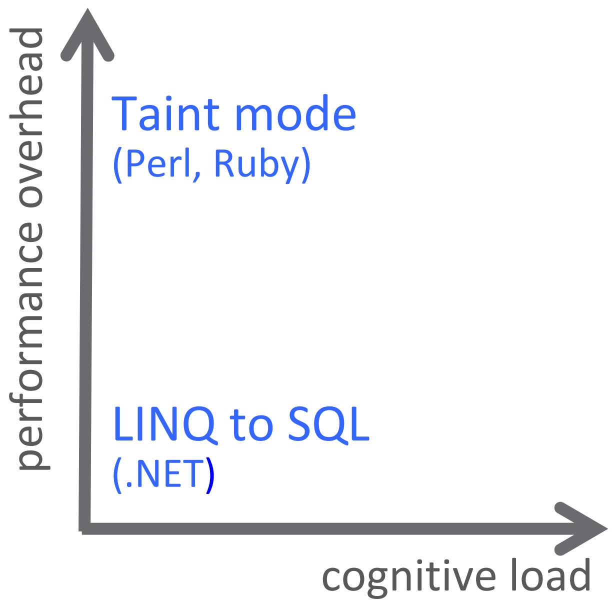


Recap: Examples of Solutions: Safe Abstractions

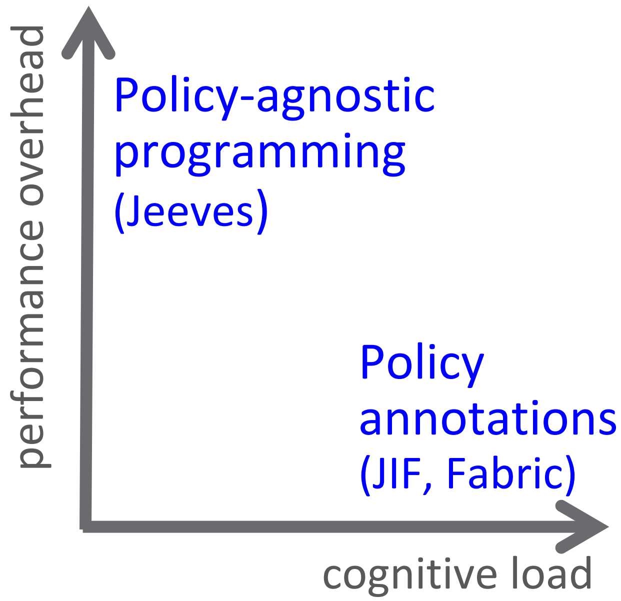
- Buffer errors



- Injections



- Information leaks



Hands-On Practice

- Focus on taint
  - Use types to introduce secure abstraction concepts
  - Use GraalVM and SL
- Task
  - Modify SL to include the type tainted string (`TaintString`) and test your implementation by adding JUnit tests
- Reflect on the pros/cons of your new language

Taint String Concepts

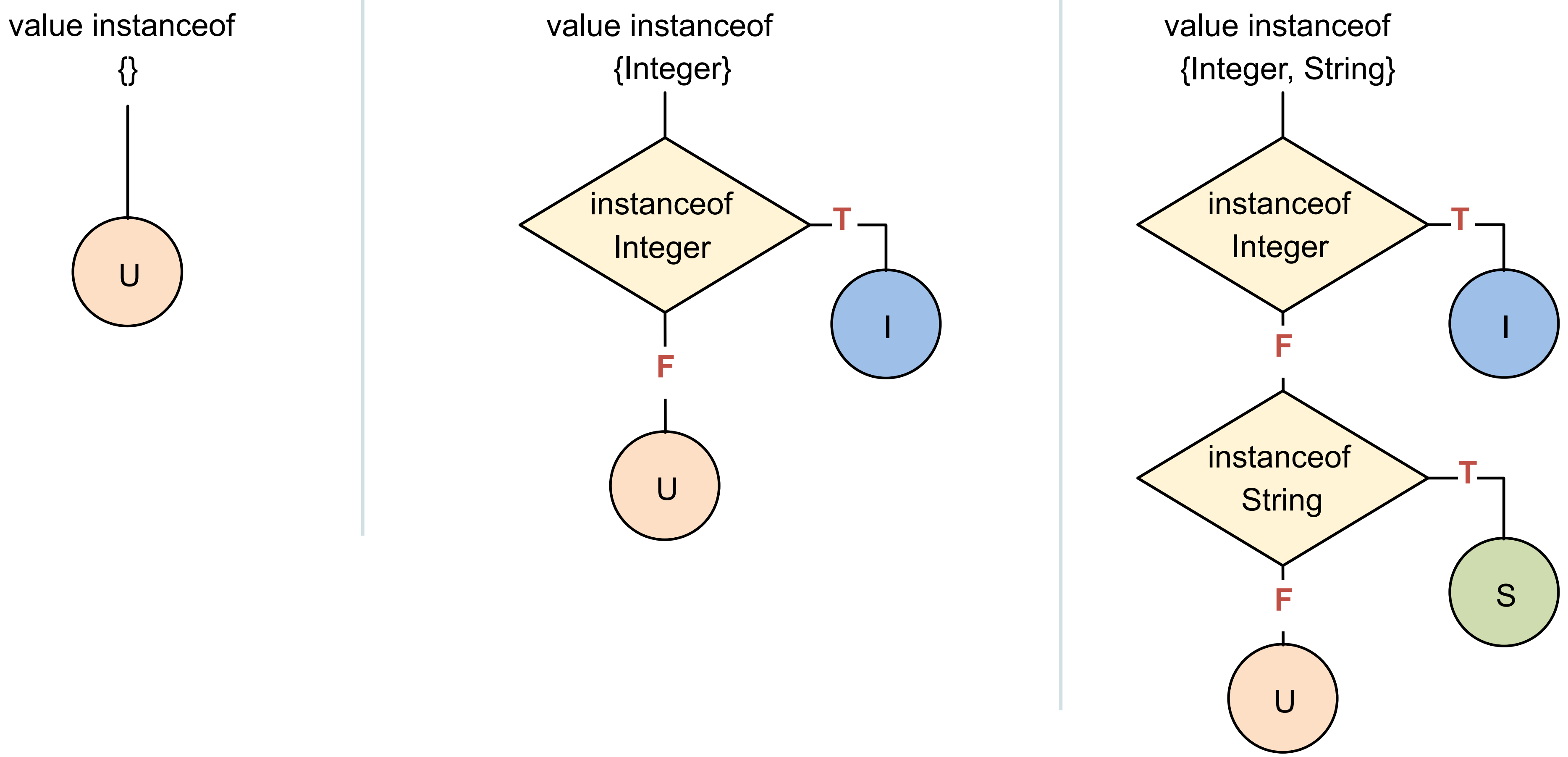
- User-input string is considered tainted
- Cannot write/print tainted string
- Tainted string can be sanitized by a specific method
- Tainted string can operate with other String and TaintString values

`TaintString` as a Type

Use Truffle DSL annotations to make changes to SL; no changes to the SL parser

- `@Specialization`
- `@CompilerDirectives.TruffleBoundary`
- `@NodeInfo`

@Specialization



Addition

```
@NodeChildren({@NodeChild("leftNode"), @NodeChild("rightNode")})
public abstract class SLBinaryNode extends SLExpressionNode { }

public abstract class SLAddNode extends SLBinaryNode {

  @Specialization(rewriteOn = ArithmeticException.class)
  protected final long add(long left, long right) {
    return ExactMath.addExact(left, right);
  }

  @Specialization
  protected final BigInteger add(BigInteger left, BigInteger right) {
    return left.add(right);
  }

  @Specialization(guards = "isString(left, right)")
  protected final String add(Object left, Object right) {
    return left.toString() + right.toString();
  }

  protected final boolean isString(Object a, Object b) {
    return a instanceof String || b instanceof String;
  }
}
```

The order of the @Specialization methods is important: the first matching specialization is selected

For all other specializations, guards are implicit based on method signature

ORACLE

Code Generated by Truffle DSL (1)

Generated code with factory method:

```
@GeneratedBy(SLAddNode.class)
public final class SLAddNodeGen extends SLAddNode {

  public static SLAddNode create(SLExpressionNode leftNode, SLExpressionNode rightNode) { ... }

  ...
}
```

The parser uses the factory to create a node that is initially in the uninitialized state

The generated code performs all the transitions between specialization states

Code Generated by Truffle DSL (2)

```
@GeneratedBy(methodName = "add(long, long)", value = SLAddNode.class)
private static final class Add0Node_ extends BaseNode_ {
  @Override
  public long executeLong(VirtualFrame frameValue) throws UnexpectedResultException {
    long leftNodeValue_;
    try {
      leftNodeValue_ = root.leftNode_.executeLong(frameValue);
    } catch (UnexpectedResultException ex) {
      Object rightNodeValue = executeRightNode_(frameValue);
      return SLTypesGen.expectLong(getNext().execute_(frameValue, ex.getResult(), rightNodeValue));
    }
    long rightNodeValue_;
    try {
      rightNodeValue_ = root.rightNode_.executeLong(frameValue);
    } catch (UnexpectedResultException ex) {
      return SLTypesGen.expectLong(getNext().execute_(frameValue, leftNodeValue_, ex.getResult()));
    }
    try {
      return root.add(leftNodeValue_, rightNodeValue_);
    } catch (ArithmeticException ex) {
      root.excludeAdd0_ = true;
      return SLTypesGen.expectLong(remove("threw rewrite exception", frameValue, leftNodeValue_, rightNodeValue_));
    }
  }

  @Override
  public Object execute(VirtualFrame frameValue) {
    try {
      return executeLong(frameValue);
    } catch (UnexpectedResultException ex) {
      return ex.getResult();
    }
}
```

The generated code can and will change at any time

@Specialization

`public @interface Specialization`

- Defines a method of a Node subclass to represent one specialization of an operation
  - Multiple specializations can be defined
  - Inputs are defined through a method signature and the annotation attributes
    - # parameters <= # nodes in the `@NodeChild` annotation declared for the enclosing operation node
  - Semantics are defined using the body of the annotated Java method

https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html

@Specialization: Annotation type specialization

`public @interface Specialization`

- Kinds of input values are declared using guards. Types of guards:
  - **Type**: optimistically assume the type of an input value. Object by default
  - **Expression**: optimistically assume the return type value is `true`. If `false`, the specialization is no longer applicable and the operation is re-specialized. Guard expressions are declared using the `Specialization.guards()` attribute.
  - **Event**: trigger re-specialization in case an exception is thrown in the specialization body. A list of such exceptions is declared using the `Specialization.rewriteOn()` attribute.
  - **Assumption**: optimistically assume that the state of an `Assumption` remains true. Assumptions are assigned using the `Specialization.assumptions()` attribute.

https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html

ORACLE

@CompilerDirectives.TruffleBoundary

public static @interface CompilerDirectives.TruffleBoundary

- Marks a method that is considered as a boundary for Truffle partial evaluation
  - For functions not designed for PE (e.g., JDK, external libraries, etc)
  - For logic that is difficult to partially evaluate

https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/CompilerDirectives.TruffleBoundary.html

ORACLE

@TruffleBoundary – Slow Path Annotation

```
public abstract class SLPrintlnBuiltin extends SLBuiltinNode {

  @Specialization
  public final Object println(Object value) {
    doPrint(getContext().getOutput(), value);
    return value;
  }

  @TruffleBoundary
  private static void doPrint(PrintStream out, Object value) {
    out.println(value);
  }
}
```

When compiling, the output stream is a constant

Why `@TruffleBoundary`? Inlining something as big as `println()` would lead to code explosion

@NodeInfo

`public @interface NodeInfo`

- Annotation for providing additional information on Nodes
- Optional elements
  - `String shortName`: short name representing the Node that can be used for debugging
- In SL
  - the `shortName` is looked up and a specialisation that executes that Node is added to the SL function registry, so that when someone calls a function with that name, the builtin is there

```
@NodeInfo(shortName = “println”)
public abstract class SLPrintlnBuiltin extends
SLBuiltinNode {
  @Specialization
  public long println(long value) { ... }
  @Specialization
  public boolean println(boolean value) { ... }
  ...
}
```

https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/NodeInfo.html

ORACLE

Add `TaintString` to SL

Functionality

- Read and write tainted strings
  - `readln()`, `println()`
- Concatenate tainted strings and strings
  - `add()`
- Compare equality of two tainted strings
  - `equal()`
- Sanitise tainted string
  - `sanitize()`

Create `TaintString`: Implement as a Wrapper to SL String

`com.oracle.truffle.sl.runtime.TaintString`

- Methods
  - `public TaintString(String)` constructor
  - `public String getValue()`
  - `public int compareTo(TaintString)`: compare two strings lexicographically

Hint: use `SLBigNumber.java` as a guide. `SLBigNumber` wraps `BigInteger` and specializes various methods

Compile often, i.e., `mvn compile`

Create SanitizeTaintStringBuiltin

`com.oracle.truffle.sl.builtins.SanitizeTaintStringBuiltin`

- Declare your sanitization methods, e.g.,
  - `public String sanitize(TaintString)`

Hint: use `SLNanoTimeBuiltin.java` as a guide

Hint: add a NodeInfo annotation

Compile and check that SanitizeTaintStringBuiltinFactory was generated in `target/generated-sources/annotations/com.oracle.truffle.sl/builtins/`

ORACLE

Modify SLContext

`com.oracle.truffle.sl.runtime.SLContext`

- Register your builtin methods by adding your `SanitizeTaintStringBuiltinFactory` to `installBuiltins()`

Modify SLReadlnBuiltin

`com.oracle.truffle.sl.builtins.SLReadlnBuiltin`

- Modify `readln()` to return a tainted string when reading from stdin
  - `TaintString readln(SLContext)`

Modify SLPrintlnBuiltin

`com.oracle.truffle.sl.builtins.SLPrintlnBuiltin`

- Modify `SLPrintlnBuiltin` to have a `println()` method that throws an `SLException` when passed a tainted string, as tainted strings cannot be written to stdout
  - `public void println(TaintString, SLContext)`

Hint: order of specializations matters

Compile: mvn compile

Modify SLAddNode

`com.oracle.truffle.sl.nodes.expression.SLAddNode`

- Add new methods to add/concatenate tainted strings
  - `protected TaintString add(TaintString left, TaintString right)`
  - `protected TaintString add(TaintString left, String right)`
  - `protected TaintString add(String left, TaintString right)`

Hint: order of specializations matters

Modify SLEqualNode

`com.oracle.truffle.sl.nodes.expression.SLEqualNode`

- Add new method to test equality of two tainted strings
  - `protected boolean equal(TaintString left, TaintString right)`

Compile: **`mvn compile`**

`Build ‘sl’ executable: mvn -Dmaven.test.skip=true package OR mvn package`

ORACLE

Add JUnit Tests

`language/tests/TaintStringTests`

- Add your tests
  - `mytest.sl`: SL test file
  - `mytest.input`: any input to `mytest.sl`
  - `mytest.output`: expected output when running `mytest.sl`
- Functionality to test
  - `readln`
  - `println`
  - `sanitize`
  - add (+)
  - `equal`

Run one test manually:
`./sl language/tests/TaintStringTests/mytest.sl`

Add JUnit Tests

`language/tests/TaintStringTests`

- Run regression test suite
  - Turn off tests for SL instrumentation: in `com.oracle.truffle.sl.test.SLInstrumentTest`, add `@Ignore` prior to the class

Run your tests and regression test suite:

`mvn test`

Debug failing tests:

`mvn –X test`

Refer to relevant `language/target/surefire-reports/<report>` for more information

Gotcha's

- SL's `String` implements a small subset of Java's `String`
- Builtin's use the convention of post-pending `Builtin` to their name in order to create the `BuiltinFactory`
- Specialization order matters
- `println` takes one string, not a comma-separated list of strings
- Trying to use internal TaintString methods for SL (user) development

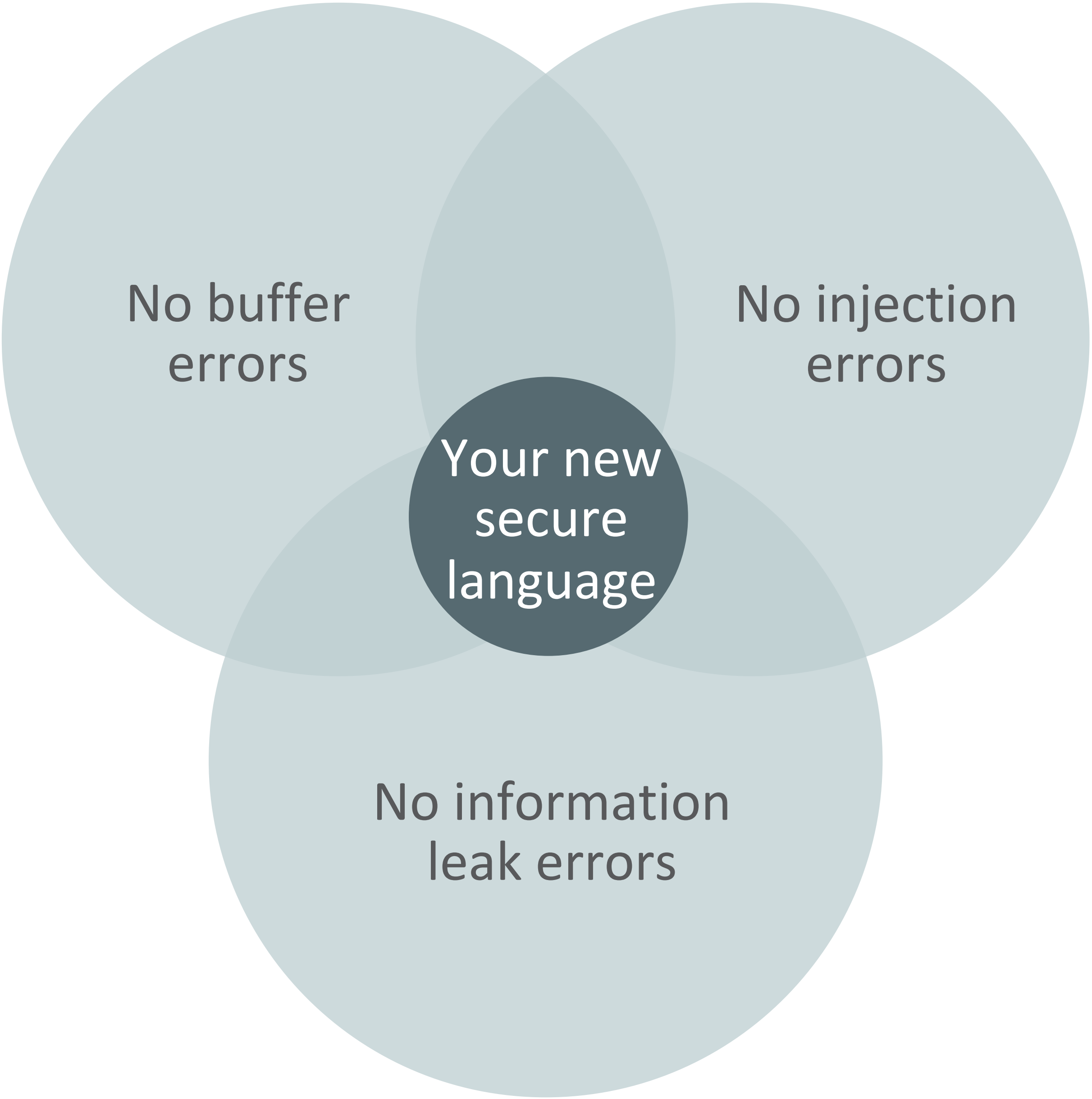
Reflect on Pros/Cons of `TaintString` in SL

- What methods are useful for sanitisation?
- What impact does `TaintString` have on the use of libraries?
- What about interoperability with other languages?
- How can you provide prevention of different types of vulnerabilities through `TaintString` concepts? E.g., for XSS and SQLi?

Homework: Hands-On Practice 2

- Focus on information leak
  - Use types to introduce secure abstraction concepts
  - Use GraalVM and SL
- Task
  - Time permitting, add the concept of leaking sensitive data on `String` and/or `Integer` (`SensitiveString`, `SensitiveInteger`)
- Reflect on the pros/cons of your new language

Concluding Remarks


No buffer errors
No injection errors
Your new secure language
No information leak errors

Internships and Postdoc Opportunities: Program Analysis (static & dynamic)

http://labs.oracle.com/locations/australia (Careers tab)

Email: cristina.cifuentes@oracle.com

Internships and Permanent Opportunities: GraalVM (optimisations, tooling, GC/runtime)

Email: thomas.wuerthinger@oracle.com

ORACLE

cristina.cifuentes@oracle.com

http://labs.oracle.com
@criscifuentes

ORACLE

Integrated Cloud

Applications & Platform Services

ORACLE