

ORACLE

What is a Secure Programming Language?

Cristina Cifuentes and Gavin Bierman

Oracle Labs

23rd January 2020

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.

5899

exploited vulnerabilities due to
buffer errors (2013-2017)



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

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>

53%

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

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

* Based on NIST's top vulnerabilities

53%

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

(labeled) errors, vulnerabilities in NVD, errors, information leak

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

* Based on NIST's top vulnerabilities

2018-2019



Latest breaking news

5970

exploited vulnerabilities due to
injection errors (2018-2019)

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

5970

exploited vulnerabilities due to
injection errors (2018-2019)

4651

exploited vulnerabilities due to
buffer errors (2018-2019)

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

5970

exploited vulnerabilities due to **injection errors** (2018-2019)

4651

exploited vulnerabilities due to **buffer errors** (2018-2019)

1980

exploited vulnerabilities due to **information leak** (2018-2019)

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

45%

(labeled) exploited vulnerabilities in NVD were injections, buffer errors and information leak (2018-2019)

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

45%

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

(labeled) errors in NVD
vulnerabilities
conditions, buffer
information leak (2018-

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

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>

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>

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>

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>

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>

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>

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>

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

Prevent
buffer errors

Java, C#,
JavaScript

Ruby,
JS 1.1

PHP
(library)

Prevent
injections

Today's Status –
Mainstream
Languages

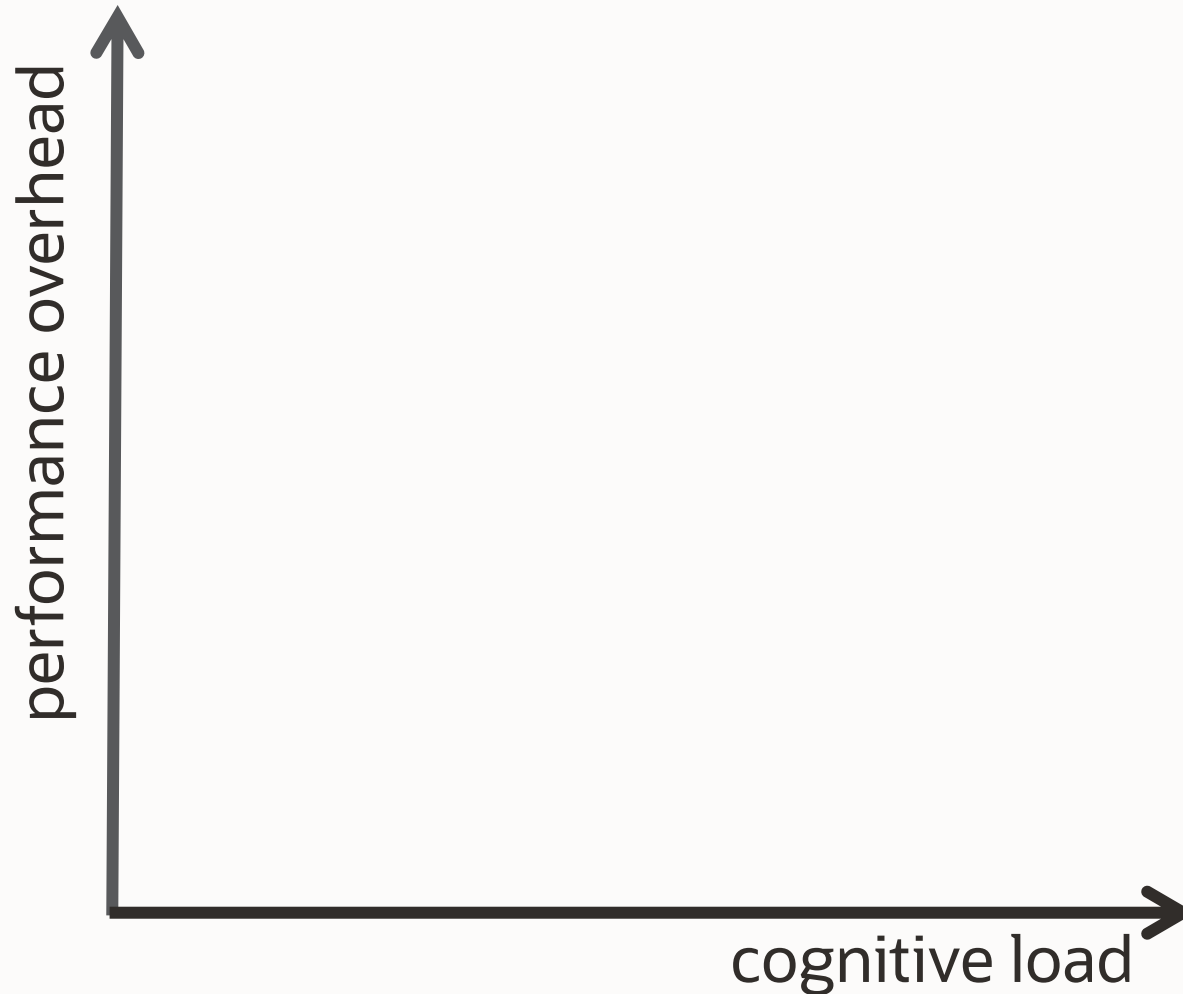


Prevent
information leaks

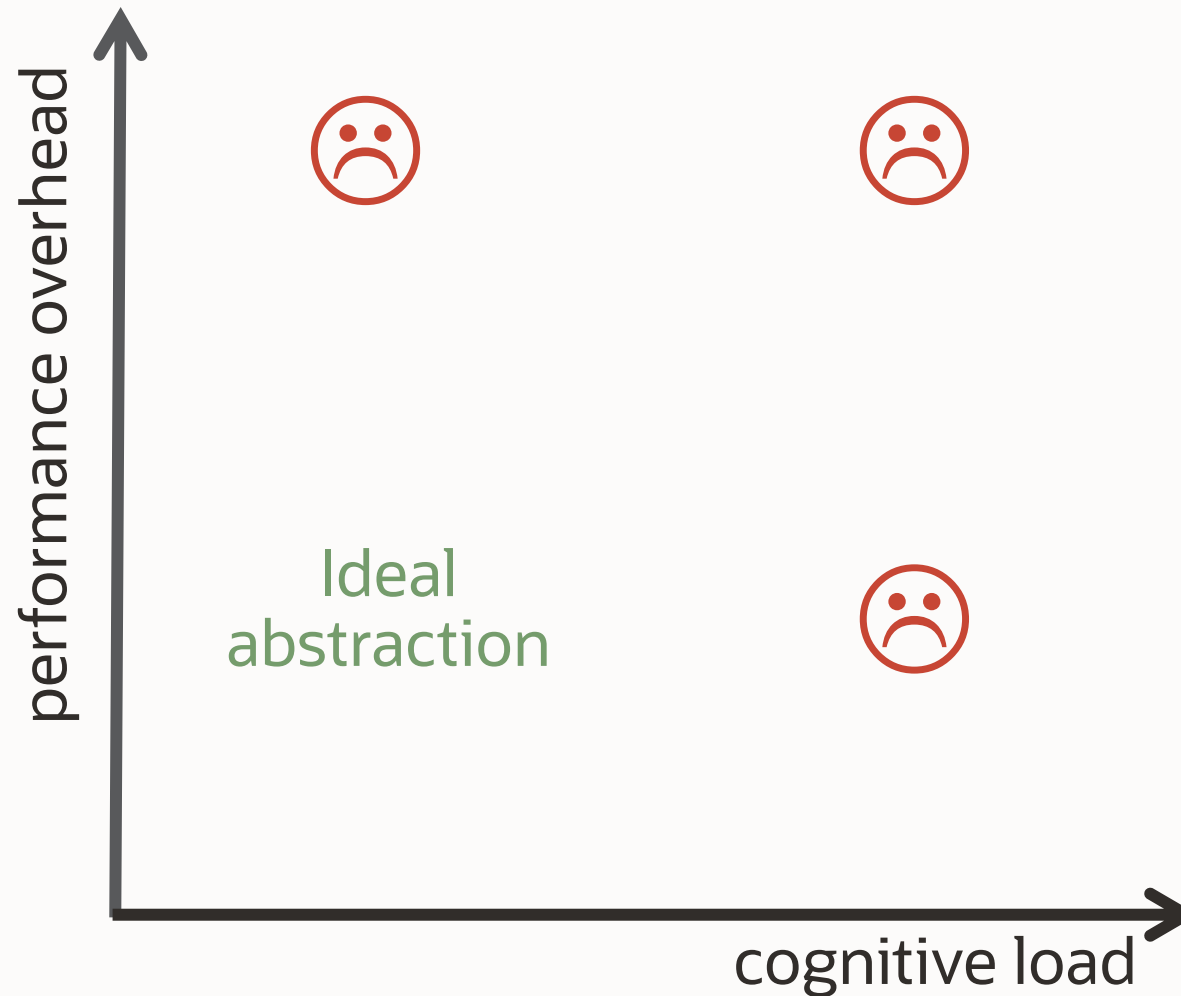
A Secure Language is One that Provides First-class Support for These Three Categories



What to Consider when Talking about Abstractions

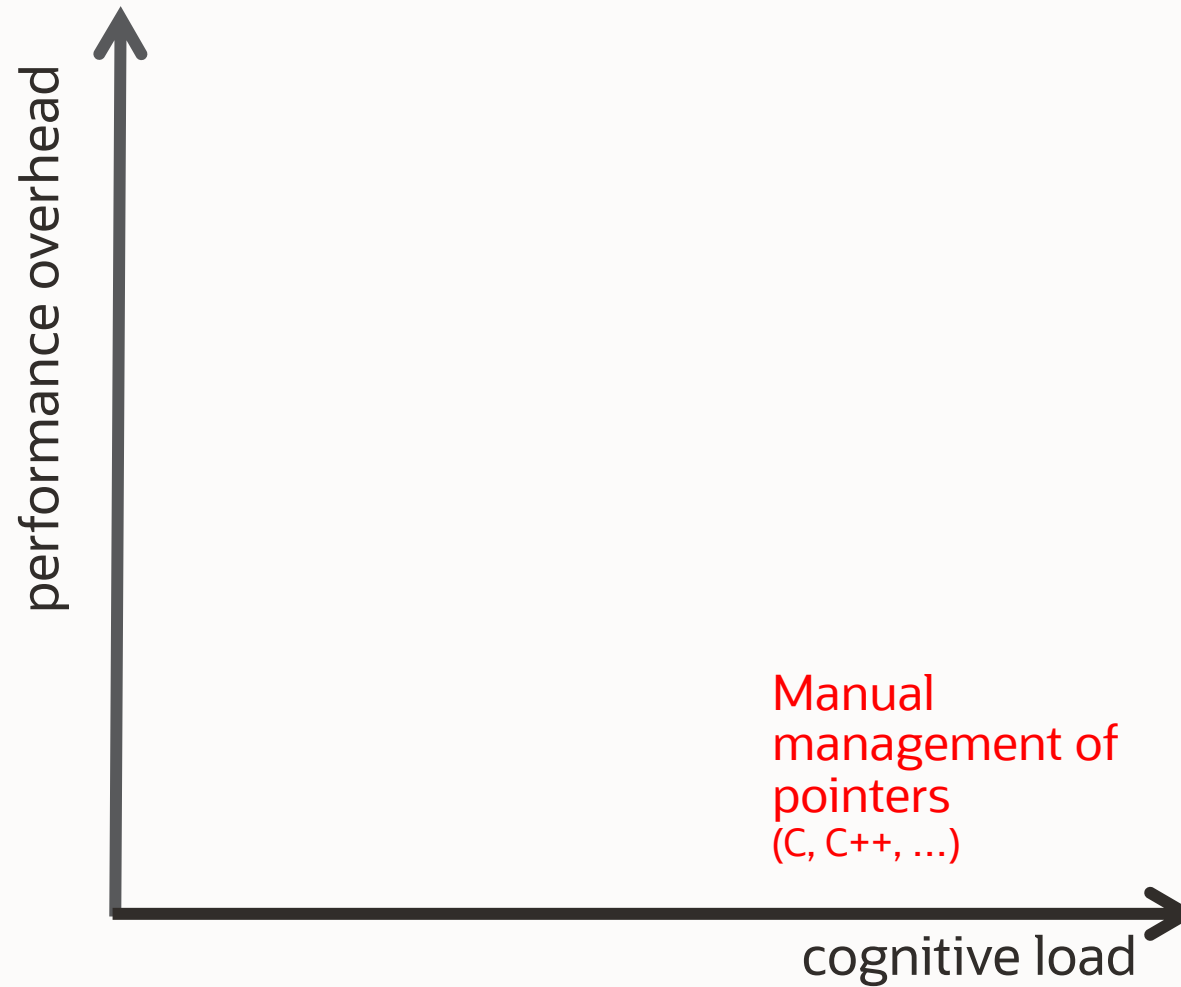


What to Consider when Talking about Abstractions

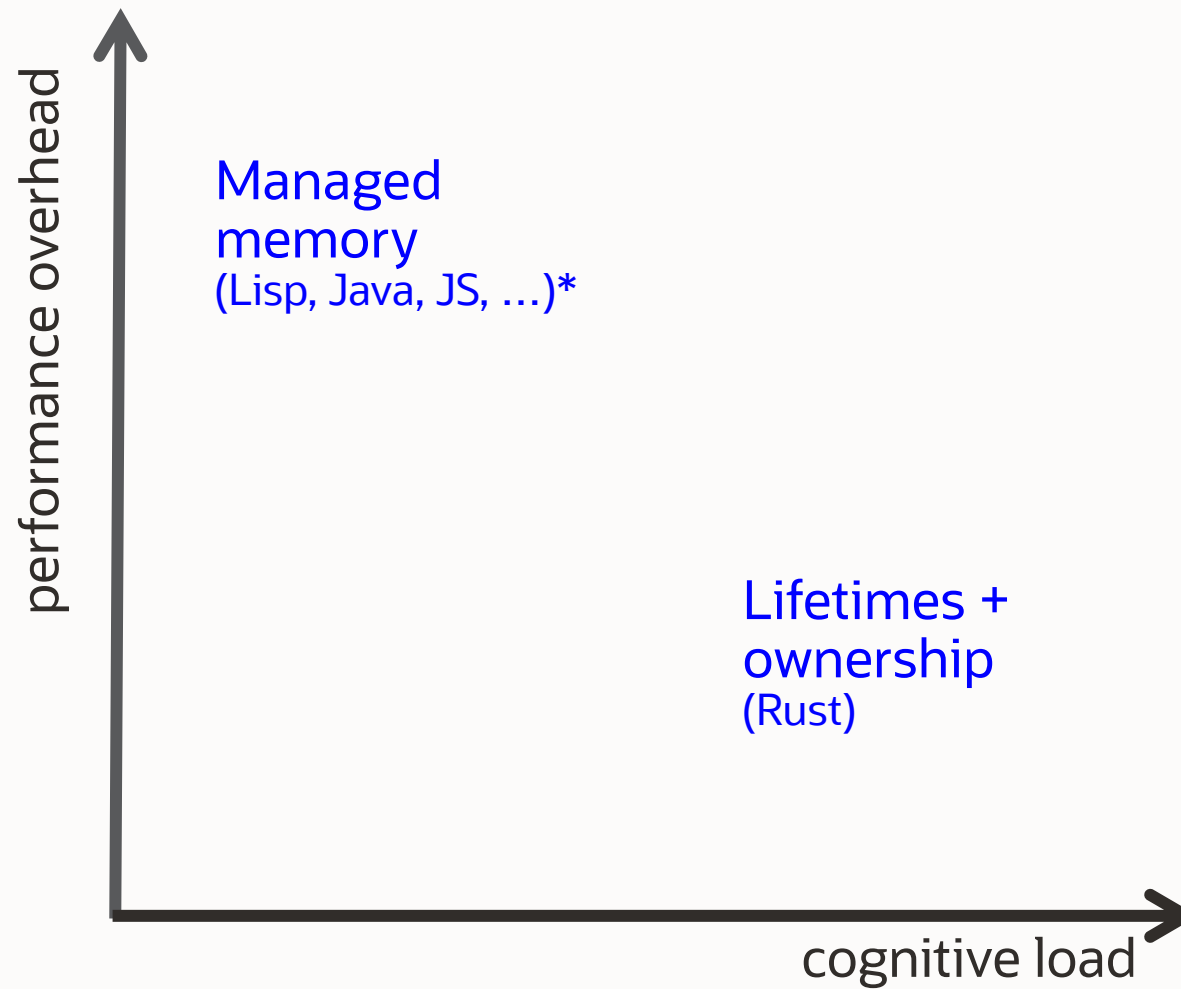


Language Support Addressing Buffer Errors

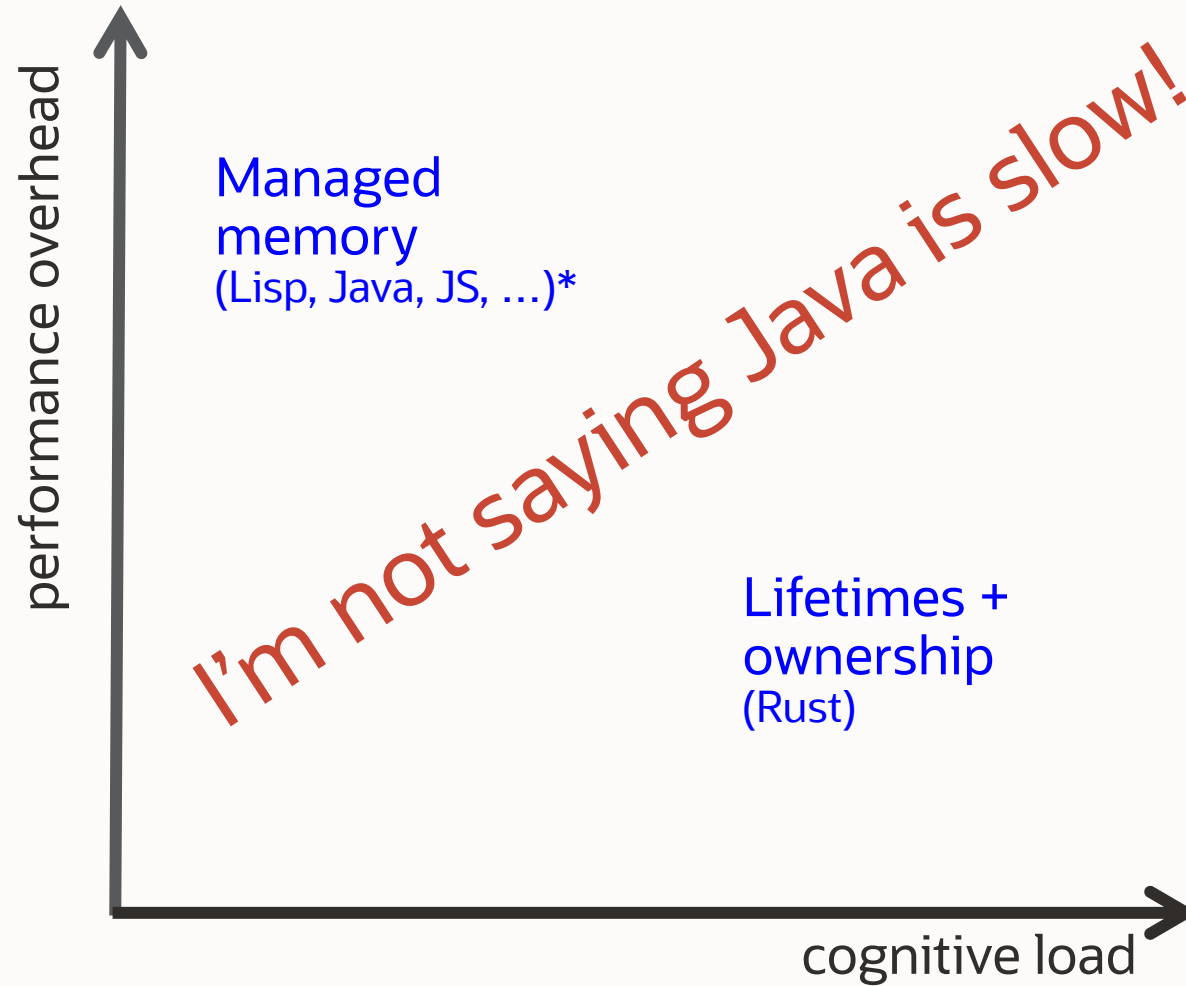
Buffer Errors – The Problem: **Unsafe Abstraction**



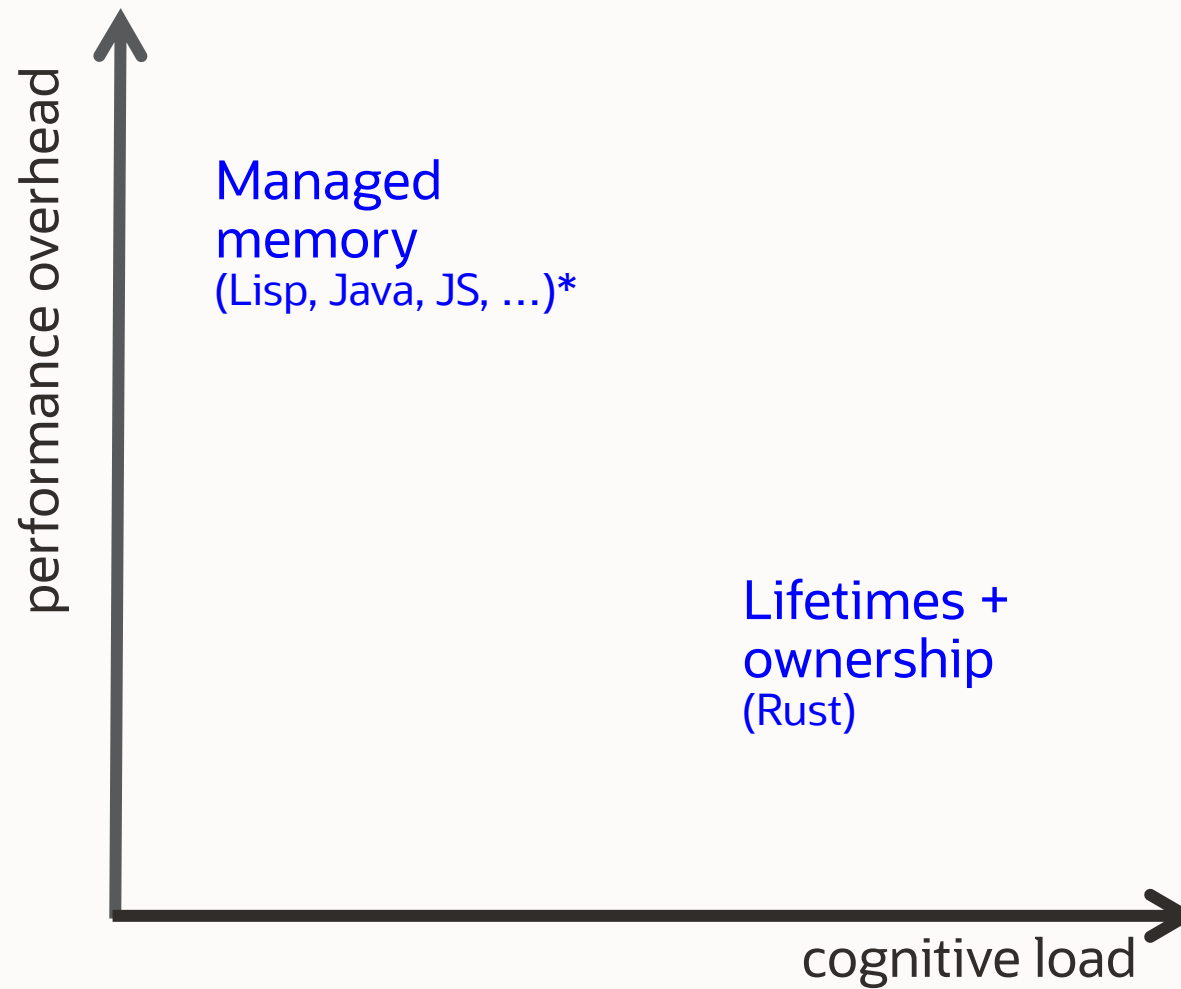
Buffer Errors – Solutions: Safe Abstractions



Buffer Errors – Solutions: Safe Abstractions



Buffer Errors – Solutions: **Safe Abstractions**



Memory Safety Through The Rust Language

Rust Language

<https://www.rust-lang.org>



Rust is a systems programming language that runs fast, prevents memory corruption, and guarantees memory and thread safety

No garbage collection

Two new concepts in the type system

- Ownership
- Lifetime / borrowing
 - Shared borrow (`&T`) – cannot mutate it
 - Mutable borrow (`&mut T`) – cannot alias it

Lifetimes

```
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. _____
```

shared borrow

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;  
  
        println!("borrow4: {}", borrow4);  
    }  
}
```

```
// error[E0502]: cannot borrow `i` as immutable  
// because it is also borrowed as mutable
```

Rust Memory Safety Guarantees

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

Rust's Unsafe Features

Must opt-in to use them

Calling foreign code

Calling unsafe code

Dereferencing a raw pointer

Rust

Ownership and lifetimes allow for memory safety guarantees

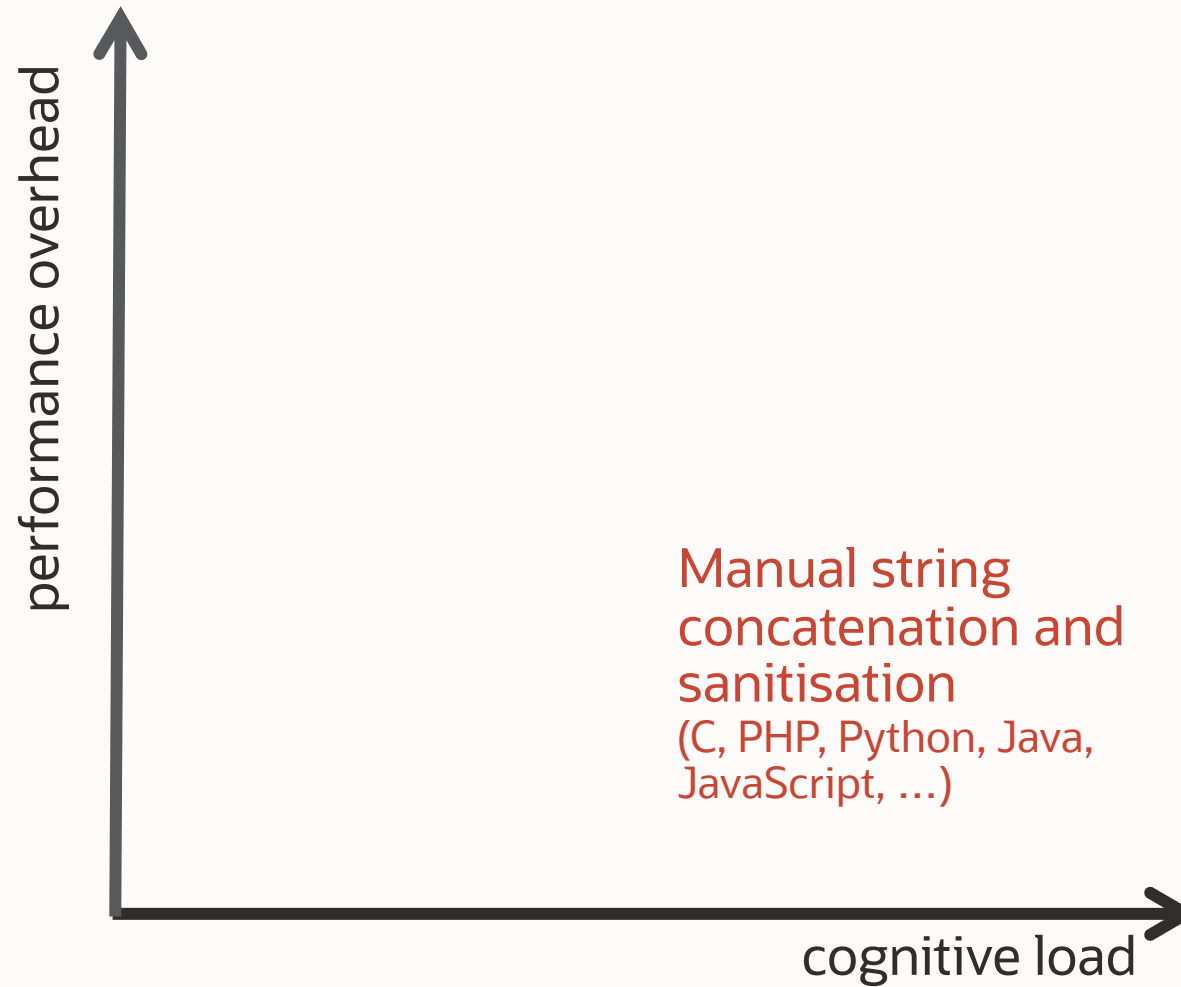
- No buffer overflows, no null pointer dereferences, no double freeing memory, no stale pointers, no data races, no arithmetic overflows

Unsafe code

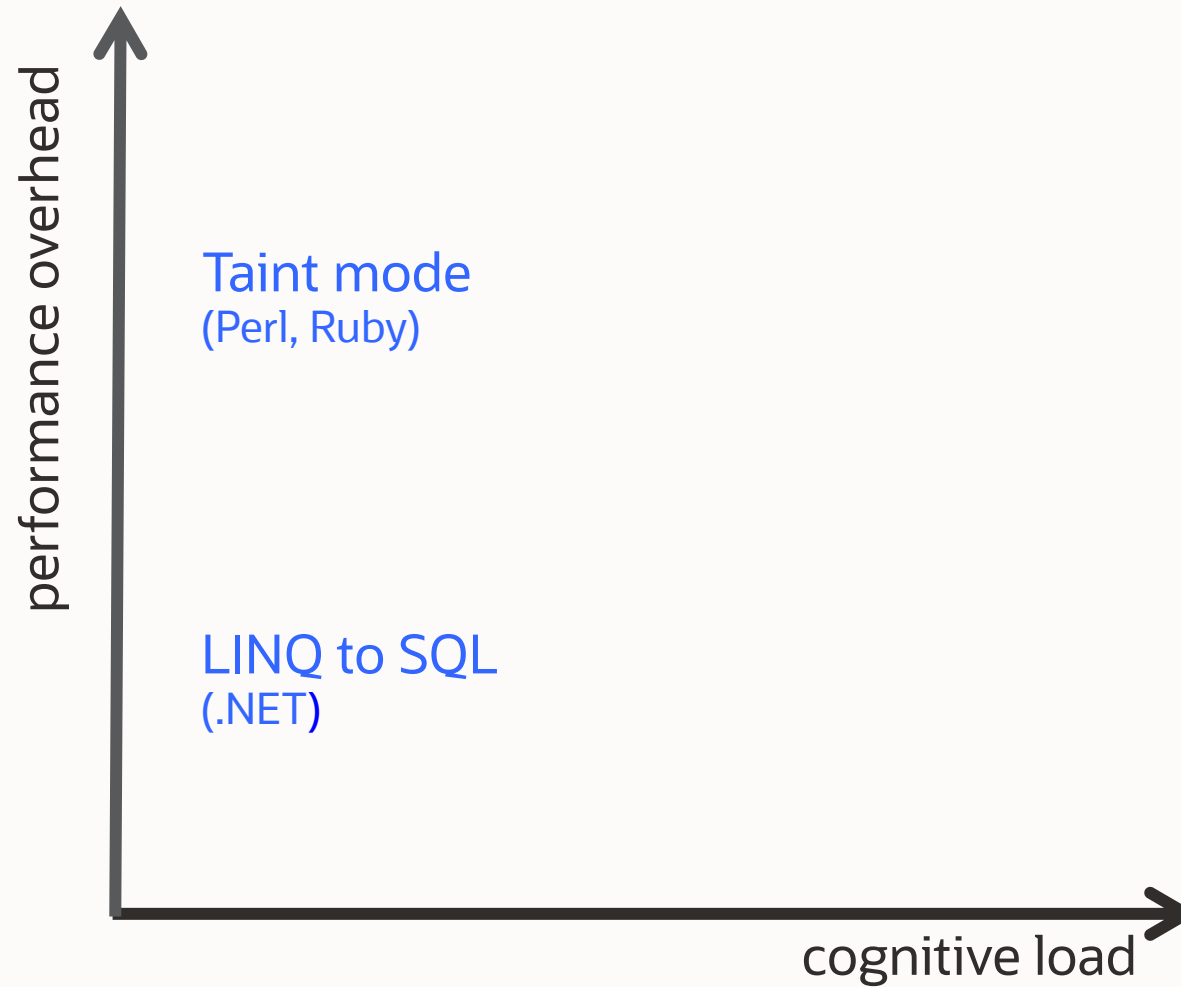
- Needed to interface with native C code
- To implement low-level libraries (e.g., Rust's own libraries, a user's library)
- Unsafe code can void memory safety guarantees

Language Support Addressing Injection Errors

Injections – The Problem: **Unsafe Abstraction**



Injections – Solutions: **Safe Abstractions**



Avoiding Injection Attacks with Perl and Ruby

Perl Language

<https://www.perl.org>



Perl is a rapid-prototyping programming language

Taint Mode – concept introduced in Perl 3, 1989

- Tracks external/input (tainted) values
- Runtime implements taint checks

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

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

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 safe level 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

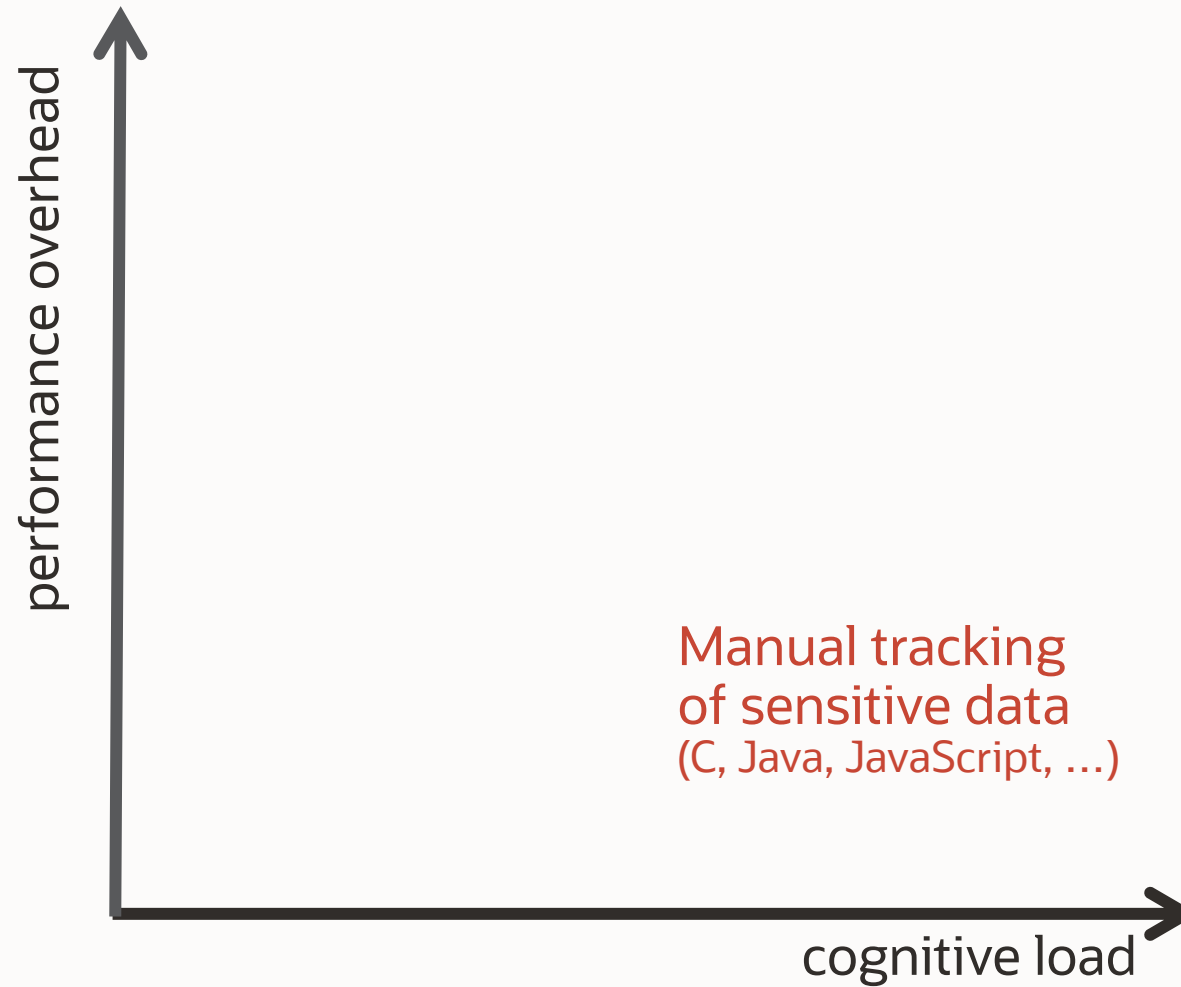
Extends Perl's taint mode to track direct data flows through SAFE modes 1-3
Programmatic taint/untaint methods

Cannot track XSS as do not track taint to `print` and `syswrite`
Do not track indirect/implicit data flows

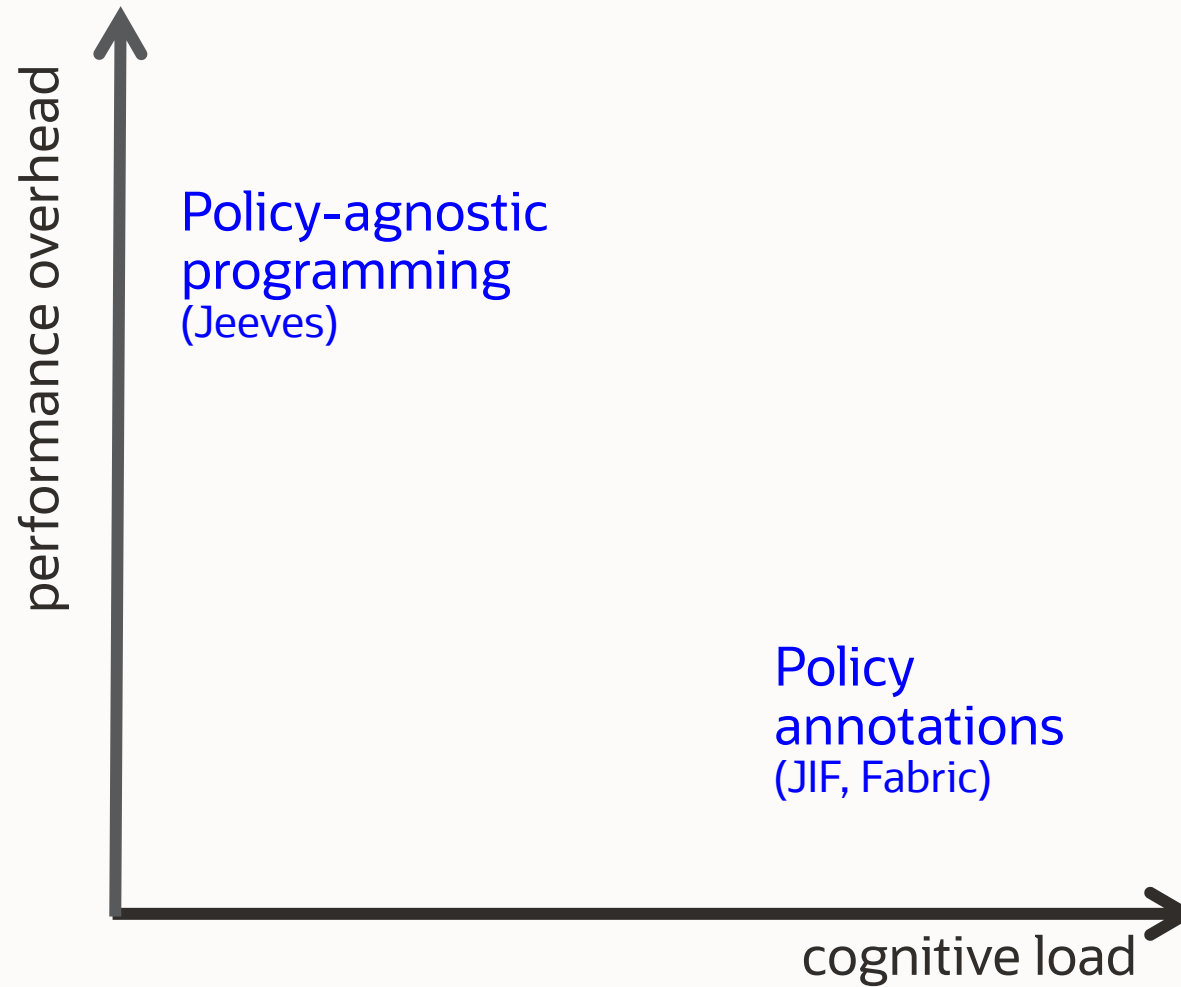
Not suitable to prevent today's Cloud injections

Language Support Addressing Information Leak Errors

Information Leaks – The Problem: **Unsafe Abstraction**



Information Leaks – Solutions: **Safe Abstractions**



Avoiding Information Leaks with Policy-Agnostic Programming

Policy-Agnostic Programming

Jeeves language and Jacqueline web framework

<https://github.com/jeanqasaur/jeeves>

Goal: factor out information flow policies so that policies can be high-level and programs can be policy-agnostic

Main concept – Faceted values

- Tracked by the language runtime
- Extended to DBs through web framework that includes relational operators
- Jeeves – extends Python with faceted values

“Preventing Information Leaks with Policy-Agnostic Programming”, Jean Yang, PhD thesis, Sep 2015

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

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>

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

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, therefore, 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 management system (as policies resolved once)

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

Recap

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

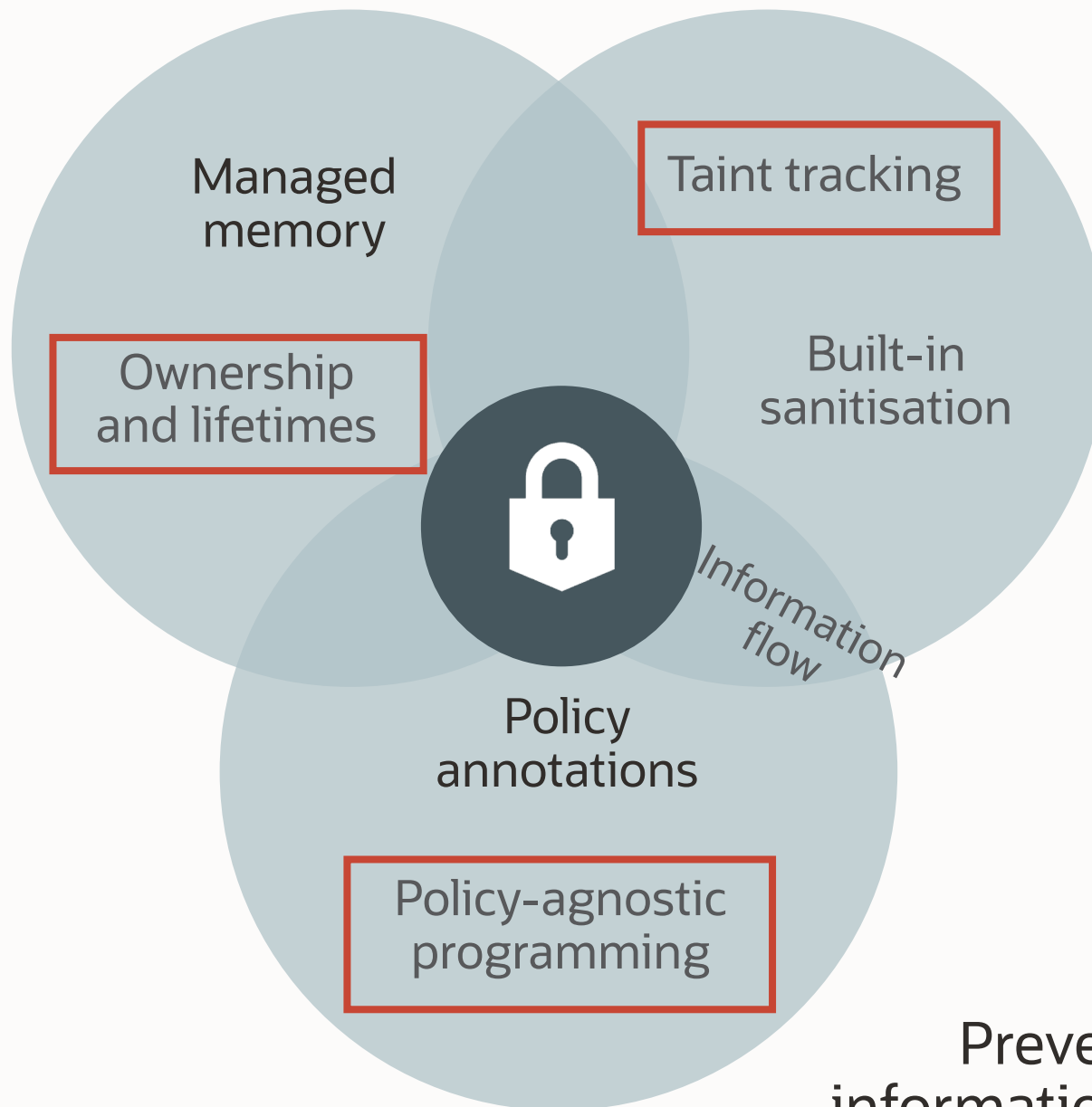


Today's mainstream languages do not support our developers in writing secure code that is free of buffer errors, injections, or information leaks.

Our mainstream languages are not
secure languages.

Secure Abstractions

Prevent buffer errors

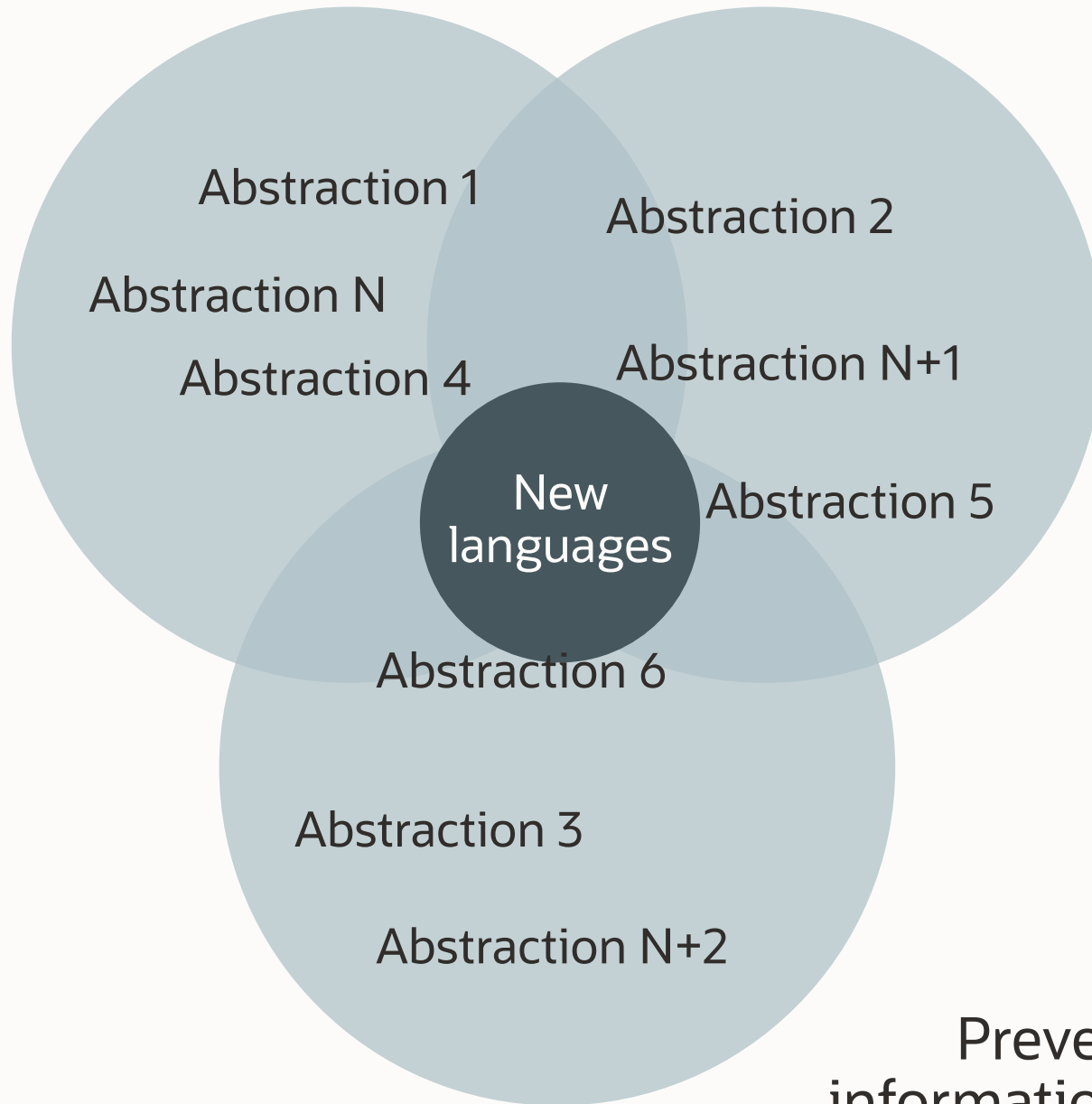


Prevent injections

Prevent information leaks



Prevent
buffer errors



Prevent
injections

Future

Prevent
information leaks



Prevent
buffer errors

Prevent
injections

Your new
secure
language

Future

Prevent
information leaks



In the **future**, what if we had one or more
secure programming languages?
Are we “done” with vulnerabilities?

Other Vulnerabilities in the Realm of Programming Languages

Resource management errors

- Use after free
- Double free
- Memory corruption
- Type casting error
- Worker termination error

Race condition errors

- Concurrent execution using shared resource without proper synchronization
- Time-of-check, Time-of-use

Recent Projects That Prevent Aspects of Resource Management Errors

Various works including Rust

- Resource management via lifetimes – no double freeing memory nor use after free issues

Project Verona

Matthew Parkinson et al, Microsoft, 2019

- 3 core ideas:
 - Data race freedom, Concurrent owners, and Linear regions
- Resource management via linear regions
 - Can't access memory outside the region
 - Can't access memory once region has been freed
- Nov 2019 first public presentation: <https://vimeo.com/376180843>
- Open sourced Jan 2020: <https://github.com/microsoft/verona>

Recent Projects That Prevent Data Race Errors

Various works including Rust and Project Verona

Pony

Sylvan Clebsch et al, Imperial College, 2014

- Actors for concurrency
- Data race free type system
- Reference capabilities attached to the path to an object
- Memory safe
- <https://github.com/ponylang/ponyc>

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 [1], and linking types [2].
Project Verona – compartmentalisation for legacy resources
- Compilation that preserves security properties via translations that are fully abstract

[1] Thomas Würthinger et al. Practical partial evaluation for high-performance dynamic language runtimes, PLDI 2017

[2] Daniel Patterson and Amal Ahmed. Linking types for multi-language software: Have your cake and eat it too. SNAPL 2017

18.5

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

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

Security is not just for expert developers.



**KEEP
CALM
AND
CARRY ON
PROGRAMMING**

It's time to introduce security
abstractions into our language
design.

ORACLE

crisrina.cifuentes@oracle.com
gavin.bierman@oracle.com

<http://labs.oracle.com>

— @criscifuentes

@GavinBierman

Cristina Cifuentes and Gavin Bierman, What is a Secure Programming Language?
3rd Summit on Advances in Programming Languages (SNAPL), LIPIcs 136, 2019.

Our mission is to help people
see data in new ways, discover insights,
unlock endless possibilities.





ORACLE