Wyvern: Improving Architecture-Based Security via a Programming Language

A/Prof Alex Potanin

Software Security is a Big Problem

18 OCT 2017 NEWS

Report: 88% of Java Apps Vulnerable to Attacks f… Known Security Defects

Michael Hill Acting Editor , Infosecurity Magazine
Email Michael Follow @MichaelInfosec

A new report from CA Veracode has exposed the pervasive risks companies face from vulnerable open source components.

Why Not Wa…

BBC Sign in | News | Sport | Weather | Shop | Reel | Travel | M…

NEWS

Home | Video | World | Asia | UK | Business | Tech | Science | Stories | Entertainment

Technology

Health records 'put at risk by security bugs'

Share

Home > Enterprise Java > Software Development

NEWS

New bug neutralizes latest Java security updates

New security vulnerability bypasses the Java plug-in's protection

By Gregg Keizer
Computerworld | JANUARY 28, 2013 12:54 PM PT

Java's new security settings, designed to block drive-by browser attacks, can be bypassed by hackers, a researcher announced Sunday.

The news came in the aftermath of several embarrassing zero-day vulnerabilities, and a recent commitment by the head of Java security that his team would fix bugs in the software.

More about Java security

- Why it's time to deprecate the Java Plug-in
- After silence on Java flaws, Oracle now says it cares
- InfoWorld's Security Adviser

The Java security provisions that can be circumvented were introduced last December with Java 7 Update 10 and let users decide which Java applets are

MORE LIKE THIS

What the latest Java flaw really me…

Two-year-old Java flaw emerges due to broken patch

Researchers find serious flaw in lat… JRE for desktops, servers

FEATURED NEWS

Teen Becomes First to Earn $1M in Bug Bounties with HackerOne

He is also the all-time top-ranked hacker on HackerOne's leaderboard, out of more than 330,000 hackers competing for the top spot.

by Tara Seals
March 4, 2019

RSAC 2019: Microsoft Zero-Day Allows Exploits to Sneak Past Sandboxes

Researchers say that Microsoft won't issue a patch for the issue.

by Tara Seals
March 5, 2019

RSAC 2019: Container Escape Hack Targets Vulnerable Linux Kernel

A proof-of-concept hack allows

BSides SF 2019: Remote-Root Bug in Logitech Harmony Hub Patched and Explained

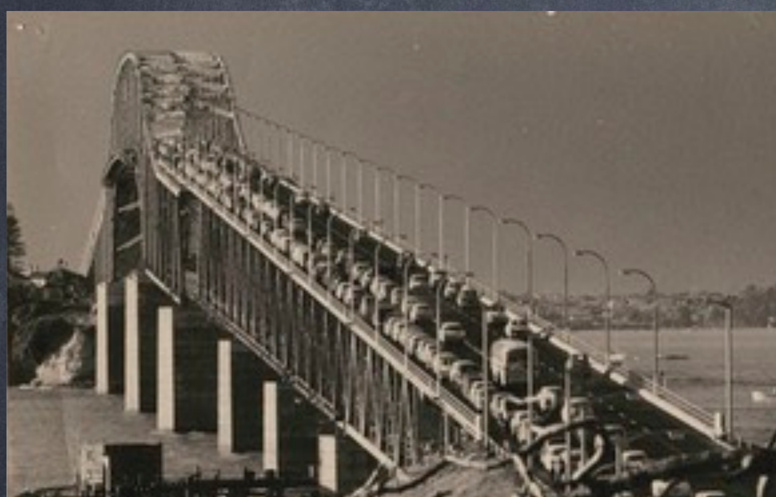
Users of Logitech's Harmony Hub get long-awaited answers about the critical bugs that left their home networks wide open to attack.

by Tom Spring

Why Systems are Vulnerable?

- We "know" how to code securely
  - Follow the rules: CERT, Oracle, ...
  - Technical advances: types, memory safety
- But we still fail too often!
- Root causes
  - Coding instead of engineering
  - Human limitations
  - Unusable tools

Our Approach: Usable Architecture-Based Security



Secure systems development

Engineering:
An architecture/design perspective

Usability:
A human perspective

λ

Formal Modelling: A mathematical perspective

The Wyvern Programming Language

- Designed for security and productivity from the ground up
- General purpose, but emphasising web, mobile, and IoT apps

http://wyvernlang.github.io/

VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

Carnegie Mellon University

The Wyvern Programming Language

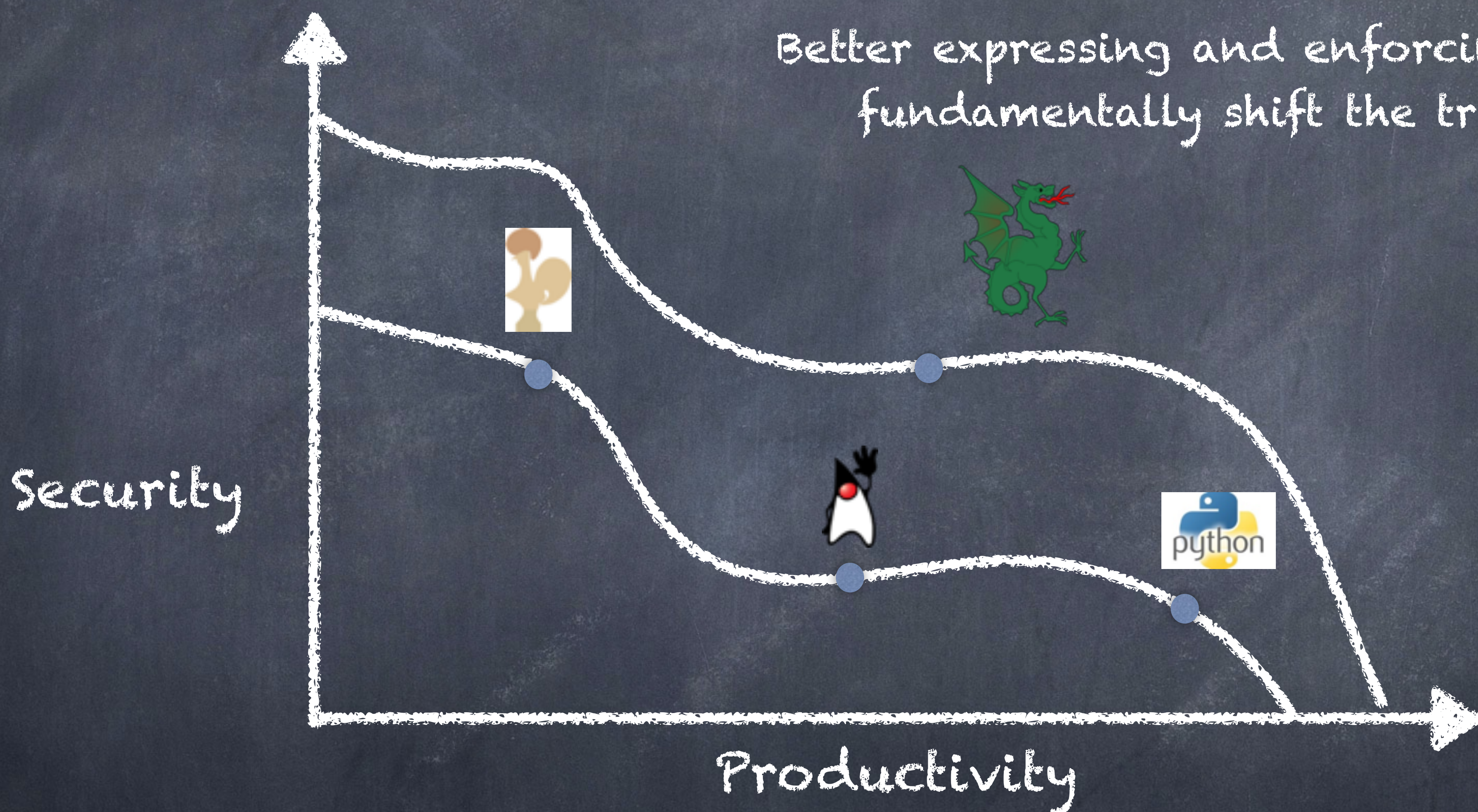
- But you might ask: "Isn't there a trade off between security and productivity?

- What is Wyvern's secret sauce?


DYNAMIC TYPING
STATIC TYPING
DONE!

Shifting the Tradeoff Curve

Better expressing and enforcing design could fundamentally shift the tradeoff curve



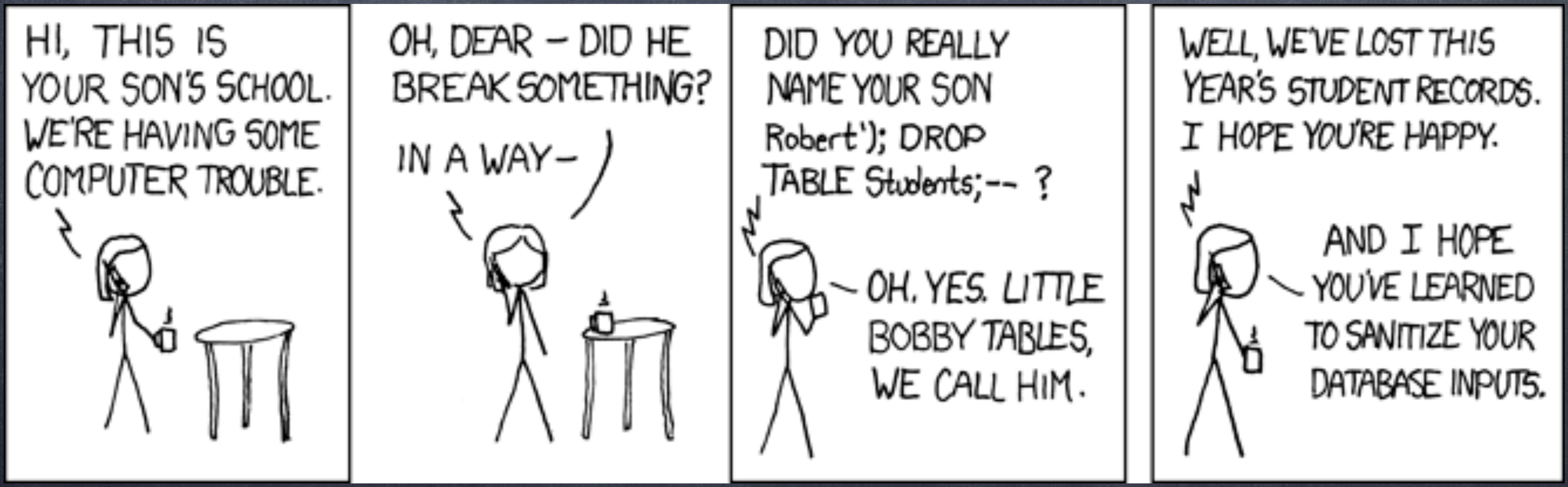
Wyvern

- Design goals
  - Sound, modern language design
    - Type- and memory- safe, mostly functional, advanced module system
  - Incorporate usability principles
  - Security mechanisms built in

Hello, world!

```
require stdout

stdout.print("Hello, world!\n")
```

SQL Command Injection



SQL Injection: a Solved Problem?

```
PreparedStatement s = connection.prepareStatement(
    "SELECT * FROM Students WHERE name = ?;");
s.setString(1, userName);
s.executeQuery();
```

Fill the hole securely

Prepare a statement with a hole

- Evaluation
  -  Usability: unnatural, verbose
  -  Design: string manipulation captures domain poorly
  -  Language semantics: largely lost – just strings
    - No type checking, IDE services, …

Wyvern: Usable Secure Programming

- A SQL query in Wyvern:

```
connection.executeQuery(~)
    SELECT * FROM Students WHERE name = {studentName}
```

~ introduces a domain-specific language (DSL) on the next indented lines

Semantically rich DSL. Can provide type checking, syntax highlighting, autocomplete, …

Safely incorporates dynamic data – as data, not a command

- Claim: the secure version more natural and more usable
  - No empirical evaluation, yet

Cyrus Omar
Assistant Professor (Sep. 2019-)
Computer Science and Engineering
University of Michigan

Technical Challenge: Syntax Conflicts

- Language extensions as libraries has been tried before
  - Example: SugarJ/Sugar* [Erdweg et al, 2010; 2013]

Is it XML or HTML?

```
import XML, HTML

val snippet = ~

    How do I <b>parse</b> this example?
```

Syntax Conflicts: Wyvern's Solution

metadata keyword indicates we are importing syntax, not just a library

```
import metadata XML, HTML
```

No ambiguity: the compiler loads the unique parser associated with the expected type XML

```
val snippet : XML = ~

    How do I <b>parse</b> this example?
```

Syntax of language completely unrestricted - indentation separates from host language

Technical Challenge: Semantics

Q: Is it safe to run custom parser at compile time?
A: Yes – immutability types used to ensure imported metadata is purely functional, has no network access, etc.

Language definition includes custom type checker – can verify query against database schema

```
import metadata SQL
val connection = SQL.connect(...)
val studentName = input(...)
connection.executeQuery(~)
    SELECT * FROM Students WHERE name = {studentName}
```

Splicing (as in genes) theory ensures capture-avoiding substitution in code generated by SQL extension – safe to use host language variables

SQL extension has access to variables and their types in Wyvern host language

Example

```
serve : (URL, HTML) -> ()
```

base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style ~
        body { font-family: %bodyFont% }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

How do you enter and exit a TSL?

- In the base language, several inline delimiters can be used to create a TSL literal:

```
`TSL code here, ``inner backticks`` must be doubled`
'TSL code here, ''inner single quotes'' must be doubled'
{TSL code here, {inner braces} must be balanced}
[TSL code here, [inner brackets] must be balanced]
<TSL code here, <inner angle brackets> must be balanced>
```

- If you use the block delimiter tilde (~), there are no restrictions on the subsequent TSL literal.
  - Indentation ("layout") determines the end of the block

How do you associate a TSL with a type?

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metadata = new : HasParser
    val parser : Parser = new
      def parse(s : TokenStream) : ExpAST =
          (* code to parse specialized HTML notation *)
```

```
objtype Parser =
  def parse(s : TokenStream) : ExpAST
```

```
casetype ExpAST =
  Var of ID
| Lam of (Var, ExpAST) | Ap of (Exp, Exp)
| CaseIntro of (TyAST, String, ExpAST) | ...
```

Why not associate a grammar with a type?

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metadata = new : HasParser
    val parser : Parser = ~
      start ::= ":body" children::start
                 `HTML.BodyElement(([], %children%))`
             | ...
```

Grammars are TSLs for Parsers!

Quotations are TSLs for ASTs!

TSL Benefits

- Modularity and Safe Composability
- Identifiability (easily see which DSL due to expected type)
- Simplicity
- Flexibility (whitespace delimited blocks => arbitrary syntax)

TSL Limitations

- Decidability of Compilation
- No editor support (coming?)

Our Approach: Usable Architecture-Based Security

DSL support in Wyvern

Engineering:
Express design in domain-specific way

Usability:
Natural syntax, enabling IDE support

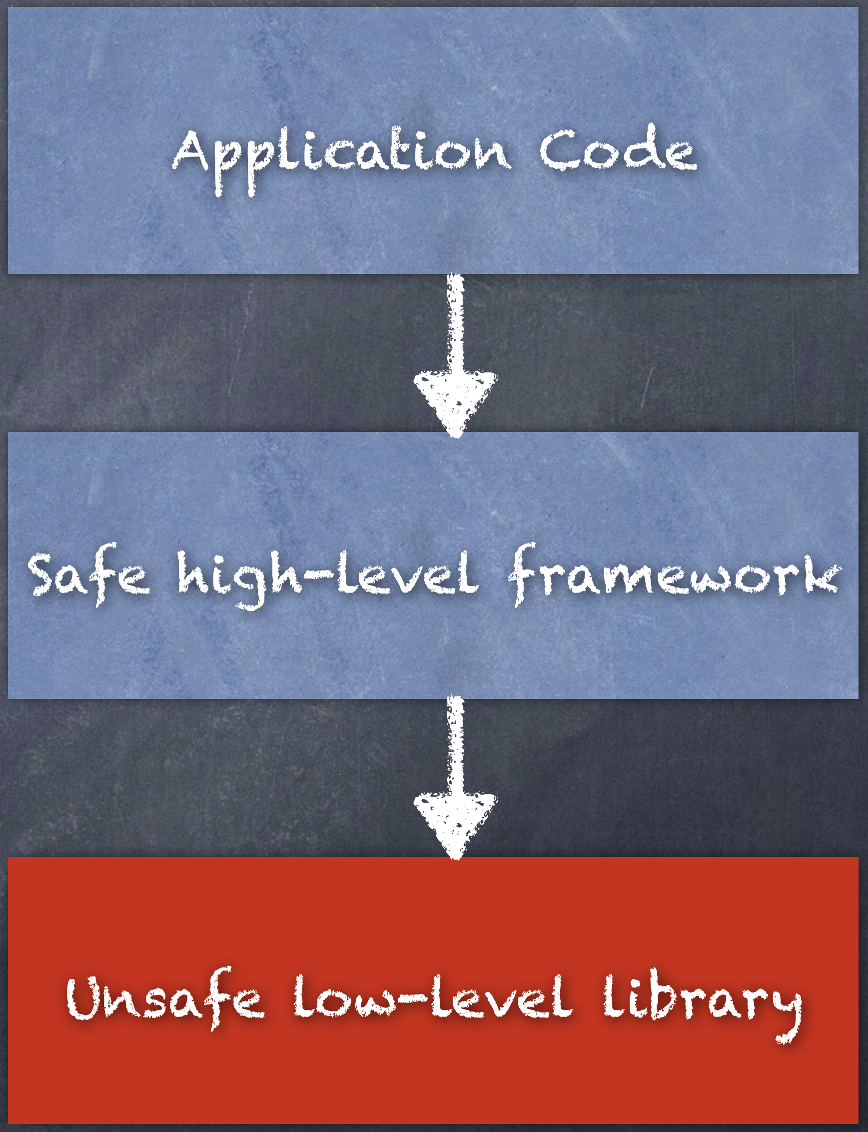
λ

Formal Modelling: Type safety, variable hygiene, conflict-free extensions

An Old Idea: Layered Architectures

[Dijkstra 1968]

- Lowest layer: an unsafe, low-level library
- Middle layer: a higher-level framework
- Top layer: the application
- Code must obey strict layering
- Many variants:
  - Secure networking framework
  - Safe SQL-access library
  - Replicated storage library

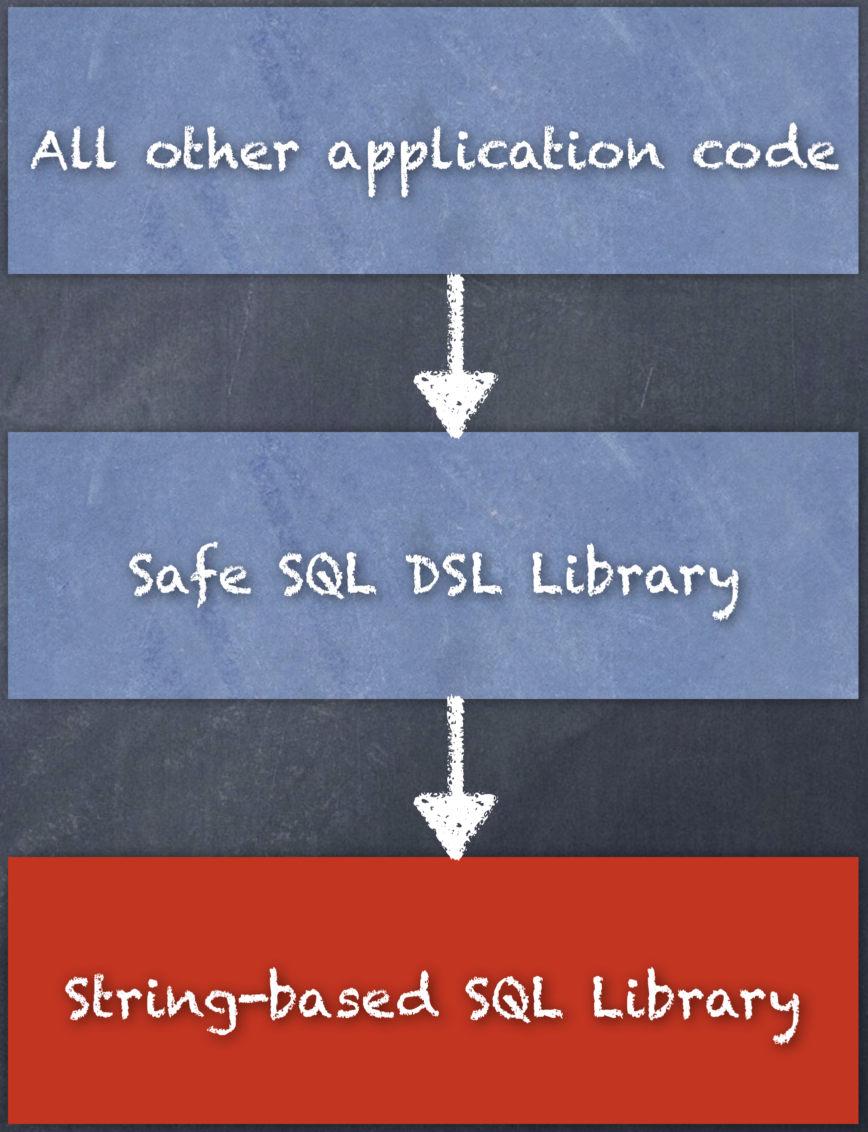


RQ: Can we use capabilities to enforce layered resource access?

* Capability: an unforgeable token controlling access to a resource [Dennis & Van Horn 1966]

Architecture: Principle of Least Privilege (PoLP)

- Every module must be able to access only the resources necessary for its legitimate purpose [Saltzer & Schroeder 75]
- Architectural layering example: Only Safe SQL library may access the low-level SQL interface



Module Linking as Architecture

```
require db.stringSQL
```

To access external resources like a database, main requires a capability from the run-time system. A capability is an unforgeable token controlling access to a resource.

```
application.run()
```

stringSQL

Module Linking as Architecture

We can import code modules, but they have no ambient authority to access resources (cf Newspeak). sqlApplication cannot access the database by itself.

```
require db.stringSQL

import db.safeSQL
import app.sqlApplication

val sql = safeSQL(stringSQL)
val application = sqlApplication(sql)

application.run()
```

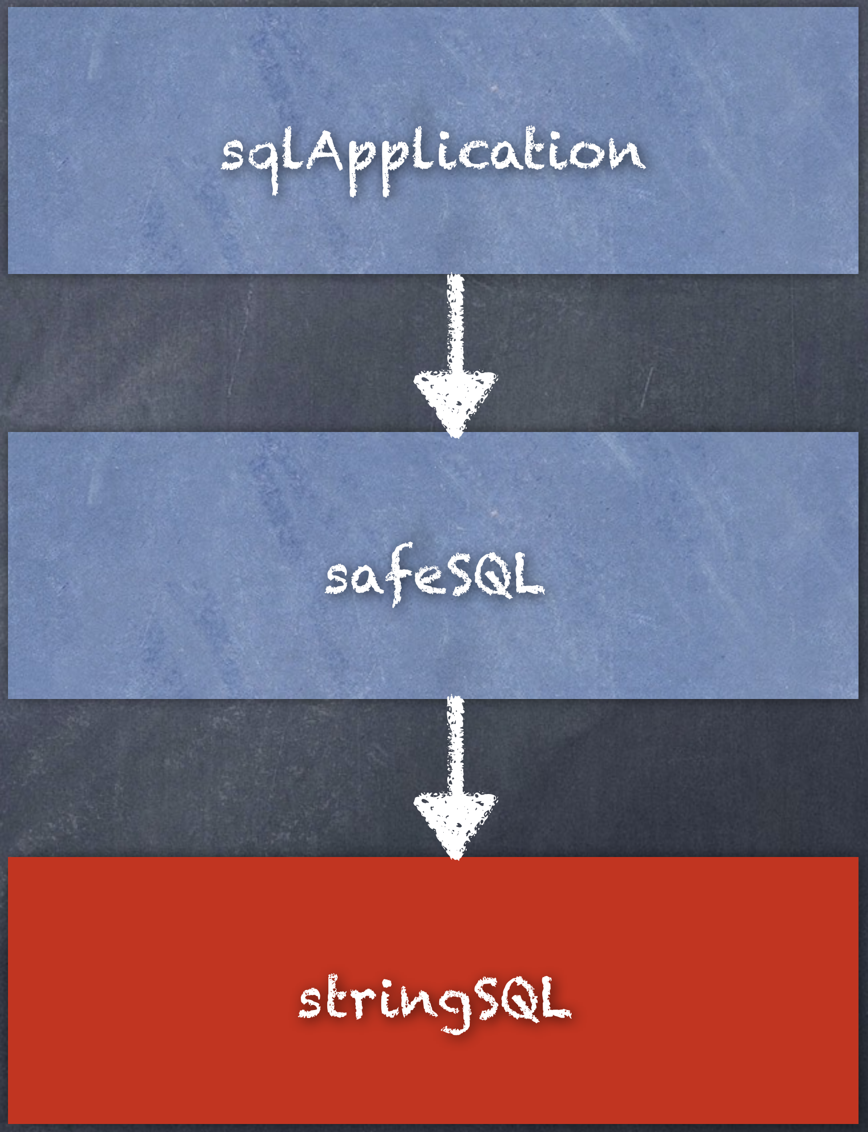
We must instantiate a sqlApplication object, passing it the resources it needs. We pass only a capability to the safe library.

Module Linking as Architecture

```
module def sqlApplication(safeSQL : db.SafeSQL)
def run() : Int
    // application code
```

```
require db.stringSQL


import db.safeSQL

import app.sqlApplication


val sql = safeSQL(stringSQL)

val application = sqlApplication(sql)


application.run()
```

```
module def safeSQL(strSQL : db.StringSQL)
// implement ADT in terms of strings
```



How Hard to Link it ALL Up?

- Most Wyvern modules don't have state, can be freely imported
- Statically tracked: stateful modules/objects and resource types

Provides access to OS resource

```
resource type File
    def write(s : String)
```

```
type SetM
    resource type Set
        def add(v : Int)
        def isMember(v : Int) : Bool
    def makeSet() : Set

module setM : SetM

module def client(aFile : File)
import setM ...
```

Type of modules is pure; no static state. Objects created by module may be stateful resources, though.

Resources must be passed in; pure modules can just be imported.

- resource types capture state or system access: other types do not
  - Useful design documentation; e.g. MapReduce tasks should be stateless
  - Supports powerful equational reasoning, safe concurrency, etc.

Checking PoLP with Effects

```
// in signature of the rawSQL module
effect UnsafeQuery
type Connection
def connect(...) : Connection
def query(q:String) : {UnsafeQuery} Data


// client code
def getData(input : String) : Data
    rawSQL.query("SELECT * FROM Students WHERE name = '" + input + "';")
```

The unsafe SQL library defines an UnsafeSQL effect

Query operations have an UnsafeQuery effect

Error: getData() must declare effect rawSQL.UnsafeQuery

Has effect rawSQL.UnsafeQuery

NB! In Wyvern Effect is a "Resource.Operation" pair.

Checking PoLP with Effects

```
// in signature of the rawSQL module
effect UnsafeQuery
type Connection
def connect(...) : Connection
def query(q:String) : {UnsafeQuery} Data


// client code
def getData(input : String) : {rawSQL.UnsafeQuery} Data
    rawSQL.query("SELECT * FROM Students WHERE name = '" + input + "';")
```

The unsafe SQL library defines an UnsafeSQL effect

Query operations have an UnsafeQuery effect

All dangerous code marked with effect

Has effect rawSQL.UnsafeQuery

NB! In Wyvern Effect is a "Resource.Operation" pair.

Effect Abstraction

- Issue: won't users of the safeSQL library have an UnsafeQuery effect, if safe SQL is built on rawSQL?

```
module def safeSQL(rawSQL : RawSQL) : SafeSQL

type SQL

    metadata ...

abstract effect SafeQuery = rawSQL.UnsafeQuery

def query(SQL) : {SafeQuery} Data

    ...
```

The safeSQL functor uses a rawSQL module

Defines a SQL ADT with metadata for parsing

The SafeQuery effect is defined in terms of UnsafeQuery. This definition is abstract – hidden from clients.

Now clients have effect safeSQL.SafeQuery

Q: Can't any library do this, potentially hiding unsafe queries?
A: Potentially, but can mechanically check only trusted libraries do so

Effect System Usability

Client Code

↓

Safe SQL DSL Library

- Isn't it a pain to declare all these effects?
  - Case in point: exception specifications in Java
- We can bound a module's effects by its capabilities
  - No need to effect-annotate the module
  - Does assume capability-safety (cf JS Frozen Realms)

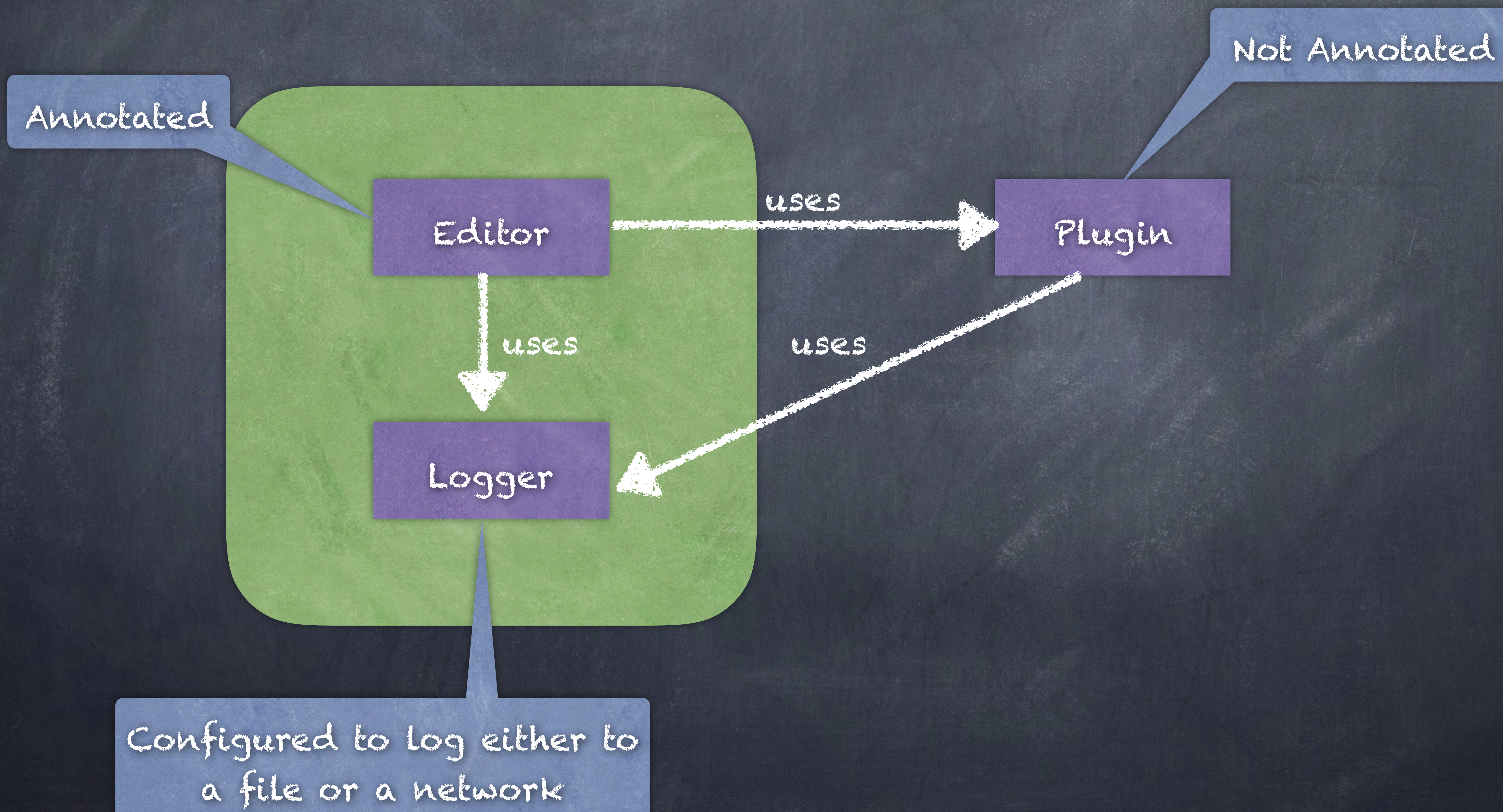
Client can have effect safeSQL.SafeQuery (and nothing else)

```
module def client(safeSQL : SafeSQL) : Client

import ...
```

If safeSQL defines higher-order functions, make sure the argument is allowed to have the SafeQuery effect (cf contravariant subtyping).

Imports may not be resources – no effects.

Wyvern Effects Example



Importing Effect Unannotated Code

For import, we want a rule of the form:

$$\frac{\cdots}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e}\ \texttt{in}\ e : \cdots\ \texttt{with}\ \cdots}\ (\varepsilon\text{-}\textsc{Import})$$

- What type and effects does the import expression have?
- What assumptions do we need?

Importing Effect Unannotated Code

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \;\texttt{with}\; \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\; x = \hat{e} \;\texttt{in}\; e : \texttt{annot}(\tau, \varepsilon_s) \;\texttt{with}\; \varepsilon_s \cup \varepsilon_1} \;(\varepsilon\text{-IMPORT1})$$

- Assume arbitrary type and effect for $\hat{e}$.
- Must be able to type e, given just that x has type $\hat{\tau}$, to ensure e uses only the capabilities provided to it.
- e is unannotated while $\hat{\tau}$ is annotated, so we erase the annotations from $\hat{\tau}$.
- e has type τ — but τ is unannotated, so we annotate with ε_S.
- Evaluating e has all effects in ε_1 and ε_S.

Importing Effect Unannotated Code

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \quad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau \quad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e} \texttt{ in } e : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT2})$$

- First version allows any capability to be passed to *e*.
- Restrict $\hat{e}$ so that its effects are contained in ε_s.
- *effects* collects all the effects captured by its argument.

$$\texttt{effects}(\{\bar{r}\}) = \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$$
$$\texttt{effects}(\hat{\tau}_1 \rightarrow_\varepsilon \hat{\tau}_2) = \texttt{effects}(\hat{\tau}_1) \cup \varepsilon \cup \texttt{effects}(\hat{\tau}_2)$$

Importing Effect Unannotated Code

$$\frac{\hat{\Gamma} \vdash \hat{e} : \hat{\tau} \texttt{ with } \varepsilon_1 \qquad \texttt{effects}(\hat{\tau}) \subseteq \varepsilon_s \qquad \texttt{ho-safe}(\hat{\tau}, \varepsilon_s) \qquad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\ x = \hat{e} \texttt{ in } e : \texttt{annot}(\tau, \varepsilon_s) \texttt{ with } \varepsilon \cup \varepsilon_1} \ (\varepsilon\text{-IMPORT3})$$

$$\texttt{effects}(\{\bar{r}\}) = \{r.\pi \mid r \in \bar{r}, \pi \in \Pi\}$$
$$\texttt{effects}(\hat{\tau}_1 \to_\varepsilon \hat{\tau}_2) = \texttt{ho-effects}(\hat{\tau}_1) \cup \varepsilon \cup \texttt{effects}(\hat{\tau}_2)$$

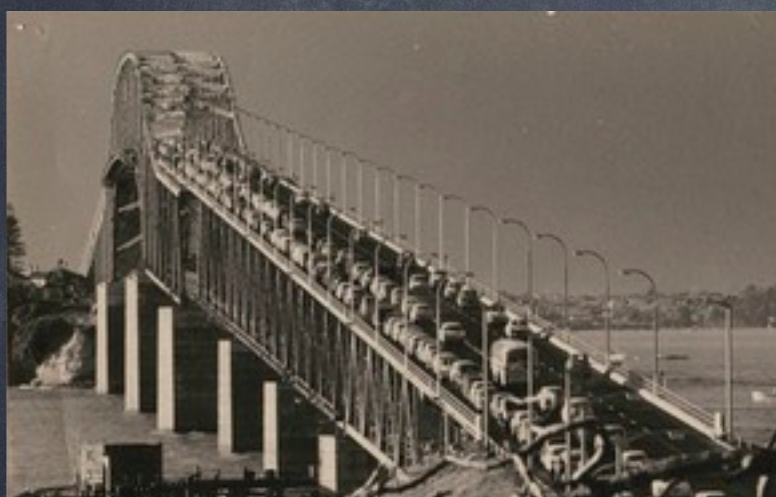
$$\texttt{ho-effects}(\{\bar{r}\}) = \varnothing$$
$$\texttt{ho-effects}(\hat{\tau}_1 \to_\varepsilon \hat{\tau}_2) = \texttt{effects}(\hat{\tau}_1) \cup \texttt{ho-effects}(\hat{\tau}_2)$$

- Need to distinguish “direct” effects from “higher-order” effects.
- And ensure safe use of resources: imported capabilities must be expecting the effects they are passed by unannotated code.

Importing Effect Unannotated Code

$$\frac{\texttt{effects}(\hat{\tau}) \cup \text{ho-}\texttt{effects}(\texttt{annot}(\tau, \varnothing)) \subseteq \varepsilon_s \qquad \hat{\Gamma} \vdash \hat{e} : \hat{\tau} \;\texttt{with}\; \varepsilon_1 \qquad \text{ho-}\texttt{safe}(\hat{\tau}, \varepsilon_s) \qquad x : \texttt{erase}(\hat{\tau}) \vdash e : \tau}{\hat{\Gamma} \vdash \texttt{import}(\varepsilon_s)\; x = \hat{e} \;\texttt{in}\; e : \texttt{annot}(\tau, \varepsilon_s) \;\texttt{with}\; \varepsilon_s \cup \varepsilon_1} \; (\varepsilon\text{-IMPORT})$$

Our Approach: Usable Architecture-Based Security



Effects and capabilities in Wyvern

Engineering: Architectural restrictions on resource use

Usability: Bound effects based on architecture

λ

Formal Modelling: effect- and capability- safety, effect bounds

Questions?