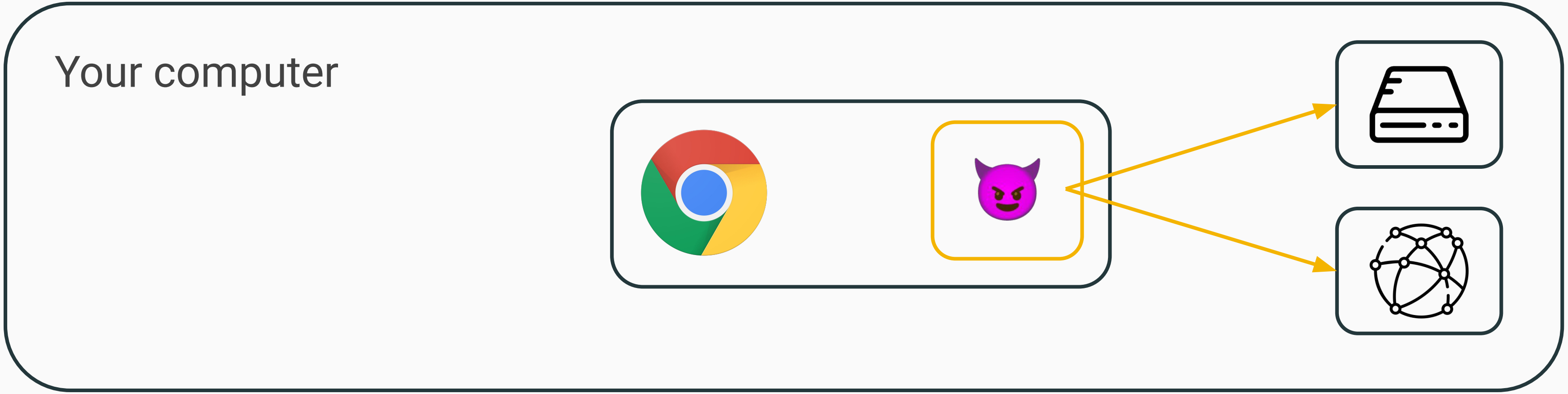
Gradient

Gradual Compartmentalization via Object Capabilities Tracked in Types

Aleksander Boruch-Gruszecki, Adrien Ghosn, Mathias Payer, Clément Pit-Claudel
OOPSLA 2024, Pasadena

We have a problem

Our software is stunningly vulnerable to *supply chain attacks*.
Examples: Log4Shell, SolarWinds, the xz/liblzma backdoor.



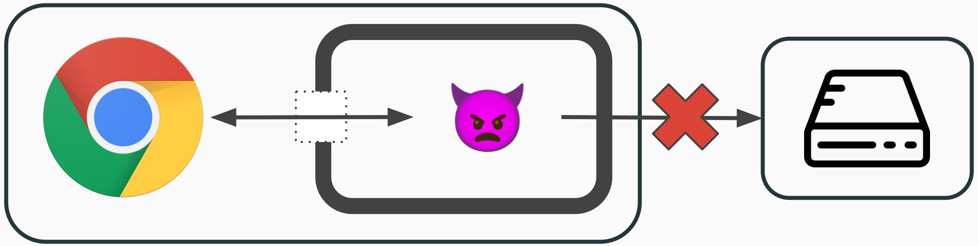
A backdoor in a lib used by Chrome could give access to *everyone's* computers.

There is a way forward

“Every [part of a] program should operate using the least [authority] necessary to complete the job.” — Principle of Least Authority/PoLA (J. Saltzer, 1974)

We want a mechanism for *compartmentalizing* parts of a program.

Basic idea: accessing the outside of the compartment is completely controlled.



Chrome does compartmentalization with processes: a heavyweight approach.

There is a way forward

The object capability model: (See Miller, “Robust Composition”, 2006)

➔ Allows enforcing the PoLA policies with code, using PL-level concepts
➔ So far, required a stop-the-world migration

What if:

- we could gradually adopt object capabilities in a program, part-by-part
- we could still introduce classical libraries
- and we were still able to enforce PoLA?

That’s the core motivation for our *gradual compartmentalization* approach, and for *Gradient*, a Scala 3 extension proposal featuring the approach.

Object capabilities

Legacy log4j

Classical code in libraries like `log4j` can access arbitrary system features.

```
def main() =
  val log = new log4j.Logger()
  log.info("Hello world!")

package log4j:
  class Logger():
    def info(msg: String) =
      open("...").write(msg)
      if shouldTriggerBackdoor(msg) then
        execute(downloadMalware())
```

Modern CapLog

CapLog is like `log4j` but for our Gradient extension: it uses object capabilities.

```
module Main(fs: Fs^, net: Net^, eval: Eval^):
  def main() =
      val CL  = new CapLog(fs, net, eval)
      val log = new CL.Logger()
      log.info("Logger created")
```

First: instantiate CapLog!

Next: instantiate Logger.

```
module CapLog(fs: Fs^, net: Net^, eval: Eval^):
  class Logger():
    def info(msg: String) =
      fs.open("...").write(msg)
      if shouldTriggerBackdoor(msg) then
        execute(eval, downloadMalware(net))
```

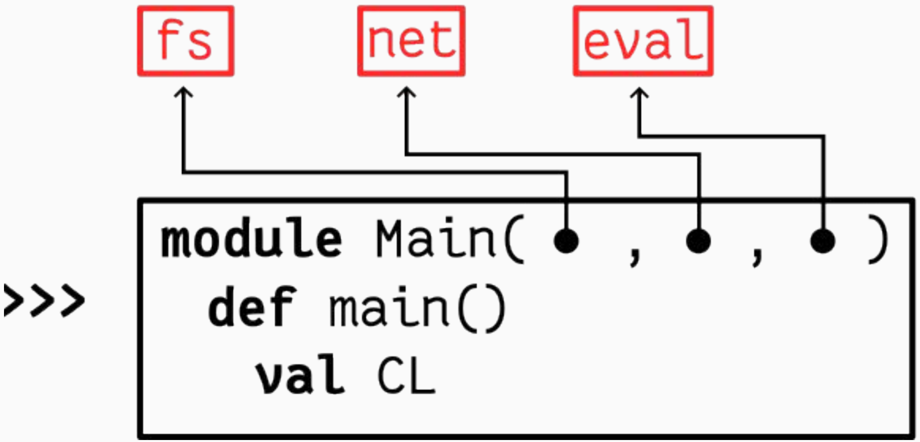
Capabilities are used to access system features.

How an object capability program starts

`fs` `net` `eval`

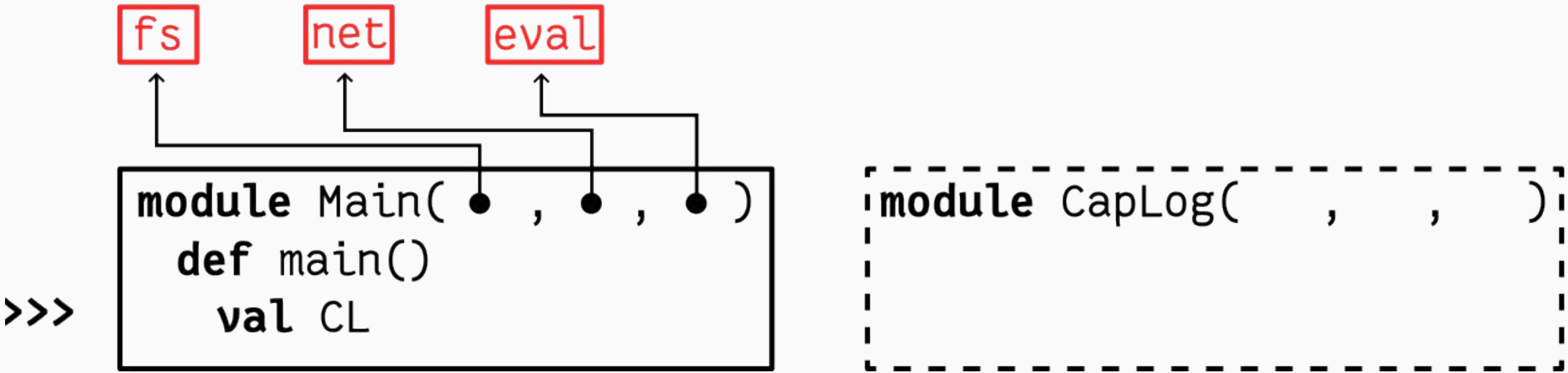
In the beginning, there are only some basic capabilities: the *devices* which let the program access the real world.

How an object capability program starts



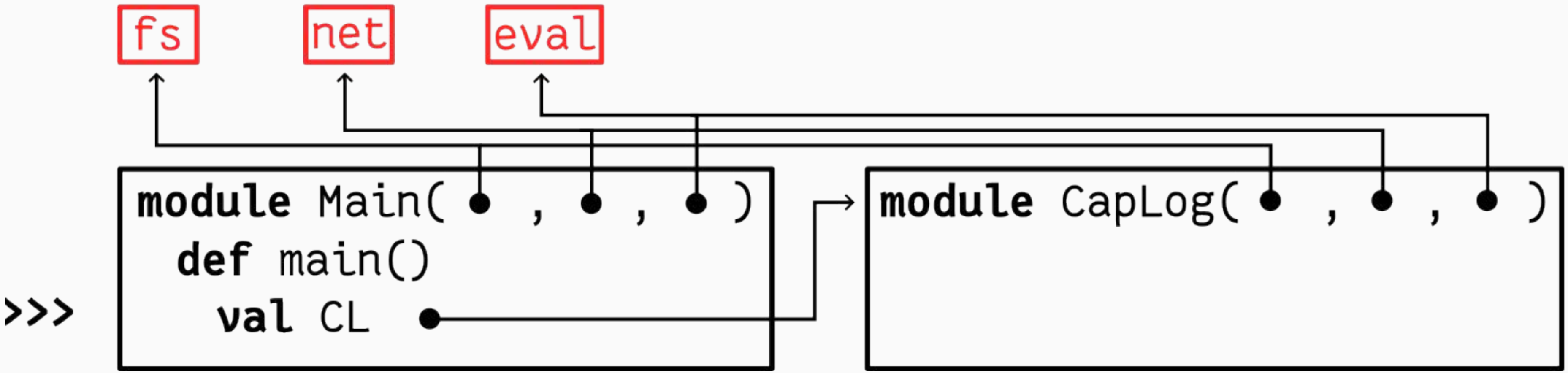
The runtime instantiates the `Main` module and then calls the `main` method.

How an object capability program starts



Main wants to instantiate (create CL, *an* instance of) CapLog.
CapLog *requests* capabilities to do its job. Main *decides* what CapLog gets.

How an object capability program starts



Main wants to instantiate (create CL, *an* instance of) CapLog.

CapLog *requests* capabilities to do its job. Main *decides* what CapLog gets.

Restricting the authority of CapLog

We want CapLog to only access files in the directory with logs.

```
module Main(fs: Fs^, net: Net^, eval: Eval^):
  def main() =
    val rfs = new RestrictedFs(fs, "/var/log/")
    // rfs can only access files in /var/log
    val CL  = new CapLog(rfs, InertNet(), InertEval())
    val log = new CL.Logger()
    log.info("Logger created")

class RestrictedFs(fs: Fs^, dir: String) extends Fs:
  def open(path: String) =
    if path.isRootedIn(dir)
      then fs.open(path)
      else throw new RuntimeException(...)
```

Inspect if the call to `open` is OK.

Object capabilities

- \+ Allow compartmentalization at the level of objects
- \+ Allow expressing arbitrary security policies with code

But, to enforce the PoLA on a library in a program, object capabilities must be used in:

1. the library
2. in its dependencies
3. in program code using the library (the *reverse* dependencies)

Can we do better?

Gradual compartmentalization

Using classical code in the object capability model

How can classical code and object capabilities interact?
What access policies can be enforced?

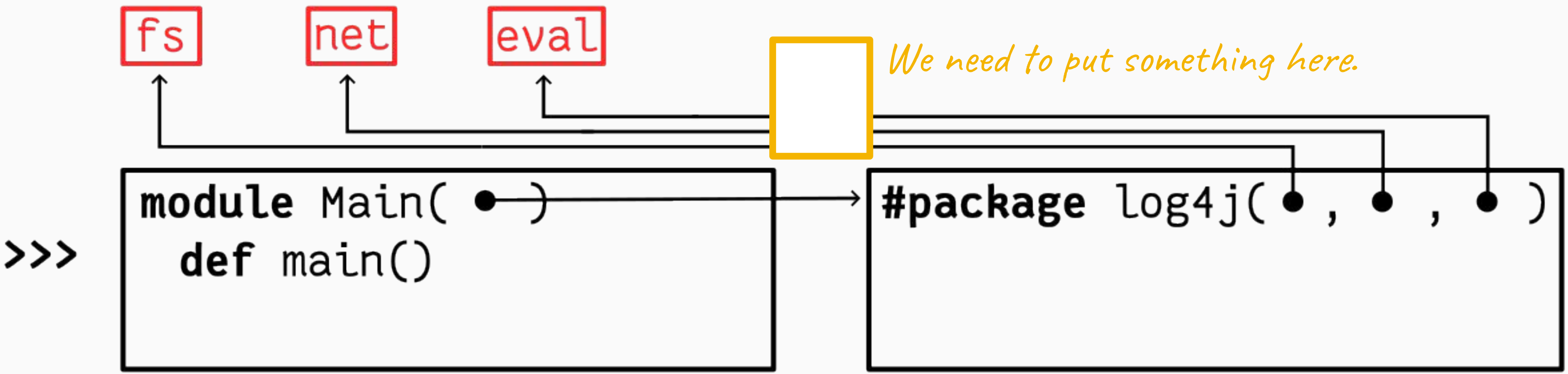
We want to integrate classical code packages into the object capability model.
Step 1: they can access system features – we treat them as object capabilities.

```
module Main(#package log4j):
  def main() =
    val log = new log4j.Logger()
    log.info("Logger created")
```

Uniform access to system features

Classical code accesses system features via *system calls*,
but object capability code can only do that via devices.

Idea: we can treat system calls like they were method calls to devices!



Controlling the authority of packages

How can we enforce the PoLA on packages of classical code?

We treat system calls like they were method calls to devices.
Idea: dynamically forbid some system calls within certain blocks!

```
module Main(#package log4j):
  def main() =
    enclosed[{fs}]:
      val log = new log4j.Logger()
      log.info("Logger created")
```

Only fs-related syscalls allowed here!

Essentially, **enclosed** blocks allow dynamically creating compartments!

(See Ghosn et al., “Enclosure: Language-Based Restriction of Untrusted Libraries”, ASPLOS 2021.)

Preventing access through mutable state

Mutable state allows “talking to” objects with access to system features.

In the paper we discuss how our approach can control mutable state access.

- Idea 1: mutable objects are capabilities.
- Idea 2: they are allocated in special memory regions.
- Policies can be enforced both on classical code and object capabilities.

In principle, the approach extends to controlling access to immutable secrets.

`enclosed` blocks

+ Allow compartmentalization at the level of code blocks
+ Work on arbitrary existing code (at the cost of flexibility)
+ Allow expressing PoLA policies with PL-level concepts

Object capabilities tracked in types

Static, type-based compartmentalization

Gradient tracks (the capture of) capabilities in types via *Capture Tracking*.
Packages may be *capture-checked*, enabling static capability access checks.

```
module M      (pack       log4j):
  def ma
    val lo              log4j.Logger()
    rest               }]:
      lo    fo(       ger created")
```

Ascriptions can be both method-level and block-level.

*Error! **main** should only capture {fs}, but instead it captures {fs, net, eval}.*

(See Boruch-Gruszecki et al., “Capturing Types”, TOPLAS 2023, presented at POPL 2024)

Static, type-based compartmentalization

Gradient tracks (the capture of) capabilities in types via *Capture Tracking*.
Packages may be *capture-checked*, enabling static capability access checks.

```
module Main(package nice4j):
  def main()^{fs} =
    val log = new nice4j.Logger()
    restricted[{log}]:
      log.info("Logger created")
```

Ascriptions can be both method-level and block-level.

(Capabilities are also visible in types of all objects, e.g., `log: Logger^{fs}`.)

Static, type-based compartmentalization

Gradient tracks (the capture of) capabilities in types via *Capture Tracking*.
Packages may be *capture-checked*, enabling static capability access checks.

```
module Main(package nice4j):
  def main()^{fs} =
    val log = new nice4j.Logger()
    restricted[{log}]:
      log.info("Logger created")
```

Ascriptions can be both method-level and block-level.

Capture ascriptions statically compartmentalize code, using the type system.

Back to danger

```
module Main(#package log4j):
  def main()^# =
    val log = new log4j.Logger()
    log.info("Logger created")
```

Errors: main may access code with unrestricted authority.

Tracked capabilities ensure that classical code runs in **enclosed** blocks.

Back to danger

```
module Main(#package log4j):
  def main()^{fs} =
    enclosed[{fs}]:
      val log = new log4j.Logger()
      log.info("Logger created")
```

Tracked capabilities ensure that classical code runs in **enclosed** blocks.

An **enclosed** block asserts what devices are accessed.

Object capabilities tracked in types

- Allow *statically* enforcing PoLA policies, at the level of blocks/objects
- Can enforce the PoLA on packages of (capture-checked) classical code
- Allow **`restricted`** blocks: a static, more flexible version of **`enclosed`**

Formal foundations

We extend $CC_{<:\square}$ (the Capture Calculus) with:

- mutability (regions, mutable references)
- object capability foundations (records, modules)
- more precise capture tracking for objects
- gradual capture tracking
    - capture-unchecked terms
    - formal **`enclosed`** blocks (a way to assert capture signatures)
    - calling capture-checked functions from capture-unchecked code

We prove that the resulting system is sound.

Validating the approach

Capture-checking a package: easier than adopting object capabilities outright.

Case study: manually capture-checking the standard Scala XML library.

The parser uses Java, accesses the filesystem and the network;
the code uses mutable objects (even if they aren't needed).

```
// before the migration
def buildString(sb: StringBuilder): StringBuilder
```

```
// after the migration
def buildString(sb: StringBuilder^): StringBuilder^{sb}
```

Out of 4200 LoC, 260 needed updates. 200/260 were as trivial as above.

- We present *gradual compartmentalization*, a hybrid approach which allows picking the best compartmentalization solution for every part of a program.
- We propose *Gradient*, a Scala 3 extension using gradual compartmentalization.
- We develop the formal foundations for Gradient.
  We add mutability, object foundations, gradual capture tracking to $CC_{<:\Box}$.
- We discuss how to implement Gradient based on existing works.
  The tasks are well-studied in isolation, the effort is like implementing a new PL.
- To validate the approach, we manually capture-check an existing library.
  No refactoring, 260/4600 LoC needed changes, 200/260 changes were trivial.
- We evaluate the performance of an Enclosure-based implementation.
  Even in pessimistic cases, the penalty can be below 1%.

Thank you!