Privacy-aware data science in Scala with monads and type level programming

Scale By the Bay 2018 (San Francisco)

David Andrzejewski, Senior Engineering Manager | November 15, 2018
2018年度最新アプリとDevSecOpsの現状
クラウド内で優れたアプリを構築、実行、保護するためのデータ駆動型の洞察

レポートをダウンロード
Current status of 2018 latest application and DevSecOps

Data-driven insight to build, execute and protect excellent apps in the cloud

Download Report
Current status of 2018 latest application and DevSecOps

Data-driven insight to build, execute and protect excellent apps in the cloud

Download Report
Me!

• Sumo Logic since 2011
  – We’re hiring!
  – We’re selling a great product!
• SF Bay Area ML meetup
  – Always looking for speakers!
  – And partners / host venues!
• Previously
  – ML research at LLNL
  – PhD @ UWisconsin (Go Badgers!)
This talk

1. Information flow control (IFC) for Data Science / ML
2. Types as guardrails
3. Composable abstractions

Privacy-aware data science in Scala with monads and type level programming
WARNING: work in progress

- Starting point, not the last word
- Hopefully you get some useful references and interesting ideas
- See future Qs / directions at end
Previously on Scale By The Bay…

- **FP for ML Panel Debate**: “Is FP the future of ML?”
  - **NO** one word: Notebooks - side-effects, mutable state, push them right to production!
  - **YES** That vision is horribly pessimistic & bleak…😉
Previously on Scale By The Bay…

- FP for ML Panel Debate: “Is FP the future of ML?”
  - NO one word: Notebooks - side-effects, mutable state, push them right to production!
  - YES That vision is horribly pessimistic & bleak… 😊

Winner*: Team YES
Previously on Scale By The Bay…

- **FP for ML Panel Debate:** “Is FP the future of ML?”
  - **NO** one word: Notebooks - side-effects, mutable state, push them right to production!
  - **YES** That vision is horribly pessimistic & bleak… 😊

**Winner*: Team YES

- [JupyterCon 2018: I don't like notebooks](#)
- [Pushing Jupyter Notebooks to Production](#)
- Cloud vendors: [AWS](#), [GCP](#), [Azure](#)
- Notebooks @ Netflix: [Part 1](#), [Part 2](#)

---

Beyond Interactive: Notebook Innovation at Netflix

By Michelle Ufford, M Pacer, Matthew Seal, and Kyle Kelley
And now... Sumo Logic Notebooks! (Docker Hub)
What if you’ve got sensitive data in your notebooks?

Talk Link 1
If you have a video, blog, or any other resource for your talk, add it here.

https://drive.google.com/file/d/1g2w31raOwJEJ2g-gCJNFVu8LCc04yw9x/view

"Just be careful"

Ad hoc API thingy

Monad library for information flow security

Unsafe data access doesn't compile

sumo logic
What if you’ve got sensitive data in your notebooks?

Talk Link 1

If you have a video, blog, or any other resource for your talk, add it here.

https://drive.google.com/file/d/1g2w31ra0wJEJ2g-gCJNFVu8LCc04yw9x/view

"Just be careful"

Ad hoc API thingy

Monadlc library for information flow security

Unsafe data access doesn't compile
### Roadmap

#### The Problem
- Risky data
- Goals
- Anti-goals

#### A Solution
- Information flow
- Monadic wrapper
- Implementation

#### …and more!
- Notebooks, revisited
- Composition
- Conclusion / Qs
2000s: Data is the New Oil

– Clive Humby & Edwina Dunn, Tesco 2006
2000s: Data is the New Oil
   – Clive Humby & Edwina Dunn, Tesco 2006

2010s: Data is nuclear waste
“This bad boy can fit so much PII”
You can’t spell “Data Science” without “Data”

Needs / use cases

• Exploration
• Cleaning / wrangling
• Iteration / prototyping
• Automated pipelines
• Model evaluation
• …and more!
You can’t spell “Data Science” without “Data”

Needs / use cases

- Exploration
- Cleaning / wrangling
- Iteration / prototyping
- Automated pipelines
- Model evaluation
- …and more!

Risks / pitfalls

- Policy violations
  - Inappropriate usage
  - Unintended access
- Statistical contamination
  - Leakage
  - Overfitting
You can’t spell “Data Science” without “Data”

### Needs / use cases
- Exploration
- Cleaning / wrangling
- Iteration / prototyping
- Automated pipelines
- Model evaluation
- …and more!

### Risks / pitfalls
- Policy violations
  - Inappropriate usage
  - Unintended access
- Statistical contamination
  - Leakage
  - Overfitting
Leakage in Data Mining: Formulation, Detection, and Avoidance – Kaufman et al (KDD 2011 Best Paper Award) - blog

- Medical: “Patient ID” field
- Sales: fields only populated for closed leads

Proposed approach: “legitimacy tagging”
Other anti-leakage techniques

- Oscar Boykin, *How to Elm-ify Your ML*
- Kaufman et al: “No time machine” requirement
Anti-goal: cryptographic concerns

• Assume
  – Data at rest encrypted, handled securely, etc
  – Data science / ML done on data “in the clear”
• No homomorphic encryption

\[
E(x_1 + x_2) \neq E(x_1) \oplus E(x_2)
\]
Anti-goal (for now?): differential privacy

\[ \ln \left( \frac{\Pr[\mathcal{T}_A(x) = t]}{\Pr[\mathcal{T}_A(x') = t]} \right) \leq \epsilon. \]

- **2017 Gödel Prize-winning work**
  - Calibrating Noise to Sensitivity in Private Data Analysis, Dwork et al

- **High-profile adopters**
  - Apple (\(\epsilon\)-budgets)
  - US Census Bureau

By moving to differential privacy, we are improving our ability to simultaneously protect data confidentiality while ensuring fitness-for-use in our published data products. Read more in this recap blog: [go.usa.gov/xPU7v](http://go.usa.gov/xPU7v)

7:55 AM - 2 Nov 2018
NOT in-scope: actual malicious adversary

```scala
val topSecret = Sec.high("SSN DATABASE")

var exfiltrator: String = null
topSecret.map(secret => {exfiltrator = secret})
exfiltrator should be("SSN DATABASE")
```
NOT in-scope: actual malicious adversary

```scala
val topSecret = Sec.high("SSN DATABASE")

var exfiltrator: String = null

val map = topSecret.map(secret => {exfiltrator = secret})

exfiltrator should be("SSN DATABASE")

println("topSecret.map(println)")
```
NOT in-scope: replacing actual organizational commitment to responsible data handling
GOAL: do useful work with data, but don’t allow…

• … direct access to sensitive / private data
• … unintended information flow aka “leakage”
GOAL: do useful work with data, but don’t allow…
• … direct access to sensitive / private data
• … unintended information flow aka “leakage”

Can the computer help us out here?
Solution requirements sketch

Your Data, In An Imaginary Box
Solution requirements sketch

Your Data, In An Imaginary Box

- **Cannot** directly access data
- **Can** manipulate data in-place
- **Can** process data to “de-secure” it
  - Code-reviewed library code only
  - (vs interactive notebook hacking)
- **Can** sequence transformations, mix & match datasets, etc
  - Security level set by “last write wins”
Cannot directly access
Can manipulate data in-place

\[ x \rightarrow f(x) \]
Can process to de-secure it
Can sequence transformations

$x \rightarrow y \rightarrow z$
Solution requirements, revisited

Your Data, In An Imaginary Box

- **Cannot** directly access data
- **Can** manipulate data in-place
- **Can** process data to “de-secure” it
  - Code-reviewed library code only
  - (vs interactive notebook hacking)
- **Can** sequence transformations, mix & match datasets, etc
  - Security level set by “last write wins”
# Solution requirements, revisited

### Your Data, In An Imaginary Box

- **Cannot** directly access data
- **Can** manipulate data in-place

- **Can** process data to “de-secure” it
  - Code-reviewed library code only
  - (vs interactive notebook hacking)

- **Can** sequence transformations, mix & match datasets, etc
  - Security level set by “last write wins”
Solution requirements, revisited

Your Data, In An Imaginary Box

- **Cannot** directly access data
- **Can** manipulate data in-place

- **Can** process data to “de-secure” it
  - Code-reviewed library code only
  - (vs interactive notebook hacking)
- **Can** sequence transformations, mix & match datasets, etc
  - Security level set by “last write wins”

HKT aka F[ _ ] aka * -> *

Functor

Monad
seplib: A simple library for static information-flow security in Haskell

newtype Sec s a
instance Functor (Sec s)
instance Monad (Sec s)

sec :: a -> Sec s a
open :: Sec s a -> s -> a

Figure 1. The Sec monad

- A Library for Light-weight Information-Flow Security in Haskell (video, code)
  – Russo, Claessen, Hughes
- Sec s a
  – s = security level type
  – a = underlying data type
Welcome to...the lattice

• “A lattice model of secure information flow” (’76)
  – Dorothy E. Denning
  – Side note: also developed early Intrusion Detection Systems (IDS) at SRI in 80’s

• Key idea: formal model of security/sensitivity *levels*

• Mechanisms can enforce model
Lattice Information Flow Control (IFC)

- Partial ordering ("poset") relation $\leftrightarrow$ allowed info flow
  - $L \subseteq H$ (do allow info flow $L \rightarrow H$)
  - $H \not\subseteq L$ (do not allow info flow $H \rightarrow L$)
- Lattice: all element pairs $(X,Y)$ have a unique
  - Greatest lower bound (glb / sup): $X \sqcap Y$
  - Least upper bound (lub / inf): $X \sqcup Y$
  - Bounded: unique top & bottom (monoid)
- Example (from original paper): subset relation
- Key idea: mechanical way to determine
  - whether code $X$ can read data $Y$
  - output level for inputs $A$ and $B$

$SC = \text{powerset } (X)$

- $A \rightarrow B$ iff $A \subseteq B$
- $A \oplus B \equiv A \cup B$
- $A \otimes B \equiv A \cap B$
- $L = \emptyset; H = X$
Key difference: (de-)classification as effect

- SecLib paper defines Monad **family**: one instance per level, eg:
  - \( \text{return} :: a \to \text{Sec} \ L \ a \)
  - \( (\ggg) :: \text{Sec} \ L \ a \to (a \to \text{Sec} \ L \ b) \to \text{Sec} \ L \ b \)
  - \( \text{return} :: a \to \text{Sec} \ H \ a \)
  - \( (\ggg) :: \text{Sec} \ H \ a \to (a \to \text{Sec} \ H \ b) \to \text{Sec} \ H \ b \)
- Families only connected via explicit transformations, data access
- How to achieve our desired classify / de-classify effects…?

![Diagram](image-url)
Our approach: last-write-wins (LWW) monadic join

- join :: Sec o (Sec i a) -> Sec r a
  - (o outer, i inner, r result)
Our approach: last-write-wins (LWW) monadic join

• \text{join} :: \text{Sec o (Sec i a)} \rightarrow \text{Sec r a}
  – (o outer, i inner, r result)

• But...implementing with SecLib-like runs afoul of Applicative laws:
  – \text{ <*>} :: \text{Sec o (a \rightarrow b)} \rightarrow \text{Sec i a} \rightarrow \text{Sec r b}
  – \text{ <*>} \text{ mab ma =}
    \text{join } \text{ mab } \text{ >>= } (\text{f } \rightarrow \text{ ma } \text{ >>= } (\text{x } \rightarrow \text{ pure (f x)}))
  – Identity: \text{pure id <*> v = v}

• Issue: LHS will always evaluate to default level chosen for \text{pure} ...
Our approach: last-write-wins (LWW) monadic join

- \( \text{join} :: \text{Sec } o \ (\text{Sec } i \ a) \to \text{Sec } r \ a \)
  - \((o \ \text{outer}, \ i \ \text{inner}, \ r \ \text{result})\)

- But...implementing with SecLib-like runs afoul of Applicative laws:
  - \(<*> :: \text{Sec } o \ (a \to b) \to \text{Sec } i \ a \to \text{Sec } r \ b\)
  - \(<*> \ \text{mab} \ \text{ma} =\)
    - \(\text{join} \ $ \ \text{mab} \ \gg = (\ f \to \ \text{ma} \ \gg = (x \to \text{pure } (f \ x)))\)
  - Identity: \(\text{pure } \text{id} \ <*> \ \text{v} = \text{v}\)

- Issue: LHS will always evaluate to default level chosen for pure ...

- Solution: define an “identity element” security level
  - N (Neutral)
  - Essentially: \textit{noncommutative monoid} over security levels

*sumo logic*
Oops! We (sort of) re-discovered **graded monads** (?)

- Recent research here: [Katsumata](https://example.com), [Orchard](https://example.com), [Uustalu](https://example.com), [Gaboardi](https://example.com)
  - FULL DISCLOSURE: I cannot understand any of it …
  - tldr: Family of monads “indexed” by a monoid (credit: Prof Russo)
  - `join :: Sec o (Sec i a) -> Sec (o ⊔ i) a`
    - level of `join` is `lub` of inputs
    - Neutral identity is bottom of lattice

---

**effect-monad**: Embeds effect systems and program logics into Haskell using graded monads and parameterised monads

---

**Sumo Logic**
Closing out the digression

- We want: LWW join to allow (certain, special) code to “de-classify”
  - Needed to add an identity element for “non-effectful” pure
  - Slightly different approach than original SecLib…
  - Yet another alternative: graded monad with lattice-based resolution
Great – how do we actually code it up?
Attempt 0: very naïve port of Haskell to Scala

class Sec[L, A](value: A) {
  protected val _value = value
}

- Issues: can’t pattern match b/c security level type L is erased
  - Maybe (probably?) this is do-able with TypeTag-related tricks (?)
  - Would love to hear how…
Attempt 1: value-encoding of security level

```java
object Privacy {
    sealed trait PrivacyLevel
    case object High extends PrivacyLevel
    case object Neutral extends PrivacyLevel
    case object Low extends PrivacyLevel
}
```
object Privacy {
    sealed trait PrivacyLevel
    case object High extends PrivacyLevel
    case object Neutral extends PrivacyLevel
    case object Low extends PrivacyLevel
}

def flatMap[S](f: T => ValSec[S]): ValSec[S] = {
    val result = f(_data)
    (this._level, result._level) match {
        case (x, Neutral) => new ValSec(x, result._data)
        case _ => result
    }
}
Attempt 1: value-encoding of security level

```scala
object Privacy {
  sealed trait PrivacyLevel
  case object High extends PrivacyLevel
  case object Neutral extends PrivacyLevel
  case object Low extends PrivacyLevel

  def flatMap[S](f: T => ValSec[S]): ValSec[S] = {
    val result = f(_data)
    (this._level, result._level) match {
      case (x, Neutral) => new ValSec(x, result._data)
      case _ => result
    }
  }

  def reveal: Option[T] = level match {
    case High => None
    case Neutral | Low => Some(_data)
  }
}
```
Attempt 2: can we push this into compile-time?

\[
\text{Sec}[\text{High}, \text{Sec}[\text{Low}, T]]
\]

- Last-write-wins truth table

- Assume:
  - Functions that construct instances output fixed / static levels
  - That is, no data-dependent levels

<table>
<thead>
<tr>
<th>Outer</th>
<th>Inner</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>N</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>N</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>N</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>
“Type-Level Computations in Scala”, Stefan Zeiger

• See very nice example with Bool
“Type-Level Computations in Scala”, Stefan Zeiger

- See very nice example with `Bool`
- Key tricks:
  - Abstract types

```scala
sealed trait Perm {
  type T
}
```
“Type-Level Computations in Scala”, Stefan Zeiger

• See very nice example with `Bool`
• Key tricks:
  – Abstract types
  – Type parameters

```scala
sealed trait Perm {
  type F[T]
}
```
See very nice example with_bool

Key tricks:
- Abstract types
- Type parameters
- Subtype constraint

```scala
sealed trait Perm {
  type F[T <: Perm]
}
```
“Type-Level Computations in Scala”, Stefan Zeiger

- See very nice example with `Bool`.
- Key tricks:
  - Abstract types
  - Type parameters
  - Subtype constraint
  - Instantiations

```scala
sealed trait Perm {
  type F[T <: Perm] 
}

object Red extends Perm {
  type F[T <: Perm] = ? 
}

object Gre extends Perm {
  type F[T <: Perm] = ? 
}

object Blu extends Perm {
  type F[T <: Perm] = ? 
}
```
“Type-Level Computations in Scala”, Stefan Zeiger

- See very nice example with `Bool`
- Key tricks:
  - Abstract types
  - Type parameters
  - Subtype constraint
  - Instantiations
  - Populate types

```scala
sealed trait Perm {
  type F[T <: Perm] 
}

object Red extends Perm {
  type F[T <: Perm] = Gre
}

object Gre extends Perm {
  type F[T <: Perm] = Red
}

object Blu extends Perm {
  type F[T <: Perm] = Blu
}
```
Simplified type-level calculation

<table>
<thead>
<tr>
<th>Inner</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Outer</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

sealed trait PrivacyLevel
sealed trait Reveable extends PrivacyLevel

object High extends PrivacyLevel
object Neutral extends PrivacyLevel with Reveable
object Low extends PrivacyLevel with Reveable
Adding a type-level function

• Outer[X <: PrivacyLevel] <: PrivacyLevel
  – Any PrivacyLevel has a type member Outer
  – Outer takes a (PrivacyLevel) type argument …
  – … and returns a (PrivacyLevel) type
• Type constructor Outer[X] acts as “type-level function”
• Use to compute Result types at compile-time

sealed trait PrivacyLevel {
  type Outer[X <: PrivacyLevel] <: PrivacyLevel
}

<table>
<thead>
<tr>
<th>Inner</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Outer</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>
Encoding our target behavior

object Neutral extends Revealable {
    type Outer[X <: PrivacyLevel] = X
}
object Low extends Revealable { type Outer[X] = Low.type }
object High extends PrivacyLevel { type Outer[X] = High.type }

<table>
<thead>
<tr>
<th>Inner</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Outer</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>
Encoding our target behavior

<table>
<thead>
<tr>
<th>Inner</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Outer</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

```java
object Neutral extends Revealable {
  type Outer[X <: PrivacyLevel] = X
}
object Low extends Revealable { type Outer[X] = Low.type }
object High extends PrivacyLevel { type Outer[X] = High.type }
```
Actually using the types in `join`:

```python
def join[T, PLO <: PrivacyLevel, PLI <: PrivacyLevel]
    (x: Sec[PLO, Sec[PLI, T]])
    : Sec[PLI#Outer[PLO], T] = {
        Sec.apply[PLI#Outer[PLO], T](x._data._data)
    }
```
Using the types in join function

```python
def join[T, PLO <: PrivacyLevel, PLI <: PrivacyLevel]
    (x: Sec[PLO, Sec[PLI, T]])
    : Sec[PLI#Outer[PLO], T] = {
        Sec.apply[PLI#Outer[PLO], T](x._data._data)
    }
```

Outer level

Inner level

2x nested

Sec

"Type function" application
Example usage

“Privacy-preserving” function: String => Int

```scala
def aggregate(x: String): Sec[Low, Int] = {
  Sec.low(x.length())
}
```

Compiler computes result types, fails to compile illegal reveal

"Sec.reveal(highSec.map(_ .length()))" shouldNot (compile)
Sec.reveal(highSec.flatMap(aggregate)) should be(highName.length())
Why, though?
Back to data science notebooks …

• IDEA: restrict which code / functions can apply “de-classify” effect
  – eg, requires special review

• Given this, other code “safe” from undesirable information flow
Why bother with “monad”, “functor”, Laws, etc?

• Consider alternative design below
  – **PROS**: beloved imperative style, very fun mutator methods
  – **CONS**: have to learn custom API, does not compose

```scala
val mySecret = high("Area51")
mySecret.update({x => x.length()})
mySecret.setSecurityLevel(Level.LOW)
mySecret.get()  // 6
```
Why bother with “monad”, “functor”, Laws, etc?

- Consider alternative design below
  - **PROS**: beloved imperative style, very fun mutator methods
  - **CONS**: have to learn custom API, does not compose

- vs map / flatMap
  - Engineers can understand it
  - Other code can interoperate with it

```scala
val mySecret = high("Area51")
mySecret.update({x => x.length()})
mySecret.setSecurityLevel(Level.LOW)
mySecret.get() // 6
```
In-context function application ($\text{fmap}$)

$$x \rightarrow f(x)$$
Security level changes as effect (flatMap)
End-to-end example

• Want to construct & execute privacy-aware data pipelines
• Some operations may “fail” and return no result
• If only there were some way to model this…
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double] // Return Low
def leaky(data: Records): Sec[Double] // Return High
def report(result: Double): String
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double]  // Return Low
def leaky(data: Records): Sec[Double]    // Return High
def report(result: Double): String

val wrapped = for {
  query <- OptionT.fromOption(getTargetQuery())
}
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double] // Return Low
def leaky(data: Records): Sec[Double] // Return High
def report(result: Double): String

val wrapped = for {
    query <- OptionT.fromOption(getTargetQuery())
    records <- getUserData(query)
}
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double] // Return Low
def leaky(data: Records): Sec[Double] // Return High
def report(result: Double): String

val wrapped = for {
  query <- OptionT.fromOption(getTargetQuery())
  records <- getUserData(query)
  result <- OptionT.liftF(analyze(records))
}
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double]  // Return Low
def leaky(data: Records): Sec[Double]  // Return High
def report(result: Double): String

val wrapped = for {
    query <- OptionT.fromOption(getTargetQuery())
    records <- getUserData(query)
    result <- OptionT.liftF(analyze(records))
} yield report(result)  // OptionT[Sec, String]
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double] // Return Low
def leaky(data: Records): Sec[Double] // Return High
def report(result: Double): String

val wrapped = for {
  query <- OptionT.fromOption(getTargetQuery())
  records <- getUserData(query)
  result <- OptionT.liftF(analyze(records))
} yield report(result) // OptionT[Sec, String]
def getTargetQuery(): Option[String]
def getUserData(query: String): OptionT[Sec, Records]
def analyze(data: Records): Sec[Double]  // Return Low
def leaky(data: Records): Sec[Double]  // Return High
def report(result: Double): String

val wrapped = for {
    query <- OptionT.fromOption(getTargetQuery())
    records <- getUserData(query)
    result <- OptionT.liftF(leaky(records))
} yield report(result)  // OptionT[Sec, String]
Questions / future work

- Scala implementation improvements (Attempt \( \geq 3 \))
- DS/ML use cases for richer security level lattices
- Data-dependent de-classification (eg, differential privacy)
- Other related work on IFC for DS/ML
- Property testing for anti-leakage
- Software tooling for (emerging) “Trustworthy AI” research
Thanks for listening! Questions / ideas?

1. Information flow control (IFC)
2. Types as guardrails
3. Composable abstractions

Privacy-aware data science in Scala with monads and type level programming
Thank you

David Andrzejewski
Senior Engineering Manager
Email: david@sumologic.com