PhD Dissertation Proposal:
The Holy Grail of Gradual Security

Tianyu Chen
Computer Science, Indiana University

January 29, 2025

My thesis:
It is possible to design a gradual IFC programming language that satisfies both noninterference and the gradual guarantee, by excluding the unknown label $\star$ from runtime security labels.

In my PhD dissertation, I propose to harmoniously combine static and dynamic enforcement of information-flow security in one programming language (PL), $\lambda^{\star}_{\texttt{IFC}}$. The distinguishing feature of $\lambda^{\star}_{\texttt{IFC}}$ is that it satisfies both the main security guarantee (noninterference) and the gradual guarantee. The proposal is organized as follows. In Section 1, I motivate my dissertation research by demonstrating that programming languages are useful in protecting software systems against real-world security threats through information-flow control (IFC). In Section 2, I review the traditional approaches to IFC, including static, dynamic, and hybrid enforcement mechanisms. In Section 3, I review prior work on gradual IFC and describe the challenges of designing a language that satisfies both noninterference and the gradual guarantee. In Section 4, I present the main enabling insight of $\lambda^{\star}_{\texttt{IFC}}$ and reiterate my thesis statement. In Section 5, I list the main technical contributions of my dissertation research. Finally in Section 6, I summarize my progress and describe the plan.

$$
\begin{aligned}
\langle RECORD \rangle ::= & \{\textbf{FirstName=}\langle ID \rangle; \\
& \textbf{LastName=}\langle ID \rangle; \\
& \textbf{SSN=}\langle SSN \rangle\} \\
\langle ID \rangle ::= & w, w \in \{\textbf{A}, ...\textbf{Z}, \textbf{a}, ...\textbf{z}\}^+ \\
\langle SSN \rangle ::= & \langle D \rangle\langle D \rangle\langle D \rangle\text{-}\langle D \rangle\langle D \rangle\text{-}\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle \\
\langle D \rangle ::= & d, d \in \{\textbf{0}, ...\textbf{9}\}
\end{aligned}
$$

Figure 1: The user input grammar for a hypothetical application

1 Security Is Important, and PL Can Help!

With the development of digital society, people are increasingly concerned about the confidentiality of their personal data and the integrity of their online assets. Increasingly relying on computing devices and the Internet in their daily life, people fear that sensitive personal information, such as social security numbers, medical records, bank account balances... may be revealed to malicious third parties. People also worry that their digital photo albums, signatures on online legal documents, spreadsheets in cloud storage... may be tampered and manipulated by potential attackers.

Indeed, the fears are justified by recent news events. In 2018, the Cambridge Analytica scandal hit the world headlines, where the data collected from 87 million social media users was misused without their consent [15, 33, 27, 32]. In the healthcare sector, from 2005 to 2019, 249 million people were affected by data breaches that caused exposure of sensitive medical data [48]. Researchers have found ways to tamper with the analytics APIs [44] and damage the integrity of metadata, such as the numbers of likes, follows, and views, of major social media platforms [43]. To deal with the security and privacy challenges of the increasingly digitalized world, the European Union introduced the General Data Protection Regulation (GDPR) to reform and regulate the collection and processing of personal data. However, studies show that business entities experience challenges in complying with GDPR or auditing for compliance [56], particularly small-to-medium size enterprises [55, 24, 28].

From a technical perspective, ensuring the security and privacy of personal data typically involves tracking and checking the flow of information. To ensure confidentiality, data must not flow to inappropriate destinations, so that sensitive personal information is not revealed; dually, to ensure integrity, data must not flow from inappropriate sources, so that valuable digital assets are not corrupted [47, 11]. In practice, such enforcement of the flow of information is often difficult to implement. Take con-

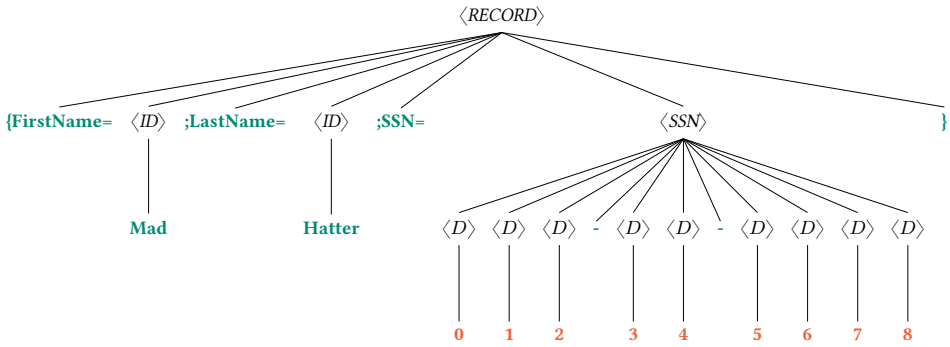

Figure 2: The parse tree generated from the example user input. All terminals are represented as labeled values: the red ones, such as the digits of SSN, are of high-security, while the green ones, such as the keys of the record and first name / last name, are of low-security.

fidentially for example, software applications accept user input where selected fields are sensitive, whose confidentiality is required during their entire life cycle including both parsing and data processing. To rule out information leaks, neither the sensitive fields, nor any data that depends on those fields, is allowed to be revealed to a low-privilege observer. Consider a web application that receives three fields from its user: (1) first name (2) last name (3) social security number, the grammar of which is defined in Figure 1, where terminals are divided into low-security and high-security. The digits d for social security number, being confidential to users of the web application, are of high-security, so they are marked red, while other terminals, such as the keys of the record and the strings w for first name / last name, being safe to disclose, are all of low-security, marked in green. Consider the following user input:

```
{FirstName=Mad;LastName=Hatter;SSN=012-34-5678}
```

It is tedious for the developer of this imaginary web application to track the security level of data and check for information leaks. Software developers tend to focus more on functionality in order to meet the tight software release schedule and budget, thus software security usually only comes as an afterthought [3, 49, 60]. Retrofitting security-related code often requires extensive modification to an existing code-base and it relies on the programmers' skills and experience to decide when and where such code should be placed. Furthermore, such modification is error-prone: one single missing check could undermine the security of the entire application.

Alternatively, the author of the web application could implement a parser for the grammar in Figure 1 in a programming language that enforces information-flow se-

curity. Each terminal in the grammar would be labeled with a security level and the programming language, instead of some ad-hoc checks implemented by the programmer of the web application, would guarantee that the high-security information is only present in those parts of the output parse tree that are marked as high-security. For example, according to the grammar in Figure 1, the example user input string is parsed into the parse tree in Figure 2, where the terminal nodes that represent digits of the social security number are of high-security, while the terminals that compose the rest of the input string are of low-security. The confidentiality of SSN is guaranteed during data processing by the programming language itself. When the web application interacts with the outside world, such as making a foreign function interface (FFI) call or storing into a database, the conceptual language should encrypt whatever values labeled as high security before they are passed into a foreign routine. The programming language-based approach of information-flow security [47] alleviates the security burden of software development, because it forms an abstraction over the flows of information and decouples security from the functionality of a software application.

2 Information-Flow Control via Static, Dynamic, and Hybrid Mechanisms

Information-flow control (IFC) ensures that information transfers within a program adhere to a security policy, for example, by preventing high-security data from flowing to a low-security channel. This adherence can be enforced statically using a type system [63, 41, 40], or dynamically using runtime monitoring [2, 4, 19, 58, 7, 64], or using static analysis to pre-compute information that facilitates runtime monitoring [36, 35, 16, 50, 46, 39]. The static and dynamic approaches have complementary strengths and weaknesses; the dynamic approach requires less effort from the programmer while the static approach provides stronger guarantees and less runtime overhead. The main theorem of an IFC system is *noninterference* [26]. Informally, noninterference states that, if the secretive user input varies, the public output of the program must stay the same.

We are going to review the literature about static and dynamic enforcement of IFC in Section 2.1 and Section 2.2 respectively. In Section 2.3, we briefly review the hybrid technique by Buiras et al. [14], which offers the programmer control over the regions where IFC is enforced statically versus dynamically in a single program.

2.1 Static IFC in Programming Languages

The interest in enforcing confidentiality and regulating the flow of information in a computer program arose with applications to defense in the 1970s [10]. Denning [17] builds an information flow model using a lattice of security labels and Denning and Denning [18] describe a static analysis for information flow, with a proof that a certified program will not transmit confidential input to non-confidential output. They distinguish between two types of information flows: explicit flow and implicit flow. Consider the assignment `x := y + z`, there are explicit flows from `y` to `x` and from `z` to `x`, because both `y` and `z` affect the value that `x` is assigned to. The certification checks whether the join (least upper bound) of the security of `y` and `z` is less than or equal to the security of `x`. Implicit flows, on the other hand, arise from the branching structure of a program. Consider the program `if x then y := y + 1 else ()`. An observer is able to learn whether `x` is true or false, by inspecting whether `y` increments. To control the implicit flow, the certification checks whether the security of `x` is less than or equal to that of `y`.

Volpano et al. [63] further develop the static enforcement and propose a typed-based approach to IFC, by defining a type system for an imperative programming language and proving its security with a type soundness proof. The type system approach benefits IFC enforcement because it is compositional: the security of the entire program is determined from the security of its constituent components; secure components form a larger secure system as long as their type signatures agree [47]. By writing a program that type checks, the software developer constructs a proof that the program is indeed secure.

We explain how a type system defends against illegal information flow using examples. Consider two IO functions: `private-input` and `publish`: the former returns a high-security boolean that represents sensitive user input information; the latter takes a low-security boolean and publishes it into a publicly visible channel.

Explicit flow Consider the program with an illegal explicit flow from private input to public output:

```
let input = private-input () in publish (¬ input)
```

The program is rejected by the type checker, because `(¬ input)` is typed at $\texttt{Bool}_{\texttt{high}}$ but `publish` expects its argument to be of $\texttt{Bool}_{\texttt{low}}$, $\texttt{high} \not\leq \texttt{low}$. As a result, the type system prevents information from leaking through explicit flow. On the other hand, the program

```
let input = private-input () in publish (¬ true)
```

is accepted by the type checker, because `(¬ true)` is typed at $\texttt{Bool}_{\texttt{low}}$, which can flow into `publish`. Indeed, the output remains constant regardless of user input, so no information leaks through the output.

Implicit flow Guarding against illegal implicit flows is more involved: the type system estimates the security of an if-conditional by joining the security of its branches with that of the branch condition (often referred to as type stamping). Consider the following program with an illegal information flow from private input to public output:

```
let input = private-input () in
  publish (if input then false else true)
```

The input influences the output through the branching structure of the program: if input is true, the then-branch is taken and the output is false; if input is false, the else-branch is taken and the output is true. The branch condition is typed at $\texttt{Bool}_{\texttt{high}}$ and the branches are typed at $\texttt{Bool}_{\texttt{low}}$, so the entire `if` has type $\texttt{Bool}_{\texttt{low} \vee \texttt{high}} = \texttt{Bool}_{\texttt{high}}$. Again, `publish` expects $\texttt{Bool}_{\texttt{low}}$, so the program is rejected. Consequently, the type system guards against the information leak through implicit flow.

Heintze and Riecke [29] and Zdancewic [65] further develop the type system approach to IFC and add support for higher-order functions. Researchers also build IFC type systems for bytecode intermediate languages [9], for object-oriented languages [1], and for reactive programming languages [13]. Although the aforementioned languages are mostly theoretical, efforts have also been made to integrate information flow control into off-the-shelf programming languages, such as Jif for Java [40] and Flow Caml for OCaml [45, 54].

2.2 Dynamic IFC in Programming Languages

IFC can also be enforced dynamically using runtime monitoring. Values typically have security labels associated with them and the runtime monitor tracks those labels during program execution. In the illegal explicit flow example

```
let input = private-input () in publish (¬ true)
```

the value of `input` is tagged with `high` and so is its negation. When `publish` is called, the function checks whether the label on the argument is `low`. The check fails, so the monitor reports a runtime error. In the constant example

```
let input = private-input () in publish (¬ true)
```

`true` is tagged with `low` and so is its negation; the monitor succeeds and publishes `false`. In the illegal implicit flow example

```
let input = private-input () in
  publish (if input then false else true)
```

the label associated with the evaluation result of the if-conditional is the join between that of the branch taken and that of the branch condition; in this case, $\texttt{low} \curlyvee \texttt{high} = \texttt{high}$. The `publish` function again checks the label against $\texttt{high} \not\leq \texttt{low}$, so it results in a runtime error.

Li and Zdancewic [37, 38] study dynamic IFC for purely functional languages (no mutable states). They add dynamic checking of information flow control to Haskell, by utilizing existing language features, specifically arrows and typeclasses.

Austin and Flanagan [4] consider IFC for Javascript, a language with mutable references. They propose a dynamic approach called *no-sensitive-upgrade* (NSU) checking, which guards against illegal implicit flows through the heap. During execution, the language runtime keeps track of a special security label, namely program counter. The program counter starts from low and is updated to high when the program branches on a high-security value. NSU terminates the execution whenever the program attempts to modify a low-security memory location under a high-security program counter. Austin and Flanagan [5] study a sound yet more flexible enforcement strategy called *permissive-upgrade*. Compared to NSU, permissive-upgrade allows more programs to run to completion. Austin and Flanagan [6], Austin et al. [7] propose *faceted values*, another approach to dynamically handle implicit flows by simulating multi-execution in a single process, which, unlike no-sensitive-upgrade and permissive-upgrade, avoids conservative monitor failures.

Stefan et al. [58, 59, 57] design a Haskell library called LIO. LIO is implemented as a domain specific language (DSL) embedded in Haskell. Even though Haskell itself is purely functional, it is worth noting that the LIO DSL does support mutable references (`LIORef`). Inspired by IFC operating systems [21, 66, 34, 62], LIO makes two unique design choices: (1) coarse-grained labeling (2) a floating current label. LIO is "coarse-grained" in that not all values are labeled by default. A programmer need to use the `toLabeled` primitive to explicitly protect a value. In this way, a programmer may choose to label a value when it is necessary to impose flow control policies and omit the labels for security-insensitive parts of the program. In LIO, the "current label" serves as a security upper bound of all values. During program execution, the LIO

runtime raises the current label to "float" above the security of all data read by the current computation. Similar to NSU, LIO performs dynamic checking on heap write operations and disallows writing to memory locations whose security is below the current label. Under the hood, LIO keeps track of the current label as a monad.

2.3 Programmer-Controlled Hybrid IFC

Similar to gradual IFC, Hybrid LIO (HLIO) [14] supports programmer's choice of static or dynamic IFC in different regions of a single program. By default, the checking is static, but a programmer can insert a `defer` clause to say that the security constraints should be checked at runtime. There are two major differences between HLIO and gradual IFC programming languages (we are going to discuss gradual IFC in the next section). In HLIO, the developer has to embed explicit `defer` into the program, while in a gradual language, the switch between static and dynamic is directed by types, with no `defer` or explicit casts needed. Moreover, there is no theorem about adding and removing `defer` in HLIO, while in a gradual language, the gradual guarantee theorem relates the runtime behavior of programs that differ only in the precision of their type annotations.

3 The Tension Between Gradual Typing and Information-Flow Control

Taking inspiration from gradual typing [51, 52], researchers have explored new ways to give programmers control over which parts of the program are secured statically versus dynamically, directed by *type annotations*. In general, gradually typed languages support the seamless transition between static and dynamic enforcement through the *precision* of type annotations. Gradual security is useful, because the programmer is free to choose when it is appropriate to increase the precision of the type annotations and put in the effort to pass the static checks and when it is appropriate to reduce the precision of type annotations, deferring the enforcement to runtime.

The main challenge in the design of gradually typed languages is controlling the flow of values (and information) between the static and dynamic regions of code, which is accomplished using runtime casts. Typically source programs are compiled to an intermediate language, called a cast calculus, that includes explicit syntax for runtime casts. Disney and Flanagan [20] design a cast calculus with IFC for a pure lambda calculus and prove noninterference. Fennell and Thiemann [22] design a cast

Table 1: Proposed sources of tension between security and the gradual guarantee

Language	Security (noninterference)	Gradual Guarantee	Type-guided classification	NSU checking	Runtime security labels
GSL_{Ref}	✓ Yes	✗ No	✓ Yes	✓ Yes	{low, high, ⋆}
GLIO	✓ Yes	✓ Yes	✗ No	✓ Yes	{low, high}
WHILE^{G}	✓ Yes	✓ Yes	✓ Yes	✗ No	{low, high, ⋆}
$\lambda^{\star}_{\text{IFC}}$	✓ Yes	✓ Yes	✓ Yes	✓ Yes	{low, high}

calculus named ML-GS with mutable references using the no-sensitive-upgrade (NSU) runtime checks of Austin and Flanagan [4]. Fennell and Thiemann [23] design a cast calculus for an imperative, object-oriented language.

The main property of gradually typed languages is the *gradual guarantee* [53], which states that removing type annotations should not change the runtime behavior. Adding type annotations should also result in the same behavior except that it may introduce more trapped errors because those new type annotations may contain mistakes. Since the formulation of the gradual guarantee as a criterion for gradually typed languages [53], researchers have explored the feasibility of satisfying both the gradual guarantee and noninterference. Toro et al. [61] identify a tension between the gradual guarantee and security enforcement. They analyze the semantics of runtime casts through the lens of Abstracting Gradual Typing [25] and propose a type-driven semantics for gradual security. However, Toro et al. [61] discover counterexamples to the gradual guarantee in the GSL_{Ref} language. They conjecture that it is not possible to enforce noninterference and satisfy the gradual guarantee at the same time.

Azevedo de Amorim et al. [8] conjecture one possible source of the tension: the *type-guided classification* performed in GSL_{Ref} [61]. They propose a new gradually typed language, GLIO, which sacrifices type-guided classification. They prove that GLIO satisfies both noninterference and the gradual guarantee using a denotational semantics. Bichhawat et al. [12] conjecture that *NSU checking* could be another possible source of the tension. As an alternative, they propose a hybrid approach that leverages static analysis ahead of program execution to determine the write effects in untaken branches. They study a simple imperative language with first-order stores and prove both noninterference and the gradual guarantee.

A ideal gradual IFC language (1) should enforce security by satisfying noninterference (2) should satisfy the gradual guarantee (3) should provide type-based reasoning through vigilance and type-guided classification (4) should not require addition static analyses prior to program execution. Contrary to the prior work, I am going to show

in my dissertation that one does not have to give up on any of the four requirements to resolve the tension between noninterference and the gradual guarantee. Instead, the real source of the tension is that GSL_{Ref} allows $\star$ as a *runtime* security label. By walking back this usual design choice, I am going to present a gradual IFC programming language $\lambda^{\star}_{\texttt{IFC}}$ and prove that $\lambda^{\star}_{\texttt{IFC}}$ satisfies both noninterference and the gradual guarantee without any sacrifices.

4 Key Enabling Insight and Thesis Statement

In GSL_{Ref}, one can write a literal such as $\texttt{true}_{\star}$ in a program. At runtime, the literal becomes a value of unknown security level. I observe that allowing $\star$ as a *runtime* security label is the main reason that GSL_{Ref} violates the gradual guarantee. That design of GSL_{Ref} was unusual because the unknown type $\star$ is traditionally used in gradual languages to represent the lack of static information, not the lack of dynamic information. The design is also unusual when compared to dynamic systems for IFC, as those systems do not use an unknown security level [2, 4, 19, 58, 7].

Based on this observation, I propose a new gradual, IFC language $\lambda^{\star}_{\texttt{IFC}}$, which (1) enforces information flow security, (2) satisfies the gradual guarantee, (3) enjoys type-based reasoning through free theorems, and (4) utilizes NSU checking to enforce implicit flows through the heap with no static analysis required. In $\lambda^{\star}_{\texttt{IFC}}$, runtime security labels do not include $\star$, only `low` and `high` (or any lattice of security labels). On the other hand, to support gradual typing, the security labels in a type annotation may include $\star$. Surprisingly, I discover that removing $\star$ from the runtime labels is sufficient to reclaim the gradual guarantee, without sacrificing type-guided classification as in GLIO or NSU checking as in WHILE^{G}. This finding is the primary contribution of this dissertation. In $\lambda^{\star}_{\texttt{IFC}}$, the security level of a literal defaults to `low`, similar to systems like Jif [42] and GLIO, but different from GSL_{Ref} and WHILE^{G}.

One might think that allowing $\star$ as a label on literals and therefore on values is necessary so that programmers can run legacy code (without any security annotations) in a gradual language, by making $\star$ the default label for literals. However, prior information-flow languages use `low` security as the default security label for literals [42] and for good reasons. The security of a literal is something that only the programmer can know. That is, the identification of high-security data in a program must be considered as an input to an information flow system, and not something that can or should be inferred. When migrating legacy code into a system that supports secure information flow, a necessary part of the process for the programmer is

to identify whether there is any high-security information in the legacy code. The choice of `low` as the default label is because most literals (if not all) in real programs are low security. In fact, it is bad practice to embed high-security literals, such as passwords, in program text.

My thesis is that a gradual IFC programming language is able to satisfy the gradual guarantee, as long as there is no $\star$ among runtime security labels:

> **My thesis:**
> **It is possible to design a gradual IFC programming language that satisfies both noninterference and the gradual guarantee, by excluding the unknown label $\star$ from runtime security labels.**

5 Technical Contributions

In general, I make the following technical contributions in my dissertation:

1. I identify the real cause of the tension between information flow security and the gradual guarantee.
2. I present $\lambda^{\star}_{\texttt{IFC}}$, the first gradual IFC language with type-guided classification that satisfies the gradual guarantee.
3. I design two coercion calculi that serve as the runtime IFC monitor: a coercion calculus for security labels and a coercion calculus for secure values.
4. I design an IFC cast calculus, named $\lambda^{c}_{\texttt{IFC}}$. $\lambda^{c}_{\texttt{IFC}}$ defines the dynamic semantics of $\lambda^{\star}_{\texttt{IFC}}$.
5. I mechanize in Agda the proofs for (1) the gradual guarantee of $\lambda^{\star}_{\texttt{IFC}}$, (2) compilation from $\lambda^{\star}_{\texttt{IFC}}$ to $\lambda^{c}_{\texttt{IFC}}$ preserves types, and (3) type safety of $\lambda^{c}_{\texttt{IFC}}$ by progress and preservation.
6. I prove noninterference for $\lambda^{\star}_{\texttt{IFC}}$.

6 Progress and Plan

I have completed the technical development of $\lambda^{\star}_{\texttt{IFC}}$. I defined the syntax, the type system, and the semantics of $\lambda^{\star}_{\texttt{IFC}}$. In particular, I defined the semantics of $\lambda^{\star}_{\texttt{IFC}}$ by

translation to a new security cast calculus $\lambda^{c}_{\texttt{IFC}}$. I defined a syntax, type system, and operational semantics for $\lambda^{c}_{\texttt{IFC}}$. I then compiled $\lambda^{\star}_{\texttt{IFC}}$ into $\lambda^{c}_{\texttt{IFC}}$ in a type-preserving way. In $\lambda^{c}_{\texttt{IFC}}$, *security coercions* serve as the runtime security monitor, in which I adapted ideas from the Coercion Calculus [30, 31] to IFC. A coercions on security labels can be an *injection* from a security label to $\star$ or a *projection* from $\star$ to a security label. Security labels on literals in $\lambda^{\star}_{\texttt{IFC}}$ become the sources of injections in $\lambda^{c}_{\texttt{IFC}}$, while runtime NSU checks are treated as a special kind of projection where the target is the security level of the memory location to modify. Information flow policies are enforced when security coercions are reduced to their normal forms. *Composing* coercions models *explicit flows*, while the action of *stamping* a coercion in normal form models *implicit flows*. In particular, security coercions are designed such that $\lambda^{\star}_{\texttt{IFC}}$ satisfies the gradual guarantee. I mechanized the proofs of (1) the gradual guarantee of $\lambda^{\star}_{\texttt{IFC}}$, (2) compilation from $\lambda^{\star}_{\texttt{IFC}}$ to $\lambda^{c}_{\texttt{IFC}}$ preserves types, and (3) type safety of $\lambda^{c}_{\texttt{IFC}}$. I proved noninterference for $\lambda^{\star}_{\texttt{IFC}}$ on paper.

I plan to finish the writing and submit my dissertation by May 2025. The tentative outline (as well as the schedule) of the dissertation is listed below.

DONE In Chapter 1, I motivate my research and describe the background. I show that security is important, and programming languages are useful for enforcing security through IFC. I review the literature on existing IFC enforcement mechanisms. I also review the tension between noninterference and the gradual guarantee. I then introduce the main technical achievement of my dissertation: a programming language called $\lambda^{\star}_{\texttt{IFC}}$ that satisfies both noninterference and the gradual guarantee. I have finished a draft of Chapter 1.

DONE In Chapter 2, I define the gradual IFC language $\lambda^{\star}_{\texttt{IFC}}$. I then demonstrate how $\lambda^{\star}_{\texttt{IFC}}$ works using examples. Finally, I define the static and the dynamic extremes of $\lambda^{\star}_{\texttt{IFC}}$. I have finished a draft of Chapter 2.

TODO In Chapter 3, I formalize the operational semantics of $\lambda^{\star}_{\texttt{IFC}}$ by defining its cast calculus $\lambda^{c}_{\texttt{IFC}}$. I take a three-step approach. First, I define a coercion calculus on security labels. Using this coercion calculus, I define security label expressions. Label expressions serve as my model of the program counter (PC) that restricts the security of memory locations where write operations are allowed to happen. Next, I define a second coercion calculus, whose purpose is to cast a program value from one type to another type. This coercion calculus serves as the representation of casts in the intermediate language $\lambda^{c}_{\texttt{IFC}}$. Finally, I present the design of my intermediate language, the cast calculus $\lambda^{c}_{\texttt{IFC}}$. I define the syntax, the type

system, and the operational semantics for $\lambda^{c}_{\texttt{IFC}}$. I then define a type-preserving compilation from $\lambda^{\star}_{\texttt{IFC}}$ to $\lambda^{c}_{\texttt{IFC}}$, so that the semantics of $\lambda^{\star}_{\texttt{IFC}}$ is given by $\lambda^{c}_{\texttt{IFC}}$. I am working on Chapter 3, and I plan to finish a draft by mid-February.

TODO In Chapter 4, I present the Agda mechanization of three major meta-theoretical results, including the gradual guarantee, compilation preserves types, and type safety. I plan to finish a draft of Chapter 4 by mid-March.

TODO In Chapter 5, I prove the noninterference theorem for $\lambda^{\star}_{\texttt{IFC}}$, by simulating its cast calculus $\lambda^{c}_{\texttt{IFC}}$ with the dynamic extreme of $\lambda^{\star}_{\texttt{IFC}}$, namely $\lambda^{\texttt{DYN}}_{\texttt{IFC}}$. The noninterference of $\lambda^{\texttt{DYN}}_{\texttt{IFC}}$ can be proved using the standard erasure technique. I plan to finish a draft of Chapter 5 by late March.

TODO In Chapter 6, I summarize this dissertation and discuss future research directions in gradual IFC. I plan to finish a draft of Chapter 6 by early April. I will keep revising my dissertation until the end of the Spring 2025 semester.

References

[1] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 2006. A logic for information flow in object-oriented programs. *ACM SIGPLAN Notices* 41, 1 (2006), 91–102.

[2] Aslan Askarov and Andrei Sabelfeld. 2009. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *2009 22nd IEEE Computer Security Foundations Symposium*. 43–59. https://doi.org/10.1109/CSF.2009.22

[3] Hala Assal and Sonia Chiasson. 2018. Security in the software development lifecycle. In *Fourteenth symposium on usable privacy and security (SOUPS 2018)*. 281–296.

[4] Thomas H Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. 113–124. https://doi.org/10.1145/1554339.1554353

[5] Thomas H Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. 1–12.

[6] Thomas H Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 165–178.

[7] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst.* 39, 3, Article 10 (may 2017), 56 pages. https://doi.org/10.1145/3024086

[8] Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling noninterference and gradual typing. In *Logic in Computer Science (LICS)*. https://doi.org/10.1145/3373718.3394778

[9] Gilles Barthe and Tamara Rezk. 2005. Non-interference for a JVM-like language. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. 103–112.

[10] D Elliott Bell and Leonard J La Padula. 1976. *Secure computer system: Unified exposition and multics interpretation*. Technical Report. MITRE CORP BEDFORD MA.

[11] Kenneth J Biba et al. 1977. Integrity considerations for secure computer systems. (1977).

[12] Abhishek Bichhawat, McKenna McCall, and Limin Jia. 2021. Gradual Security Types and Gradual Guarantees. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00015

[13] Aaron Bohannon, Benjamin C Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. 2009. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security*. 79–90.

[14] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/2784731.2784758

[15] Carole Cadwalladr. 2018. Facebook suspends data firm hired by Vote Leave over alleged Cambridge Analytica ties. *The Guardian* (2018).

[16] Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 463–475. https://doi.org/10.1109/ACSAC.2007.37

[17] Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243. https://doi.org/10.1145/360051.360056

[18] Dorothy E Denning and Peter J Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (1977), 504–513.

[19] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *2010 IEEE Symposium on Security and Privacy*. 109–124. https://doi.org/10.1109/SP.2010.15

[20] Tim Disney and Cormac Flanagan. 2011. Gradual Information Flow Typing. In *Workshop on Script to Program Evolution*.

[21] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the Asbestos operating system. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 17–30.

[22] L. Fennell and P. Thiemann. 2013. Gradual Security Typing with References. In *2013 IEEE 26th Computer Security Foundations Symposium*. 224–239. https://doi.org/10.1109/CSF.2013.22

[23] Luminous Fennell and Peter Thiemann. 2015. LJGS: Gradual Security Types for Object-Oriented Languages. In *Workshop on Foundations of Computer Security (FCS)*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.9

[24] M da C Freitas and Miguel Mira da Silva. 2018. GDPR Compliance in SMEs: There is much to be done. *Journal of Information Systems Engineering & Management* 3, 4 (2018), 30.

[25] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL 2016)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

[26] Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–11.

[27] Felipe González, Yihan Yu, Andrea Figueroa, Claudia López, and Cecilia Aragon. 2019. Global reactions to the cambridge analytica scandal: A cross-language social media study. In *Companion Proceedings of the 2019 world wide web conference*. 799–806.

[28] Ralf Christian Härting, Raphael Kaim, Nicole Klamm, and Julian Kroneberg. 2021. Impacts of the New General Data Protection Regulation for small-and medium-sized enterprises. In *Proceedings of Fifth International Congress on Information and Communication Technology: ICICT 2020, London, Volume 1*. Springer, 238–246.

[29] Nevin Heintze and Jon G Riecke. 1998. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 365–377.

[30] Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (June 1994), 197–230. https://doi.org/10.1016/0167-6423(94)00004-2

[31] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

[32] Joanne Hinds, Emma J Williams, and Adam N Joinson. 2020. "It wouldn't happen to me": Privacy concerns and perspectives following the Cambridge Analytica scandal. *International Journal of Human-Computer Studies* 143 (2020), 102498.

[33] S Kitchgaessner. 2017. Cambridge Analytica used data from Facebook and Politico to help Trump. *The Guardian* 26 (2017).

[34] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 321–334. https://doi.org/10.1145/1323293.1294293

[35] Gurvan Le Guernic. 2007. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. IEEE, 218–232. https://doi.org/10.1109/CSF.2007.10

[36] Gurvan Le Guernic and Thomas Jensen. 2005. Monitoring information flow. In *Proc. Workshop on Foundations of Computer Security*. 19–30.

[37] Peng Li and Steve Zdancewic. 2006. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE, 12–pp.

[38] Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical Computer Science* 411, 19 (2010), 1974 – 1994. https://doi.org/10.1016/j.tcs.2010.01.025 Mathematical Foundations of Programming Semantics (MFPS 2006).

[39] Scott Moore and Stephen Chong. 2011. Static analysis for efficient hybrid information-flow control. In *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE, 146–160. https://doi.org/10.1109/CSF.2011.17

[40] Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 228–241. https://doi.org/10.1145/292540.292561

[41] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. *SIGOPS Oper. Syst. Rev.* 31, 5 (oct 1997), 129–142. https://doi.org/10.1145/269005.266669

[42] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. *Jif 3.0: Java information flow*. http://www.cs.cornell.edu/jif

[43] Masarah Paquet-Clouston, Olivier Bilodeau, and David Décary-Hétu. 2017. Can we trust social media data? social network manipulation by an iot botnet. In *Proceedings of the 8th international conference on social media & society*. 1–9.

[44] Jürgen Pfeffer, Katja Mayer, and Fred Morstatter. 2018. Tampering with Twitter's sample API. *EPJ Data Science* 7, 1 (2018), 50.

[45] François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 319–330.

[46] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 186–199. https://doi.org/10.1109/CSF.2010.20

[47] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[48] Adil Hussain Seh, Mohammad Zarour, Mamdouh Alenezi, Amal Krishna Sarkar, Alka Agrawal, Rajeev Kumar, and Raees Ahmad Khan. 2020. Healthcare data breaches: insights and implications. In *Healthcare*, Vol. 8. MDPI, 133.

[49] Anuradha Sharma and Praveen Kumar Misra. 2017. Aspects of enhancing security in software development life cycle. *Advances in Computational Sciences and Technology* 10, 2 (2017), 203–210.

[50] Paritosh Shroff, Scott Smith, and Mark Thober. 2007. Dynamic Dependency Monitoring to Secure Information Flow. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. 203–217. https://doi.org/10.1109/CSF.2007.20

[51] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92. http://scheme2006.cs.uchicago.edu/13-siek.pdf

[52] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (LCNS, Vol. 4609)*. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[53] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics)*. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[54] Vincent Simonet and Inria Rocquencourt. 2003. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*. 152–165.

[55] Sean Sirur, Jason RC Nurse, and Helena Webb. 2018. Are we there yet? Understanding the challenges faced in complying with the General Data Protection Regulation (GDPR). In *Proceedings of the 2nd International Workshop on Multimedia Privacy and Security*. 88–95.

[56] Yelena Smirnova and Victoriano Travieso-Morales. 2024. Understanding challenges of GDPR implementation in business enterprises: a systematic literature review. *International Journal of Law and Management* (2024).

[57] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796816000241

[58] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*. 95–106. https://doi.org/10.1145/2034675.2034688

[59] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2012. Flexible dynamic information flow control in the presence of exceptions. *arXiv preprint arXiv:1207.1457* (2012).

[60] Curtis Steward Jr, Luay A Wahsheh, Aftab Ahmad, Jonathan M Graham, Cheryl V Hinds, Aurelia T Williams, and Sandra J DeLoatch. 2012. Software security: The dangerous afterthought. In *2012 Ninth International Conference on Information Technology-New Generations*. IEEE, 815–818.

[61] Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4, Article 16 (Dec. 2018), 55 pages. https://doi.org/10.1145/3229061

[62] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazieres. 2007. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems (TOCS)* 25, 4 (2007), 11–es. https://doi.org/10.1145/1314299.1314302

[63] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187. https://doi.org/10.3233/JCS-1996-42-304

[64] Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. IEEE Press, Piscataway, NJ, USA. https://doi.org/10.1109/SP40001.2021.00002

[65] Stephan Arthur Zdancewic. 2002. *Programming languages for information security*. Ph. D. Dissertation. https://www.proquest.com/docview/251799337

[66] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. 2011. Making information flow explicit in HiStar. *Commun. ACM* 54, 11 (2011), 93–101. https://doi.org/10.1145/2018396.2018419