FLOWSTLC: An Information Flow Control Type System Based On Graded Modality

1st Zhige Chen
Dept. of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
12413315@mail.sustech.edu.cn

2nd Junqi Huang
Dept. of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
12212226@mail.sustech.edu.cn

3rd Zhiyuan Cao
Dept. of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
12311109@mail.sustech.edu.cn

***Abstract*—In this project, we introduce the design and implementation of a simple functional programming language with a type system that enforces secure information flow. Building on the simply-typed lambda calculus (STLC), we extend the type system with a graded modality with a security semiring. Inspired by *Graded Modal Dependent Type Theory* [1], our system statically prevents unauthorized flows of sensitive data into public outputs, ensuring the noninterference property [2].**

***Index Terms*—Computer security, information flow, noninterference, security-type systems**

I. INTRODUCTION

Modern software often handles sensitive data that must remain confidential. The *informational flow control* (or IFC) aims to ensure that high-security inputs of the program do not improperly influence low-security outputs. For example, private user information like passwords should never be leaked to public logs. Formally, the security policy of *noninterference* requires that variations in high-security inputs have no observable effect on low-security outputs. Intuitively, this means that changing a secret value should never cause any change in what an outside observer at low level sees. Violating the IFC principle can lead to information leak, so enforcing noninterference is critical for system security.

Type systems [3] [4] have been proved effective at enforcing noninterference: they associate each value with a *security label* and constrain operations so that a well-typed program is guaranteed not to leak high-security information. For example, consider the following function:

$$\text{foo} = \lambda x : \texttt{Nat}.\lambda y : \texttt{Nat}.x + y \tag{1}$$

Then such coeffect systems might allow the function to be typed as

$$\text{foo} : \texttt{Nat}^{\texttt{Sec}} \to \texttt{Nat}^{\texttt{Pub}} \to \texttt{Nat}^{\texttt{Sec}} \tag{2}$$

where the first parameter is marked as a high-security input, and the second parameter is marked as a low-security input. If a low-security result depends on a high-security input, the type system would reject the program. Thus, those security type systems can automatically verify that programs adhere to confidentiality policies.

However, existing IFC type systems can be rather inflexible or coarse-grained. Many system treat security labels in a rigid, lattice-based way [5], and often difficult to coexist with other sophisticated type system constructs. To address these issues, we propose the FLOWSTLC, a more flexible and extensible type system that track informational flows with *graded modalities*. More specifically, we will design a simply typed lambda-calculus with integrated *graded necessity* for the semiring $\{\texttt{Pub} \sqsubseteq \texttt{Sec}\}$, where `Pub` represents low-security information, and `Sec` represents high-security information. This system will allow us precisely control how the values interact as a type of *coeffects* [6] [7], while remain easy to extend with other program reasoning constructs like the *usage analysis* [8].

II. FLOWSTLC: THE CORE CALCULUS

A. Syntax

The FLOWSTLC is a graded extension to the simply typed lambda-calculus resembling the Fuzz type system of Reed et al. [9]. It is also similar to coeffect calculi of Brunel et al. [10] and Gaboardi et al. [11].

The syntax of FLOWSTLC is a straightforward extension of STLC with two additional constructs for introducing and eliminating the graded necessity type $\Box_\ell T$. We also add built-in boolean, natural numbers, and unit type to it:

$$\begin{aligned}
\text{Term } t \ ::= \quad & x \mid t\ t \mid \lambda x.t \mid [t] \mid \textbf{let}\ [x] = t\ \textbf{in}\ t \\
& \texttt{true} \mid \texttt{false} \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t \mid n \\
& \texttt{iszero}\ t \mid () \mid \textbf{let}\ () = t\ \textbf{in}\ t \\
\text{Type } T \ ::= \quad & T \to^{\ell} T \mid \Box_r T \mid \texttt{Unit} \mid \texttt{Bool} \mid \texttt{Nat}
\end{aligned}$$

The syntax $[t]$ promotes a term to a graded modality, and **let** $[x] = t_1$ **in** t_2 eliminate the modalities by checking whether t_2 uses t_1 w.r.t. its grade, and, if satisfied, "unbox" the modality and substitute it into t_2. The function type $T \to^{\ell} T$ records how will the function use its argument by a grade ℓ. The graded modality $\Box_\ell T$ is a type constructor where ℓ comes from the *security level algebra*, i.e., a semiring $\mathbb{S} = \texttt{Pub} \sqsubseteq \texttt{Sec}$. The built-in booleans, natural numbers, and

unit types are easy and well-studied extensions, and we will omit the explanation of these constructs here.

Typing judgements are of the regular form $\Gamma \vdash t : T$ with the typing contexts of the form:

$$\text{Context } \Gamma \ ::= \ \emptyset \mid \Gamma, x : [T]_\ell$$

Contexts are either empty $\emptyset$, or can be extended with a *graded assumption* $x : [T]_\ell$. For graded assumptions, their grades ℓ capture their substructural behavior by describing how can they be used in a term, in this case, the grade capture the security level of information. We will use $dom(\Gamma)$ to denote the variables assigned by context Γ.

B. Typing

FLOWSTLC has all three standard typing rules of STLC:

$$\frac{}{\mathbf{0} \cdot \Gamma, x : [T]_{\mathbf{1}} \vdash x : T} \text{ T-VAR}$$

$$\frac{\Gamma, x : [T_1]_\ell \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \to^\ell T_2} \text{ T-ABS}$$

$$\frac{\Gamma_1 \vdash t_1 : T_{11} \to^\ell T_{12} \qquad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1 + \ell \cdot \Gamma_2 \vdash t_1 \ t_2 : T_{12}} \text{ T-APP}$$

The exchange rule and weakening rule that allow permutation and expansion of contexts is implicit here, instead we enables the permutation by a permutation lemma (see Permutation lemma). The differences with the standard rules are that these rules integrate grades into the premises and conclusions. For example, the T-VAR rule enforces that we only use x by requiring all other assumptions have a $\texttt{Sec}$ grade. And the abstraction rule T-ABS records the usage of variable x in function body by marking the function's type with the same grade of the assumption that assigns x. For the application rule T-APP, the concatenation combines all graded assumptions by the semiring addition $+$ (for more information see the Appendix A.3).

The next three rules build the structure of the security label semiring:

$$\frac{\Gamma \vdash t : T}{\ell \cdot \Gamma \vdash [t] : \square_\ell T} \text{ T-PRO}$$

$$\frac{\Gamma_1 \vdash t_1 : \square_\ell T_1 \qquad \Gamma_2, x : [T_1]_\ell \vdash t_2 : T_2}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } [x] = t_1 \textbf{ in } t_2 : T_2} \text{ T-LET}$$

$$\frac{\Gamma, x : [T_2]_{\ell_1} \vdash t : T_1 \qquad \ell_2 \sqsubseteq \ell_1}{\Gamma, x : [T_2]_{\ell_2} \vdash t : T_1} \text{ T-APPROX}$$

The *Promotion* rule T-PRO "promotes" a regular term t to a graded modality by propagating its resource requirement to the context by a *scalar multiplication* (for complete definition of scalar multiplication, see the Appendix A.3). The T-LET rule provides a way to eliminate graded modality via substitution, where graded value is "unboxed" and substituted into a graded assumption with matching grades. The context concatenation is also used in the conclusion. Finally, the *approximation* rule T-APPROX allows us to "loosen" the requirement if $\ell_2 \sqsubseteq \ell_1$.

The remaining rules specify the types of booleans, natural numbers, unit values, and conditionals, etc.

$$\frac{}{\mathbf{0} \cdot \Gamma \vdash \texttt{true} : \texttt{Bool}} \text{ T-TRUE}$$

$$\frac{}{\mathbf{0} \cdot \Gamma \vdash \texttt{false} : \texttt{Bool}} \text{ T-FALSE}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{Bool} \quad \Gamma_2 \vdash t_2 : T \quad \Gamma_2 \vdash t_3 : T \quad \ell \sqsubseteq \mathbf{1}}{\ell \cdot \Gamma_1 + \Gamma_2 \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : T} \text{ T-COND}$$

$$\frac{}{\mathbf{0} \cdot \Gamma \vdash 0 : \texttt{Nat}} \text{ T-ZERO}$$

$$\frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{succ } t : \texttt{Nat}} \text{ T-SUCC}$$

$$\frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{pred } t : \texttt{Nat}} \text{ T-PRED}$$

$$\frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{iszero } t : \texttt{Bool}} \text{ T-ISZERO}$$

$$\frac{}{\mathbf{0} \cdot \Gamma \vdash () : \texttt{Unit}} \text{ T-UNIT}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{Unit} \qquad \Gamma_2 \vdash t_2 : T}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } () = t_1 \textbf{ in } t_2 : T} \text{ T-UNITELIM}$$

A noteworthy point of these rules is the premise $\ell \sqsubseteq 1$ of rule T-COND. This requirement is added to prevent the *control flow attack* via *implicit flow*, in which the attacker uses some form of equality checking and control flow to leak information. For example, the following typing derivation is invalid in our system:

$$\frac{x : [\texttt{Bool}]_{\texttt{Pub}} \vdash x : \texttt{Bool} \quad \vdash 0 : \texttt{Nat} \quad \vdash 42 : \texttt{Nat} \quad \texttt{Sec} \sqsubseteq \mathbf{1}}{x : [\texttt{Bool}]_{\texttt{Sec}} \vdash \textbf{if } x \textbf{ then } 0 \textbf{ else } 42 : \texttt{Nat}}$$

This is the same solution in Abel and Bernardy [12] and Choudhury et al. [13].

C. Operational Semantics

Once the type-checker shows that a program is well-typed, the AST is interpreted to execute following a standard call-by-value evaluation strategy. To make proving the type preservation theorem more easy, we specify the operational semantics of FLOWSTLC in the small-step style. We first specify the value as lambda abstractions and literals:

$$\text{Value } v \ ::= \ \lambda x : T . t \mid n \mid \texttt{true} \mid \texttt{false} \mid ()$$

The call-by-value reduction relation $t \to t'$ is then defined as follows:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1 \ t_2 \rightsquigarrow t_1' \ t_2} \text{ E-APP1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{v_1 \ t_2 \rightsquigarrow v_1 \ t_2'} \text{ E-APP2}$$

$$\frac{}{(\lambda x : T . t) \ v \rightsquigarrow [x \mapsto v]t} \text{ E-APPABS}$$

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']} \text{ E-PRO}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let } [x] = t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let } [x] = t_1' \textbf{ in } t_2} \text{ E-LET-EVAL}$$

$$\frac{}{\textbf{let } [x] = [v] \textbf{ in } t \rightsquigarrow [x \mapsto v]t} \text{ E-LET-UNBOX}$$

The operational behavior of primitive values, functions, and conditionals is then specified in a intuitive manner:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \rightsquigarrow \textbf{if } t_1' \textbf{ then } t_2 \textbf{ else } t_3} \text{ E-IF-EVAL}$$

$$\frac{}{\textbf{if } \texttt{true} \textbf{ then } t_2 \textbf{ else } t_3 \rightsquigarrow t_2} \text{ E-IF-TRUE}$$

$$\frac{}{\textbf{if } \texttt{false} \textbf{ then } t_2 \textbf{ else } t_3 \rightsquigarrow t_3} \text{ E-IF-FALSE}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let } () = t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let } () = t_1' \textbf{ in } t_2} \text{ E-UNIT-ELIM-EVAL}$$

$$\frac{}{\textbf{let } () = () \textbf{ in } t \rightsquigarrow t} \text{ E-UNIT-ELIM}$$

$$\frac{t \rightsquigarrow t'}{\texttt{succ } t \rightsquigarrow \texttt{succ } t'} \text{ E-SUCC}$$

$$\frac{t \rightsquigarrow t'}{\texttt{pred } t \rightsquigarrow \texttt{pred } t'} \text{ E-PRED}$$

$$\frac{t \rightsquigarrow t'}{\texttt{iszero } t \rightsquigarrow \texttt{iszero } t'} \text{ E-ISZERO}$$

$$\frac{}{\texttt{pred } 0 \rightsquigarrow 0} \text{ E-PRED-ZERO}$$

$$\frac{}{\texttt{pred succ } t \rightsquigarrow t} \text{ E-PRED-SUCC}$$

$$\frac{}{\texttt{iszero } 0 \rightsquigarrow \texttt{true}} \text{ E-ISZERO-ZERO}$$

$$\frac{}{\texttt{iszero succ } t \rightsquigarrow \texttt{false}} \text{ E-ISZERO-SUCC}$$

D. Graded Modality

The central innovation of FLOWSTLC is the use of a graded modality $\square_\ell T$ to track and control information flow. This modality is parameterized by a security grade ℓ drawn from the semiring $\mathbb{S}$, where the ordering signifies that `Sec` information is more sensitive than `Pub`.

Intuitively, a term of type $\square_\ell T$ is a value that should be used according to some security policies, the grade ℓ attached to the modality acts as a capability or a requirement: it specifies the minimum security level of the context necessary to "unbox" and use the value.

The power of this approach lies in the algebraic structure of the security semiring. The semiring operations (meet for addition, join for multiplication) naturally model the combination of security constraints:

- **Context Concatenation** uses the join operation ($\sqcap$) to combine graded assumptions for the same variable. This reflects the principle that if a value can be used publicly, them it should also can be used secretly elsewhere. For example, using a variable in two different parts of a program, one requiring `Sec` and the other `Pub`, results in a combined requirement of `Pub` ($\texttt{Sec} \sqcap \texttt{Pub} = \texttt{Pub}$).
- **Context Scalar Multiplication** uses the meet operation ($\sqcup$). to strengthen the security requirements of a context. This is crucial in the T-PRO rule for ensuring that a promoted term does not leak its contents to a lower grade.

In essence, the graded modality $\square_\ell T$ serves as a security-aware container. The type system's rules ensure that the "capability" ℓ needed to open this container is always respected, thereby statically guaranteeing that high-sensitivity data cannot flow into low-sensitivity outputs.

E. A Simple Example

We demonstrate the system's capability of enforcing noninterferencee by showing the system would reject the following program:

$$\lambda x : \square_{\texttt{Sec}} T.\textbf{let } [y] = x \textbf{ in } y : \square_{\texttt{Sec}} T \to^{\texttt{Pub}} T$$

since it try to use a `Sec` value to compute a `Pub` value. To type this term, we first need to show that

$$x : [\square_{\texttt{Sec}} T]_{\texttt{Pub}} \vdash \textbf{let } [y] = x \textbf{ in } y : \square_{\texttt{Sec}} T \to^{\texttt{Pub}} T$$

then according to the premises of T-LET rule, we need to show the following

1) $x : [\square_{\texttt{Sec}} T]_{\texttt{Pub}} \vdash x : \square_{\texttt{Sec}} T$
2) $y : [T]_{\texttt{Sec}} \vdash y : T$

The (1) is trivial to show, but there is no rule that allows the derivation of (2). T-VAR rule doesn't help here because it requires that the used variable must have a `Pub` grade. So the term is ill-typed.

III. METATHEORY

A. Type Safety

Lemma 1 (Permutation)**.** *If $\Gamma \vdash t : T$ and Δ is a permutation of Γ, then $\Delta \vdash t : T$ and the derivation depth of the latter is the same as the former.*

The proof for the permutation lemma is trivial since assumptions in contexts are unrelated in a simply-typed context. Then we prove the well-typedness of substitution, this lemma is then used to establish syntactic type safety.

Lemma 2 (Well-typed Substitution)**.** *If $\Gamma_1 \vdash t_1 : T_1$ and $\Gamma_2, x : [T_1]_\ell \vdash t_2 : T_2$, then we have $\Gamma_1 + \ell \cdot \Gamma_2 \vdash [x \mapsto t_1]t_2 : T_2$.*

We then state the type safety lemma as follows:

Theorem 1 (Type Preservation)**.** *If $\Gamma \vdash t : T$, then either t is a value or there exists a t' s.t. $t \rightsquigarrow t'$ with $\Gamma \vdash t : T$.*

B. Strong Normalization

Theorem 2 (Strong Normalization)**.** *If t is a closed and well-typed FlowSTLC term, then t is normalizable.*

C. Fundamental Theorems and Noninterference

Our definition of type semantics by building a logical relation model following the work of Rajani et al. [14]. The key difference between our approach and the work of Rajani et al. is that our calculus use a coeffect-based analysis instead of effect-based analysis.

IV. Bidirectional Type Checking Algorithm

V. Implementation and Examples

VI. Related Work

Language-based IFC has a rich literature. Sabelfeld and Myers [3] survey a variety of security type systems that enforce noninterference via typing. Seminal work by Volpano et al. [15] introduced a static type system for a simple imperative language, ensuring well-typed programs satisfy noninterference. Extensions include JFlow [16] and JIF [17] for Java, which associate security labels with Java types to track flows, and FlowCaml [18] for OCaml, which similarly adds a security type-checker. Generally, these systems label each variable as high or low and enforce rules so that no operation can use a high value in a low context.

More recently, graded type theories have been proposed to generalize such analyses. Graded type systems annotate types with additional information (”grades”) to capture various properties of programs. For example, the Granule language [8] uses graded modalities to track the effects and coeffects of program. In information flow use-cases, types are graded by a security lattice, allowing automatic enforcement of noninterference. Moon et al. introduce the Graded Modal Dependent Type Theory (GRTT) [1], which extends dependent type system with graded modality and show that the grades can be effective at reasoning of programs. This work demonstrates that graded modality can capture fine-grained flow policies in types. Similarly, Marshall and Orchard 19 demonstrated a graded-modal framework that can enforce confidentiality and even integrity simultaneously.

Other approaches include dynamic IFC (e.g. LIO monad in Haskell [20]) or hybrid systems, but our focus is purely static. In summary, while classical security type sytems (JIF, FlowCaml, etc.) enforce noninterference via fixed lattice-labels. Our project draws on these ideas: we adapt graded modalities to a simple functional language to track information flows more flexibly.

VII. Further Work and Conclusion

In summary, FLOWSTLC has demonstrated that a simple lambda calculus with graded necessity can enforce secure information flow. By instantiating grades with the security semiring $\mathbb{S}$, our system statically tracks data labels at the type level. And with typing rules that enforcing the noninterference property, we guarantee that well-typed FLOWSTLC programs cannot leak secrets.

Despite these positive results, FLOWSTLC in its current form is an *idealized model* with clear limitations. For one, the core calculus is deliberately small and omits many practical language features. It has only base types and functions (no records, no algebraic data types, no recursive types, etc.), and it does not include polymorphism or dependent typing. As observed in prior work, static IFC models with minimal features have historically been considered ”too limited or too restrictive to be used in practice” [16]. Thus, our model inherits this limitation of expressiveness. Second, FLOWSTLC exists only as a theoretical calculus and has not been integrated into a full-blown programming language. We consider that bridging this gap is a non-trivial challenge. For example, a recent survey noted that even in a rich language like Rust, implementing a sound IFC system requires ”building an ad-hoc effect tracking system using bleeding-edge features of the Rust compiler”. Finally, we have not evaluated performance or conducted real-world case studies. There are no benchmarks on typing or runtime costs, nor have we applied FlowSTLC to any non-trivial programs. In contrast, systems like JFlow and Cocoon [21] have shown that static IFC checking can incur negligible overhead. Without such evaluation, the practical impact of FLOWSTLC remains speculative.

Given these limitations, there are many promising directions for future work. We highlight several concrete directions:

- **Extending the type system with dependent types and polymorphism.** One natural extension is to enrich FLOWSTLC with more expressive typing disciplines. In particular, adding *dependent types* could allow security policies to depend on program values and thus encode more precise confidentiality and declassification properties. Prior work (e.g. DepSec for Idris language [22]) shows that dependent types increase the expressiveness of static IFCs. Similarly, introducing *parametric polymorphism* would permit writing generic secure functions without duplicating code.
- **Combining graded modalities with effect, usage, or resource typing.** Another interesting direction is to study how the security grading interacts with other static analyses. Graded type theories generalize both effect systems (via graded monads) and coeffect/usage systems (via graded comonads). For example, FLOWSTLC might be extended with an effect system that tracks side-effects or I/O on secret data (similar to Koka’s effect typing system [23] [24]). Alternatively, one could explore *coeffect-like* analyses where the context is graded alongside security. Integrating graded security labels with *resource-* or *capability-aware typing* could yield richer guarantees.
- **Embedding FLOWSTLC in a practical language or compiler.** To bring theory closer to practice, a implementation of FlowSTLC’s typing discipline in a real programming environment is essential. One approach is to create a domain-specific language (DSL) or library for an existing language (e.g. Haskell or OCaml) that enforces these graded security types. Recent work (e.g. [21]) shows that it is possible to add IFC to Rust without modifying the compiler. Similarly, the Granule project [8] demonstrates that a language with graded, linear, and dependent types can be realized in practice. Following these examples, we could adapt FLOWSTLC’s type checker into an implementation (for instance, a GHC plugin) to test on larger codebases. Such an embedding would enable empirical measurement of type-checking performance and runtime costs. It would also allow exploration of practical issues (language interoperability, tooling integration, etc.) that

are vital for any real applications of our outcome.

In conclusion, FLOWSTLC provides a formal foundation for secure information flow via graded modal types, but many challenges remain. Future work will pursue richer type features, deeper integration with program reasoning (effects, resources), and practical implementation efforts. We are optimistic that bridging these gaps will bring secure flow typing closer to real-world programming. By pursuing these directions, we hope to develop a mature framework that combines the rigorous guarantees of our type-theoretic approach with the expressiveness and practicality needed for deployed systems.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to the Theoretical Computer Science StackExchange user taquetgauche, who reviewed our early prototype type systems and provided valuable feedback on how to proceed.

We are also deeply thankful to all members of the Theoretical Computer Science Society of SUSTech for their continued support and for the many insightful ideas they shared with us throughout this project.

REFERENCES

[1] B. Moon, H. Eades III, and D. Orchard, "Graded modal dependent type theory," in *European Symposium on Programming*. Springer International Publishing Cham, 2021, pp. 462–490.

[2] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*. Springer, 2007, pp. 291–307.

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 147–160.

[5] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[6] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: Unified static analysis of context-dependence," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2013, pp. 385–397.

[7] ——, "Coeffects: a calculus of context-dependent computation," *ACM SIGPLAN Notices*, vol. 49, no. 9, pp. 123–135, 2014.

[8] D. Orchard, V.-B. Liepelt, and H. Eades III, "Quantitative program reasoning with graded modal types," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–30, 2019.

[9] J. Reed, M. Gaboardi, and B. Pierce, "Distance makes the types grow stronger: A calculus for differential privacy (extended version)."

[10] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic, "A core quantitative coeffect calculus," in *European Symposium on Programming Languages and Systems*. Springer, 2014, pp. 351–370.

[11] M. Gaboardi, S.-y. Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, "Combining effects and coeffects via grading," *ACM SIGPLAN Notices*, vol. 51, no. 9, pp. 476–489, 2016.

[12] A. Abel and J.-P. Bernardy, "A unified view of modalities in type systems," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–28, 2020.

[13] P. Choudhury, H. Eades III, R. A. Eisenberg, and S. Weirich, "A graded dependent type system with a usage-aware semantics," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–32, 2021.

[14] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 233–246.

[15] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of computer security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[16] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[17] K. Pullicino, "Jif: Language-based information-flow security in java," *arXiv preprint arXiv:1412.8639*, 2014.

[18] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*, 2003, pp. 152–165.

[19] D. Marshall and D. Orchard, "Graded modal types for integrity and confidentiality," *arXiv preprint arXiv:2309.04324*, 2023.

[20] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Proceedings of the 4th ACM Symposium on Haskell*, 2011, pp. 95–106.

[21] A. Lamba, M. Taylor, V. Beardsley, J. Bambeck, M. D. Bond, and Z. Lin, "Cocoon: Static information flow control in rust," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 166–193, 2024.

[22] S. Gregersen, S. E. Thomsen, and A. Askarov, "A dependently typed library for static information-flow control in idris," in *International Conference on Principles of Security and Trust*. Springer, 2019, pp. 51–75.

[23] D. Leijen, "Koka: Programming with row polymorphic effect types," *arXiv preprint arXiv:1406.2061*, 2014.

[24] ——, "Algebraic effects for functional programming," *Technical Report. MSR-TR-2016–29. Microsoft Research technical report, Tech. Rep.*, 2016.

APPENDIX

A. Complete Specification of FLOWSTLC

1) Syntax:

Term	t	::=	x	*variable*
		|	$t\ t$	*application*
		|	$\lambda x.t$	*abstraction*
		|	$[t]$	*packing*
		|	**let** $[x] = t$: T **in** t	*unpacking*
		|	`true`	*true constant*
		|	`false`	*false constant*
		|	**if** t **then** t **else** t	*conditional*
		|	n	*natural number*
		|	`iszero` t	*zero test*
		|	$()$	*unit*
		|	**let** $() = t$ **in** t	*unit elimination*
Number	n	::=	0	*zero constant*
		|	`succ` n	*successor*
		|	`pred` n	*predecessor*
Type	T	::=	`Unit`	*unit type*
		|	`Nat`	*natural number type*
		|	`Bool`	*boolean type*
		|	$T \to^{\ell} T$	*function type*
		|	$\Box_r T$	*graded modality*
Value	v	::=	λx: $T.t$	*abstraction value*
		|	n	*natural number value*
		|	`true`	*true value*
		|	`false`	*false value*
		|	$()$	*unit value*
Context	Γ	::=	$\emptyset$	*empty context*
		|	Γ, x: $[T]_r$	*graded assumption*
Security	ℓ	::=	`Sec`	*high-security label*
		|	`Pub`	*low-security label*

2) Security Level Semiring:

Definition 1 (Security Level Semiring)**.** *The security level semiring $\mathbb{S}$ is a two-point lattice of security levels $\mathbb{S} = \{\texttt{Pub} \sqsubseteq \texttt{Sec}\}$ with*

- $\boldsymbol{0} = \texttt{Sec}$
- $\boldsymbol{1} = \texttt{Pub}$
- *Addition as the meet: $r + s = r \sqcap s$*
- *Multiplication as the join $r \cdot s = r \sqcup s$*

See Appendix B for a proof that this algebra is a indeed a semiring.

3) Auxiliary Definitions:

Definition 2 (Context Concatenation)**.**

$$\emptyset + \Gamma = \Gamma$$
$$\Gamma + \emptyset = \Gamma$$
$$(\Gamma, x : [T]_r) + (\Gamma', x : [T]_s) = (\Gamma + \Gamma'), x : [T]_{r+s}$$

Definition 3 (Context Scalar Multiplication)**.**

$$r \cdot \emptyset = \emptyset$$
$$r \cdot (\Gamma, x : [T]_s) = (r \cdot \Gamma), x : [T]_{(r \cdot s)}$$

4) Typing Rules:

$$\frac{}{\boldsymbol{0} \cdot \Gamma, x : [T]_{\boldsymbol{1}} \vdash x : T}\ \text{T-VAR}$$

$$\frac{\Gamma, x : [T_1]_\ell \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to^{\ell} T_2}\ \text{T-ABS}$$

$$\frac{\Gamma_1 \vdash t_1 : T_{11} \to^{\ell} T_{12} \qquad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1 + \ell \cdot \Gamma_2 \vdash t_1\ t_2 : T_{12}}\ \text{T-APP}$$

$$\frac{\Gamma \vdash t : T}{\ell \cdot \Gamma \vdash [t] : \Box_\ell T}\ \text{T-PRO}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_\ell T_1 \qquad \Gamma_2, x : [T_1]_\ell \vdash t_2 : T_2}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : T_2}\ \text{T-LET}$$

$$\frac{\Gamma, x : [T_2]_{\ell_1} \vdash t : T_1 \qquad \ell_2 \sqsubseteq \ell_1}{\Gamma, x : [T_2]_{\ell_2} \vdash t : T_1}\ \text{T-APPROX}$$

$$\frac{}{\boldsymbol{0} \cdot \Gamma \vdash \texttt{true} : \texttt{Bool}}\ \text{T-TRUE}$$

$$\frac{}{\boldsymbol{0} \cdot \Gamma \vdash \texttt{false} : \texttt{Bool}}\ \text{T-FALSE}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{Bool} \quad \Gamma_2 \vdash t_2 : T \quad \Gamma_2 \vdash t_3 : T \quad \ell \sqsubseteq \boldsymbol{1}}{\ell \cdot \Gamma_1 + \Gamma_2 \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 : T}\ \text{T-COND}$$

$$\frac{}{\boldsymbol{0} \cdot \Gamma \vdash 0 : \texttt{Nat}}\ \text{T-ZERO}$$

$$\frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{succ}\ t : \texttt{Nat}}\ \text{T-SUCC}$$

$$\frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{pred}\ t : \texttt{Nat}}\ \text{T-PRED}$$

$$\frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{iszero}\ t : \texttt{Bool}}\ \text{T-ISZERO}$$

$$\frac{}{\boldsymbol{0} \cdot \Gamma \vdash () : \texttt{Unit}}\ \text{T-UNIT}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{Unit} \qquad \Gamma_2 \vdash t_2 : T}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ () = t_1\ \textbf{in}\ t_2 : T}\ \text{T-UNITELIM}$$

5) Evaluation Rules:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{E-APP1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{v_1\ t_2 \rightsquigarrow v_1\ t_2'}\ \text{E-APP2}$$

$$\frac{}{(\lambda x : T.t)\ v \rightsquigarrow [x \mapsto v]t}\ \text{E-APPABS}$$

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\ \text{E-PRO}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ [x] = t_1'\ \textbf{in}\ t_2}\ \text{E-LET-EVAL}$$

$$\frac{}{\textbf{let}\ [x] = [v]\ \textbf{in}\ t \rightsquigarrow [x \mapsto v]t}\ \text{E-LET-UNBOX}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \rightsquigarrow \textbf{if}\ t_1'\ \textbf{then}\ t_2\ \textbf{else}\ t_3}\ \text{E-IF-EVAL}$$

$$\frac{}{\textbf{if } \texttt{true } \textbf{then } t_2 \textbf{ else } t_3 \rightsquigarrow t_2} \text{ E-IF-TRUE}$$

$$\frac{}{\textbf{if } \texttt{false } \textbf{then } t_2 \textbf{ else } t_3 \rightsquigarrow t_3} \text{ E-IF-FALSE}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let } () = t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let } () = t_1' \textbf{ in } t_2} \text{ E-UNIT-ELIM-EVAL}$$

$$\frac{}{\textbf{let } () = () \textbf{ in } t \rightsquigarrow t} \text{ E-UNIT-ELIM}$$

$$\frac{t \rightsquigarrow t'}{\texttt{succ } t \rightsquigarrow \texttt{succ } t'} \text{ E-SUCC}$$

$$\frac{t \rightsquigarrow t'}{\texttt{pred } t \rightsquigarrow \texttt{pred } t'} \text{ E-PRED}$$

$$\frac{t \rightsquigarrow t'}{\texttt{iszero } t \rightsquigarrow \texttt{iszero } t'} \text{ E-ISZERO}$$

$$\frac{}{\texttt{pred } 0 \rightsquigarrow 0} \text{ E-PRED-ZERO}$$

$$\frac{}{\texttt{pred succ } t \rightsquigarrow t} \text{ E-PRED-SUCC}$$

$$\frac{}{\texttt{iszero } 0 \rightsquigarrow \texttt{true}} \text{ E-ISZERO-ZERO}$$

$$\frac{}{\texttt{iszero succ } t \rightsquigarrow \texttt{false}} \text{ E-ISZERO-SUCC}$$

B. Proofs

1) Security Level Semiring:

Proof.

- **Associativity of addition**: $(a + b) + c = a + (b + c)$
  This is trivial since the semiring addition is meet, and join is associative in a lattice.
- **Commutativity of addition**: $a + b = b + a$
  This is also trivial since meet is commutative in a lattice.
- **Additive identity**: $a + \mathbf{0} = a$ for all a
  Since $0 = \texttt{Sec}$, and $\texttt{Sec}$ be the maximum in $\mathbb{S}$, we have
  $$\begin{aligned} \texttt{Sec} \sqcap \texttt{Sec} &= \texttt{Sec} \\ \texttt{Pub} \sqcap \texttt{Sec} &= \texttt{Pub} \end{aligned} \tag{3}$$
- **Associativity of multiplication**: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  This is trivial by the associativity of join operation.
- **Multiplication distributes over addition**: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$
  The distributivity trivially holds since $\mathbb{S}$ is a two-point lattice.
- **Multiplicative identity**: $a \cdot \mathbf{1} = a$ for all a
  Since $1 = \texttt{Pub}$ and $\texttt{Pub} \sqsubset \texttt{Sec}$, we have
  $$\begin{aligned} \texttt{Pub} \sqcup \texttt{Pub} &= \texttt{Pub} \\ \texttt{Pub} \sqcup \texttt{Sec} &= \texttt{Sec} \end{aligned} \tag{4}$$
- **Multiplication by 0 annihilates**: $\mathbf{0} \cdot a = a \cdot \mathbf{0} = \mathbf{0}$ for all a
  Since $0 = \texttt{Sec}$ and $\texttt{Pub} \sqsubset \texttt{Sec}$, we have
  $$\begin{aligned} \texttt{Sec} \sqcup \texttt{Sec} &= \texttt{Sec} \\ \texttt{Pub} \sqcup \texttt{Sec} &= \texttt{Sec} \end{aligned} \tag{5}$$

Thus the security level algebra is a semiring. □

2) Well-typed substitution: We state several technical lemmas without proof:

Lemma 3 (Restriction Collapse)**.** *For two typing contexts Γ_1 and Γ_2, we have*

$$\Gamma_{1|\Gamma_2} + \Gamma_{2|\overline{\Gamma_2}} = \Gamma_1$$

Lemma 4 (Context Shuffle 1)**.** *For typing contexts Γ_1, Γ_1', and Γ_2, variable x and type T, we have*

$$(\Gamma_1, x : T, \Gamma_1') + \Gamma_2 = (\Gamma_1 + \Gamma_{2|\Gamma_1}), x : T, (\Gamma_1' + \Gamma_{2|\overline{\Gamma_1}})$$

Lemma 5 (Context Shuffle 2)**.** *For typing contexts Γ_1, Γ_2, and Γ_2', variable x and type T, we have*

$$\Gamma_1 + (\Gamma_2, x : T, \Gamma_2') = (\Gamma_{1|\Gamma_2}), x : T, (\Gamma_1' + \Gamma_{2|\overline{\Gamma_1}})$$

Lemma 6 (Context Shuffle 3)**.**

Lemma 7 (Distribution of Scalar Multiplication over Context Concatenation)**.** *For typing context Γ and two security levels $\ell_1, \ell_2 \in \mathbb{S}$, we have*

$$(\ell_1 \cdot \Gamma) + (\ell_2 \cdot \Gamma) = (\ell_1 + \ell_2) \cdot \Gamma$$

Lemma 8.

Proof. By induction on a derivation of the form $\Gamma, x : S \vdash t : T$. For a given derivation, we proceed by cases on the final typing rule used in the proof.

- **Case** T-VAR: $t = z$ with $z : T \in (\Gamma, x : S)$
  We need to consider two subcases:
  1) if $z = x$, then $[x \mapsto s]z \to [x \mapsto s]x \to s$. So we need to show $\Gamma + \Delta \vdash s : S$. To get this first we apply the T-WEAK rule to the assumption $\Delta \vdash s : S$,
  2) otherwise $[x \mapsto s]z \to z$, and the result is immediate.
- **Case** T-ABS: $t = \lambda y : T_2.t_1$, $T = T_2 \to T_1$, $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$
  By convention we may assume that $x \neq y$ and $y \notin FV(s)$. Then by the permutation lemma and weakening rule, we have $\Gamma, \Delta, y : T_2, x : S \vdash t_1 : T_1$. Again we apply the two rules on the assumption $\Delta \vdash s : S$ to get $\Gamma, \Delta, y : T_2 \vdash s : S$. By the induction hypothesis, we have $\Gamma, \Delta, y : T_2 \vdash [x \mapsto s]t_1 : T_1$, then by T-ABS $\Gamma, \Delta \vdash \lambda y : T_2.[x \mapsto s]t_1 : T_2 \to T_1$, and this is our desired result.
- **Case** T-APP: $t = t_1\ t_2$, $\Gamma, x : S \vdash t_1 : T_2 \to T_1$, $\Gamma, x : S \vdash t_2 : T_2$, $T = T_1$
  By the induction hypothesis, we have $\Gamma, \Delta \vdash [x \mapsto s]t_1 : T_2 \to T_1$ and $\Gamma, \Delta \vdash [x \mapsto s]t_2 : T_2$. Then by T-APP, $\Gamma, \Delta \vdash [x \mapsto s]t_1\ [x \mapsto s]t_2 : T_1$, i.e., $\Gamma, \Delta \vdash [x \mapsto s](t_1\ t_2) : T_1$.
- **Case** T-WEAK: $t = t_1$, $\Gamma' \vdash t_1 : T$ where $\Gamma' \subseteq (\Gamma, x : S)$ and $x : S \in \Gamma'$
  By the induction hypothesis, we have $\Gamma', \Delta \vdash [x \mapsto s]t_1 : T$, then apply the T-WEAK, we have $\Gamma, \Delta \vdash [x \mapsto s]t_1 : T$.

- **Case** T-DER: $t = t_1$, $T = T_1$, $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$
  By the induction hypothesis and permutation lemma, we have $\Gamma, \Delta, y : T_2 \vdash [x \mapsto s]t_1 : T_1$. Then by the T-Der rule, we immediately have our desired result $\Gamma, \Delta, y : [T_2]_{\texttt{Pub}} \vdash [x \mapsto s]t_1 : T_1$.
- **Case** T-PRO:
  This case is impossible since T-PRO rule requires a purely graded context, and $x : S$ would violate the premise.
- **Case** T-LET: $t =$ **let** $[y] = t_1$ **in** t_2, $\Gamma_1, x : S \vdash t_1 : \Box_\ell T_1$, $\Gamma_2, y : [T_1]_\ell, x : S \vdash t_2 : T_2$, $T = T_2$
  Again by convention we may assume that $x \neq y$ and $y \notin FV(s)$. First we apply the induction hypothesis to get $\Gamma_1, \Delta \vdash [x \mapsto s]t_1 : \Box_\ell T_1$ and $\Gamma_2, \Delta, y : [T_1]_\ell \vdash [x \mapsto s]t_2 : T_2$. And it follows from the T-LET rule that $(\Gamma_1 + \Gamma_2), \Delta \vdash$ **let** $[y] = [x \mapsto s]t_1$ **in** $[x \mapsto s]t_2 : T_2$, i.e., $(\Gamma_1 + \Gamma_2), \Delta \vdash [x \mapsto s]($**let** $[y] = t_1$ **in** $t_2) : T_2$.
- **Case** T-APPROX: $t = t_1$, $\Gamma, x : S, y : [T_2]_{\ell_1} \vdash t_1 : T_1$, $\ell_1 \sqsubseteq \ell_2$, $T = T_1$.
  From the induction hypothesis and permutation lemma we can get $\Gamma, \Delta, y : [T_2]_{\ell_1} \vdash [x \mapsto s]t_1 : T_1$. Then we apply the T-APPROX rule to get our desired result $\Gamma, \Delta, y : [T_2]_{\ell_2} \vdash [x \mapsto s]t_1 : T_1$.

□

3) Well-typed Graded Substitution:

Proof. Similar to how we prove the regular substitution lemma, we proceed by induction on a derivation of the form $\Gamma, x : [S]_\ell \vdash t : T$.

- **Case** T-VAR: $t = z$ with $z : T \in (\Gamma, x : [S]_\ell)$
  Unlike the case for regular substitution lemma, z cannot be x since there is not rule that can introduce $\Gamma, \ell \cdot \Delta \vdash s : S$. So $z \neq x$, and we have $[x \mapsto s]z \to z$, and the result is immediate.
- **Case** T-ABS: $t = \lambda y : T_2.t_1$, $T = T_2 \to T_1$, $\Gamma, x : [S]_\ell, y : T_2 \vdash t_1 : T_1$
  By convention we may assume that $x \notin FV(s)$. Then by
- **Case** T-APP:
- **Case** T-WEAK:
- **Case** T-DER:
- **Case** T-PRO:
- **Case** T-LET:
- **Case** T-APPROX:

□

4) Type Preservation:

Proof. □