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] have been proved effective at enforcing noninterference: they associate each value with aa security label and constrain operations so that a well-typed program is guaranteed not to leak high-security information. For example, a simple rule is that the result of a computation must have the hightest security label of any input used in its computation. 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 [4], 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 $\{\texttt{Secret} \sqsubseteq \texttt{Public}\}$. This system will allow us precisely control how the values interact as a type of coeffects, while remain easy to extend with other program reasoning constructs like the *usage analysis* [5].

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.* [6]. It is also similar to coeffect calculi of Brunel *et al.* [7] and Gaboardi *et al.* [8].

The syntax of FLOWSTLC is a straightforward extension of STLC with two additional constructs for introducing and eliminating the graded necessity type $\Box_\ell T$:

$$\begin{aligned} \textsf{Term } t &::= x \mid t\ t \mid \lambda x.t \mid [t] \mid \textbf{let } [x] = t \textbf{ in } t \\ \textsf{Type } T &::= T \to T \mid \Box_r T \end{aligned}$$

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 graded modality $\Box_\ell T$ is a type constructor where ℓ comes from the *security level algebra*, i.e., a semiring $(\{\texttt{Public}, \texttt{Secret}\}, \sqcap, \texttt{Public}, \sqcup, \texttt{Secret})$.

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

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

Contexts are either empty $\emptyset$, or can be extended with either a regular assumption $x : T$ or a *graded assumption* $x : [T]_\ell$. The regular assumptions can be used exactly the same way as in STLC. 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, plus a explicit weakening rule:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR}$$

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

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

$$\frac{\Gamma \vdash t : T}{\Gamma, \Gamma' \vdash t : T} \text{ T-WEAK}$$

The exchange rule that allows permutation of contexts is implicit here, instead we enables the permutation by a permutation lemma (see Permutation lemma). The only difference with the standard rules is the application rule applies a *context concatenation* $\Gamma_1 + \Gamma_2$. The concatenation combines two contexts if they contain disjoint set of regular assumptions. Then the concatenation will combine all regular assumptions together and merge graded assumptions by the semiring addition $+$ (for more information see the Appendix).

The remaining rules build the structure of the security label semiring and connect the regular assumptions to the graded ones:

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma, x : [T_1]_{\texttt{Secret}} \vdash t : T_2} \text{ T-DER}$$

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

where $[\Gamma]$ denotes a context containing only graded assumptions.

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

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

The *Dereliction* rule T-DER converts a regular assumption to a graded assumption marked with `Secret`, since the assumption is potentially being used by the term t. 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). 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 $\ell_1 \sqsubseteq \ell_2$.

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:

$$\textsf{Value}\ v\ ::=\ \lambda x\colon T.t$$

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

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

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

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

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

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

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

D. Graded Modality

E. A Simple Example

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

$$\lambda x : \Box_{\texttt{Secret}} T.\textbf{let}\ [y] = x\ \textbf{in}\ [y] : \Box_{\texttt{Secret}} T \to \Box_{\texttt{Public}} T$$

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

$$x : \Box_{\texttt{Secret}} T \vdash \textbf{let}\ [y] = x\ \textbf{in}\ [y] : \Box_{\texttt{Secret}} T \to \Box_{\texttt{Public}} T$$

then we need to show the following

1) $x : \Box_{\texttt{Secret}} T \vdash x : \Box_{\texttt{Secret}} T$
2) $y : [T]_{\texttt{Secret}} \vdash [y] : \Box_{\texttt{Public}} T$

The (1) is trivial to show, but to prove the (2) typing judgement we need to use the T-Pro rule on the judgement $y : T \vdash y : T$ (easy to obtain by T-VAR rule), which requires us to show

$$y : [T]_\ell \vdash y : T \quad \text{and} \quad \ell \cdot \texttt{Public} = \texttt{Secret}$$

which is clearly impossible with $\texttt{Secret} \sqsubseteq \texttt{Public}$.

III. Metatheory

A. Substitution

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. The permutation lemma enables us to state the two substitution lemmas in a cleaner form, we first prove the well-typedness of substitution through regular assumptions:

Lemma 2 (Well-typed Substitution)**.** *If $\Gamma, x : S \vdash t : T$ and $\Delta \vdash s : S$, then $\Gamma, \Delta \vdash [x \mapsto s]t : T$.*

Lemma 3 (Well-typed Graded Substitution)**.** *If $\Gamma, x : [S]_\ell \vdash t : T$ and $[\Delta] \vdash s : S$, then $\Gamma, \ell \cdot \Delta \vdash [x \mapsto s]t : T$.*

B. Type Preservation

Theorem 1 (Type Preservation)**.**

C. Progression

D. Strong Normalization

E. Noninterference

F. Type Checking Algorithm

IV. Implementation and Examples

V. 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.* [9] introduced a static type system for a simple imperative language, ensuring well-typed programs satisfy noninterference. Extensions include JFlow [10] and JIF [11] for Java, which associate security labels with Java types to track flows, and FlowCaml [12] 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 [5] 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 13 demonstrated a graded-modal framework that can enforce confidentiality and even integrity simultaneously.

Other approaches include dynamic IFC (e.g. LIO monad in Haskell [14]) 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.

VI. 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 two-point security semiring $\{\texttt{Secret} \sqsubseteq \texttt{Public}\}$, 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" [10]. 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 [15] 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 [16]) 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 [17] [18]). 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. [15]) shows that it is possible to add IFC to Rust without modifying the compiler. Similarly, the Granule project [5] 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] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] ——, "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*
Type	T	::=	$T \to T$	*function type*
		|	$\square_r T$	*graded modality*
Value	v	::=	λx: $T.t$	*abstraction value*
Context	Γ	::=	$\emptyset$	*empty context*
		|	Γ, x: T	*assumption*
		|	Γ, x: $[T]_r$	*graded assumption*
Security	ℓ	::=	`Secret`	
		|	`Public`	

2) Security Level Semiring:

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

- $0 = \texttt{Public}$
- $1 = \texttt{Secret}$
- *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)**.**

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

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:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR}$$

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

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

$$\frac{\Gamma \vdash t : T}{\Gamma, \Gamma' \vdash t : T}\ \text{T-WEAK}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma, x : [T_1]_{\texttt{Secret}} \vdash t : T_2}\ \text{T-DER}$$

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

where $[\Gamma]$ denotes a context containing only graded assumptions.

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

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

5) Evaluation Rules:

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

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

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

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

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

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

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 meet 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**: $a + 0 = a$ for all a
  Since $0 = \texttt{Public}$ and $\texttt{Secret} \sqsubseteq \texttt{Public}$, we have

$$\begin{aligned} \texttt{Public} \sqcap \texttt{Secret} &= \texttt{Secret} \\ \texttt{Secret} \sqcap \texttt{Public} &= \texttt{Secret} \\ \texttt{Secret} \sqcap \texttt{Secret} &= \texttt{Secret} \\ \texttt{Public} \sqcap \texttt{Public} &= \texttt{Public} \end{aligned} \tag{1}$$

- **Associativity of multiplication**: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  This is trivial since the semiring multiplication is join, and join is associative in a lattice.
- **Multiplication distributes over addition**: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$
  Since the lattice is a chain (and thus distributive), this holds.
- **Multiplicative identity**: $1 \cdot a = a \cdot 1 = a$ for all a

Since $1 = \texttt{Secret}$ and $\texttt{Secret} \sqsubseteq \texttt{Public}$, we have

$$\begin{aligned} \texttt{Public} \sqcup \texttt{Public} &= \texttt{Public} \\ \texttt{Public} \sqcup \texttt{Secret} &= \texttt{Public} \\ \texttt{Secret} \sqcup \texttt{Public} &= \texttt{Public} \\ \texttt{Secret} \sqcup \texttt{Secret} &= \texttt{Secret} \end{aligned} \tag{2}$$

- **Multiplication by 0 annihilates**: $0 \cdot a = a \cdot 0 = 0$ for all a
  Since $0 = \texttt{Public}$ and $\texttt{Secret} \sqsubseteq \texttt{Public}$, we have

$$\begin{aligned} \texttt{Public} \sqcup \texttt{Public} &= \texttt{Public} \\ \texttt{Public} \sqcup \texttt{Secret} &= \texttt{Public} \\ \texttt{Secret} \sqcup \texttt{Public} &= \texttt{Public} \end{aligned} \tag{3}$$

Thus the security level algebra is a semiring. □

2) Well-typed substitution:

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 we apply the T-WEAK rule and the permutation lemma to the assumption $\Delta \vdash s : S$ which will give us desired result.
  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 $\Gamma, \Delta, y : [T_2]_{\texttt{Public}} \vdash [x \mapsto s]t_1 : T_1$.
- **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: $t = \textbf{let}\ [y] = t_1\ \textbf{in}\ t_2$, $\Gamma_1, x : S \vdash t_1 : \square_\ell T_1$, $\Gamma_2, y : [T_1]_\ell, x : S \vdash t_2 : T_2$, $T = T_2$
  Again by convention we may assume that $x \neq y$ and $y \notin FV(s)$. First we apply the induction hypothesis to get $\Gamma_1, \Delta \vdash [x \mapsto s]t_1 : \square_\ell T_1$ 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 $(\Gamma_1 + \Gamma_2), \Delta \vdash \textbf{let}\ [y] = [x \mapsto s]t_1\ \textbf{in}\ [x \mapsto s]t_2 : T_2$, i.e., $(\Gamma_1 + \Gamma_2), \Delta \vdash [x \mapsto s](\textbf{let}\ [y] = t_1\ \textbf{in}\ t_2) : T_2$.
- **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:
- **Case** T-ABS:
- **Case** T-APP:
- **Case** T-WEAK:
- **Case** T-DER:
- **Case** T-PRO:
- **Case** T-LET:
- **Case** T-APPROX:

□

4) Type Preservation:

Proof. □