The Art of Letting Go: Formal Declassification with Privacy Budgets in Velvet

Abstract

Differential privacy (DP) offers mechanisms with strong privacy guarantees to extract useful information from sensitive data, but it is largely concerned with only two states for data: public or private. In contrast, traditional secure *information-flow control (IFC)* systems (implementing e.g., *multi-level security (MLS)* or *role-based access control (RBAC)* models) prevent unauthorized data leakage across multiple security domains, but offer very little flexibility for extracting useful insights from sensitive data without compromising security at some stage of the process. This paper addresses the challenge of combining the two in the form of *budgeted declassification*: how to release sanitized information under a quantitative privacy budget within a generic IFC system. We present Velvet, a framework that seamlessly integrates DP into a language-based information flow control setting. The core idea is to treat declassification as a first-class citizen in the type system, allowing sensitive data to be declassified only through DP mechanisms with a specified privacy loss ϵ. We design a type system and formal semantics that enforce that every data release complies with DP requirements and label constraints, ensuring end-to-end guarantees of both security – noninterference (NI) up to ϵ-bounded leaks giving rise to a novel notion of *differential noninterference (DNI)* – and privacy. We also present a prototype implementation, demonstrating that Velvet is expressive and practical; with which users can write simple programs to perform analyses on sensitive data with multiple labels, having the system automatically handle policy enforcement and noise injection. Evaluation on several scenarios shows that Velvet requires an order of magnitude fewer lines of code to achieve the same functionality compared to ad-hoc solutions on top of existing DP or RBAC mechanisms (such as GUPT [25] and Apache Ranger [1]), while improving result accuracy for a given privacy budget and incurring minimal performance overhead (or even improving performance in some cases). Velvet enables safe and convenient data analysis in high-assurance systems by unifying the benefits of information flow control and differential privacy.

ACM Reference Format:

. 2025. The Art of Letting Go: Formal Declassification with Privacy Budgets in Velvet. In . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Effective data management in security-sensitive environments often necessitates a nuanced approach to controlling access to sensitive information. To this end, differential privacy (DP) systems have emerged as a powerful tool for dealing with privacy concerns when analyzing sensitive data, offering strong guarantees against the risk of identification or information leakage [13] when releasing data to the public. By introducing mathematical rigor, including through privacy budgets and controlled noise injection, DP offers a means to quantify and minimize information leakage, while maintaining a certain level of data utility and accuracy. However, DP only views data as either public or private, with no intermediate state or the ability to distinguish between different observers or private sources – as would be usual in many real-life scenarios that deal with data from multiple sources with different levels of trustworthiness or users with different roles and access rights.

Access control models, or more specifically, information-flow control (IFC) models, are designed to manage the flow of information in a way preventing unauthorized access to sensitive data. Such models traditionally rely on classification and strict rules to manage information flow [5, 22], by the use of some form of *label*-like levels in a lattice as in multi-level security (MLS) or tags and roles as in role-based access control (RBAC). While these models provide robust security guarantees, their rigidity in not allowing secure information release between labels (or removal of tags etc.) poses challenges in dynamic, real-world applications where diverse data sources, user roles, and operational needs require more sophisticated access mechanisms capable of controlled mixing of data.

Indeed, MLS schemes (like the Bell-LaPadula [6] model) have long been employed to govern the flow of information across confidentiality and integrity levels in high-assurance environments. Yet, settings with more users, larger numbers of data sources, and more complex operational needs often demand more flexible approaches. Big data platforms, for instance, frequently use policy-based or attribute-based access control frameworks – such as Apache Ranger [1] or Apache Sentry [2] – to define finer-grained and dynamically adaptable rules. These systems strive to accommodate evolving organizational roles, continuous analytics requirements, and vast data volumes without sacrificing security.

One key challenge is reconciling the tension between maintaining strong security guarantees and enabling practical data sharing. In many contexts, such as cross-domain collaborations or privacy-sensitive analytics, strict isolation of data is impractical. Instead, mechanisms are needed to allow controlled relaxation of access constraints – commonly referred to as "declassification." Declassification enables data classified with "higher" or "stronger" confidentiality labels to be accessed under specific, controlled conditions, balancing operational usability with security [5, 16, 21]. However, inadequately handled declassification may give rise to information leaks and serious privacy concerns [5, 18, 32] – which is why declassification is frequently considered a last resort or an "escape

hatch" in security policies; its presence can only indicate whether additional confidential data was exposed, but not *how much*.

This paper explores a novel approach to access control by integrating DP with data under different security labels and notably with corresponding declassification mechanisms. Such an integration provides a pathway for secure and adaptable access control, enabling necessary data sharing without compromising the privacy of sensitive information, even across different security domains. As integrating DP requires reasoning about the privacy characteristics of programs, we propose a framework that allows reasoning about the security and privacy properties of programs, without introducing highly specific constructs or requiring extensive manual analysis of each core algorithm as is commonly done in domain-specific implementations (e.g., specialized matrix operations in Duet [28]).

In short, our approach extends the concept of IFC with first-class DP-aware "budgeted declassifications," ensuring each release of sensitive data is accounted for under a quantitative privacy metric. We define a *dependent* type system (as introduced in [19]) that tracks both security labels and differential privacy budgets, enabling reasoning about how data flows and where noise insertion or partial declassification can occur safely.

In summary, the contributions of this paper are as follows:

- We introduce a novel approach for *budgeted declassification*, where every release of sensitive data is measured and regulated by a DP budget. This approach extends traditional IFC by quantifying and preventing repeated or cumulative leaks.
- We introduce a new language and framework, named Velvet, with a type system that explicitly tracks *both* security labels and DP budgets, enabling developers to write programs that mix label-based restrictions with noise injection.
- We define a property called *differential noninterference (DNI)* which ensures that the system's security guarantees are preserved even when declassifying data under a DP budget.
- We provide a formalism to reason about the confidentiality of multi-label data and the quantitative privacy provided by DP.
- We compare Velvet against three baseline approaches: (1) traditional RBAC/MLS without DP, (2) strict DP, and (3) ad-hoc differentially private declassification. Our evaluation highlights how Velvet addresses limitations such as coarse "public/private" logic, error-prone ad-hoc declassification, and complexity in specifying leak limits while reducing the development effort to produce queries capable of mixing data from different security domains in a differentially private manner. Velvet leads to an order of magnitude fewer lines of code, with comparable or better accuracy and performance when compared to ad-hoc implementations.

In the remainder of this paper, Section 2 presents some background for our work, and discusses the challenges of existing approaches to declassification. Section 3 motivates our unified approach, and introduces our Velvet framework with its syntax and semantics. Section 4 introduces our novel DNI property, and Section 5 proves the key lemmas needed to show that our language guarantees it. Section 6 evaluates several scenarios based on our prototype compiler, and Section 7 discusses some related work, discussing their relevance to the problem and our approach. Finally, Section 8 concludes the paper with some final remarks. Additionally, the appendix consists of the proofs that were omitted in Section 4 and Section 5.

2 Background

Flexible, well-defined mechanisms for controlling data exposure have become crucial in modern secure computing. Differential privacy (DP) has emerged as a powerful tool for managing privacy in data analysis, orthogonally IFC systems have long been used to enforce strict access controls based on security labels. While DP provides strong guarantees against identification and information leakage, it typically operates under a binary classification of data, as opposed to IFC models which often rely on classification of data into multiple tiers of sensitivity or confidentiality; Such a classification can be enforced through language-based techniques that embed labels in the type system (e.g., Jif [26], Paragon [8], LIO [30]), or through policy-driven frameworks used in large-scale infrastructures such as Apache Ranger and Sentry.

2.1 Differential Privacy (DP)

Introduced by Dwork et al. [13], DP provides a rigorous way to limit the impact of any single record on the output, typically by injecting carefully calibrated noise. The central notion is a privacy budget, often denoted by ϵ, which quantifies the maximum permissible divergence in outputs when one record changes. Smaller values of ϵ imply stronger privacy (more noise or stricter controls), while larger values allow more precise – but also more revealing – outputs. This principle has proven valuable in statistical analysis and machine learning [14], as it allows analysts to extract aggregate insights while bounding the risk of identifying individual records. Its relevance to declassification arises from the fact that DP effectively *controls* how much sensitive detail is released. Instead of an all-or-nothing "downgrade" from a secure tier to a public one , data can be shared in a differentially private manner that explicitly measures and throttles potential leakage.

DP considers sensitive data as a collection of records $x \in D^n$, where x_i represents the record (or data) contributed by an individual i. The *distance* between two such collections $x, y \in D^n$ is defined as $d(x, y) = |\{i \mid x_i \neq y_i\}|$, which counts the number of records that differ between x and y. This leads to a definition of *neighboring datasets* as two datasets that differ by at most one record, i.e., $d(x, y) \leq 1$. Then, a mechanism $\mathcal{M}$ is said to be ϵ-differentially private if and only if for all neighboring datasets $x, y \in D^n$ and all subsets of outputs $S \subseteq O$, the following holds:

$$\Pr[\mathcal{M}(x) \in S] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(y) \in S]$$

This property gives a probabilistic guarantee that the distributions of the output of the mechanism for any such neighboring datasets are indistinguishable (up to a limit permitted by the value ϵ), thus protecting individual privacy. The parameter ϵ can be seen as a privacy budget, controlling the trade-off between privacy and utility: smaller values of ϵ yield stronger privacy guarantees but less accurate results; it can also be extended to account for the concept of *function sensitivity*, which quantifies how much a function's output can change when a single record in its input is modified.

This concept is crucial for maintaining an overall global privacy budget across a program (or query).

For instance, the function $g(x) = x + 1$ has a sensitivity of 1, as changing the input x by one unit would change the output by 1. In general, the sensitivity of a function g can be defined as $S(g) = \max_{x,y:d(x,y)=1} |d(g(x), g(y))|$, where d is some distance metric on the corresponding output space.

The Laplace mechanism is a common way to achieve ϵ-DP [13], by adding noise drawn from a Laplace distribution at location 0 with scale $\frac{S(g)}{\epsilon}$ to the output of the function g. For simplicity, we will use $\mathcal{L}(b) \sim f$ for the probability density function $f(x) = \frac{1}{2b} e^{\left(-\frac{|x|}{b}\right)}$ going forward to refer to sampling the Laplace distribution with scale b at location 0:

$$\mathcal{M}_g(x) = g(x) + \mathcal{L}\left(\frac{S(g)}{\epsilon}\right)$$

2.2 Information-flow Control and Its Challenges

Multi-level security (MLS) systems are a classic instantiation of IFC, particularly in environments handling highly sensitive information, such as governmental, military, and financial institutions [6]. The fundamental premise is a strict, hierarchical classification—examples include "Top Secret," "Secret," "Confidential," and "Unclassified." These labels and the relationships between them are usually user-specified, and are not bound to any particular set (e.g. "Private"/"Public" or "High"/"Low"/"Public"); usually formalized as a relation (or possibly an ordering) $\prec$ over the set of labels, where $\ell_1 \prec \ell_2$ indicates that ℓ_1 is "less secure" than ℓ_2. Labeling and access restrictions help ensure that data in higher tiers remain inaccessible to lower tiers. Although this structure preserves confidentiality, it can become overly rigid when operational contexts demand selective data releases. Cross-domain collaboration often requires finer-grained control than a simple top-down approach can accommodate. For instance, intelligence agencies may wish to share limited intelligence with allies, or healthcare organizations might need to grant selective access to patient records [22] to e.g. researchers or other organizations.

As such, overly strict IFC deployments can lead to underutilization of valuable data, as ad-hoc or unmanaged "downgrading" from high to low levels risks introducing unforeseen security vulnerabilities. This tension has prompted research on mechanisms that allow systematic relaxation of confidentiality policies without compromising foundational guarantees. Declassification plays a pivotal role in these efforts by letting data flow to lower tiers under explicit, well-defined conditions.

Still, while MLS is historically significant, many modern systems employ more flexible or fine-grained access control mechanisms. Attribute-based access control (ABAC) and RBAC frameworks, such as Apache Ranger or Sentry, dynamically adapt to organizational roles and large-scale data analytics demands [1, 2]. Language-based IFC frameworks like Jif, Paragon, and LIO likewise offer expressive labeling systems embedded in the code itself.

2.3 Declassification

While strict enforcement of confidentiality ensures that high-sensitivity data does not flow to inappropriate confidentiality domains, real-world applications frequently require partial disclosure, known as *declassification*, to accommodate legitimate sharing needs; That is to say, declassification acknowledges that certain operational contexts require partial release of high-sensitivity information [3].

Declassification mechanisms must balance two competing goals. On one hand, they aim to prevent unauthorized leakage by restricting how and when sensitive data is made visible. On the other hand, they must provide enough flexibility to support a variety of operational and analytical tasks. Overly rigid controls can impede collaboration or data-driven insights, while permissive or ad-hoc releases can allow subtle leaks that undermine core security guarantees. Many prior works describe how to formalize declassification by using noninterference (NI) [20] principles but adding explicit "escape hatches" under policy constraints [3, 11]. Some efforts introduce quantitative metrics to limit how much information an attacker could learn from a given declassification step, reflecting the growing need for mechanisms that track not just *whether* information is released but also *how much*.

2.4 Declassification in IFC and Beyond

Even within the simple MLS, formalizing how and when such releases occur proves nontrivial. An overly permissive approach may unravel the provided protections by creating new covert channels or attack surfaces [18, 32], whereas strictly minimal releases can hinder legitimate needs. Many declassification models extend the principle of NI, which originally demanded that high-security inputs never affect observations at lower levels. In practice, NI is often relaxed to permit certain, justifiable flows [3].

Systems that rely on NI-based declassification must strike a delicate balance between security and usability. Rigid adherence to NI can block essential data flows, leading to repeated, situational policy exceptions that erode overall security posture. Research on more adaptive frameworks aims to codify these exceptions in a structured way – some introduce advanced policy languages or quantitative notions of how much data is being revealed, trying to maintain NI as a guiding ideal while trying to accommodate real-world data sharing.

2.5 Adapting to Various Access Control Schemes

All these approaches face the same fundamental challenge: deciding how to *downgrade* or *declassify* confidential data without losing key security properties. Each scheme uses its own notion of "levels," "labels," or "roles," so declassification must be adapted accordingly. A system with just two levels – "Public" and "Private" – may interpret any release of private data as a declassification event, while multi-level or role-based systems require more nuanced policies for partial sharing. Some ongoing research explores approximate knowledge or distributed policy checks (e.g., ANOSY [21]), emphasizing how real-world declassification often involves both local and global reasoning about risk.

3 The Velvet Framework

This section first motivates our unified approach to DP with differently labeled data, before providing an overview of our approach and introducing it via a formal language for corresponding programs (queries).

3.1 Motivation for a Unified Approach

Bringing DP into systems that manage multiple security constraint categories, classes, levels, or simply labels introduces an opportunity to quantify and limit the information disclosed by each declassification event. Traditional IFC setups treat a downgrade as a binary permission, whereas a DP-based approach can release noisy aggregates that preserve much of the data's utility while bounding privacy loss. Prior work on probabilistic and quantitative methods for declassification [5, 32] lays the foundation for bridging label-based security and formal leakage metrics, but direct integration of DP with corresponding policies is still an open area of exploration.

Despite the many advances in declassification and DP, practical integration remains nontrivial. Conventional IFC models can become overly strict, inadvertently stifling legitimate collaboration, while simple declassification rules risk uncontrolled leakage over time. Pure DP analyses, meanwhile, often assume a clear boundary between sensitive and nonsensitive data, but real-world systems feature multiple tiers, roles, or domains.

By mapping each declassification action to a privacy cost, one can distribute DP budgets across different users, roles, or classification levels. Highly sensitive data might impose a small ϵ allowance, limiting outputs to heavily perturbed summaries, while less sensitive domains might have a more generous allowance, enabling higher-fidelity information release. This form of *budgeted declassification* helps systems strike a balance between strict security for critical assets and flexibility for analytics or shared intelligence.

An approach that *unifies* IFC labeling with DP-based controls could offer fine-grained, auditable releases that remain verifiably within a predefined privacy budget. Systems capable of dynamically adjusting how much noise is introduced – or how large a budget is allocated – stand to accommodate changing trust relationships and evolving data needs. Such a model is especially valuable in settings like healthcare and financial analytics, where complete suppression of sensitive data is neither realistic nor desirable, and in intelligence-sharing contexts where partial yet secure disclosure is often a necessity. With careful coordination between policy definitions, IFC enforcement, and DP-based budget management, declassification can evolve from a one-off exception to a systematically governed process within secure computation.

3.2 Overview

Velvet provides a unified approach to integrating a general notion of labels – representing security or privacy constraints – with a DP budget. Unlike purely lattice-based methods, Velvet does not require a partial order among labels; rather, it only assumes the existence of definable transitions from one label to another, although a partial order may well be implied given certain transition sets. When data transitions from one label to another, *declassification* may occur, along with an appropriate consumption (or adjustment) of the DP budget.

The framework is thus sufficiently abstract to support a wide range of labeling conventions, including MLS, various RBAC schemes, or simple binary classifications (e.g., private vs public) as in classical DP. The central mechanism governing data re-labeling and DP budget consumption is the $\texttt{dpDowngrade}(\epsilon)$ relation introduced shortly, which specifies when a label transition is permissible and how it affects the budget.

3.3 Generic Label Model and Declassification

Labels in Velvet need not form a strict lattice. Instead, labels may be sets of roles, tags, or attributes, or they may follow a hierarchical structure akin to MLS. We denote the set of all labels as $\mathbb{L}$. The important requirement is that valid transitions between labels be well-defined: for any two labels ℓ_1 and ℓ_2, the system must determine whether re-labeling from ℓ_1 to ℓ_2 is allowed and, if so, how it affects the DP budget of the data being transitioned.

To allow for this flexibility, we define a partial ordering relation ($\ll$) over the set of labels, which should evaluate to whether a label is considered strictly "less secure" than another label.

The transition process is captured by the $\texttt{dpDowngrade}(\epsilon)$ relation, expressed as $\texttt{dpDowngrade}(\epsilon)(\ell_1, \epsilon_1, \ell_2, \epsilon_2)$, which holds if relabeling data from ℓ_1 (with current available budget ϵ_1) to ℓ_2 (with requested budget ϵ_2) is permissible with a cost of ϵ under the system's policy. This relation must satisfy the following properties:

- $\epsilon \leq \epsilon_1$, ensuring that the available budget is not exceeded,
- $\ell_2 \ll \ell_1$, indicating that the security policy permits the transition (directly or transitively), and
- $\forall \ell_i, \epsilon_i \, \exists \epsilon_4 \, \epsilon_5. \, \ell_2 \ll \ell_i \ll \ell_1 \wedge \texttt{dpDowngrade}(\epsilon_4)(\ell_1, \epsilon_1, \ell_i, \epsilon_i) \wedge \texttt{dpDowngrade}(\epsilon_5)(\ell_i, \epsilon_i, \ell_2, \epsilon_2) \Rightarrow \epsilon_4 + \epsilon_5 = \epsilon$, ensuring that the total cost of a multi-step transition is consistent with the individual steps.

Intuitively ℓ_1 and ℓ_2 are the original and target labels, ϵ_1 and ϵ_2 represent the original and requested DP budgets, and ϵ is the given budget spent to perform the transition. If the transition is disallowed by the policy, the relation must not hold.

Note, that for the purposes of this section, the form ϵ can be assumed to be a value in $\mathcal{R}^+$ (but keep in mind that this assumption will be revisited in later sections).

Notice how the $\texttt{dpDowngrade}(\epsilon)$ relation is *not* necessarily transitive, $\ell_1 \ll \ell_2 \wedge \ell_2 \ll \ell_3$ does not necessarily imply $\ell_1 \ll \ell_3$. This property allows users to define complex label structures that may not fit in a strict lattice, as is the case in many real-world systems (particularly in RBAC and MLS schemes [17, 31]).

In practice, users are expected to provide a less strict formulation of the $\texttt{dpDowngrade}(\epsilon)$ relation, expressed as:

$$\texttt{dpDowngrade}'(\ell_1,\epsilon_1,\ell_2,\epsilon_2) = \begin{cases} \epsilon \Rightarrow \texttt{dpDowngrade}(\epsilon)(\ell_1,\epsilon_1,\ell_2,\epsilon_2) \\ \bot \Rightarrow \nexists \epsilon \, \texttt{dpDowngrade}(\epsilon)(\ell_1,\epsilon_1,\ell_2,\epsilon_2) \end{cases}$$

where:

- ϵ is a privacy budget cost that would make $\texttt{dpDowngrade}$ hold, ensuring that the transition is valid.
- $\bot$ indicates that the transition is not allowed under the system's policy. (This could also be defined as ∞, as no possible available budget would make the transition valid.)

This formulation allows for a more user-friendly approach, while still enforcing the necessary constraints on differential privacy budget management and security policies; as `dpDowngrade` allows the system to pick the "best" path to perform multi-step declassifications automatically, without explicit user input.

Different instantiations of `dpDowngrade` yield varied security and privacy behaviors:

- Standard binary DP labels (classical private/public). If only two labels exist (ℓ_1 = `Public` $\ll$ ℓ_2 = `Private`), then any transition from private to public should deduct or adjust the DP budget in a manner consistent with classical differential privacy. This deduction may, for instance, subtract ϵ_2 from ϵ_1.
- MLS-style hierarchies. Traditional MLS can be captured by a set of partially-ordered labels (e.g., $\ell_1 \ll \ell_2 \ll \ell_3$ with ℓ_1 = `Unclassified`, ℓ_2 = `Secret`, and ℓ_3 = `TopSecret`).
- Tag-/role-based schemes. Labels may be defined as sets of roles or tags (e.g., {`Doctor`, `ICU`}), and valid transitions allow selective removal of these tags. `dpDowngrade` then specifies how much budget is expended whenever certain sensitive tags are removed (if allowed at all). If no quantitative privacy enforcement is required, the relation may not hold, thereby ignoring the DP budget aspect and reverting to standard MLS rules.
- Tag-/role-based schemes. Labels may be defined as sets of roles or tags (e.g., {`Doctor`, `ICU`}), and valid transitions allow selective removal of these tags. `dpDowngrade` then specifies how much budget is expended whenever certain sensitive tags are removed (if allowed at all).

3.4 Syntax

In Figure 1 we present the syntax of our language. We directly embed into the language the labels denoted as ℓ, natural numbers denoted as n, and non-negative reals r and s representing either sensitivity or budget constants. We denote by $\tilde{s}$ and $\tilde{\epsilon}$ symbolic names representing the sensitivity of an expression or function return value, and the privacy budget associated with a type respectively; the value represented by this symbol may be dependent on surrounding context (such as the types of function parameters).

The plain types τ comprise integer types `int` and lists of a statically known size `list` κ n. For simplicity, we will assume that the only primitive type is an abstract integer; however further primitives can be added easily. The annotated types κ include plain types annotated with the privacy budget variable denoted as $\tilde{\epsilon}$ and label $\tau^{\ell}_{\tilde{\epsilon}}$, the product type $\kappa \times \kappa$, and, finally, function types over annotated types with sensitivity $\kappa \xrightarrow{s} \kappa$.

The core primitives of our language are the two expressions `declassify`$(e, \ell, \epsilon_1, s, \epsilon_2)$ and `noise`(e, ϵ, s). The former attempts to declassify e from e's current label and budget to label ℓ with target budget ϵ_1 while spending ϵ_2 security budget from e, where ϵ_1 and ϵ_2 are symbolic budget expressions presented later in the syntax. The latter applies noise to e spending budget ϵ. In addition to the above, we have standard expressions for integer literals, variables, list indexing, tuple member access, list literals, operators and function calls, variable abstraction, iteration/folding, conditionals, and function literals. Note, the syntactic form [`for` x `in` $n_1..n_2$: e] is equivalent exactly to a static list constructor $[e_{n_1}, \ldots, e_{n_2-1}]$ where $e_i = e[i/x]$ for $i \in [n_1, n_2)$. Syntax elements highlighted in green may be omitted in user input, as they can be synthesized from the context by the type checker; for instance, s in `noise`(e, ϵ, s) may be omitted, as it is always equal to the sensitivity of e (which is tracked by the type system).

$$\boxed{\ell \in \mathbb{L} \quad n \in \mathbb{N} \quad r, s \in \mathbb{R}^+ \cup \{\infty\}}$$

τ	::=	`int`	[integer]
	|	`list` κ n	[statically sized list]
κ	::=	$\tau^{\ell}_{\tilde{\epsilon}}$	[annotated types]
	|	$\kappa \times \kappa$	[tuples]
	|	$\kappa \xrightarrow{s} \kappa$	[functions]
e	::=	`declassify`$(e, \ell, \epsilon, s, \epsilon)$	[declassification]
	|	`noise`(e, ϵ, s)	[explicit laplace noise]
	|	n	[integers]
	|	x	[variables]
	|	$e[e]$	[list index]
	|	$e.n$	[tuple index]
	|	$[e, \ldots, e]$	[list constructor]
	|	[`for` x `in` $n..n$: e]	[repeated list constructor]
	|	$op(e)$	[application]
	|	`let` x = e `in` e	[let binding]
	|	`for` x `in` e `index` x `accum` x = e `do` e	[fold]
	|	`if` e `then` e `else` e	[conditional]
	|	$proc$	
op	::=	$+ \mid - \mid * \mid / \mid , \mid \cdots \mid e$	
$proc$	::=	`proc` $(x : \kappa, \ldots) \xrightarrow{s} \kappa$ e	
ϵ	::=	$\tilde{\epsilon}$	[runtime budget variable]
	|	r	[constant budget]
	|	$r * \epsilon \mid \epsilon + \epsilon \mid \min(\epsilon, \epsilon)$	

Figure 1: Main syntax of the Velvet language. Parts highlighted in green can be omitted by the programmer.

Finally, we describe epsilon expressions ϵ, which can be a symbolic variable $\tilde{\epsilon}$, an exact value $r \in \mathbb{R}^+$, a product of an exact value and epsilon expression, a sum or a minimum of two expressions.

3.5 Semantics

We interpret computations in Velvet as computations in a probabilistic monad. The probability monad $\mathcal{D}(A)$ represents probability distributions over a set A, with $\text{return}(a) \in \mathcal{D}(A)$ denoting the point mass distribution at $a \in A$. For distribution $\mu \in \mathcal{D}(A)$ and a function $f : A \to \mathcal{D}(A')$, the *bind* operator, denoted $\mu \gg\!= f$ is the distribution obtained by sampling $a \sim \mu$ and then sampling $a' \sim f(a)$:

$$\mu \gg\!= f = \int_{a \in A} \int_{a' \in A'} \mu(da) \cdot f(a)(da')$$

To illustrate how the bind operation works, consider the following example:

Example 3.1. Consider the sets $A = \{a_0, a_1\}$, $A' = \{a'_0, a'_1\}$, and,

$$\mu = \frac{1}{2}a_0 + \frac{1}{2}a_1 \quad \text{and} \quad f(a) = \begin{cases} \frac{1}{4}a'_0 + \frac{3}{4}a'_1 & \text{if } a = a_0 \\ a'_0 & \text{if } a = a_1 \end{cases}$$

distributions as defined above. Then, $\mu \gg\!= f = \frac{1}{2}f(a_0) + \frac{1}{2}f(a_1)$ which yields $\frac{1}{2}a'_0 + \frac{1}{2}\left(\frac{1}{4}a'_0 + \frac{3}{4}a'_1\right) = \frac{5}{8}a'_0 + \frac{3}{8}a'_1$.

In Figure 2 we present denotational semantics of Velvet expressions denotes as $[\![e]\!](\sigma, \tilde{\epsilon}_r) \in \mathcal{D}(A)$ for a respective A, where σ is an environment mapping variables to values, and $\tilde{\epsilon}_r$ is an environment mapping symbolic budget variables $\tilde{\epsilon}$ to their respective values. The notation $\tilde{\epsilon}_r(\epsilon)$ is used as a shorthand for evaluating the symbolic expression ϵ in the context of the environment $\tilde{\epsilon}_r$ (which can be derived from the output $\tilde{\mathcal{E}}$ of a typing judgement as described in Section 3.6).

The projection operator $.n$ extracts the n-th element from a tuple; allowing access to individual components of product types, which is used in the semantic rule for tuple indexing expressions.

For conditional expressions, we use a branching function called $\text{branch}(e_T, e_F, \sigma, \tilde{\epsilon}_r)$ which selects between the distributions of the true-branch e_T and false-branch e_F based on the condition's evaluation. When a condition evaluates to a non-zero value, the true-branch is taken; otherwise, the false-branch is selected. Formally, this function can be expressed as:

$$\text{branch}(e_T, e_F, \sigma, \tilde{\epsilon}_r) = v_{\text{cond}} \mapsto \int_{v \neq 0} v_{\text{cond}}(dv) \cdot [\![e_T]\!](\sigma, \tilde{\epsilon}_r)(O) + \int_{v=0} v_{\text{cond}}(dv) \cdot [\![e_F]\!](\sigma, \tilde{\epsilon}_r)(O)$$

The fold function implements the recursive behavior of for-loops, processing list elements sequentially while accumulating results. It operates by evaluating the body expression e for each list element, passing the current value, index, and accumulator, then recursively continuing with the updated accumulator:

$$\begin{aligned} \text{fold}(e, [], d, a, \sigma, \tilde{\epsilon}_r) &= \text{return}(a) \\ \text{fold}(e, [v_h | v_t], d, a, \sigma, \tilde{\epsilon}_r) &= [\![e]\!](\sigma[x := v_h,\ i := d,\ a := a], \tilde{\epsilon}_r) \ggg (a' \mapsto \text{fold}(e, v_t, d+1, a', \sigma, \tilde{\epsilon}_r)) \end{aligned}$$

The semantics given below assign each expression e a function $[\![e]\!] : \Sigma, \tilde{\mathcal{E}}_r \to \mathcal{D}(v)$.

$$\begin{aligned}
[\![n]\!](\sigma, \tilde{\epsilon}_r) &= \text{return}(n) && \text{[literal]} \\
[\![x]\!](\sigma, \tilde{\epsilon}_r) &= \text{return}(\sigma(x)) && \text{[variable lookup]} \\
[\![[e_1, \ldots, e_n]]\!](\sigma, \tilde{\epsilon}_r) &= [\![e_1]\!](\sigma, \tilde{\epsilon}_r) \ggg (v_1 \mapsto \ldots [\![e_n]\!](\sigma, \tilde{\epsilon}_r) \ggg (v_n \mapsto \text{return}([v_1, \ldots, v_n]))) && \text{[list constructor]} \\
[\![e.n]\!](\sigma, \tilde{\epsilon}_r) &= [\![e]\!](\sigma, \tilde{\epsilon}_r) \ggg (v \mapsto \text{return}(v.n)) && \text{[tuple indexing]} \\
[\![e[i]]\!](\sigma, \tilde{\epsilon}_r) &= [\![i]\!](\sigma, \tilde{\epsilon}_r) \ggg (d \mapsto [\![e]\!](\sigma, \tilde{\epsilon}_r) \ggg (v \mapsto \text{return}(v[d]))) && \text{[list indexing]} \\
[\![op(e)]\!](\sigma, \tilde{\epsilon}_r) &= [\![op]\!](\sigma, \tilde{\epsilon}_r) \ggg (f \mapsto [\![e]\!](\sigma, \tilde{\epsilon}_r) \ggg (v \mapsto f(v))) && \text{[application]} \\
[\![\texttt{proc}(x_1 : \kappa, \ldots, x_n : \kappa) \xrightarrow{s} \kappa\ e]\!](\sigma, \tilde{\epsilon}_r) &= \text{return}(f) \\
&\quad \text{where } f = a_n \mapsto a_{n-1} \mapsto \cdots \mapsto a_n [\![e]\!](\sigma[x_1 \mapsto a_1, \ldots], \tilde{\epsilon}_r) && \text{[procedure]} \\
[\![\texttt{noise}(e, \epsilon, s)]\!](\sigma, \tilde{\epsilon}_r) &= [\![e]\!](\sigma, \tilde{\epsilon}_r) \ggg (v \mapsto v + \mathcal{L}(s/\tilde{\epsilon}_r(\epsilon))) && \text{[declassification]} \\
[\![\texttt{declassify}(e, \ell_t, \epsilon_2, s, \epsilon)]\!](\sigma, \tilde{\epsilon}_r) &= [\![e]\!](\sigma, \tilde{\epsilon}_r) \ggg (v \mapsto v + \mathcal{L}(s/\tilde{\epsilon}_r(\epsilon))) && \text{[explicit noise]} \\
[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!](\sigma, \tilde{\epsilon}_r) &= [\![e_1]\!](\sigma, \tilde{\epsilon}_r) \ggg (v \mapsto [\![e_2]\!](\sigma[x \mapsto v], \tilde{\epsilon}_r)) && \text{[let binding]} \\
[\![\texttt{if } e_{\text{cond}} \texttt{ then } e_T \texttt{ else } e_F]\!](\sigma, \tilde{\epsilon}_r) &= [\![e_{\text{cond}}]\!](\sigma, \tilde{\epsilon}_r) \ggg \text{branch}(e_T, e_F, \sigma) && \text{[conditional]} \\
[\![\texttt{for } x \texttt{ in } e_{\text{list}} \texttt{ index } i \texttt{ accum } a = e_{\text{accum}} \texttt{ do } e]\!](\sigma, \tilde{\epsilon}_r) &= [\![e_{\text{accum}}]\!](\sigma, \tilde{\epsilon}_r) \ggg (a_0 \mapsto [\![e_{\text{list}}]\!](\sigma) \ggg (l \mapsto \text{fold}(e, l, 0, a_0, \sigma))) && \text{[for loop]}
\end{aligned}$$

Figure 2: Semantic rules for the Velvet language.

3.5.1 Metric. To evaluate distance, we recursively define metrics:

$$\begin{aligned} d_{\texttt{int}}(n_1, n_2) &= |n_1 - n_2| \\ d_{\kappa_1 \to^{s} \kappa_2}(f_1, f_2) &= \max_{v:\kappa_1} d_{\kappa_2}(f_1(v), f_2(v)) \\ d_{\texttt{list}\ \kappa\ n}(l_1, l_2) &= \max_i d_\kappa(v_i^1, v_i^2), \text{ where } l_j = [v_i^j, v_2^j, \ldots] \\ d(\sigma_1, \sigma_2) &= \textstyle\sum_{x \in \text{dom}(\sigma_1) \cup \text{dom}(\sigma_2)} d(\sigma_1(x), \sigma_2(x)) \end{aligned}$$

We also adapt the following distance between distributions $\mu_1, \mu_2 \in \mathcal{D}(X)$ from Fuzz [29] to our setting:

$$d(\mu_1, \mu_2) = \max_{x \in X} \left| \ln \left(\frac{\mu_1(x)}{\mu_2(x)} \right) \right|$$

3.6 Typing Rules

In this section we present the typing rules for Velvet (see Figure 3). The typing judgement for expressions has shape $\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \kappa \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma'$ (with any or all of $\tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma'$ omitted to mean that they remain unchanged) where:

- the *sensitivity context* $\tilde{\mathcal{S}}$ maps sensitivity variables denoted as $\tilde{s}$ to their values,
- the *epsilon context* $\tilde{\mathcal{E}}$ maps symbolic budget variables denoted as $\tilde{\epsilon}$ to their values,
- the *typing environment* Γ assigns types κ and sensitivity variables $\tilde{s}$ to variables x,
- e is the expression being typed,
- $\tilde{s}$ is the sensitivity of e with respect to the inputs, and
- $\text{sensvars}(C)$ is the set of sens. variables in environment C,
- $\text{epsvars}(C)$ is the set of epsilon variables in environment C,
- $\text{vars}(e)$ is the set of variables in expression e,
- κ is the type of the expression.

Note that the judgement defined above is not well-formed unless $\text{epsvars}(\Gamma) \subseteq \tilde{\mathcal{E}}$ and $\text{sensvars}(\Gamma) \subseteq \tilde{\mathcal{S}}$.

$$\boxed{\Gamma \vdash x \overset{\tilde{s}}{:} \kappa \Leftrightarrow x \overset{\tilde{s}}{:} \kappa \in \Gamma \quad C[a \overset{*}{\mapsto} b] \Leftrightarrow C[a \mapsto b] \wedge a \notin \mathrm{dom}(C)}$$

(Int-Literal)
$$\frac{\tilde{\mathcal{S}}' = \tilde{\mathcal{S}}[\tilde{s} \overset{*}{\mapsto} 0] \qquad \tilde{\mathcal{E}}' = \tilde{\mathcal{E}}[\tilde{\epsilon} \overset{*}{\mapsto} \epsilon]}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash n \overset{\tilde{s}}{:} \mathrm{int}^{\ell}_{\tilde{\epsilon}} \dashv \tilde{\mathcal{E}}', \tilde{\mathcal{S}}'}$$

(Variable)
$$\frac{\Gamma \vdash x \overset{\tilde{s}}{:} \kappa \quad \tilde{\mathcal{S}}' = \tilde{\mathcal{S}}[\tilde{s} \mapsto \tilde{\mathcal{S}}(\tilde{s}) + 1]}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash x \overset{\tilde{s}}{:} \kappa \dashv \tilde{\mathcal{E}}, \tilde{\mathcal{S}}'}$$

(List-Access)
$$\frac{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} (\mathrm{list}\ \tau^{\ell_2}_{\tilde{\epsilon}_2}\ n)^{\ell_1}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \qquad \ell_3 = \max_{<:}(\ell_1, \ell_2) \qquad \tilde{\mathcal{E}}'' = \tilde{\mathcal{E}}'[\tilde{\epsilon}_3 \overset{*}{\mapsto} \min(\tilde{\mathcal{E}}'(\tilde{\epsilon}_1), \tilde{\mathcal{E}}'(\tilde{\epsilon}_2))]}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}', \Gamma \vdash e[i] \overset{\tilde{s}}{:} \tau^{\ell_3}_{\tilde{\epsilon}_3} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}''}$$

(Tuple-Access)
$$\frac{n \in \{0, 1\} \qquad \tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \kappa_0 \times \kappa_1 \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}'}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e.n \overset{\tilde{s}}{:} \kappa_n \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}'}$$

(Noise)
$$\frac{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \tau^{\ell}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \qquad \epsilon_2 \le \tilde{\mathcal{E}}'(\tilde{\epsilon}_1) \qquad \tilde{\mathcal{E}}'' = \tilde{\mathcal{E}}'[\tilde{\epsilon}_1 \mapsto \tilde{\mathcal{E}}'(\tilde{\epsilon}_1) - \epsilon_2]}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash \mathtt{noise}(e, \epsilon_2, \tilde{\mathcal{S}}'(\tilde{s})) \overset{\tilde{s}}{:} \tau^{\ell}_{\epsilon_2} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}''}$$

(Declassify)
$$\frac{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \mathrm{int}^{\ell_1}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \quad \mathrm{dpDowngrade}(\epsilon)(\ell_1, \tilde{\mathcal{E}}'(\epsilon_1), \ell_2, \epsilon_2) \qquad \tilde{\mathcal{E}}'' = \tilde{\mathcal{E}}'[\tilde{\epsilon}_1 \mapsto \tilde{\mathcal{E}}'(\tilde{\epsilon}_1) - \epsilon][\tilde{\epsilon}_2 \overset{*}{\mapsto} \epsilon_2]}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash \mathtt{declassify}(e, \ell_2, \epsilon_2, \tilde{\mathcal{S}}(\tilde{s}), \epsilon) \overset{\tilde{s}}{:} \mathrm{int}^{\ell_2}_{\epsilon_2} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}''}$$

(Application)
$$\frac{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash op \overset{\tilde{s}'}{:} \kappa_1 \to^{s} \kappa_2 \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \qquad \tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma \vdash e \overset{\tilde{s}_1}{:} \kappa_1 \dashv \tilde{\mathcal{S}}'', \tilde{\mathcal{E}}'' \qquad \tilde{\mathcal{S}}''' = \tilde{\mathcal{S}}''[\tilde{s}_2 \overset{*}{\mapsto} \tilde{\mathcal{S}}''(\tilde{s}_1) \cdot s][\tilde{s}'' \mapsto \tilde{\mathcal{S}}''(\tilde{s}'') \cdot s]_{\forall \tilde{s}'' \in \mathrm{sensvars}(\Gamma(\mathrm{vars}(e)))}}{\tilde{\mathcal{S}}'', \tilde{\mathcal{E}}, \Gamma \vdash op(e) \overset{\tilde{s}_2}{:} \kappa_2 \dashv \tilde{\mathcal{S}}''', \tilde{\mathcal{E}}''}$$

(Let)
$$\frac{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}_1}{:} \kappa_1 \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \qquad \tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma[x \overset{\tilde{s}_1}{\mapsto} \kappa_1] \vdash e' \overset{\tilde{s}_2}{:} \kappa_2 \dashv \tilde{\mathcal{S}}'', \tilde{\mathcal{E}}''}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e' \overset{\tilde{s}_2}{:} \kappa_2 \dashv \tilde{\mathcal{S}}'', \tilde{\mathcal{E}}''}$$

(List-Ctor)
$$\frac{\tilde{\mathcal{S}}_0 = \tilde{\mathcal{S}} \quad \tilde{\mathcal{E}}_0 = \tilde{\mathcal{E}} \qquad \forall i \in [1, n] \quad \tilde{\mathcal{S}}_{i-1}, \tilde{\mathcal{E}}_{i-1}, \Gamma \vdash e_i \overset{\tilde{s}_i}{:} \kappa \dashv \tilde{\mathcal{S}}_i, \tilde{\mathcal{E}}_i \qquad \tilde{\mathcal{S}}' = \tilde{\mathcal{S}}_n[\tilde{s} \overset{*}{\mapsto} \max(\tilde{\mathcal{S}}_n(\tilde{s}_i), \ldots)]}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash [e_1, \ldots, e_n] \overset{\tilde{s}}{:} \mathrm{list}\ \kappa\ n \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}_n}$$

(For)
$$\frac{\begin{array}{c}\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e_{\mathrm{acc}} \overset{\tilde{s}}{:} \kappa_2 \dashv \tilde{\mathcal{S}}'_2, \tilde{\mathcal{E}}'_2 \qquad \tilde{\mathcal{S}}'_2, \tilde{\mathcal{E}}'_2, \Gamma \vdash e_{\mathrm{lst}} \overset{\tilde{s}}{:} (\mathrm{list}\ \kappa\ n)^{\ell_1}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \\ \tilde{\mathcal{S}}'' = \tilde{\mathcal{S}}'[\tilde{s} \mapsto \tilde{\mathcal{S}}'(\tilde{s}) \cdot n][\tilde{s}' \overset{*}{\mapsto} 0][\tilde{s}'_1 \overset{*}{\mapsto} 0][\tilde{s}'_2 \overset{*}{\mapsto} 0] \qquad \tilde{\mathcal{E}}'' = \tilde{\mathcal{E}}'[\tilde{\epsilon} \overset{*}{\mapsto} \tilde{\mathcal{E}}'(\tilde{\epsilon}_1)/n] \\ \tilde{\mathcal{S}}'', \tilde{\mathcal{E}}'', \Gamma[x \overset{\tilde{s}'}{\mapsto} \kappa^{\ell_1}_{\tilde{\epsilon}}][x_i \overset{\tilde{s}'_1}{\mapsto} \mathrm{int}][x_a \overset{\tilde{s}'_2}{\mapsto} \kappa_2] \vdash e \overset{\tilde{s}}{:} \kappa_2 \dashv \tilde{\mathcal{S}}''', \tilde{\mathcal{E}}'''\end{array}}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash \mathtt{for}\ x\ \mathtt{in}\ e_{\mathrm{lst}}\ \mathtt{index}\ x_i\ \mathtt{accum}\ x_a = e_{\mathrm{acc}}\ \mathtt{do}\ e \overset{\tilde{s}}{:} \kappa_2 \dashv \tilde{\mathcal{S}}''', \tilde{\mathcal{E}}'''}$$

(If)
$$\frac{\begin{array}{c}\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \mathrm{int} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}' \qquad \tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma \vdash e_1 \overset{\tilde{s}_1}{:} \kappa \dashv \tilde{\mathcal{S}}_1, \tilde{\mathcal{E}}_1 \\ \tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma \vdash e_2 \overset{\tilde{s}_2}{:} \kappa \dashv \tilde{\mathcal{S}}_2, \tilde{\mathcal{E}}_2 \qquad \tilde{\mathcal{S}}_{12} = \tilde{\mathcal{S}}_1 \uplus \tilde{\mathcal{S}}_2 \qquad \tilde{\mathcal{S}}'' = \tilde{\mathcal{S}}_{12}[\tilde{s}' \overset{*}{\mapsto} \tilde{\mathcal{S}}_{12}(\tilde{s}_1) + \tilde{\mathcal{S}}_{12}(\tilde{s}_2)]\end{array}}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \overset{\tilde{s}'}{:} \kappa \dashv \tilde{\mathcal{S}}_1 \uplus \tilde{\mathcal{S}}_2, \tilde{\mathcal{E}}_1 \uplus \tilde{\mathcal{E}}_2}$$

(Procedure)
$$\frac{\tilde{\mathcal{S}}' = \tilde{\mathcal{S}}[\tilde{s}_1 \overset{*}{\mapsto} 0] \ldots [\tilde{s}_n \overset{*}{\mapsto} 0] \qquad \tilde{\mathcal{S}}', \tilde{\mathcal{E}}, \Gamma[x_1 \overset{\tilde{s}_1}{\mapsto} \kappa_1] \ldots [x_n \overset{\tilde{s}_n}{\mapsto} \kappa_n] \vdash e \overset{\tilde{s}}{:} \kappa_r \dashv \tilde{\mathcal{S}}'_i, \tilde{\mathcal{E}}'_i}{\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash \mathtt{proc}\ (x_1 : \kappa_1, \ldots) \to \kappa_r\ e \overset{0}{:} \kappa_1 \overset{0}{\to} \ldots \overset{\tilde{s}}{\to} \kappa_r \dashv \biguplus_i (\tilde{\mathcal{S}}'_i), \biguplus_i (\tilde{\mathcal{E}}'_i)}$$

Figure 3: Core typing rules for the Velvet language.

The intuition behind sensitivity tracking is that if we have $\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \kappa$, then substituting all symbols $\tilde{s}_i \in \tilde{\mathcal{S}}$ corresponding to x_i in Γ with the distance between v_i and v'_i, and computing $\tilde{s}$, we get the bound for the distance between e with inputs v_i and v'_i, i.e., $d(e[\bar{v}/\bar{x}], e[\bar{v}'/\bar{x}]) \le \tilde{\mathcal{S}}[d(v_i, v'_i)/\tilde{s}_i, \ldots](\tilde{s})$.

For example, suppose we have $f(g, w) = g(w, w)$ where f is a higher-order function, taking a function g as input, as well as an integer w. To correctly establish the type of f, we need to know the sensitivity of g (denoted by $\tilde{s}'$); which can be formulated as follows in the type system as defined in Figure 3:

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash f : (\mathtt{int} \to \mathtt{int} \overset{\tilde{s}'}{\to} \mathtt{int}) \to \mathtt{int} \overset{2 * \tilde{s}'}{\to} \mathtt{int}$$

This means that regardless of what the sensitivity of g is, the sensitivity of f is at most twice that of g. Concretely, for $g(w, z) = w + z$ of type $\mathtt{int} \to \mathtt{int} \overset{2}{\to} \mathtt{int}$, we have $\tilde{\mathcal{S}}(\tilde{s}') = 2$, therefore the upper bound for $d(f(g, w), f(g, w'))$ is 4 for $d(w, w') = 1$.

The typing rules below rely on a few operators and relations to handle declassification and track sensitivity: The $\mathtt{dpDowngrade}(\epsilon)$ relation specifies the budget ϵ for declassifying a value from label ℓ_1 to ℓ_2. Additionally, $\uplus$ is the symbolic sensitivity union operator which combines two contexts $\tilde{\mathcal{S}}_1$ and $\tilde{\mathcal{S}}_2$ using the merge function $\{\tilde{s} \mapsto s_1\} \uplus \{\tilde{s} \mapsto s_2\} = \{\tilde{s} \mapsto \max(s_1, s_2)\}$. We also use notation $\tilde{\mathcal{S}}[\tilde{s} \mapsto s]$ to define a mapping which is the same as $\tilde{\mathcal{S}}$, but with $\tilde{s}$ being mapped to s, and $\tilde{\mathcal{S}}[\tilde{s} \overset{*}{\mapsto} s]$ to define a mapping which is the same as $\tilde{\mathcal{S}}$, but with additionally a fresh variable (here called $\tilde{s}$) being mapped to s.

As promised in Section 3.3, we will now show that the simplifying assumption made there is no longer necessary, and that ϵ can itself be a symbolic expression (as defined by the ϵ rule in Figure 1),

with the operator $\geq$ denoting the comparison of such symbolic expressions. That operator can be defined recursively on the simplified, lexicographically sorted form (in terms of the $\tilde{\epsilon}$ variables) of the expressions.

The summation of expressions is defined for $\epsilon_1 = \sum_i r_i \tilde{\epsilon}_i$ and $\epsilon_2 = \sum_i t_i \tilde{\epsilon}_i$ in the standard form as $r * \epsilon_1 + t * \epsilon_2 = \sum_i (rr_i + tt_i) * \tilde{\epsilon}_i$.

Then, the $\min(\epsilon_1, \epsilon_2)$ form is defined as:

$$\min(\epsilon_1, \epsilon_2) = \begin{cases} \epsilon_1 & \text{if } \forall i\ r_i \leq t_i \\ \epsilon_2 & \text{if } \forall i\ t_i \leq r_i \\ \bot & \text{otherwise} \end{cases}$$

We also use the $\max_{\ll}(\ell_1, \ell_2)$ notation in Figure 3, defined as:

$$\max_{\ll}(\ell_1, \ell_2) = \begin{cases} \ell_1 & \text{if } \ell_1 \ll \ell_2 \\ \ell_2 & \text{if } \ell_2 \ll \ell_1 \\ \bot & \text{otherwise} \end{cases}$$

4 Differential Noninterference

In this section we present our novel formal property, DNI, which combines classical NI with DP guarantees including on declassification. The key idea is to require that, for any two initial memories that are indistinguishable at a given observer level (with some maximum observable label), the output distributions produced by a well-typed program in the Velvet system must differ only by the privacy loss induced by controlled declassifications.

To formally define the property we will use the following conventions and notations:

- for any label $\ell \in \mathbb{L}$, an observer with access to label ℓ can only see data with labels ℓ or ℓ' such that $\ell' \ll \ell$,
- m and m' denote two memory states,
- $\Gamma \vDash \sigma \Leftrightarrow \forall x.\ \vdash \sigma(x) : \Gamma(x)$, denoting that σ is well-typed with regards to Γ.

Definition 4.1 (Access-controlled Low Equivalence). Two states $\Gamma \vDash \sigma$ and $\Gamma \vDash \sigma'$ are *ℓ-equivalent*, denoted as $\Gamma \vdash \sigma \approx_\ell \sigma'$ iff for all $\Gamma(x) = \tau_\epsilon^{\ell'}$, s.t., $\ell' \ll \ell$ we have $\sigma(x) = \sigma'(x)$.

Following the existing works on DP support in programming languages, we introduce a *metric* on the values of each of the plain types denoted as $d_\tau(-,-)$ for plain type τ.

Definition 4.2 (Neighbor Memories). States σ and σ', are Γ-*neighbors* iff $\Gamma \vDash \sigma, \sigma'$ and for all x $\Gamma(x) = \tau_\epsilon^\ell \Rightarrow d_\tau(\sigma(x), \sigma'(x)) \leq 1$.

In this section we introduce the main privacy property, namely differential non-interference.

We call context Γ *epsilon closed* iff $\text{epsvars}(\Gamma) = \emptyset$.

Definition 4.3 (Differential Noninterference). Program e satisfies *differential noninterference* at label ℓ w.r.t. an epsilon-closed Γ and runtime epsilon context instance $\tilde{\epsilon}_r$ iff for all labels $\ell' \ll \ell$, all Γ-neighbor states σ and σ', s.t., $\Gamma \vdash \sigma \approx_{\ell'} \sigma'$, and for every measurable set of outputs $O \subseteq \mathcal{O}$, it holds that

$$\Pr[[\![e]\!](\sigma, \tilde{\epsilon}_r) \in O] \leq$$
$$\exp\left(\sum_{x:\Gamma(x)=\tau_r^{\ell^*}} [\ell' \ll \ell^*] \cdot r\right) \cdot \Pr[[\![e]\!](\sigma', \tilde{\epsilon}_r) \in O].$$

Note, in particular, that this property is a generalization of classical NI, which states that the output distributions of e are identical for all low-equivalent memories m and m' (here σ and σ'); and reduces to classic differential privacy when only a single observer with access to label ℓ is considered (i.e. public vs private).

```
let zip = proc(xs: list T 100 lab{L1} eps{e1},
  ys: list U 100 lab{L2} eps{e2})
  -> list (T lab{L1} eps{e1} × U lab{L2} eps{e2}) 100
    [for i in 0..100: (xs[i], ys[i])]
in let wavg = proc(xs: list int 100 lab{L1} eps{e1},
    ys : list int 100 lab{L2} eps{e2}) -> int lab{L2} eps{0}
  let x = for v in zip(xs, ys) index i accum x = 0 do (
    x + declassify(
      // spend e1/100 declassifying x[i]
      v.0, e1 / 100,
      // to label L2 with target budget e2/100
      L2, e2 / 100) * v.1
  )
  in res = x / 100
  // Spend e2/100 and add noise to result
  in noise(res, 0)
in wavg(...)
```

Figure 4: Weighted average program in Velvet

```
let noisymax = proc(
  xs: list int 100 lab{L1} eps{e1}
) -> int (for x in xs index i accum s = (0, 0) do
  let score = declassify(x, e1/100, Public, 0) in
    if score > s.0 then
      (score, declassify(x, 0, Public, 0))
    else s).1
in noisymax(...)
```

Figure 5: Classic report noisy max program in Velvet

```
let fold_mul = proc(
  xs: list (int lab{L1} eps{e1}, int lab{L2} eps{e2}) 500
) -> list int lab{L1} eps{e1} 500
  [for i in 0..500:
    xs[i].0 * declassify(xs[i].1, e2, L1, e1)]
in let reduce_add = proc(
  xs: list int lab{L1} eps{e1} 500
) -> int lab{L1} eps{e1}
  for x in xs index i accum s = 0 do s + x
in declassify(reduce_add(fold_mul(zip(
  ..., ...
))), e1, Public, 0)
```

Figure 6: Vector dot product calculation (as a fold) in Velvet

5 Proof Outline

We prove that every well-typed program in Velvet satisfies the following property, where $\tilde{\mathcal{E}}(\Gamma)$ denotes a context Γ with all epsilon variables evaluated according to $\tilde{\mathcal{E}}$. We denote by $[\![\tilde{\mathcal{E}}]\!]$ mapping from epsilon vars to their respective values resulting from evaluating all expressions in $\tilde{\mathcal{E}}$.

Theorem 5.1. *If $\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \tau_{\epsilon'}^{\ell'} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}'$ then e satisfies DNI at label ℓ' w.r.t. $(\tilde{\mathcal{E}} - \tilde{\mathcal{E}}')(\Gamma)$ and $[\![\tilde{\mathcal{E}}]\!]$.*

5.1 Assumptions and Setup

In this section, we will make use of the following assumptions:

- two memories σ and σ' are *low-equivalent* with label ℓ denoted as $\sigma \approx_\ell \sigma'$,
- the overall privacy budget observable at ℓ is $\sum_{d \in D_\ell} \epsilon_{(d)}$, where every declassification event in the semantics is stored together with its cost,
- for any well-typed expression e and measurable event O, the execution probability $\Pr[[\![e]\!](\sigma, \tilde{\epsilon}_r) \in O]$ is defined according to Figure 2.

The proof of the Theorem 5.1 requires the following lemma, which is proven in Appendix A.

Lemma 5.2 (Sensitivity Preservation). *Given a typing judgment,*

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \tau \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}'$$

then for all memories m, m', if $d(m, m') \leq 1$, we have:

$$d([\![e]\!](m, \tilde{\epsilon}_r), [\![e]\!](m', \tilde{\epsilon}_r)) \leq \tilde{\mathcal{S}}'(\tilde{s})$$

For brevity, we next show two nontrivial lemmas required for the proof of the Sensitivity Preservation Lemma.

Lemma 5.3 (Sensitivity of Variables). *Given a typing judgement,*

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash x \overset{\tilde{s}}{:} \kappa \dashv \tilde{\mathcal{E}}, \tilde{\mathcal{S}}'$$

then for all memories m, m', if $d(m, m') \leq 1$, we have:

$$d([\![x]\!](m, \tilde{\epsilon}_r), [\![x]\!](m', \tilde{\epsilon}_r)) \leq \tilde{\mathcal{S}}'(\tilde{s})$$

Proof. Suppose $e = x$. By definition, $\Gamma \vdash x \overset{\tilde{s}}{:} \kappa$ and:

$$\tilde{\mathcal{S}}' = \tilde{\mathcal{S}}[\tilde{s} \mapsto \tilde{\mathcal{S}}(\tilde{s}) + 1]$$

The semantics for variable lookup gives:

$$[\![x]\!](m, \tilde{\epsilon}_r) = m(x)$$

Given that the sensitivity annotation in Γ inductively guarantees that, for any two memories m and m' with $d(m, m') \leq 1$, we have,

$$d(m(x), m'(x)) \leq \tilde{\mathcal{S}}(x)$$

then, by the additional increment of sensitivity recorded in $\tilde{\mathcal{S}}'$ (i.e. $\tilde{\mathcal{S}}'(\tilde{s}) > 0$), we conclude:

$$d([\![x]\!](m, \tilde{\epsilon}_r), [\![x]\!](m', \tilde{\epsilon}_r)) \leq \tilde{\mathcal{S}}'(\tilde{s})$$

□

Lemma 5.4 (Sensitivity of Noise). *Given a typing judgement,*

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e_1 \overset{\tilde{s}}{:} \tau^{\ell}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}'$$

then for all memories m, m', if $d(m, m') \leq 1$, we have:

$$d([\![\texttt{noise}(e_1, \epsilon_2, s)]\!](m, \tilde{\epsilon}_r), [\![\texttt{noise}(e_1, \epsilon_2, s)]\!](m', \tilde{\epsilon}_r)) \leq \tilde{\mathcal{S}}'(\tilde{s}).$$

Proof. Suppose $e = \texttt{noise}(e_1, \epsilon_2, s)$. By the induction hypothesis, for any memories m, m' with $d(m, m') \leq 1$, we have:

$$d([\![e_1]\!](m, \tilde{\epsilon}_r), [\![e_1]\!](m', \tilde{\epsilon}_r)) \leq \tilde{\mathcal{S}}'(\tilde{s})$$

The evaluation of the noise expression is defined by the semantics as:

$$[\![\texttt{noise}(e_1, \epsilon_2, s)]\!](m, \tilde{\epsilon}_r) = [\![e_1]\!](m, \tilde{\epsilon}_r) \ggg \big(v \mapsto v + \mathcal{L}(s/\tilde{\epsilon}_r(\epsilon_1))\big).$$

Since $\mathcal{L}(s/\tilde{\epsilon}_r(\epsilon_1))$ is independent of the value of $[\![e_1]\!](m, \tilde{\epsilon}_r)$, the difference is exactly that of the outputs of e_1, and thus we obtain:

$$d([\![\texttt{noise}(e_1, \epsilon_2, s)]\!](m, \tilde{\epsilon}_r), [\![\texttt{noise}(e_1, \epsilon_2, s)]\!](m', \tilde{\epsilon}_r)) \leq \tilde{\mathcal{S}}'(\tilde{s}).$$

□

5.2 Key Lemmas

The proof of DNI proceeds by induction over the typing of Velvet. We highlight several key cases:

Base cases. Here, we demonstrate a few base sub-lemmas required for the proof of Theorem 5.1.

Lemma 5.5 (Constants). *For any constant n:*

$$\Pr[[\![\texttt{n}]\!](\sigma) \in O] = \Pr[[\![\texttt{n}]\!](\sigma') \in O]$$

Proof Outline. As constants are independent of the state, the observable distribution is trivially the same. □

Lemma 5.6 (Variable Lookup). *If x is a variable with security label of at most ℓ:*

$$\Pr[[\![\texttt{x}]\!](\sigma) \in O] \;=\; \Pr[[\![\texttt{x}]\!](\sigma') \in O]$$

Proof Outline. As $\sigma \approx_\ell \sigma'$ implies $\sigma(x) = \sigma'(x)$, the output distributions are equal. □

Declassification and noise injection. The declassification construct is written as $\texttt{declassify}(e, L, \epsilon, s, \epsilon')$, and its semantics are defined by first evaluating e and then applying a noise injection. Crucially, the measure of $\texttt{declassify}(e, L, \epsilon, s, \epsilon')$ is given by

$$\Pr\Big[[\![\texttt{declassify}(e, L, \varepsilon, s, \epsilon')]\!](\sigma) \in O\Big] \;=\; \Pr\Big[[\![\texttt{noise}\,(e\,, \epsilon_1\,, s)]\!](\sigma) \in O\Big],$$

where $\exists \epsilon_1. dpDowngrade(\epsilon_1)(L_e,\, \epsilon_{L_e},\, L,\, \epsilon')$. Thus, the extra privacy cost from declassification is incorporated by the Laplace mechanism $\mathcal{L}(\frac{s}{\epsilon_1})$. The following lemma captures this insight.

Lemma 5.7 (DNI for Declassify).

$$\Pr[[\![\texttt{declassify}(e, \epsilon, L, \epsilon', s)\epsilon'']\!](\sigma) \in O] \;\leq\; \exp\Big(\sum_{d \in D_\ell(e)} \epsilon_{(d)} + \epsilon_1\Big) \cdot \Pr[[\![\texttt{declassify}(e, \epsilon, L, \epsilon', s)\epsilon'']\!](\sigma') \in O].$$

Proof Outline. Based on the semantics, the evaluation of

$$\texttt{declassify}(e, L, \epsilon', s, \epsilon'')$$

is equivalent to that of $\texttt{noise}\,(e\,, \epsilon_1\,, s)$, with

$$\epsilon_1 = dpDowngrade'(L_e,\, \epsilon_{L_e},\, L,\, \epsilon').$$

Then, if the induction hypothesis yields,

$$\Pr[[\![e]\!](\sigma) \in O] \;\leq\; \exp\Big(\sum_{d \in D_\ell(e)} \epsilon_{(d)}\Big) \cdot \Pr[[\![e]\!](\sigma') \in O]$$

The privacy bound for $\texttt{declassify}(e, L, \epsilon', s, \epsilon'')$ inherits the bound from e and is multiplied by $\exp(\epsilon_1)$ to account for the extra noise injection. □

Lemma 5.8 (DNI for Noise). *The noise injection construct satisfies*

$$\Pr[[\![\texttt{noise}\,(e,\epsilon,s)]\!](\sigma) \in O] \;\leq\; \exp\Big(\sum_{d\in D_\ell(e)} \epsilon_{(d)} + \epsilon\Big) \cdot \Pr[[\![\texttt{noise}\,(e,\epsilon,s)]\!](\sigma') \in O].$$

Proof. This directly follows from the privacy guarantee of the Laplace mechanism. □

Interesting control flow constructs. In what follows, we will outline the proofs for the constructs `if` and `for`.

Lemma 5.9 (DNI for Conditionals). *If e_T and e_F satisfy DNI, then the conditional*

$$\texttt{if}\ e\ \texttt{then}\ e_T\ \texttt{else}\ e_F$$

satisfies

$$\Pr[[\![\texttt{if}\ e\ \texttt{then}\ e_T\ \texttt{else}\ e_F]\!](\sigma) \in O] \;\leq\; \exp\Big(\sum_{d\in D_\ell} \epsilon_{(d)}\Big) \cdot \Pr[[\![\texttt{if}\ e\ \texttt{then}\ e_T\ \texttt{else}\ e_F]\!](\sigma') \in O].$$

Proof Outline. In our semantics the conditional expression is interpreted via a branch function that, based on the distribution for the condition, selects the output distribution from either the `then` branch or the `else` branch. More precisely, if $branch(e_T, e_F, \sigma)$ returns the distribution corresponding to the condition, then for every measurable set O we have

$$branch(e_T, e_F, \sigma)(v)(O) = \begin{cases} [\![e_T]\!](\sigma)(O), & v \neq 0, \\ [\![e_F]\!](\sigma)(O), & v = 0. \end{cases}$$

As the branch function is defined as an integral over the condition's distribution, the monotonicity of the integration operators ensures that the privacy cost of the conditional is at most the sum of the costs incurred in the two branches. Thus, if both e_T and e_F satisfy DNI, then the overall conditional satisfies

$$\Pr\Big[[\![\texttt{if}\ e\ \texttt{then}\ e_T\ \texttt{else}\ e_F]\!](\sigma) \in O\Big] \leq \exp\Big(\sum_{d\in D_\ell} \epsilon_{(d)}\Big) \cdot \Pr\Big[[\![\texttt{if}\ e\ \texttt{then}\ e_T\ \texttt{else}\ e_F]\!](\sigma') \in O\Big].$$

□

Lemma 5.10 (DNI for Loops). *Assume that the sub-expressions e_{accum}, e_{list}, and e_{body} each satisfy DNI. Then the* `for` *loop*

$$\texttt{for}\ x\ \texttt{in}\ e_{list}\ \texttt{index}\ i\ \texttt{accum}\ a = e_{accum}\ \texttt{do}\ e_{body}$$

satisfies DNI.

Proof Outline. This result follows from the compositionality of the privacy bounds in the recursive fold underlying the `for` construct, and the monotonicity of the binding operator, which together ensure that the accumulated privacy cost is bounded by the sum of the costs of the individual iterations (whose count is determined by the length of the list and is known statically). □

6 Evaluation

To assess the practicality and effectiveness of Velvet, we conduct a series of benchmarks comparing it against existing systems. Our evaluation focuses on performance, accuracy, and code complexity, highlighting the benefits of integrating DP into IFC type systems.

6.1 Implementation

The Velvet language is implemented as a compiler that parses the described syntax using a handwritten parser. It enforces the type-checking rules outlined in Section 3 to ensure correctness across all expression forms. The compiler then generates C++ source code, which is subsequently compiled into native binaries using a standard C++ toolchain. This approach ensures efficient execution while preserving the strong guarantees established during typechecking.

6.2 Experimental Setup

We evaluate Velvet by comparing it against two categories of systems: (i) a conventional access-control platform with ad-hoc DP (Apache Hive [9] with Ranger policies), and (ii) a dedicated DP system with simulated label-awareness (GUPT [25]). Three queries were benchmarked on a 32-core AMD Ryzen 9950X CPU with 128GB of system memory:

- a *weighted average* (500 entries, $\epsilon = 1$) query (see Figure 4),
- a *report noisy max* (500 entries, $\epsilon = 500$) query (see Figure 5),
- an alternative fold-based implementation of *vector dot product* (500 entries, $\epsilon = 1$) (see Figure 6).

Note that the noisy max mechanism is given a higher budget as it operates on individual values rather than aggregates; using a lower budget would lead to significant accuracy loss (this applies to all systems), which is why we chose to use a higher budget for this query.

Each configuration was tested and measured over 500 runs, and we report average latency, result accuracy, and code complexity as a proxy for developer effort. Reported latencies may include compilation times, which is noted where applicable. To assess code complexity, we measure the number of lines of code (LOC) required to implement the same functionality in each system; the measures given here are the total LoC required to implement the query, including the noise addition and policy setup – which may be shared across queries, if applicable.

Our findings are summarized in Table 1, Table 2 and Table 3. The full experimental setup, including the code for the queries and the details of the systems used for comparison, is available in the supplementary material.

6.3 Performance

In the RBAC scenario (weighted average), Velvet incurs negligible overhead when enabling DP: the DP-enabled version runs in 0.443s vs. 0.436s without DP. By contrast, integrating DP into Apache Hive increases runtime from 0.2448s to 0.8337s—over 3× slower. Using an alternative (more functional style) implementation of the weighted average/vector dot product query, manually handling noise addition and declassification leads to significant accuracy loss (26.8% vs 58.7%) and runtime overhead (mainly due to the coding style), whereas Velvet largely retains the accuracy of the original query (59.9% vs 67.0%) while not incurring much performance overhead (0.535s vs 0.443s). In the MLS scenario (noisy max), Velvet again maintains low overhead (0.665s DP vs. 0.577s non-DP), whereas the Apache Hive baseline took 2.2786s without DP and had high variance with DP (0.8037s avg). GUPT's runtime was faster in isolation, but it excludes compilation and policy costs; Velvet's comparable runtime (0.0037s) matches this performance.

6.4 Accuracy and Utility

In the weighted average query, Velvet achieves higher result accuracy (67%) than both ad-hoc DP with Apache Hive (58.7%) as well as GUPT (58%). For noisy max ($\epsilon = 500$), Velvet again outperforms others: 65.2% accurate vs. 60.2% for ad-hoc DP with Apache Hive and 47.8% for GUPT. For the dot product, Velvet achieves 59.9% accuracy, which is the highest among the three systems (GUPT: 49.3%; Hive: 26.8%) – this is despite the algorithm being written in a functional programming style, which proves to make managing noise manually much more difficult to reason about, as well as less efficient. These results indicate that Velvet's type-directed noise calibration yields better utility under the same privacy budget.

6.5 Code Complexity

Velvet dramatically reduces development effort. The weighted average query required only 17 lines in Velvet, compared to 83 for the same effect with Apache Hive, and 48 for GUPT. Similarly, the noisy max query took 9 lines in Velvet versus 87 in Hive and 60 in GUPT. In the more functional style implementation of the vector dot product, Velvet required 13 lines of code, compared to the 95 lines in Hive and 58 in GUPT. This reduction is mainly due to Velvet's type system, which automatically infers noise parameters and declassification labels, while the other systems require manual specification of these parameters; as well as the need to write custom operators for the ad-hoc DP systems.

Table 1: Weighted average results (500 entries, $\epsilon = 1$)

System	Type	Avg time (s)	Range (s)	Acc%	LoC
Ranger	RBAC	0.2448	0.242–0.248	–	83
Ranger	DP-RBAC	0.8337	0.604–2.257	58.7	83
Velvet	RBAC	0.436	0.405–0.447	–	17
Velvet	DP-RBAC	0.443	0.393–0.445	67.0	17
GUPT*	DP-MLS	0.0172	0.014–0.024	58.9	48
Velvet*	DP-RBAC	0.0087	0.005–0.015	–	17

* Excludes compilation time.

Table 2: Report noisy max results (500 entries, $\epsilon = 500$)

System	Type	Avg time (s)	Range (s)	Acc%	LoC
Ranger	MLS	2.2786	2.2590–2.2880	–	87
Ranger	DP-MLS	0.8037	0.0040–2.1650	60.2	87
Velvet	MLS	0.577	0.576–0.711	–	9
Velvet	DP-MLS	0.665	0.647–0.717	65.2	9
GUPT*	DP-MLS	0.00249	0.0023–0.0090	47.8	60
Velvet*	DP-MLS	0.0037	0.0031–0.0166	–	9

* Excludes compilation time.

6.6 Discussion

Across all benchmarks, Velvet delivers superior trade-offs: better result accuracy, drastically lower code complexity, and performance comparable to or better than retrofitted systems. Its integration of DP into IFC yields principled, auditable declassification with minimal developer burden, and its type system ensures that privacy budgets are statically enforced while cleanly allowing for multiple programming styles (as elaborated on in Section 6.4).

Table 3: Vector dot product results (500 entries, $\epsilon = 1$)

System	Type	Avg time (s)	Range (s)	Acc%	LoC
Ranger	MLS	2.3798	2.3700–2.3930	–	95
Ranger	DP-MLS	2.8262	2.0040–3.2430	26.8	95
Velvet	MLS	1.8203	0.1838–1.8203	–	13
Velvet	DP-MLS	0.5356	0.1838–1.8203	59.9	13
GUPT*	DP-MLS	0.0035	0.0018–0.0190	49.3	58
Velvet*	DP-MLS	0.0037	0.0031–0.0166	–	13

* Excludes compilation time.

7 Related Work

In this section, we will review the existing work on or related to declassification, IFC, and how DP applies to the combination of the two; and in doing so, compare the existing work with how Velvet addresses the same problems.

7.1 IFC and Declassification

Information-flow control (IFC), particularly multi-level security (MLS), has long served as a foundational approach to enforcing confidentiality in secure computing systems. Classic MLS frameworks, such as Bell-LaPadula [6], introduced hierarchical confidentiality levels, subsequently expanded by models such as Biba [7] and Clark-Wilson [10]. While these models provide robust theoretical underpinnings, their rigid hierarchical structures often prove insufficiently flexible for practical data-sharing scenarios.

Declassification has thus emerged as a critical research area within IFC, aimed at facilitating controlled information release to balance security with operational needs. Early advances such as the noninterference (NI) model [20] introduced controlled declassification, further refined in lattice-based [12] and decentralized label models [27]. Despite these advancements, traditional IFC frameworks frequently lack the quantitative privacy guarantees necessary for contemporary applications.

7.2 Personalized Differential Privacy in IFC

Integrating differential privacy (DP) with IFC has become an active research area aimed at strengthening privacy guarantees in data-sharing environments. Traditional DP approaches, such as those by McSherry and Mironov [24] and Dwork et al. [14], typically ensure uniform privacy protections, disregarding individual privacy preferences or varying sensitivities across data contributors.

To address this limitation, recent works have explored personalized differential privacy (PDP), a generalized DP framework that allows individuals to specify varying privacy requirements for their data. Jorgensen et al. [23] introduced PDP as a mechanism to tailor privacy levels at the individual level, thus enhancing overall data utility by allocating privacy protection according to actual user preferences. Similarly, Ebadi et al. [15] presented a provenance-based accounting approach to PDP, enabling fine-grained tracking

of privacy budgets at the individual level and better utilization of privacy resources.

These personalized DP approaches demonstrate significant promise for integration with IFC systems, offering nuanced and quantifiable privacy guarantees that align well with modern data-sharing contexts.

7.3 Formal Methods in IFC and DP

Formal methods have been extensively utilized to verify and analyze security properties in IFC systems. Models such as Non-Interference [20], Relational Hoare Logic [26], and Probabilistic Relational Hoare Logic [4] provide rigorous frameworks for reasoning about secure information flows. However, these methods traditionally emphasize security and confidentiality without explicitly accounting for privacy guarantees like those provided by DP.

Integrating formal methods with DP offers a promising pathway for verifying privacy-preserving properties alongside traditional security properties. Such integration can help precisely capture the quantifiable privacy assurances introduced by DP within the rigorous analytical frameworks traditionally used for IFC. This combined approach provides a more comprehensive methodology for verifying security and privacy, essential for modern, data-intensive computing environments.

8 Conclusion

We introduced Velvet, a novel language and framework that unifies differential privacy and information flow control by enabling *budgeted declassification*. Our system enforces privacy budgets within a language-based IFC setting, ensuring that each release of sensitive data is mediated by differential privacy and subject to static analysis.

The evaluation demonstrates that Velvet achieves better result utility and dramatically reduces implementation complexity (by an order of magnitude) compared to ad-hoc integrations in existing systems like Apache Ranger and GUPT, while our type system ensures strong security and privacy guarantees – as well as minimizing runtime overhead.

By making declassification a first-class citizen in secure language design, and formulating it using DP, Velvet enables expressive, safe, and efficient analytics over data with multiple security labels (and/or levels). Its ability to enforce fine-grained privacy policies without sacrificing usability makes it well-suited for high-assurance domains like healthcare, finance, and intelligence sharing.

References

[1] apacheranger [n. d.]. Apache Ranger. https://ranger.apache.org/. Online; accessed January 23, 2025.

[2] apachesentry [n. d.]. Apache Sentry. https://sentry.apache.org/. Online; accessed January 23, 2025.

[3] Anindya Banerjee, Roberto Giacobazzi, and Isabella Mastroeni. 2007. What You Lose is What You Leak: Information Leakage in Declassification Policies. *Electronic Notes in Theoretical Computer Science* 173 (April 2007), 47–66. https://doi.org/10.1016/j.entcs.2007.02.027

[4] Gilles Barthe, Benjamin Grégoire, Justin Hsu, Pierre-Yves Strub, and Dominique Unruh. 2012. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *Proceedings of the 2012 ACM SIGPLAN Symposium on Principles of Programming Languages*. 193–206. https://doi.org/10.1145/2103656.2103679

[5] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Transactions on Programming Languages and Systems* 35, 3 (Nov. 2013), 1–49. https://doi.org/10.1145/2492061

[6] David Elliot Bell and Leonard J. LaPadula. 1973. *Secure Computer Systems: Mathematical Foundations and Model.* Technical Report MTR-2547. MITRE Corp.

[7] Kenneth J. Biba. 1977. *Integrity Considerations for Secure Computer Systems.* Technical Report MTR-3153. MITRE Corp.

[8] Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for Practical Programming with Information-Flow Control. In *Programming Languages and Systems*, Chung-chieh Shan (Ed.). Springer International Publishing, Cham, 217–232.

[9] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1773–1786. https://doi.org/10.1145/3299869.3314045

[10] David D. Clark and David R. Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. 184–194. https://doi.org/10.1109/SP.1987.10001

[11] Raimil Cruz and Éric Tanter. 2019. Polymorphic Relaxed Noninterference. In *2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019*. IEEE, 101–113. https://doi.org/10.1109/SECDEV.2019.00021

[12] 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

[13] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. *Calibrating Noise to Sensitivity in Private Data Analysis.* Springer Berlin Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[14] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407. https://doi.org/10.1561/0400000042

[15] Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now it's Getting Personal. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 69–81. https://doi.org/10.1145/2676726.2677005

[16] Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now it's Getting Personal. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM. https://doi.org/10.1145/2676726.2677005

[17] Sebastian Eggert and Ron van der Meyden. 2017. Dynamic intransitive noninterference revisited. *Formal Aspects of Computing* 29, 6 (2017), 1087–1120. https://doi.org/10.1007/s00165-017-0430-6

[18] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '13)*. ACM. https://doi.org/10.1145/2429069.2429113

[19] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. *SIGPLAN Not.* 48, 1 (Jan. 2013), 357–370. https://doi.org/10.1145/2480359.2429113

[20] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. *Proceedings of the 1982 IEEE Symposium on Security and Privacy* (1982), 11–20. https://doi.org/10.1109/SP.1982.10014

[21] Sankha Narayan Guria, Niki Vazou, Marco Guarnieri, and James Parker. 2022. ANOSY: approximated knowledge synthesis with refinement types for declassification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. ACM, 15–30. https://doi.org/10.1145/3519939.3523725

[22] Samuel Haney, Michael Shoemate, Grace Tian, Salil Vadhan, Andrew Vyrros, Vicki Xu, and Wanrong Zhang. 2023. Concurrent Composition for Interactive Differential Privacy with Adaptive Privacy-Loss Parameters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23, Vol. 9)*. ACM, 1949–1963. https://doi.org/10.1145/3576915.3623128

[23] Zach Jorgensen, Ting Yu, and Graham Cormode. 2015. Conservative or liberal? Personalized differential privacy. In *2015 IEEE 31st International Conference on Data Engineering*. 1023–1034. https://doi.org/10.1109/ICDE.2015.7113353

[24] Frank McSherry and Ilya Mironov. 2009. Differentially Private Recommender Systems: Building Privacy into the Netflix Prize Contenders. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 627–636. https://doi.org/10.1145/1557019.1557090

[25] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David E. Culler. 2012. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings*

of the 2012 ACM SIGMOD International Conference on Management of Data. 349–360. https://doi.org/10.1145/2213836.2213876

[26] 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* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

[27] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. 129–142. https://doi.org/10.1145/268998.266669

[28] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 172 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360598

[29] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/1863543.1863568

[30] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. *SIGPLAN Not.* 46, 12 (Sept. 2011), 95–106. https://doi.org/10.1145/2096148.2034688

[31] Ron Van Der Meyden. 2007. What, indeed, is intransitive noninterference?. In *Proceedings of the 12th European Conference on Research in Computer Security* (Dresden, Germany) *(ESORICS'07)*. Springer-Verlag, Berlin, Heidelberg, 235–250.

[32] Hao Zhu, Yi Zhuang, and Xiang Chen. 2014. Quantitative Robust Declassification. *Applied Mathematics & Information Sciences* 8, 4 (July 2014), 2055–2061. https://doi.org/10.12785/amis/080464

A Proofs

In the following proofs, we interchangeably use the notation $|a-b|$ to refer to the distance metric $d(a,b)$ defined in Section 3.5.1.

Proof of Theorem 5.2. We prove by induction on the structure of the typing derivation a stronger statement, namely, that for all

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \tau \dashv \tilde{\mathcal{S}}^*, \tilde{\mathcal{E}}^*.$$

and for all memory states m, m' satisfying

$$|m - m'| \le 1,$$

we have

$$\left|[\![e]\!](m, \tilde{\epsilon}_r) - [\![e]\!](m', \tilde{\epsilon}_r)\right| \le \mathrm{Sens}(e),$$

where we define $\mathrm{Sens}(e') := \tilde{\mathcal{S}}^*(\tilde{s})$. (For brevity we assume a fixed metric on memories so that the notation $|m - m'| \le 1$ makes sense elementwise.)

We now consider each case in the derivation.

Case 1 (Int-Literal): Suppose

$$e = n.$$

By (Int-Literal), we have

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash n \overset{\tilde{s}}{:} \mathrm{int}^{\ell}_{\tilde{\epsilon}} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

with $\tilde{\mathcal{S}}'$ defined as $\tilde{\mathcal{S}}[\tilde{s} \overset{+}{\mapsto} 0]$. By the semantics,

$$[\![n]\!](\sigma, \tilde{\epsilon}_r) = n,$$

so for any memories m, m' (recalling that the evaluation of n is independent of the memory),

$$\left|[\![n]\!](m, \tilde{\epsilon}_r) - [\![n]\!](m', \tilde{\epsilon}_r)\right| = |n - n| = 0 \le 0 = \tilde{\mathcal{S}}'(\tilde{s}).$$

Thus the property holds in this case.

Case 2 (Variable): Suppose

$$e = x.$$

By the (Variable) rule, we have

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash x \overset{\tilde{s}}{:} \kappa \dashv \tilde{\mathcal{E}}, \tilde{\mathcal{S}}',$$

where by definition $\Gamma \vdash x \overset{\tilde{s}}{:} \kappa$ and

$$\tilde{\mathcal{S}}' = \tilde{\mathcal{S}}[\tilde{s} \mapsto \tilde{\mathcal{S}}(\tilde{s}) + 1].$$

The semantics for variable lookup gives

$$[\![x]\!](m, \tilde{\epsilon}_r) = m(x).$$

Given that the sensitivity annotation in Γ guarantees that, for any two memories m and m' with $|m - m'| \le 1$, we have

$$|m(x) - m'(x)| \le \tilde{\mathcal{S}}(x),$$

then, by the additional increment of sensitivity recorded in $\tilde{\mathcal{S}}'$ (i.e. $\tilde{\mathcal{S}}'(\tilde{s}) \ge \tilde{\mathcal{S}}(x)$), we conclude

$$\left|[\![x]\!](m, \tilde{\epsilon}_r) - [\![x]\!](m', \tilde{\epsilon}_r)\right| \le \tilde{\mathcal{S}}'(\tilde{s}).$$

Case 3 (Noise): Suppose

$$e = \mathtt{noise}(e_1, \epsilon_2, s).$$

By the rule (Noise) we have a premise

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e_1 \overset{\tilde{s}}{:} \tau^{\ell}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

By the induction hypothesis, for any memories m, m' with $|m - m'| \le 1$, we have

$$\left|[\![e_1]\!](m, \tilde{\epsilon}_r) - [\![e_1]\!](m', \tilde{\epsilon}_r)\right| \le \tilde{\mathcal{S}}'(\tilde{s}).$$

The evaluation of the noise expression is given by

$$[\![\mathtt{noise}(e_1, \epsilon_2, s)]\!](m, \tilde{\epsilon}_r) = [\![e_1]\!](m, \tilde{\epsilon}_r) \ggg \left(v \mapsto v + \mathcal{L}(s/\tilde{\epsilon}_r(\epsilon_1))\right).$$

Given that the sample $\mathcal{L}(s/\tilde{\epsilon}_r(\epsilon_1))$ is independent of v, the difference is exactly that of the outputs of e_1, and thus we obtain:

$$\left|[\![\mathtt{noise}(e_1, \epsilon_2, s)]\!](m, \tilde{\epsilon}_r) - [\![\mathtt{noise}(e_1, \epsilon_2, s)]\!](m', \tilde{\epsilon}_r)\right| \le \tilde{\mathcal{S}}'(\tilde{s}).$$

Case 4 (Declassify): Suppose

$$e = \mathtt{declassify}(e_0, \ell_2, \epsilon_2, s, \epsilon).$$

The (Declassify) rule gives as premises the typing derivation for e_0:

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e_0 \overset{\tilde{s}}{:} \mathrm{int}^{\ell_1}_{\tilde{\epsilon}_1} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

and a condition involving dpDowngrade:

$$\mathrm{dpDowngrade}(\epsilon)(\ell_1, \tilde{\mathcal{E}}'(\tilde{\epsilon}_1), \ell_2, \epsilon_2).$$

By the induction hypothesis on e_0, for any m, m' with $|m - m'| \le 1$, we have

$$\left|[\![e_0]\!](m, \tilde{\epsilon}_r) - [\![e_0]\!](m', \tilde{\epsilon}_r)\right| \le \tilde{\mathcal{S}}'(\tilde{s}).$$

The semantics for declassification

$$[\![\mathtt{declassify}(e_0, \ell_2, \epsilon_2, s, \epsilon)]\!](m, \tilde{\epsilon}_r) =$$
$$[\![e_0]\!](m, \tilde{\epsilon}_r) \ggg \left(v \mapsto v + \mathcal{L}(s/\tilde{\epsilon}_r(\epsilon))\right)$$

mirrors the noise case. By a similar coupling argument, the additional noise ensures that the overall difference remains bounded by s, which by the rule's conditions matches (or is upper bounded by) $\tilde{\mathcal{S}}'(\tilde{s})$.

Case 5 (Application): Suppose

$$e = op(e_1)$$

The rule (Application) tells us that we have derivations

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash op \overset{\tilde{s}'}{:} \kappa_1 \xrightarrow{s} \kappa_2 \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

and

$$\tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma \vdash e_1 \overset{\tilde{s}_1}{:} \kappa_1 \dashv \tilde{\mathcal{S}}'', \tilde{\mathcal{E}}''.$$

By the induction hypothesis applied to e_1, for all m, m' with $|m - m'| \leq 1$ we have

$$\left|[\![e_1]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e_1]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \tilde{\mathcal{S}}''(\tilde{s}_1).$$

The semantics for application is:

$$[\![op(e_1)]\!](m, \tilde{\epsilon}_\mathrm{r}) = [\![op]\!](m, \tilde{\epsilon}_\mathrm{r}) \ggg \Big(f \mapsto [\![e_1]\!](m, \tilde{\epsilon}_\mathrm{r}) \ggg f\Big).$$

The operator op evaluates to a function whose own sensitivity (with respect to its argument) is s, i.e. for all $f \in [\![op]\!](m, \tilde{\epsilon}_\mathrm{r})$ and for any v, v', $|v - v'| \leq \alpha$, $|f(v) - f(v')| \leq \alpha s$.

$$\left|[\![e_1]\!](m, \tilde{\epsilon}_\mathrm{r}) \ggg f - [\![e_1]\!](m', \tilde{\epsilon}_\mathrm{r}) \ggg f\right| \leq \tilde{\mathcal{S}}''(\tilde{s}_1) \cdot s.$$

Now, since $[\![op]\!](m, \tilde{\epsilon}_\mathrm{r}) = [\![op]\!](m', \tilde{\epsilon}_\mathrm{r})$ we can conclude

$$\left|[\![op(e_1)]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![op(e_1)]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \tilde{\mathcal{S}}''(\tilde{s}_1) \cdot s = \tilde{\mathcal{S}}^*(\tilde{s}_1) \cdot \tilde{\mathcal{S}}^*(\tilde{s}).$$

By the requirement of well-typedness, this product is bounded above by the sensitivity annotation produced in the rule, namely $\tilde{\mathcal{S}}'''(\tilde{s}_2)$, so the property holds.

Case 6 (Let): Suppose

$$e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2.$$

The (Let) rule gives premises

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e_1 \overset{\tilde{s}_1}{:} \kappa_1 \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

and

$$\tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma[x \mapsto \kappa_1] \vdash e_2 \overset{\tilde{s}_2}{:} \kappa_2 \dashv \tilde{\mathcal{S}}'', \tilde{\mathcal{E}}''.$$

By the induction hypothesis on e_1, for all m, m' with $|m - m'| \leq 1$ we have

$$\left|[\![e_1]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e_1]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \tilde{\mathcal{S}}'(\tilde{s}_1).$$

Let v denote the common value obtained for e_1 when using a suitable coupling of the random choices (if any). Then, extending the memory with $x \mapsto v$, the induction hypothesis on e_2 (in the extended environment) ensures that

$$\left|[\![e_2]\!](m[x \mapsto v], \tilde{\epsilon}_\mathrm{r}) - [\![e_2]\!](m'[x \mapsto v], \tilde{\epsilon}_\mathrm{r})\right| \leq \tilde{\mathcal{S}}''(\tilde{s}_2).$$

Since the let-binding simply sequences these evaluations, the overall change is bounded by the sensitivity indicated in the typing rule.

Case 7 (If): Suppose

$$e = \texttt{if}\ e_\mathrm{cond}\ \texttt{then}\ e_T\ \texttt{else}\ e_F.$$

The (If) rule ensures:

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e_\mathrm{cond} \overset{\tilde{s}}{:} \mathrm{int} \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

and

$$\tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma \vdash e_T \overset{\tilde{s}_T}{:} \kappa \dashv \tilde{\mathcal{S}}_T, \tilde{\mathcal{E}}_T, \quad \tilde{\mathcal{S}}', \tilde{\mathcal{E}}', \Gamma \vdash e_F \overset{\tilde{s}_F}{:} \kappa \dashv \tilde{\mathcal{S}}_F, \tilde{\mathcal{E}}_F.$$

By the induction hypothesis, a perturbation of at most 1 in memory ensures that the value of e_cond changes by at most some bound (which is encoded in the sensitivity of the guard) and similarly for e_T and e_F. Although the branch taken may differ between m and m', the difference would have to come from one of the branches. Thus, we have:

$$\left|[\![e]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \max\{\tilde{\mathcal{S}}_T(\tilde{s}_T), \tilde{\mathcal{S}}_F(\tilde{s}_F)\},$$

which by definition is bounded by $\tilde{\mathcal{S}}''(\tilde{s}')$.

Case 8 (List-Access): Suppose the expression is of the form $e[i]$. By the semantics we have:

$$[\![e[i]]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) =$$
$$[\![i]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) \ggg \Big(idx \mapsto [\![e]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) \ggg \big(v \mapsto \mathrm{return}(v[idx])\big)\Big).$$

By the induction hypothesis for the subexpressions, we know:

$$\left|[\![e]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \mathrm{Sens}(e),$$

and

$$[\![i]\!](m, \tilde{\epsilon}_\mathrm{r}) = [\![i]\!](m', \tilde{\epsilon}_\mathrm{r}),$$

so that the same index idx is chosen in both evaluations. Using our distance metric for lists, the difference between the two lists, viewed as l and l', satisfies:

$$|\, l - l' \,| = n \rightarrow \forall k\ |l[k] - l'[k]| \leq n.$$

As the accessed index is fixed by the index expression, we will only be concerned with the difference between the two values at the same index, which is bounded by the sensitivity of the list i.e. $|l[idx] - l'[idx]| \leq \mathrm{Sens}(e)$. Thus, we can conclude that:

$$\left|[\![e[i]]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e[i]]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \mathrm{Sens}(e),$$

Case 9 (Tuple-Access): Consider an expression $e.n$ that projects the nth component from a tuple. Its semantics is given by:

$$[\![e.n]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) = [\![e]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) \ggg \big(v \mapsto \mathrm{return}(v.n)\big).$$

By the induction hypothesis for e, for any memories m, m' with $|m - m'| \leq 1$ we have:

$$\left|[\![e]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \mathrm{Sens}(e),$$

where $v = (v_0, v_1)$ is a two-component tuple. Since the projection π_n simply selects one component, it follows directly that:

$$|v.n - v'.n| \leq \mathrm{Sens}(e),$$

whenever the tuple v differs from v' by at most $\mathrm{Sens}(e)$ (as the sensitivity of the projection is at most the sensitivity inherent in that component). Thus, the sensitivity bound is preserved under tuple access.

Case 10 (For): Consider a for expression of the form

$$\texttt{for}\ x\ \texttt{in}\ e_\mathrm{list}\ \texttt{index}\ i\ \texttt{accum}\ a = e_\mathrm{accum}\ \texttt{do}\ e.$$

Its operational semantics is defined as:

$$[\![\texttt{for}\ x\ \texttt{in}\ e_\mathrm{list}\ \texttt{index}\ i\ \texttt{accum}\ a = e_\mathrm{accum}\ \texttt{do}\ e]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) =$$
$$[\![e_\mathrm{accum}]\!](\sigma, \tilde{\epsilon}_\mathrm{r}) \ggg \Big(a_0 \mapsto [\![e_\mathrm{list}]\!](\sigma) \ggg \big(l \mapsto \mathrm{fold}(e, l, 0, a_0, \sigma)\big)\Big).$$

Given that the initial accumulator evaluation is equivalent to that of a let-binding, we can apply the same reasoning as in the let case to obtain:

$$\left|[\![e_\mathrm{accum}]\!](m, \tilde{\epsilon}_\mathrm{r}) - [\![e_\mathrm{accum}]\!](m', \tilde{\epsilon}_\mathrm{r})\right| \leq \mathrm{Sens}(e_a),$$

then, we can continue in two stages:

(1) Evaluation of the List. By the induction hypothesis for e_{list}, if m and m' are two memories at distance at most 1 then:

$$\left|[\![e_{\text{list}}]\!](m, \tilde{\epsilon}_{\text{r}}) - [\![e_{\text{list}}]\!](m', \tilde{\epsilon}_{\text{r}})\right| \leq 1.$$

Interpreting the output as lists l and l', our distance metric guarantees that the two lists differ in at most one component by 1.

(2) The Folding Process. Assume that the fold is defined inductively over the list elements. Suppose

$$\text{fold}(e, l, a_0, \sigma)$$

applies a function corresponding to e, with x bound to the current list element and the accumulator updated accordingly. By the induction hypothesis for e (with the extended environment that binds x appropriately), the change in the output from processing a single element is bounded by a constant $c = \text{Sens}(e)$. Since l and l' differ in at most one position, the only possible additional difference between

$$\text{fold}(e, l, 0, a_0, \sigma) \quad \text{and} \quad \text{fold}(e, l', 0, a_0, \sigma)$$

arises from the differing element. Thus by a simple case analysis, we conclude that:

$$|\text{fold}(e, l, 0, a_0, \sigma) - \text{fold}(e, l', 0, a_0, \sigma)| \leq c,$$

where c is bounded by $\text{Sens}(e) \leq n \cdot \text{Sens}(e)$. Finally, composing the two parts via the bind structure in the operational semantics, we obtain:

$$|[\![\texttt{for } x \texttt{ in } e_{\text{list}} \texttt{ index } i \texttt{ accum } a = e_{\text{accum}} \texttt{ do } e]\!](m, \tilde{\epsilon}_{\text{r}}) - [\![\texttt{for } x \texttt{ in } e_{\text{list}} \texttt{ index } i \texttt{ accum } a = e_{\text{accum}} \texttt{ do } e]\!](m', \tilde{\epsilon}_{\text{r}})| \leq c,$$

Conclusion. By structural induction on the typing derivation, we have shown that for every expression

$$\tilde{\mathcal{S}}, \tilde{\mathcal{E}}, \Gamma \vdash e \overset{\tilde{s}}{:} \tau \dashv \tilde{\mathcal{S}}', \tilde{\mathcal{E}}',$$

and for all memories m, m' such that $|m - m'| \leq 1$, it follows that

$$\left|[\![e]\!](m, \tilde{\epsilon}_{\text{r}}) - [\![e]\!](m', \tilde{\epsilon}_{\text{r}})\right| \leq \tilde{\mathcal{S}}'(\tilde{s}).$$

This completes the proof. □