

Differentially Private Runtime Monitoring

Bernd Finkbeiner^{} and Frederik Scheerer^{}

CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany
{finkbeiner, frederik.scheerer}@cispa.de

Abstract. Modern stream-based monitors collect detailed statistics of the runtime behavior of the system under observation. If the system runs in a privacy-sensitive context, this poses the risk of disclosing sensitive information. Differential privacy is the state-of-the-art approach for protecting sensitive information, however, integrating it into runtime monitoring is challenging: temporal operators can cause individual input values to influence multiple outputs over time, leading to repeated disclosure of private information. We propose an approach that automatically enforces differential privacy in stream-based monitoring specifications by analyzing temporal dependencies and injecting carefully calibrated noise into the specification. To preserve the utility of the outputs, we identify strategically chosen positions in the specification for noise injection and leverage tree-based mechanisms to mitigate the accuracy loss caused by noise injected into aggregation operators. We demonstrate the practicality and effectiveness of our approach in a case study on monitoring public transportation usage.

Keywords: Runtime Verification · Stream-Based Monitoring · Privacy

1 Introduction

Runtime monitoring is a powerful technique for ensuring a system’s correctness during operation, especially in complex, real-world applications where static verification is difficult to obtain. In many cases, a binary yes/no verdict is insufficient, and instead, stream-based languages output values that summarize and aggregate the system’s current state over time. At the same time, runtime monitoring is increasingly deployed in privacy-sensitive domains, where richer monitor outputs increase the risk of revealing private information.

For example, consider a car-rental company that continuously monitors the distance driven by each vehicle in order to schedule maintenance, detect abnormal wear, or ensure contractual conditions. Even if the monitor reports only aggregated values, such as average daily distance driven, these outputs may still reveal sensitive information about the renter’s behavior, including typical usage patterns and inferred routines when observed over time. Similar concerns arise when monitoring commercial vehicle fleets, where runtime monitors are used to assess efficiency, safety, or regulatory compliance but inadvertently expose routes

or operating schedules of drivers. While runtime monitoring is essential for ensuring safety, performance, and compliance in such systems, it is crucial to ensure that the monitoring process does not become a source of privacy violations.

Differential privacy (DP) [20] is widely regarded as the gold standard for providing formal privacy guarantees. Informally, it ensures that the presence or absence of a single individual in the dataset has minimal impact on the output of a query. This is typically achieved by adding carefully calibrated noise to the output and thereby producing a mechanism with provable privacy guarantees.

Integrating differential privacy into stream-based monitoring, however, introduces unique challenges due to the temporal nature of streams and the temporal operators describing them. In contrast to static queries, monitors repeatedly release information over time, often derived from overlapping portions of the input data. Releasing information derived from a single data point multiple times increases the overall privacy loss, as each release leaks additional information. With temporal operators such as sliding-window aggregations, we observe exactly this behavior, and to reason about privacy in this setting, it is necessary to understand how individual inputs propagate through the temporal operators.

This observation introduces two key challenges. First, we must be able to precisely characterize and compute the impact of individual inputs on the outputs, accounting for the effects of asynchronous streams and temporal operators, such as sliding-window aggregations. Second, beyond providing formal privacy guarantees, we must ensure that the resulting private monitors remain accurate. In particular, temporal operators amplify the influence of individual inputs, and the required amount of noise may degrade the quality of monitoring outputs.

In this paper, we present a runtime monitoring framework that enables users without privacy expertise to deploy monitors that prevent the leakage of sensitive information. Our framework integrates differential privacy by analyzing the specification and automatically injecting appropriate amounts of noise. It strategically selects positions in the specification that are beneficial to the accuracy of the output, and leverages tree-based constructs to further improve monitoring accuracy. We demonstrate our approach using the stream-based specification language RTLola [5], which has already been applied in several real-world, privacy-sensitive areas [9,24,7,28]. In a case study on public transportation usage, we show that our approach enables private monitoring with high accuracy.

The remainder of this paper is organized as follows. Section 2 gives an intuition of our approach using a motivating example, and Section 3 introduces our formalization of differential privacy for stream-based monitors. In Section 4, we detail the sensitivity calculations for stream-based specifications, while Section 5 discusses our strategies for improving the accuracy of the private monitors. Finally, Section 6 explores extensions of our approach to stricter privacy guarantees, and Section 7 reports on experimental results.

1.1 Related Work

Closest to our work is the approach of Henzinger et al. [32], which studies private runtime monitoring based on private function evaluation while maintaining an

internal state. Further, prior work has explored alternative notions of privacy in LTL monitoring [1,2]. However, these approaches do not rely on differential privacy and do not support the rich output provided by stream-based languages.

We are the first to integrate DP directly into runtime verification frameworks. Previous work has focused on enforcing DP on database-style queries, especially SQL-like languages [33,10,37,38,39,40,42]. Some related work also considers queries on data streams [36]. All these approaches, however, are designed for database-style queries and emphasize operators such as joins, rather than the temporal operators used in runtime verification.

An algorithmic idea adopted in our work is tree-based mechanisms for continual queries under DP. Dwork et al. [21] introduced tree-based mechanisms for continual counting queries and Chan et al. [14] independently proposed a similar approach for infinite streams. Tree-based mechanisms were later extended to other aggregations such as sums [41,47,43,13]. In addition to tree-based approaches, matrix-based mechanisms have been proposed for streaming data [19,36,30,31], offering alternative trade-offs between accuracy and efficiency. Some approaches also enforce DP for sliding window aggregations [11,46,15], a temporal operator commonly used in stream-based monitoring. Most streaming DP works assume event-level privacy and independent data; however, some approaches support user-level privacy [26,18,16,12] or correlated data [23,45].

We demonstrate our approach using the stream-based specification language RTLola [5], an extension of Lola [17] that incorporates asynchronous streams and real-time functionality. RTLola has been applied in multiple domains, including several privacy-sensitive domains such as monitoring sensor data from private vehicles [9], analyzing network traffic [24], monitoring automatic decision and prediction systems [7], and processing medical health data [28]. Although our approach is presented in the context of RTLola, the underlying ideas extend to other stream-based specification languages such as TeSSLa [34] and Striver [29].

2 Motivating Example

We begin with a motivating example that illustrates the key ideas of our approach. The methods will be detailed in the following sections. We work in the setting of stream-based monitoring, where specifications define sets of streams, representing infinite sequences of values. Input streams represent incoming data from the monitored system, while output streams are defined through stream equations and compute new streams by transforming and aggregating the inputs.

Consider Figure 1, which shows an example for the stream-based specification language RTLola [5]. It models a monitoring scenario in which user feedback is continually collected and evaluated over time. Each user’s feedback consists of new values added to the two input streams: a **score**, ranging from 1 (best) to 6 (worst), and a **conf** value indicating positive (1), neutral (0), or negative (-1) confidence. The specification combines these inputs into a single adjusted rating between 0 and 17, and emits the result as a new value of the output stream **adj**. Using a sliding window aggregation, the stream **davg** computes the average

```

1 input score : Int64 // ∈ {1,2,3,4,5,6}
2 input conf : Int64 // ∈ {-1,0,1}
3 output adj := (6-score)*3 + conf + 1
4 output davg @1d :=
5     adj.aggregate(over: 3d, using: avg).defaults(to: 0.0)
6 output low @1d := min(low.offset(by: -1).defaults(to: 15.0), davg)
7 output high @1d := max(high.offset(by: -1).defaults(to: 0.0), davg)
8 #[public]
9 output range @1d := (low, high)

```

Fig. 1: An example RTLola specification calculating statistics of user feedback.

rating over the last three days. Finally, the streams `low` and `high` track the minimum and maximum daily averages observed so far. The `range` stream, consisting of both statistics, is marked as public using RTLola’s annotation mechanism [6], indicating that its values are output by the monitor to the public.

Individual user feedback is privacy-sensitive. While we assume the monitor itself is trusted, its public outputs go into an untrusted world. Therefore, the monitor must ensure that no private information is leaked by observing the publicly emitted values of `range` over time. To incorporate differential privacy into a stream-based specification, we must understand how a change to a single input influences observable outputs. Offset accesses propagate information across time, as seen in the recursive definitions of `low` and `high`, where a single rating influences arbitrarily many future outputs. Similarly, in sliding-window aggregations, each input contributes to multiple overlapping windows, so a single change affects multiple results.

Dependencies are made explicit in the *dependency graph* [5] of the specification. Each stream in the specification corresponds to a node, and stream accesses are represented by directed edges in the graph, annotated with the access kind. The dependency graph for the example is shown in Figure 2. In the graph, the private input streams are marked with a red outline, but streams that (indirectly) depend on private inputs also contain private information. The extent to which private information influences a stream is quantified by the *sensitivity*. Intuitively, the sensitivity of a stream bounds how much its values can change when a single user’s feedback is modified. In a temporal setting, this captures both how strongly values change and how often this change reappears over time. Sensitivity propagates through the dependency graph, accumulating at arithmetic operations and amplifying across temporal operators.

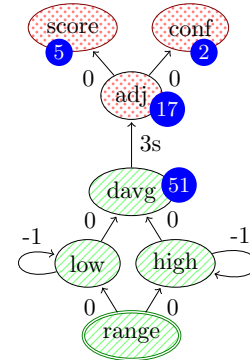


Fig. 2: The dependency graph for the specification in Figure 1.

For instance, the sensitivity of the `score` stream is 5, reflecting its input range. The `adj` stream combines multiple pieces of private information and therefore has higher sensitivity. The sliding window further amplifies this: A single value contributes to three overlapping windows, resulting in a sensitivity of 51 for `davg`. This sensitivity information allows us to track how private information flows through the specification. Based on this information, we select a subset of nodes at which the monitor adds noise to each value. Adding noise to a node renders the corresponding information in the stream privacy-preserving, which in turn makes all downstream computations safe to publish. This yields a placement problem: where should noise be added so that all public outputs are protected, while keeping the loss of accuracy as small as possible?

In the example, we select the `davg` stream and add noise calibrated to its sensitivity. This choice ensures that all values flowing into the recursive definitions of `low` and `high` are already privacy-preserving. Applying noise earlier, for example at the inputs, would reduce accuracy because each aggregated value would be noisy by itself. After the aggregation, a single rating can be hidden among multiple others, making its individual contribution smaller and requiring less perturbation relative to the result. Now, as this node is privacy-preserving, it and all downstream nodes are marked as green. As all public outputs are among these green nodes, the overall specification satisfies the desired privacy guarantees.

In this example, the sliding window amplifies the sensitivity linearly with the window size. Later in the paper, we show how tree-based aggregation methods can substantially increase accuracy for longer windows.

3 Differential Privacy of Stream-Based Monitors

This section formalizes differential privacy for stream-based runtime monitoring. We first summarize the standard definitions and mechanisms of differential privacy, stated with respect to a general adjacency relation. We then present the evaluation model used to represent monitor executions and define an event-level adjacency notion capturing the privacy requirements for stream-based monitors.

3.1 Differential Privacy Background

Differential privacy [20] is the de facto standard for ensuring privacy, grounded in a rigorous mathematical definition. Originally introduced in the context of databases, it provides a strong guarantee: the presence or absence of any individual in the dataset has only a minimal impact on the output of a query on the database. Intuitively, a randomized mechanism is differentially private if an observer cannot distinguish whether a particular record was part of the database based on the output of the mechanism.

Definition 1 (Differential Privacy [22]). *Let $\mathcal{D}$ be a domain and $\sim \subseteq \mathcal{D} \times \mathcal{D}$ a symmetric adjacency relation. A randomized mechanism $\mathcal{K} : \mathcal{D} \rightarrow \mathcal{R}$ is ϵ -differentially private w.r.t. $\sim$ if for all $D, D' \in \mathcal{D}$ with $D \sim D'$ and for all*

$S \subseteq \mathcal{R}$,

$$\Pr[\mathcal{K}(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[\mathcal{K}(D') \in S],$$

where the probability is over the randomness of $\mathcal{K}$.

We state the definition in its general form with respect to an arbitrary adjacency relation. The *adjacency relation* $\sim$ specifies which pairs of inputs should be considered indistinguishable by an observer. In the classical database setting, the relation captures neighboring databases that differ by a single row. The Probability $\Pr[\mathcal{K}(D) \in S]$ captures the chance that the mechanism $\mathcal{K}$ produces an output in the set S . Differential privacy bounds how much this probability can change between adjacent inputs. The *privacy parameter* ε quantifies the strength of the privacy guarantee. Smaller values of ε correspond to stronger privacy. Choosing ε involves balancing the level of privacy with the output utility, depending on the privacy requirements of the data and the intended application.

To design differentially private mechanisms, we must understand how much the output of a query can change between two adjacent inputs. This quantity is called the sensitivity of the query.

Definition 2 (Sensitivity [22]). For $f : \mathcal{D} \rightarrow \mathbf{R}^d$, the L_1 sensitivity of f w.r.t. $\sim$ is $\Delta_{\sim} f = \max_{\substack{D, D' \\ D \sim D'}} \|f(D) - f(D')\|_1$.

Intuitively, the sensitivity measures the maximum possible change in the output between two adjacent inputs. For instance, a query that counts the number of rows satisfying a certain predicate has sensitivity 1, since modifying or removing a single row can change the count by at most one. In contrast, a query that sums numerical values can have a much larger sensitivity, depending on the possible ranges of those values.

Once the sensitivity of a query is known, differential privacy can be achieved by adding Laplace noise calibrated to that sensitivity.

Theorem 1 (Laplace Mechanism [22]). For $f : \mathcal{D} \rightarrow \mathbf{R}^d$, the mechanism $\mathcal{K}(\mathcal{D}) = f(\mathcal{D}) + \eta$ that adds independently generated noise η with distribution $\eta_i \sim \text{Lap}(\Delta_{\sim} f / \varepsilon)$ to each of the d output terms satisfies ε -differential privacy w.r.t. $\sim$.

While the Laplace mechanism establishes privacy guarantees for individual mechanisms, composition theorems describe how these guarantees change when multiple mechanisms are combined. Under *sequential composition* [22], publishing the outputs of multiple mechanisms results in an additive privacy loss: combining mechanisms with privacy parameters ε_1 and ε_2 gives an overall privacy guarantee of $(\varepsilon_1 + \varepsilon_2)$ with respect to the same adjacency relation. As a result, repeatedly releasing noisy versions of the same query gradually weakens privacy, so individual mechanisms must use smaller privacy parameters to stay within a fixed overall budget. *Post-processing* [22] ensures that any transformation applied to the output of a differentially private mechanism preserves its privacy guarantee. Once a mechanism has produced a privacy-preserving output, it can be safely transformed without adding any extra privacy loss. Together, these properties let us reason modularly about privacy in stream-based monitors.

3.2 Evaluation Models for Runtime Monitoring

To formalize differential privacy in the context of runtime monitoring, we need a model allowing us to compare different monitor executions. For this purpose, we introduce the notion of an evaluation model. Let $\text{Time} = \mathbb{N}$ be the set of discrete timestamps, $\mathbb{V}_\perp = \mathbb{R} \cup \{\perp\}$ the set of optional stream values, and $\mathbb{R}^+$ the set of non-negative real numbers.

Definition 3 (Evaluation Model [5]). *An evaluation model $\omega \in \mathbb{W}$ consists of timed data streams over a set of input stream references $\text{ID}^\uparrow$ and output stream references $\text{ID}^\downarrow$. It is represented as the combination of a stream map and a timing map:*

$$\begin{aligned} \text{Stream} &:= \text{Time} \rightarrow \mathbb{V}_\perp \\ \text{StreamMap} &:= (\text{ID}^\uparrow \uplus \text{ID}^\downarrow) \rightarrow \text{Stream} \\ \text{TimeMap} &:= \text{Time} \rightarrow \mathbb{R}^+ \\ \mathbb{W} &:= \text{StreamMap} \times \text{TimeMap}. \end{aligned}$$

Intuitively, an evaluation model captures all values produced by the monitor during one execution. The stream mapping assigns each input and output stream reference to a stream of values indexed by discrete timestamps. Because the monitor operates asynchronously, a stream may have no new value at a given timestamp. This absence is represented by the symbol $\perp$. The time mapping associates each discrete timestamp with its corresponding real-time timestamp.

For convenience, given an evaluation model $\omega = (\text{streams}, \text{times})$, we define $\omega(t) := \text{times}(t)$ to denote the real-time timestamp of $t \in \text{Time}$, and $\omega(s) := \text{streams}(s)$ to denote the stream assigned to reference $s \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$.

3.3 Adjacency of Evaluation Models

To reason about differential privacy in the context of monitor executions, we define a suitable adjacency relation between evaluation models. Different choices of adjacency correspond to different privacy guarantees and reflect what aspect of the monitored behavior is intended to be protected. In this work, we adopt an *event-level* [21] notion of privacy: Two evaluation models are adjacent if their input streams differ by exactly one event occurring at a single timestamp. Moreover, we assume that only the content of an event is sensitive, not its timing. Consequently, adjacent evaluation models share the same timing map, ensuring that temporal information remains unchanged:

Definition 4 (Event-Level Adjacency). *Two evaluation models $\omega, \omega' \in \mathbb{W}$ are considered event-level adjacent, if*

1. *For all $t \in \text{Time}$, it holds that $\omega(t) = \omega'(t)$,*
2. *For all $t \in \text{Time}$ and $i \in \text{ID}^\uparrow$, it holds that $\omega(i)(t) = \perp \Leftrightarrow \omega'(i)(t) = \perp$,*
3. *There exists at most one $t \in \text{Time}$, such that $\exists i \in \text{ID}^\uparrow. \omega(i)(t) \neq \omega'(i)(t)$.*

Further, we consider them s -distant if $\forall t \in \text{Time}. \forall i \in \text{ID}^\uparrow. |\omega(i)(t) - \omega'(i)(t)| \leq s$.

The notion of s -distance restricts how much two adjacent evaluation models may differ in their input values: each input can change by at most s . This assumption reflects many realistic scenarios, where input domains are naturally bounded, for instance, sensor readings limited by the hardware, or demographic data such as age constrained within feasible ranges.

Our notion of privacy protects individual datapoints. For example, it can hide a single detour when tracking a person's car, but it cannot hide the daily commute to work. This choice is motivated by our approach using static analysis, and stronger notions of adjacency would require significant amounts of noise, making it difficult to produce accurate monitors. In Section 6, we nevertheless explore strategies for supporting stronger notions of privacy within our framework.

4 Sensitivity Calculations for Stream-Based Languages

The sensitivity of a function measures how much its output can change for two adjacent inputs. For stream-based monitors, we analyze how much output streams may differ between two adjacent and valid evaluation models. We introduce a formal framework for the sensitivity of streams, discuss practical challenges in stream-based computations, and establish static bounds that enable differentially private monitoring.

4.1 Specifications

A *specification* φ consists of a set of stream equations $\{x := \varphi_x \mid x \in \text{ID}^\downarrow\}$, where each φ_x is a stream expression over $\text{ID}^\uparrow \uplus \text{ID}^\downarrow$. Each output stream is additionally annotated with a *padding type* that determines when it is evaluated relative to the input streams. An evaluation model $\omega \in \mathbb{W}$ is *valid* for φ if every output stream value $\omega(x)(t)$ is computed according to the defining equation φ_x for all $t \in \text{Time}$.

4.2 Stream Expressions

Sensitivities are defined recursively over stream expressions. Expressions may reference a stream y , a constant c , an offset $o \in \mathbb{N}$, or be combined using a binary operator $\circ \in \{+, -, \cdot\}$:

$$\begin{aligned} \text{exp} ::= & \text{sync}(y) \mid \text{offset}(y, o) \mid \text{hold}(y) \mid \text{aggr}(y, W, f) \\ & \mid \text{default}(\text{exp}, \text{exp}) \mid \text{ite}(\text{exp}, \text{exp}, \text{exp}) \mid \text{exp} \circ \text{exp} \mid c \end{aligned}$$

We distinguish between *synchronous accesses*, namely `sync` and `offset`, which access the value of y at the current (resp. offset) discrete timepoint, and *asynchronous accesses*, such as `hold`, which returns the most recent available value, and `aggr`, which applies the aggregation function $f \in \{\Sigma, \text{last}, \text{count}\}$ over a window W of values of y . As asynchronous accesses may be undefined at certain time

points, we provide the `default` operator, which supplies a fallback value whenever an access fails. A formal definition of the semantics of stream expressions is given in Appendix A.

4.3 Timing Abstraction

Since the sensitivity analysis should be independent of the specific stream values but may depend on the timing of events, we separate the timing information of an evaluation model. For this purpose, we introduce a *pacing model*, which captures only the temporal structure of stream evaluations.

Definition 5 (Pacing Model). *A pacing model $p \in \mathbb{P}$ captures the timing of stream evaluations independent of stream values. It consists of a pacing and timing map, where $\mathbb{B} = \{\perp, \top\}$ denotes the set of boolean values:*

$$\begin{aligned} \text{PacingMap} &:= (\text{ID}^\uparrow \uplus \text{ID}^\downarrow) \rightarrow \text{Time} \rightarrow \mathbb{B} \\ \text{TimeMap} &:= \text{Time} \rightarrow \mathbb{R}^+ \\ \mathbb{P} &:= \text{PacingMap} \times \text{TimeMap} \end{aligned}$$

Analogously to the evaluation model, we introduce shorthand notation. For a pacing model $p = (\text{pacings}, \text{times})$, we define $p(t) := \text{times}(t)$ for a time $t \in \text{Time}$, and $p(s) := \text{pacings}(s)$ for a stream $s \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$. Further, we say an evaluation model *behaves according to* a pacing model if all real-time timestamps are equal and a stream has a value exactly if the corresponding pacing is true.

To enable an automatic analysis of specifications, we impose assumptions on the pacing of stream evaluations. Among other checks, such as dependency and value type analysis, each specification is statically verified to satisfy these pacing constraints before execution. These constraints guarantee that the monitoring algorithm produces only valid evaluation models and that the resulting monitor operates within bounded memory requirements.

Definition 6 (Correctly Typed Pacing Model). *For a pacing model $p \in \mathbb{P}$ to be considered correctly typed according to a specification φ , it must hold that*

1. *For each synchronous access `sync` and offset from output stream $x \in \text{ID}^\downarrow$ to stream $y \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$, it must hold that whenever x evaluates, y evaluates as well: $\forall t \in \text{Time}. p(x)(t) \Rightarrow p(y)(t)$.*
2. *For each sliding window aggregation `aggr` from output stream $x \in \text{ID}^\downarrow$ over a stream $y \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$, it must hold that x is periodic with some period δ_x , i.e. $\forall t \in \text{Time}. p(x)(t) \Rightarrow \exists k \in \mathbb{N}. p(t) = k \cdot \delta_x$.*

A specification φ is *well-typed* if it statically satisfies the pacing constraints above. For every valid evaluation model $\omega \in \mathbb{W}$ of a well-typed specification, there exists a correctly-typed pacing model $p \in \mathbb{P}$ that ω behaves according to.

4.4 Per-Event Sensitivity

To measure how much the values of an output stream can differ between two adjacent evaluation models, we introduce *per-event sensitivity*, which captures the maximum impact a single changed input has on a stream value at a given timestamp, taking into account the timing constraints imposed by a pacing model.

Definition 7 (Per-Event Sensitivity). *Given a specification φ , a pacing model $p \in \mathbb{P}$, and two timestamps $t, t' \in \text{Time}$, the per-event sensitivity defines how much the value of a stream $x \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$ at timestamp t' can change, if the inputs at timestamp t are changed by at most s .*

$$\Delta_{t,p,s}^{t'} : \text{ID}^\uparrow \uplus \text{ID}^\downarrow \rightarrow \mathbb{R}^+$$

$$\Delta_{t,p,s}^{t'}(x) = \begin{cases} s & \text{if } x \in \text{ID}^\uparrow \wedge t' = t \\ 0 & \text{if } x \in \text{ID}^\uparrow \wedge t' \neq t \\ \Delta_{t,p,s}^{t'}(\varphi_x) & \text{if } x \in \text{ID}^\downarrow \end{cases}$$

where $\Delta_{t,p,s}^{t'}(\varphi_x)$ computes the sensitivity for the stream expression φ_x of x . Given stream expressions e , e_1 and e_2 , the constant c , and the stream reference y , we define

$$\begin{aligned} \Delta_{t,p,s}^{t'}(e_1 + e_2) &= \Delta_{t,p,s}^{t'}(e_1) + \Delta_{t,p,s}^{t'}(e_2) \\ \Delta_{t,p,s}^{t'}(c \cdot e) &= |c| \cdot \Delta_{t,p,s}^{t'}(e) \\ \Delta_{t,p,s}^{t'}(\text{sync}(y)) &= \Delta_{t,p,s}^{t'}(y) \\ \Delta_{t,p,s}^{t'}(\text{offset}(y, o)) &= \Delta_{t,p,s}^{\text{last}(y,t',p,o)}(y) \\ \Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \Sigma)) &= \sum_{t'' \in \text{wtimes}(y,t',p,W)} \Delta_{t,p,s}^{t''}(y) \end{aligned}$$

where $\text{last}(y, t, p, o)$ returns the timestamp of the o -th most recent value of y , and $\text{wtimes}(y, t, p, W)$ returns timestamps of y from the W sized window behind t . Consider Appendix B for the full definition with all operators.

For an input stream, the sensitivity depends on the timestamp t' being considered. If $t' = t$, the sensitivity is exactly s , reflecting the allowed change at that time. For all other timestamps, the input sensitivity is zero, since only a single input value may be modified. For an output stream x , the sensitivity is determined by its defining stream expression φ_x in the specification. Most operations compute values at the current timestamp, so recursion does not shift the timestamp. An exception is the *offset*-operator: The sensitivity at timestamp t' depends on the sensitivity of the value retrieved from the past. As this is determined by the pacing model, the function *last* identifies the timestamp of the accessed value. In the case of summation over sliding windows, the sensitivity at time t' accumulates contributions from all aggregated values within the window.

Our intuition of the per-event sensitivity function is captured by this theorem:

Theorem 2. *Let $\omega, \omega' \in \mathbb{W}$ be valid evaluation models for an acyclic specification φ which are s -distant, event-level adjacent, and differ at timestamp t . Then, for each stream $x \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$ and each timestamp $t' \in \text{Time}$,*

$$|\omega(x)(t') - \omega'(x)(t')| \leq \Delta_{t,p,s}^{t'}(x),$$

where both evaluation models define a unique and common pacing model $p \in \mathbb{P}$.

The proof is by structural induction over the defining stream expressions and can be found in Appendix D. Note that throughout this chapter, the dependency graph is assumed acyclic. Dependency cycles are handled in Section 5.1.

4.5 Value-Dependent Operators

So far, we have assumed that an expression's sensitivity depends only on the sensitivities of the accessed streams. For some operators, such as multiplication and conditionals, the sensitivity also depends on the actual values. For instance, in a conditional $\text{ite}(c, e_1, e_2)$, the output may switch between e_1 and e_2 for neighboring executions, and the resulting sensitivity depends on the maximum difference between the two branches. Similarly, for a multiplication, the output sensitivity scales with the operand values. To handle such operators, we extend the analysis to propagate *value ranges* alongside sensitivities. Each expression is associated with an interval that over-approximates the values it may produce at runtime. Using this range information, the per-event sensitivity function $\Delta_{t,p,s}^{t'}$ can be extended as follows:

Definition 8 (Extended Per-Event Sensitivity). *Let $\Delta_{t,p,s}^{t'}(x)$ be defined as in Definition 7 for all standard operators. For operators whose sensitivity depends on the input values, we extend it by*

$$\Delta_{t,p,s}^{t'}(x) := \begin{cases} ub(x) - lb(x) & \text{if } t \rightsquigarrow_p^x t' \\ 0 & \text{otherwise,} \end{cases}$$

where $t \rightsquigarrow_p^x t'$ indicates that a change of inputs at time t can influence the value of stream x at time t' under the pacing model p , and $ub(\cdot)$ and $lb(\cdot)$ are the propagated value bounds. Find the definitions in Appendix C.

Note that sliding-window aggregations do not preserve bounded value ranges, as their output range grows unboundedly even when the output sensitivity is bounded. As a consequence, we impose an additional typing constraint: value-dependent operators are only permitted on expressions for which a finite value range can be statically derived. Note that bounded value ranges can always be re-established explicitly using the `clamp` operator, which enforces fixed lower and upper bounds on the produced values.

4.6 Asynchronous Accesses

Stream-based specifications allow asynchronous stream evaluations, so a single input value may be accessed multiple times by dependent streams. For some asynchronous operators, such as sliding-window aggregations, well-typed pacing restrictions ensure that the number of contributions of a single input is bounded. However, the *hold access* operator retrieves the most recent value of another stream, allowing a single input to influence an unbounded number of timestamps.

Consider a stream c that accesses the latest value of a whenever b occurs:

```
1 | output c @b := a.hold().defaults(to: 0)
```

Because the timing of a and b need not be known in advance, it is impossible to determine how often a single value of a is accessed, and thus how much it may contribute to c .

In general, unrestricted hold accesses prevent bounding the total effect of a single input across all timestamps. However, in practice, these accesses can often be constrained either by bounding the time span during which a value is relevant or by limiting the number of accesses. For instance, a bounded hold can be expressed as a sliding-window aggregation, e.g., `aggr(a, 5s, last)` or by explicitly restricting the number of accesses, e.g., `hold5(a)` with a newly introduced hold variant. Once such bounds are made explicit, the contribution of a single input value becomes finite, and a static global sensitivity bound can be derived.

4.7 Static Bounds

While the per-event sensitivity Δ provides a way to quantify the sensitivity at a specific timestamp, it depends on a concrete pacing model and assumes knowledge of the timestamp at which the inputs differ, which is both unavailable at runtime. To address this, we overapproximate the total L_1 -sensitivity of Δ accumulated across all timestamps, by deriving a static upper bound for each stream. The only assumption we make is that the pacing model is well-typed, a property statically verified on the specification. Under this assumption, the following result establishes the existence of a finite bound for every stream.

Theorem 3 (Sensitivity Bound). *Given a well-typed specification φ with an acyclic dependency graph without unbounded hold operators and a distance value s , it holds that*

$$\forall x \in (\text{ID}^\uparrow \uplus \text{ID}^\downarrow). \exists b_{\varphi, x, s}. \forall p \in \mathbb{P}_\varphi. \forall t \in \text{Time}. \left(\sum_{t' \in T_{p, x}} \Delta_{t, p, s}^{t'}(x) \right) \leq b_{\varphi, x, s}.$$

where $\mathbb{P}_\varphi$ denotes all correctly-typed pacing models according to φ and $T_{p, x} = \{t' \in \text{Time} \mid p(x)(t')\}$ denotes the timestamps at which stream x is evaluated.

Intuitively, this theorem ensures that each stream x has a finite, statically derivable upper bound $b_{\varphi, x, s}$ on its total L_1 -sensitivity given an s -bounded change to

the inputs. This upper bound can be computed recursively along the dependency graph. For example, the bound for an input stream is given by s , the bound for an addition is the sum of the bounds of its operands, and the bound for a synchronous access coincides with the bound of the accessed stream. Precise bounds for each operator and the full proof can be found in Appendix F.

With this static bound available, differential privacy can be achieved by injecting appropriately scaled noise into each value of an output stream.

Theorem 4. *Given a well-typed specification φ with an acyclic dependency graph without unbounded hold operators and a distance value s , and $\omega \in \mathbb{W}$ a valid evaluation model for φ . Let $\mathcal{K}$ be a randomized mechanism $\mathcal{K}_x : \mathbb{W} \rightarrow \text{Stream}$ which produces a noisy version of the output stream x by adding independent Laplace noise to each value:*

$$\mathcal{K}_x(\omega) := \lambda t.\omega(x)(t) + \eta_t,$$

where each term η_t is drawn independently from the Laplace distribution with

$$\eta_t \sim \text{Lap}\left(\frac{b_{\varphi,x,s}}{\varepsilon}\right).$$

Then, $\mathcal{K}_s$ satisfies ε -private s -distant event-level differential privacy.

The result shows that by calibrating Laplace noise to the global sensitivity bound $b_{\varphi,x,s}$, we can guarantee ε -differential privacy for all s -distant, event-level adjacent executions. The proof follows immediately from the combination of Theorem 2 and Theorem 3 with Theorem 1 and can be found in Appendix G. While the randomized mechanism described above releases only a single output stream, standard composition theorems of differential privacy ensure that multiple output streams can also be released with appropriately adjusted privacy parameters.

5 Accurate Monitoring under Differential Privacy

Differential privacy of individual streams can be ensured by appropriately adding noise to their values. Simply adding noise to all inputs would guarantee privacy, but it can lead to poor utility of the results. In this section, we present strategies to maintain accurate monitoring while preserving privacy.

5.1 Ensuring Privacy for Unbounded Specifications

Certain language features, such as unbounded hold operators or circular definitions, can cause a single input event to influence an unbounded number of outputs. This makes it impossible to compute a finite global sensitivity for the entire specification directly. However, every specification can still be made differentially private by controlling where in the specification noise is injected. The key idea is to partition the dependency graph of the specification into two segments:

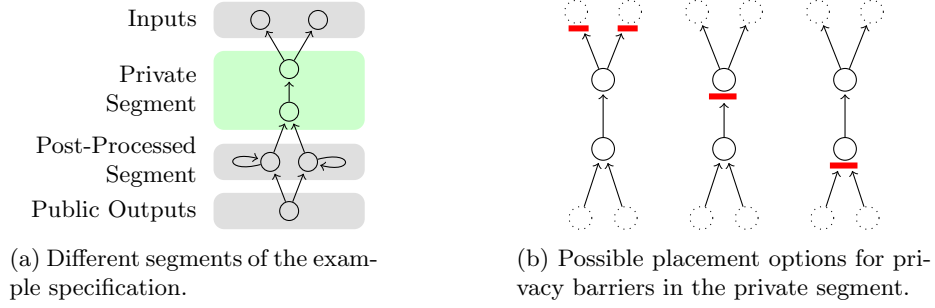


Fig. 3: Dependency Graph examples for the RTLola specification in Figure 1.

- An upper *private segment*, which is acyclic and contains only computations for which the sensitivity can be statically bounded, and
- A lower *post-processed segment*, which may include features that induce unbounded sensitivity.

The segments are illustrated in Figure 3a for the specification from Figure 1. Noise can be safely injected at the boundaries between the segments, ensuring that all values entering the post-processed segment already satisfy differential privacy. Downstream computations, no matter how complex or unbounded, can then be treated as post-processing, which cannot weaken the privacy guarantees.

5.2 Privacy Barriers

Adding noise at the transition between private and post-processed segments ensures differential privacy for any specification. However, it is not always optimal: injecting noise earlier in the specification can significantly improve the utility of the outputs. To formalize where noise can be injected into the specification, we introduce the notion of *privacy barriers*, which separate private inputs from the public outputs in the dependency graph:

Definition 9 (Privacy Barriers). Let $ID^\uparrow$ be the set of input streams and $ID_{pub}^\downarrow \subseteq ID^\downarrow$ the set of public output streams. A set $\mathcal{C} \subseteq ID^\uparrow \uplus ID^\downarrow$ is a valid set of privacy barriers if every path in the dependency graph from any $i \in ID^\uparrow$ to any $o \in ID_{pub}^\downarrow$ passes through exactly one $c \in \mathcal{C}$.

Any valid set of privacy barriers is a valid option for adding noise. At each barrier, noise is injected, and all downstream computations are treated as post-processing. Figure 3b shows all valid options for the example specification.

A natural goal is to choose a barrier placement that maximizes the utility of the monitored outputs. However, such an optimal placement does not exist: different choices can favor short-term vs. long-term accuracy, or prioritize the accuracy of different output streams against others. A placement that is locally optimal for one output stream may be suboptimal for another, leading to

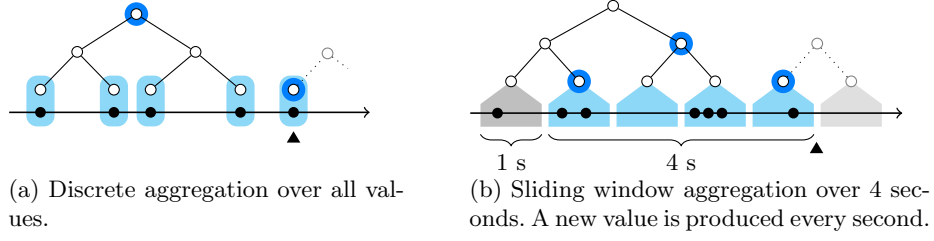


Fig. 4: Tree-Based Aggregation Examples.

inherent trade-offs between streams. We therefore do not prescribe a single optimal strategy, but instead suggest several heuristics for selecting privacy barriers whose trade-offs can be predicted and reasoned about.

The *input-only* heuristic injects noise at input streams, ensuring that all computations operate on perturbed data. The *deep* heuristic targets the transition from private to post-processed segments. This preserves the accuracy of intermediate computations as much as possible, but may require a lot of barriers and, therefore, more noise due to the composition theorem. The *post-aggregation* heuristic places barriers immediately after the first aggregation. Compared to adding noise before aggregations, this often reduces the overall noise required since the private input is combined with multiple others before being perturbed. The *minimal barriers* heuristic selects the smallest set of valid barriers to reduce the number of noise injections. Fewer injection points reduce runtime cost but may lead to a higher variance compared to more fine-grained barrier placements. Importantly, all placement strategies preserve the same differential privacy guarantees.

Beyond these, other heuristics are conceivable, for example, tracking the propagated error symbolically through the specification, similar to approaches in monitoring under uncertainty [27].

5.3 Tree-Based Aggregations

In Section 5.1, we showed that we exclude cycles in the dependency graph from the privacy segment as they introduce unbounded sensitivity. In practice, however, circular definitions are most commonly used to aggregate all values of a stream over time. Beyond sliding-window aggregations, many stream-based languages support discrete all-aggregations, which aggregate all stream values since the start of the monitor. The most common loops in specifications can therefore automatically be rewritten as all-aggregations.

To introduce a barrier at an all-aggregation, noise must be added to each released aggregate in order to preserve differential privacy. However, independently perturbing each aggregated result leads to unbounded privacy loss, since each private value contributes to all future aggregations. A standard solution to this problem is provided by tree-based mechanisms for differential privacy under

continual observation [21,14,41,47,43,13]. In these approaches, noisy intermediate aggregates are precomputed and reused across multiple releases. Rather than repeatedly accessing the private data, aggregates are constructed from these perturbed subaggregates. Concretely, the stream is partitioned into intervals organized in a binary tree, where each node represents an aggregate over a fixed interval and is perturbed only once. Aggregates at any time step are obtained by combining the noisy values of a small number of nodes in the tree whose intervals form a partition of the required range. Several approaches have been proposed to handle unbounded time with these tree-based mechanisms [14,47]. In this work, we adopt a construction based on a geometric series that distributes the privacy budget across the infinite height of the tree.

Particular to our setting is that aggregations are not restricted to input streams, where only a single event must be hidden. Aggregations may also be applied to output streams, where temporal operators have already propagated a single change to multiple stream values. Our tree-based construction accommodates this, as it only reasons about the total L_1 -sensitivity of the aggregated stream. Formal details and proofs are provided in Appendix H.

Figure 4a illustrates this construction for a discrete all-aggregation. Each leaf of the tree corresponds to a value of the aggregated stream. At the position marked with the black triangle, the aggregation over all values is computed by combining the precomputed noisy aggregate over the first four values (the blue node at the top of the tree) with the noisy aggregate of the fifth value at the current position.

In contrast to all-aggregations, whose aggregation is done over a growing prefix of the stream, sliding-window aggregations operate over a time window that shifts with the current evaluation point. The pacing type requirement in Definition 6 ensures that aggregation results are released at a fixed frequency. This restriction was originally introduced to guarantee that monitoring can be performed using finite memory. To this end, values are first aggregated into buckets, where the bucket size is determined by the aggregating stream’s frequency and the length of the window [44]. We can improve the utility of sliding window aggregations by adopting the tree-based approach, where the buckets form the leaves of the tree, as illustrated in Figure 4b. To compute, for example, the aggregation over the last four seconds, one sums the noisy values in the blue-marked nodes of the tree. Using this tree-based construction, in the example, each private value is only released three times, compared to four times without the tree-based approach when each aggregated result is perturbed independently.

6 Stricter Privacy Guarantees

Our approach assumes that the values at the individual input stream positions are independent, but this assumption does not hold in all applications. For example, consecutive sensor readings, such as GPS coordinates, are often correlated. In such cases, standard event-level differential privacy underestimates the required amount of noise. One way to address this is to adopt w -consecutive

Specification	no-priv	priv
Fraud Detection (J.1)	3 ms	7 ms
Geofence Small [4]	118 ms	278 ms
Geofence Large [4]	594 ms	1368 ms
Peak Detection [8]	3 ms	8 ms
Public Transport (J.2)	5 ms	12 ms
Driving Emissions [9]	123 ms	247 ms
Sensor Validation [8]	2 ms	7 ms

Table 1: Analysis runtime comparison for different specifications.

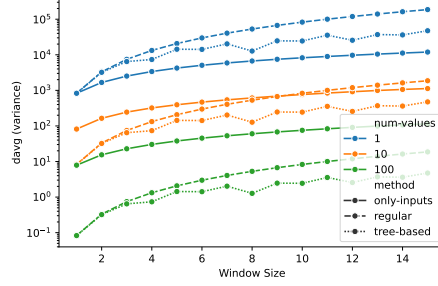


Fig. 5: Output variances for different sliding window aggregations.

differential privacy (w -DP) [35], which hides the effect of up to w consecutive input values. If correlations are limited to w consecutive values, w -DP provides meaningful privacy guarantees. Extending our approach to w -DP is straightforward, since our mechanisms already handle correlated outputs: we can apply the same techniques to w -sized groups of input values.

Another complementary strategy is to use a preprocessing monitor. In this approach, a first monitor transforms correlated inputs into independent values or aggregated summaries. These independent streams are then fed into a second, privacy-preserving monitor that ensures differential privacy. For example, the first monitor could extract individual trips from raw GPS data. The second monitor then computes the private outputs over these independent summaries. This strategy of using a preprocessing monitor can even ensure user-level privacy [21] by ensuring the second privacy-preserving monitor receives a single event per user.

7 Implementation and Evaluation

This section evaluates the practical applicability of our approach. We first describe the implementation of the privacy analysis within the RTLola framework and then assess the impact of different design choices using synthetic examples and a case study on monitoring public transportation data.

7.1 Implementation

Our implementation extends the RTLola frontend [5] with a privacy analysis that is executed after the dependency analysis. This analysis identifies all valid privacy barriers in the dependency graph and applies the heuristics described in Section 5.2 to select the barriers at which noise is injected. Using propagated sensitivity and range information, noise expressions are inserted into the intermediate representation of the specification, and tree-based aggregation is

translated into corresponding RTLola constructs. This way, no modifications to the existing backend implementations [3,25] are required for private monitoring.

We evaluate the runtime of the privacy analysis. The results are summarized in Table 1, where the *priv* column reports the maximum runtime across all heuristics. The measurements were obtained on a system with a 13th Gen Intel Core i7-1355U processor over 50 runs. We observe that the runtime more than doubles for all specifications. This increase is primarily due to the fact that, following the privacy analysis, which introduces new expressions and streams in the specification, we rerun earlier stages such as dependency analysis and type checking. Despite this overhead, all runtimes remain well below a few seconds, rendering the additional cost negligible for an analysis performed once prior to monitor deployment.

7.2 Impact of Tree-Based Aggregations

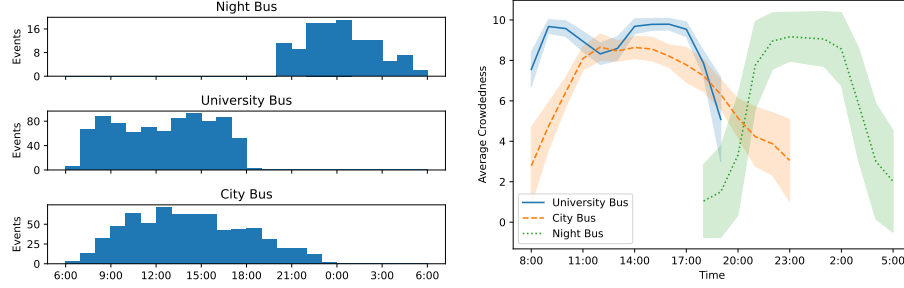
Next, we assess the impact of the tree-based aggregation on the accuracy of the output using the example shown in Figure 1. Accuracy is quantified by computing, for each timestamp, the variance across 100 runs of the private monitor ($\epsilon = 1$) on the same input trace and then averaging this variance across all timestamps. Lower variance indicates higher accuracy of the output.

We compare how accuracy is influenced by the structure of the aggregation window. First, we vary the number of buckets in the window. Increasing the number of buckets causes individual values to contribute to multiple aggregates, which in turn requires more perturbation to ensure privacy. Second, we vary the number of values per bucket. More values per bucket allow individual values to be more effectively obscured by others, thereby reducing the variance.

We compare three perturbation strategies. As a baseline, we add noise directly to each input before aggregation. We then consider the basic sensitivity-based perturbation described in Section 4 and the tree-based aggregation approach from Section 5.3. Figure 5 shows that for small windows, all three methods achieve similar utility. As the window size grows, the differences become more pronounced, with the tree-based approach yielding significantly lower variance than the sensitivity-based method. When each bucket contains only a single value, directly perturbing the inputs yields the highest accuracy. However, this approach rapidly degrades in utility as the number of values per bucket increases.

7.3 Case Study

To demonstrate the applicability of our approach in a realistic setting, we present a case study from the public transportation domain. We implemented a system that crowdsources information about public transportation usage by allowing users to report trips on specific bus lines. The reported events are processed by an RTLola monitor, which aggregates the data and displays anonymized usage statistics to users. The aggregated results provide insights into temporal usage patterns, such as the expected level of crowding on a bus line at different times



(a) Histogram of the number of events per hour per bus line. (b) The average crowdedness per hour output by the differentially private monitor.

Fig. 6: Results from monitoring synthetic public transportation data.

of day. Since the collected data reflects individual movement behavior, it is inherently privacy-sensitive. By relying on our privacy-preserving framework, all privacy guarantees are enforced automatically by the frontend analysis, allowing the application to be implemented without manual reasoning about privacy concerns. The corresponding RTLola specification is provided in Appendix J.2.

We evaluate our case study using synthetic data. Specifically, we consider three bus lines and generate a varying number of users depending on the time of day, as shown in Figure 6a. The first is a university bus, highly frequented but only during a shorter period of the day. The second is a city bus, which is frequently used throughout daytime hours. The third is a night bus, which also exhibits high usage per bus, but with significantly fewer buses and users overall.

In Figure 6b, we show the average crowdedness per time of day as reported by the private monitor ($\epsilon = 1$, deep heuristic), with the standard deviation (SD) shown as error bars. The SD indicates that the confidence of the monitor output depends on the number of users. To estimate the SD introduced by differential privacy, we execute the monitor 1500 times and calculate statistics over these runs. During high-traffic daytime hours, the differentially private estimates closely match the true crowdedness. At night, the smaller number of participants leads to a higher SD. But even during nighttime hours with fewer participants, the monitor output closely approximates the actual crowdedness.

8 Conclusion

In this paper, we have presented a method for integrating differential privacy into runtime monitoring to ensure system correctness while preserving the privacy of sensitive system data. Our approach analyzes the specifications and automatically enforces privacy guarantees without requiring any user interaction. As a result, users without privacy expertise can benefit from differential privacy without having to reason about sensitivities, noise calibration, or privacy budgets, which is particularly challenging given temporal dependencies in stream-based

specifications. Finally, privacy is enforced in a utility-aware manner by injecting noise at carefully selected points in the specification, thereby preserving the accuracy of monitoring outcomes while still providing rigorous privacy guarantees. We have implemented our approach in the stream-based monitoring framework RTLola, demonstrating that privacy-preserving runtime monitoring can be achieved automatically and with minimal loss of accuracy.

Our work lays the foundation for deploying runtime monitoring in privacy-critical domains where such techniques have traditionally been difficult to apply. This includes areas such as medical or financial systems, where continuous monitoring is essential, but the underlying data is highly privacy-critical. We believe that making privacy guarantees an integral part of runtime monitoring is a step toward making runtime verification applicable in a broader range of real-world, safety-critical and privacy-sensitive systems.

Acknowledgments. This work was partially supported by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660) and PreCePT (No. 521273327), and by the European Research Council (ERC) Grant HYPER (No. 101055412).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement. The artifacts and resources associated with this article are accessible via <https://doi.org/10.5281/zenodo.19694096>.

References

1. Abbas, H.: Private runtime verification: work-in-progress. In: Proceedings of the International Conference on Embedded Software Companion, New York, NY, USA, October 13-18, 2019. p. 11. ACM (2019). <https://doi.org/10.1145/3349568.3351552>
2. Banno, R., Matsuoka, K., Matsumoto, N., Bian, S., Waga, M., Suenaga, K.: Oblivious online monitoring for safety ltl specification via fully homomorphic encryption. In: International Conference on Computer Aided Verification. pp. 447–468. Springer (2022)
3. Baumeister, J., Correnson, A., Finkbeiner, B., Scheerer, F.: An intermediate program representation for optimizing stream-based languages. In: Piskac, R., Rakamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15933, pp. 393–407. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_20
4. Baumeister, J., Finkbeiner, B., Kohn, F., Löhr, F., Manfredi, G., Schirmer, S., Torens, C.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14682, pp. 207–218. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_10

5. Baumeister, J., Finkbeiner, B., Kohn, F., Scheerer, F.: A tutorial on stream-based monitoring. In: Platzter, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 14934, pp. 624–648. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_33
6. Baumeister, J., Finkbeiner, B., Scheerer, F.: Active monitoring with rtlola: A specification-guided scheduling approach. In: Könighofer, B., Torfah, H. (eds.) *Runtime Verification - 25th International Conference, RV 2025, Graz, Austria, September 15-19, 2025, Proceedings. Lecture Notes in Computer Science*, vol. 16087, pp. 181–201. Springer (2025). https://doi.org/10.1007/978-3-032-05435-7_11
7. Baumeister, J., Finkbeiner, B., Scheerer, F., Siber, J., Wagenpfeil, T.: Stream-based monitoring of algorithmic fairness. In: Gurfinkel, A., Heule, M. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 15696, pp. 60–81. Springer (2025). https://doi.org/10.1007/978-3-031-90643-5_4
8. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: Rtlola cleared for take-off: Monitoring autonomous aircraft. In: Lahiri, S.K., Wang, C. (eds.) *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3
9. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: Rtlola on board: Testing real driving emissions on your phone. In: Groote, J.F., Larsen, K.G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 12652, pp. 365–372. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_20
10. Blocki, J., Blum, A., Datta, A., Sheffet, O.: Differentially private data analysis of social networks via restricted sensitivity. In: Kleinberg, R.D. (ed.) *Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013*, pp. 87–96. ACM (2013). <https://doi.org/10.1145/2422436.2422449>
11. Cao, J., Xiao, Q., Ghinita, G., Li, N., Bertino, E., Tan, K.: Efficient and accurate strategies for differentially-private sliding window queries. In: Guerrini, G., Paton, N.W. (eds.) *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pp. 191–202. ACM (2013). <https://doi.org/10.1145/2452376.2452400>
12. Cao, Y., Yoshikawa, M., Xiao, Y., Xiong, L.: Quantifying differential privacy under temporal correlations. In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pp. 821–832. IEEE Computer Society (2017). <https://doi.org/10.1109/ICDE.2017.132>
13. Cardoso, A.R., Rogers, R.: Differentially private histograms under continual observation: Streaming selection into the unknown. In: Camps-Valls, G., Ruiz, F.J.R., Valera, I. (eds.) *International Conference on Artificial Intelligence and Statistics, AISTATS 2022, 28-30 March 2022, Virtual Event. Proceedings of Machine Learn-*

- ing Research, vol. 151, pp. 2397–2419. PMLR (2022), <https://proceedings.mlr.press/v151/rivera-cardoso22a.html>
14. Chan, T.H., Shi, E., Song, D.: Private and continual release of statistics. *ACM Trans. Inf. Syst. Secur.* **14**(3), 26:1–26:24 (2011). <https://doi.org/10.1145/2043621.2043626>
 15. Chen, Q., Ni, Z., Zhu, X., Xia, P.: Differential privacy histogram publishing method based on dynamic sliding window. *Frontiers Comput. Sci.* **17**(4), 174809 (2023). <https://doi.org/10.1007/S11704-022-1651-2>
 16. Chen, Y., Machanavajjhala, A., Hay, M., Miklau, G.: Pegasus: Data-adaptive differentially private stream processing. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. pp. 1375–1388. ACM (2017). <https://doi.org/10.1145/3133956.3134102>
 17. D’Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: *12th International Symposium on Temporal Representation and Reasoning (TIME 2005)*, 23–25 June 2005, Burlington, Vermont, USA. pp. 166–174. IEEE Computer Society (2005). <https://doi.org/10.1109/TIME.2005.26>
 18. Dong, W., Luo, Q., Yi, K.: Continual observation under user-level differential privacy. In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21–25, 2023*. pp. 2190–2207. IEEE (2023). <https://doi.org/10.1109/SP46215.2023.10179466>
 19. Dvijotham, K.D., McMahan, H.B., Pillutla, K., Steinke, T., Thakurta, A.: Efficient and near-optimal noise generation for streaming differential privacy. In: *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27–30, 2024*. pp. 2306–2317. IEEE (2024). <https://doi.org/10.1109/FOCS61266.2024.00135>
 20. Dwork, C., McSherry, F., Nissim, K., Smith, A.D.: Calibrating noise to sensitivity in private data analysis. In: Halevi, S., Rabin, T. (eds.) *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4–7, 2006, Proceedings. Lecture Notes in Computer Science*, vol. 3876, pp. 265–284. Springer (2006). https://doi.org/10.1007/11681878_14
 21. Dwork, C., Naor, M., Pitassi, T., Rothblum, G.N.: Differential privacy under continual observation. In: Schulman, L.J. (ed.) *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5–8 June 2010*. pp. 715–724. ACM (2010). <https://doi.org/10.1145/1806689.1806787>
 22. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.* **9**(3–4), 211–407 (2014). <https://doi.org/10.1561/0400000042>
 23. Fan, L., Xiong, L.: Real-time aggregate monitoring with differential privacy. In: Chen, X., Lebanon, G., Wang, H., Zaki, M.J. (eds.) *21st ACM International Conference on Information and Knowledge Management, CIKM’12, Maui, HI, USA, October 29 - November 02, 2012*. pp. 2169–2173. ACM (2012). <https://doi.org/10.1145/2396761.2398595>
 24. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings. Lecture Notes in Computer Science*, vol. 10012, pp. 152–168. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_10

25. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24
26. Feng, S., Mohammady, M., Wang, H., Li, X., Qin, Z., Hong, Y.: DPI: ensuring strict differential privacy for infinite data streaming. In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. pp. 1009–1027. IEEE (2024). <https://doi.org/10.1109/SP54263.2024.00124>
27. Finkbeiner, B., Fränzle, M., Kohn, F., Kröger, P.: Stream-based monitoring under measurement noise. In: Ábrahám, E., Abbas, H. (eds.) *Runtime Verification - 24th International Conference, RV 2024, Istanbul, Turkey, October 15-17, 2024, Proceedings. Lecture Notes in Computer Science*, vol. 15191, pp. 22–39. Springer (2024). https://doi.org/10.1007/978-3-031-74234-7_2
28. Finkbeiner, B., Keller, A., Schmidt, J., Schwenger, M.: Robust monitoring for medical cyber-physical systems. In: *Proceedings of the Workshop on Medical Cyber Physical Systems and Internet of Medical Things*. pp. 17–22 (2021)
29. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: *International Conference on Runtime Verification*. pp. 282–298. Springer (2018)
30. Henzinger, M., Upadhyay, J., Upadhyay, S.: Almost tight error bounds on differentially private continual counting. In: Bansal, N., Nagarajan, V. (eds.) *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*. pp. 5003–5039. SIAM (2023). <https://doi.org/10.1137/1.9781611977554.CH183>
31. Henzinger, M., Upadhyay, J., Upadhyay, S.: A unifying framework for differentially private sums under continual observation. In: Woodruff, D.P. (ed.) *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*. pp. 995–1018. SIAM (2024). <https://doi.org/10.1137/1.9781611977912.38>
32. Henzinger, T.A., Karimi, M., Thejaswini, K.S.: Privacy-preserving runtime verification. In: Huang, C., Chen, J., Shieh, S., Lie, D., Cortier, V. (eds.) *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*. pp. 2774–2787. ACM (2025). <https://doi.org/10.1145/3719027.3765137>
33. Johnson, N.M., Near, J.P., Song, D.: Towards practical differential privacy for SQL queries. *Proc. VLDB Endow.* **11**(5), 526–539 (2018). <https://doi.org/10.1145/3187009.3177733>, <http://www.vldb.org/pvldb/vol11/p526-johnson.pdf>
34. Kallwies, H., Leucker, M., Schmitz, M., Schulz, A., Thoma, D., Weiss, A.: Tessler—an ecosystem for runtime verification. In: *International Conference on Runtime Verification*. pp. 314–324. Springer (2022)
35. Kellaris, G., Papadopoulos, S., Xiao, X., Papadias, D.: Differentially private event sequences over infinite streams. *Proc. VLDB Endow.* **7**(12), 1155–1166 (2014). <https://doi.org/10.14778/2732977.2732989>, <http://www.vldb.org/pvldb/vol7/p1155-kellaris.pdf>
36. Li, S., Wen, Y., Wang, Z., Wang, W., Zhang, L., Meng, D.: Streamdp: Continual observation of real-world data streams with differential privacy. In: *IEEE International Performance, Computing, and Communications Conference, IPCCC 2024, Orlando, FL, USA, November 22-24, 2024*. pp. 1–10. IEEE (2024). <https://doi.org/10.1109/IPCCC59868.2024.10850443>

37. McSherry, F.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM* **53**(9), 89–97 (2010). <https://doi.org/10.1145/1810891.1810916>
38. Mohan, P., Thakurta, A., Shi, E., Song, D., Culler, D.E.: GUPT: privacy preserving data analysis made easy. In: Candan, K.S., Chen, Y., Snodgrass, R.T., Gravano, L., Fuxman, A. (eds.) *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. pp. 349–360. ACM (2012). <https://doi.org/10.1145/2213836.2213876>
39. Narayan, A., Haeberlen, A.: Djoin: Differentially private join queries over distributed databases. In: Thekkath, C., Vahdat, A. (eds.) *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. pp. 149–162. USENIX Association (2012), <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/narayan>
40. Nissim, K., Raskhodnikova, S., Smith, A.D.: Smooth sensitivity and sampling in private data analysis. In: Johnson, D.S., Feige, U. (eds.) *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*. pp. 75–84. ACM (2007). <https://doi.org/10.1145/1250790.1250803>
41. Perrier, V., Asghar, H.J., Kaafar, D.: Private continual release of real-valued data streams. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society (2019), <https://www.ndss-symposium.org/ndss-paper/private-continual-release-of-real-valued-data-streams/>
42. Proserpio, D., Goldberg, S., McSherry, F.: Calibrating data to sensitivity in private data analysis. *Proc. VLDB Endow.* **7**(8), 637–648 (2014). <https://doi.org/10.14778/2732296.2732300>, <http://www.vldb.org/pvldb/vol7/p637-proserpio.pdf>
43. Qardaji, W.H., Yang, W., Li, N.: Understanding hierarchical methods for differentially private histograms. *Proc. VLDB Endow.* **6**(14), 1954–1965 (2013). <https://doi.org/10.14778/2556549.2556576>, <http://www.vldb.org/pvldb/vol6/p1954-qardaji.pdf>
44. Schwenger, M.: *Statically-analyzed stream monitoring for cyber-physical Systems*. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2022), <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/33890>
45. Song, S., Wang, Y., Chaudhuri, K.: Pufferfish privacy mechanisms for correlated data. In: Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., Suci, D. (eds.) *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. pp. 1291–1306. ACM (2017). <https://doi.org/10.1145/3035918.3064025>
46. Upadhyay, J.: Sublinear space private algorithms under the sliding window model. In: Chaudhuri, K., Salakhutdinov, R. (eds.) *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. Proceedings of Machine Learning Research, vol. 97*, pp. 6363–6372. PMLR (2019), <http://proceedings.mlr.press/v97/upadhyay19a.html>
47. Wang, T., Chen, J.Q., Zhang, Z., Su, D., Cheng, Y., Li, Z., Li, N., Jha, S.: Continuous release of data streams under both centralized and local differential privacy. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. pp. 1237–1253. ACM (2021). <https://doi.org/10.1145/3460120.3484750>

A Operator Semantics

Given an evaluation model ω and a corresponding pacing model p , we define the semantics of the operators as follows. Note that for the semantics and the following proofs, we assume there is a separate stream definition for each subexpression.

$$\begin{aligned}
\omega(e_1 + e_2)(t) &= \omega(e_1)(t) + \omega(e_2)(t) \\
\omega(e_1 \cdot e_2)(t) &= \omega(e_1)(t) \cdot \omega(e_2)(t) \\
\omega(\text{sync}(y))(t) &= \omega(y)(t) \\
\omega(\text{offset}(y, o))(t) &= \begin{cases} \omega(y)(\text{last}(y, t, o, p)) & \text{if } \text{last}(y, t, o, p) \text{ exists} \\ \perp & \text{otherwise} \end{cases} \\
\omega(\text{hold}(y))(t) &= \begin{cases} \omega(y)(\text{last}(y, t, 0, p)) & \text{if } \text{last}(y, t, 0, p) \text{ exists} \\ \perp & \text{otherwise} \end{cases} \\
\omega(\text{defaults}(e, dft))(t) &= \begin{cases} \omega(e)(t) & \text{if } \omega(e)(t) \neq \perp \\ \omega(dft)(t) & \text{otherwise} \end{cases} \\
\omega(\text{ite}(c, e_1, e_2))(t) &= \begin{cases} \omega(e_1)(t) & \text{if } \omega(c)(t) = \top \\ \omega(e_2)(t) & \text{otherwise} \end{cases} \\
\omega(\text{aggr}(y, W, f))(t) &= f(\{\omega(y)(t') \mid t' \in \text{wtimes}(y, t, p, W)\}) \\
\omega(\text{hold}_{\text{bound}}(y))(t) &= \begin{cases} \omega(y)(\text{last}(y, t, 0, p)) & \text{if } \text{last}(y, t, 0, p) \text{ exists} \\ & \wedge \text{holdn}(x, y, t, p) \leq \text{bound} \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

We use the following predicate to determine how often a specific value of y is accessed from x :

$$\text{holdn}(x, y, t, p) = |\{t' \in \text{Time} \mid p(x)(t') \wedge t' \leq t \wedge \text{last}(y, t', 0, p) = \text{last}(y, t, 0, p)\}|$$

B Definition Δ

Given stream expressions e , e_1 and e_2 , the constant c , and the stream reference y , we define

$$\begin{aligned}
\Delta_{t,p,s}^{t'}(e_1 + e_2) &= \Delta_{t,p,s}^{t'}(e_1) + \Delta_{t,p,s}^{t'}(e_2) \\
\Delta_{t,p,s}^{t'}(c \cdot e) &= |c| \cdot \Delta_{t,p,s}^{t'}(e) \\
\Delta_{t,p,s}^{t'}(c) &= 0 \\
\Delta_{t,p,s}^{t'}(\text{sync}(y)) &= \Delta_{t,p,s}^{t'}(y) \\
\Delta_{t,p,s}^{t'}(\text{offset}(y, o)) &= \Delta_{t,p,s}^{\text{last}(y, t', p, o)}(y) \\
\Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \Sigma)) &= \sum_{t'' \in \text{wtimes}(y, t', p, W)} \Delta_{t,p,s}^{t''}(y) \\
\Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \text{last})) &= \begin{cases} \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(y) & \text{if } \text{last}(y, t', 0, p) \geq t' - W \\ 0 & \text{otherwise} \end{cases} \\
\Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \#)) &= 0 \\
\Delta_{t,p,s}^{t'}(\text{hold}(y)) &= \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(y) \\
\Delta_{t,p,s}^{t'}(\text{hold}_{\text{bound}}(y)) &= \begin{cases} \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(y) & \text{if } \text{holdn}(x, y, t', p) \leq \text{bound} \\ 0 & \text{otherwise} \end{cases} \\
\Delta_{t,p,s}^{t'}(\text{default}(e, \text{dft})) &= \max(\Delta_{t,p,s}^{t'}(e), \Delta_{t,p,s}^{t'}(\text{dft})) \\
\Delta_{t,p,s}^{t'}(x) &= \begin{cases} \text{ub}(x) - \text{lb}(x) & \text{if } t \rightsquigarrow_p^x t' \\ 0 & \text{otherwise,} \end{cases} \quad \text{for value-dependent operators}
\end{aligned}$$

with

$$\begin{aligned}
\text{last}(y, t, p, o) &= \max_o \{t' \in \text{Time} \mid t' < t \wedge p(y)(t')\} \\
\text{wtimes}(y, t, p, W) &= \{t' \in \text{Time} \mid p(t) - W \leq p(t') \leq p(t) \wedge p(y)(t')\}.
\end{aligned}$$

$\rightsquigarrow$ is defined in Appendix E.

C Bounds of Expressions

The *lower bound* of a stream is defined as

$$\begin{aligned}
\text{lb} : \mathbf{ID}^\uparrow \uplus \mathbf{ID}^\downarrow &\rightarrow \mathbb{R} \cup \{-\infty\} \\
\text{lb}(x) &= \begin{cases} -\infty & \text{if } x \in \mathbf{ID}^\uparrow \\ \text{lb}(\varphi_x) & \text{if } x \in \mathbf{ID}^\downarrow, \end{cases}
\end{aligned}$$

and recursively on expressions as

$$\begin{aligned}
lb(\text{sync}(y)) &= lb(y) \\
lb(\text{offset}(y, -)) &= lb(y) \\
lb(f_1 + f_2) &= lb(f_1) + lb(f_2) \\
lb(f_1 \cdot f_2) &= \min(ub(f_1) \cdot ub(f_2), lb(f_1) \cdot ub(f_2), \\
&\quad ub(f_1) \cdot lb(f_2), lb(f_1) \cdot lb(f_2)) \\
lb(\text{aggr}(y, W, S, \Sigma)) &= -\infty \\
lb(\text{aggr}(y, W, S, \#)) &= 0 \\
lb(\text{ite}(c, f_1, f_2)) &= \min(lb(f_1), lb(f_2)) \\
lb(\text{clamp}(f, nlb, nub)) &= nlb.
\end{aligned}$$

The *upper bound* of a stream is defined as

$$\begin{aligned}
ub : \mathbf{ID}^\uparrow \uplus \mathbf{ID}^\downarrow &\rightarrow \mathbb{R} \cup \{\infty\} \\
ub(x) &= \begin{cases} \infty & \text{if } x \in \mathbf{ID}^\uparrow \\ ub(\varphi_x) & \text{if } x \in \mathbf{ID}^\downarrow, \end{cases}
\end{aligned}$$

and recursively on expressions as

$$\begin{aligned}
ub(\text{sync}(y)) &= ub(y) \\
ub(\text{offset}(y, -)) &= ub(y) \\
ub(f_1 + f_2) &= ub(f_1) + ub(f_2) \\
ub(f_1 \cdot f_2) &= \max(ub(f_1) \cdot ub(f_2), lb(f_1) \cdot ub(f_2), \\
&\quad ub(f_1) \cdot lb(f_2), lb(f_1) \cdot lb(f_2)) \\
ub(\text{aggr}(y, W, S, \Sigma)) &= \infty \\
ub(\text{aggr}(y, W, S, \#)) &= 0 \\
ub(\text{ite}(c, f_1, f_2)) &= \max(ub(f_1), ub(f_2)) \\
ub(\text{clamp}(f, nlb, nub)) &= nub.
\end{aligned}$$

D Proof for Theorem 2

Proof. Proof by structural induction over the defining stream expression of x .

Base Case: If x is an input stream, then, if $t' = t$, $|\omega(x)(t') - \omega'(x)(t')| \leq s$ by assumption (s -distant), and $\Delta_{t,p,s}^{t'}(x) = s$ by definition of Δ . If $t' \neq t$, then $|\omega(x)(t') - \omega'(x)(t')| = 0$ by assumption (event-level adjacent and differ at timestamp t), and $\Delta_{t,p,s}^{t'}(x) = 0$ by definition of Δ .

Induction Hypothesis: Assume that for all subexpressions e of the stream expression for x , for every timestamp t' , it holds that

$$|\omega(x)(t') - \omega'(x)(t')| \leq \Delta_{t,p,s}^{t'}(x).$$

Induction Step: Consider the possible operations in the stream expression:

– **Addition:** $e = e_1 + e_2$

$$\begin{aligned}
 |\omega(e)(t') - \omega'(e)(t')| &= |(\omega(e_1)(t') + \omega(e_2)(t')) \\
 &\quad - (\omega'(e_1)(t') + \omega'(e_2)(t'))| \\
 &= |(\omega(e_1)(t') - \omega'(e_1)(t')) \\
 &\quad + (\omega(e_2)(t') - \omega'(e_2)(t'))| \\
 &\leq |\omega(e_1)(t') - \omega'(e_1)(t')| \\
 &\quad + |\omega(e_2)(t') - \omega'(e_2)(t')| \quad \text{Triangle inequality} \\
 &\leq \Delta_{t,p}^{t'}(f_1) + \Delta_{t,p}^{t'}(f_2) \quad \text{Induction hypothesis} \\
 &= \Delta_{t,p}^{t'}(f) \quad \text{Def. } \Delta
 \end{aligned}$$

– **Scaling by constant:** $e = c \cdot e'$

$$\begin{aligned}
 |\omega(e)(t') - \omega'(e)(t')| &= |c \cdot (\omega(e')(t') - \omega'(e')(t'))| \\
 &= |c| \cdot |\omega(e')(t') - \omega'(e')(t')| \quad \text{absolute value} \\
 &\leq |c| \cdot \Delta_{t,p,s}^{t'}(e') \quad \text{Induction hypothesis} \\
 &= \Delta_{t,p,s}^{t'}(e) \quad \text{Def. } \Delta
 \end{aligned}$$

– **Offset access:** $e = \text{offset}(y, o)$

$$\begin{aligned}
 |\omega(e)(t') - \omega'(e)(t')| &= |\omega(y)(\text{last}(X, t', p, o)) \\
 &\quad - \omega'(y)(\text{last}(y, t', p, o))| \quad \text{behaves according to } p \\
 &\leq \Delta_{t,p,s}^{\text{last}(y, t', o, p)}(X) \quad \text{Induction hypothesis} \\
 &= \Delta_{t,p,s}^{t'}(e) \quad \text{Def. } \Delta
 \end{aligned}$$

– **Synchronous access:** $e = \text{sync}(y)$

$$\begin{aligned}
 |\omega(e)(t') - \omega'(e)(t')| &= |\omega(y)(t') - \omega'(y)(t')| \\
 &\leq \Delta_{t,p,s}^{t'}(y) \quad \text{Induction hypothesis} \\
 &= \Delta_{t,p,s}^{t'}(e) \quad \text{Def. } \Delta
 \end{aligned}$$

– **Window access (sum):** $e = \text{aggr}(y, W, \sigma)$

$$\begin{aligned}
& |\omega(e)(t') - \omega(e)(t'')| \\
&= \left| \sum_{t'' \in \text{wtimes}(y, t', p, W)} \omega(y)(t'') - \sum_{t'' \in \text{wtimes}(y, t', p, W)} \omega'(y)(t'') \right| && \text{behaves according to } p \\
&= \left| \sum_{t'' \in \text{wtimes}(y, t', p, W)} \omega(y)(t'') - \omega'(y)(t'') \right| && \begin{array}{l} \text{wtimes is finite} \\ (p \text{ is well-typed}) \end{array} \\
&\leq \sum_{t'' \in \text{wtimes}(X, t', p, W)} |\omega(X)(t'') - \omega'(y)(t'')| && \text{absolute value} \\
&\leq \sum_{t'' \in \text{wtimes}(y, t', p, W)} \Delta_{t,p,s}^{t'}(y) && \text{Induction hypothesis} \\
&= \Delta_{t,p,s}^{t'}(e) && \text{Def. } \Delta
\end{aligned}$$

– **Window access (count):** $e = \text{aggr}(y, W, \#)$

$$\begin{aligned}
|\omega(e)(t') - \omega'(e)(t')| &= \left| \sum_{t'' \in \text{wtimes}(y, t', p, W)} 1 - \sum_{t'' \in \text{wtimes}(y, t', p, W)} 1 \right| && \text{behaves according to } p \\
&= 0 = \Delta_{t,p,s}^{t'}(e) && \text{Def. } \Delta
\end{aligned}$$

– **Window access (last):** $e = \text{aggr}(y, W, \text{last})$

$$\begin{aligned}
|\omega(e)(t') - \omega'(e)(t')| &= |\omega(y)(\text{last}(y, t', 0, p)) - \omega'(y)(\text{last}(y, t', 0, p))| \\
&\leq \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(y) && \text{Induction hypothesis} \\
&= \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(e) && \text{Def. } \Delta
\end{aligned}$$

– **Hold:** $e = \text{hold}(y)$. The same as for last aggregation:

$$\begin{aligned}
|\omega(e)(t') - \omega'(e)(t')| &= |\omega(y)(\text{last}(y, t', 0, p)) - \omega'(y)(\text{last}(y, t', 0, p))| \\
&\leq \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(y) && \text{Induction hypothesis} \\
&= \Delta_{t,p,s}^{\text{last}(y, t', 0, p)}(e) && \text{Def. } \Delta
\end{aligned}$$

For the extension of Δ presented in Section 4.5, it holds that

$$\omega(e)(t') \geq lb(e) \quad \text{and} \quad \omega'(e)(t') \leq ub(e).$$

From this, it follows that

$$|\omega(e)(t') - \omega'(e)(t')| \leq ub(x) - lb(x).$$

The recursion ends because the dependency graph is acyclic.

E Definition $\rightsquigarrow$

A change in the inputs at time t influences the value of x at time t' if

$$\begin{aligned}
t \rightsquigarrow_p^x t' &:= \begin{cases} t = t' & \text{if } x \in \text{ID}^\uparrow \\ t \rightsquigarrow_p^{\varphi_x} t' & \text{if } x \in \text{ID}^\downarrow \end{cases} \\
t \rightsquigarrow_p^{\text{sync}(y)} t' &:= t \rightsquigarrow_p^y t' \\
t \rightsquigarrow_p^{\text{offset}(y,o)} t' &:= t \rightsquigarrow_p^y \text{last}(y, t', o, p) \\
t \rightsquigarrow_p^{\text{hold}(y)} t' &:= t \rightsquigarrow_p^y \text{last}(y, t', 0, p) \\
t \rightsquigarrow_p^{e_1 \circ e_2} t' &:= t \rightsquigarrow_p^{e_1} t' \vee t \rightsquigarrow_p^{e_2} t' \quad \text{for all binary operators } \circ \\
t \rightsquigarrow_p^{\text{ite}(c, e_1, e_2)} t' &:= t \rightsquigarrow_p^c t' \vee t \rightsquigarrow_p^{e_1} t' \vee t \rightsquigarrow_p^{e_2} t'
\end{aligned}$$

Theorem 5. *Given an acyclic specification φ without unbounded hold operators, then it holds that*

$$\exists n_x. \forall p \in \mathbb{P}_\varphi. \forall t \in \text{Time}. |\{t' \in T_e \mid t \rightsquigarrow_p^x t'\}| \leq n_x$$

the number of timepoints of x that are influenced by a change of an input at any timepoint is bounded by a constant n_x .

Proof. Proof by structural induction over stream-expression x :

Base Case: If x is an input stream, there is exactly one time point t' where $t = t'$.

Induction Hypothesis: Assume that for all subexpressions e of the stream expression for x , it holds that

$$\exists n_e. \forall p \in \mathbb{P}_\varphi. \forall t \in T_e. |\{t' \in T_e \mid t \rightsquigarrow_p^e t'\}| \leq n_e.$$

Induction Step: Consider the possible operations in the stream expression:

- **Binary operators:** $e = e_1 \circ e_2$. Then we set $n_e = n_{e_1} + n_{e_2}$ and it holds that

$$\begin{aligned}
|\{t' \in T_x \mid t \rightsquigarrow_p^e t'\}| &= |\{t' \in T_x \mid t \rightsquigarrow_p^{e_1} t' \vee t \rightsquigarrow_p^{e_2} t'\}| \\
&\leq |\{t' \in T_x \mid t \rightsquigarrow_p^{e_1} t'\}| \\
&\quad + |\{t' \in T_x \mid t \rightsquigarrow_p^{e_2} t'\}| \\
&\leq n_{e_1} + n_{e_2} = n_e
\end{aligned}$$

- **Synchronous access:** $e = \text{sync}(y)$. Then we set $n_e = n_y$ and it holds that

$$\begin{aligned}
|\{t' \in T_x \mid t \rightsquigarrow_p^e t'\}| &= |\{t' \in T_x \mid t \rightsquigarrow_p^y t'\}| \\
&\leq |\{t' \in T_y \mid t \rightsquigarrow_p^y t'\}| \quad p \text{ well-typed } (T_x \subseteq T_y) \\
&\leq n_y = n_e
\end{aligned}$$

– **Offset access:** $e = \text{offset}(y, o)$. Then we set $n_e = n_y$ and it holds that

$$\begin{aligned} |\{t' \in T_x \mid t \rightsquigarrow_p^e t'\}| &= |\{t' \in T_x \mid t \rightsquigarrow_p^y \text{last}(y, t', o, p)\}| \\ &= |\{t' \in T_y \mid t \rightsquigarrow_p^y t'\}| \\ &= n_y = n_e \end{aligned}$$

Because p is well-typed, for every $t'' \in T_y$, there exists only one $t' \in T_x$ for which $\text{last}(y, t', o, p) = t''$.

The proofs for the other operators follow a very similar structure to the proof in Appendix F. The recursion ends because the dependency graph is acyclic.

F Proof for Theorem 3

Proof. Proof by structural induction over stream-expression x :

Base Case: If x is an input stream, then we set $b = s$, and it holds that:

$$\begin{aligned} \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(I) &= \sum_{t' \in T_x} \begin{cases} s & \text{if } t' = t \\ 0 & \text{otherwise} \end{cases} \\ &\leq s = b. \end{aligned}$$

Induction Hypothesis: Assume that for all subexpressions e of the stream expression for x , it holds that

$$\exists b_e. \forall t. \left(\sum_{t' \in T_x} \Delta_{t,p,s}^{t'} e \right) \leq b_e.$$

Induction Step: Consider the possible operations in the stream expression:

– **Addition:** $e = e_1 + e_2$. Then we set $b_e = b_{e_1} + b_{e_2}$ and it holds that

$$\begin{aligned} \sum_{t' \in T_x} \Delta_{t,p,s}^{t'} e &= \sum_{t' \in T_x} \left(\Delta_{t,p,s}^{t'}(e_1) + \Delta_{t,p,s}^{t'}(e_2) \right) && \text{Def. } \Delta \\ &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e_1) + \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e_2) && \Delta \text{ non-negativ} \\ &\leq b_{e_1} + b_{e_2} && \text{Induction Hypothesis} \\ &= b_e \end{aligned}$$

- **Scaling by constant:** $e = c \cdot e'$. Then we set $b_e = |c| \cdot b_{e'}$ and it holds that

$$\begin{aligned}
\sum_{t' \in T_x} \Delta_{t,p,s}^{t'} e &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'} (c \cdot e') \\
&= \sum_{t' \in T_x} |c| \cdot \Delta_{t,p,s}^{t'}(e') && \text{Def. } \Delta \\
&= |c| \cdot \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e') \\
&\leq |c| \cdot b_{e'} = b_e && \text{Induction Hypothesis}
\end{aligned}$$

- **Synchronous access:** $e = \text{sync}(y)$. Then we set $b_e = b_y$ and it holds that

$$\begin{aligned}
\sum_{t' \in T_x} \Delta_{t,p,s}^{t'} e &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(\text{sync}(y)) \\
&= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(y) && \text{Def. } \Delta \\
&= \sum_{t' \in T_y} \Delta_{t,p,s}^{t'}(y) && p \text{ well-typed } (T_x \subseteq T_y) \\
&\leq b_y = b_e && \text{Induction Hypothesis}
\end{aligned}$$

- **Offset access:** $e = \text{offset}(y, o)$. Then we set $b_e = b_y$ and it holds that

$$\begin{aligned}
\sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e) &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(\text{offset}(y, o)) \\
&= \sum_{t' \in T_x} \Delta_{t,p,s}^{last(y, t', p, o)}(y) && \text{Def. } \Delta \\
&= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(y) && last \text{ is a bijection} \\
&= \sum_{t' \in T_y} \Delta_{t,p,s}^{t'}(y) && p \text{ well-typed } (T_x \subseteq T_y) \\
&= b_y = b_e && \text{Induction Hypothesis}
\end{aligned}$$

$last$ is a bijection because of the assumption that p is correctly typed.

- **Window Aggregation (sum):** $e = \text{aggr}(y, W, \Sigma)$. Because we know p is well-typed, there exists a frequency δ_x of the stream containing e . Then we set $b_e = (\lfloor \frac{W}{\delta_x} \rfloor + 1) \cdot b_y$ and it holds that:

$$\begin{aligned}
\sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e) &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \Sigma)) \\
&= \sum_{t' \in T_x} \sum_{t'' \in \text{wtimes}(y, t', p, W)} \Delta_{t,p,s}^{t''}(y) && \text{Def. } \Delta \\
&= \sum_{t' \in T_y} \Delta_{t,p,s}^{t'}(y) \cdot m(t')
\end{aligned}$$

where $m(t') = |\{t'' \in T_x \mid t' \in wtimes(y, t'', p, W)\}|$. We claim that

$$\forall t' \in T_y. m(t') \leq \left\lfloor \frac{W}{\delta_x} \right\rfloor + 1.$$

and then

$$\begin{aligned} \sum_{t' \in T_y} \Delta_{t,p,s}^{t'}(y) \cdot m(t') &\leq \sum_{t' \in T_y} \left(\Delta_{t,p,s}^{t'}(y) \cdot \left(\left\lfloor \frac{W}{\delta_x} \right\rfloor + 1 \right) \right) && \text{Claim} \\ &= \left(\left\lfloor \frac{W}{\delta_x} \right\rfloor + 1 \right) \cdot \sum_{t' \in T_y} \Delta_{t,p,s}^{t'}(y) && \text{Def. } \Sigma \\ &\leq \left(\left\lfloor \frac{W}{\delta_x} \right\rfloor + 1 \right) \cdot b_y = b_e && \text{Ind. Hypothesis} \end{aligned}$$

It remains to show that our claim $m(t') \leq \left\lfloor \frac{W}{\delta_x} \right\rfloor + 1$ holds. $m(t')$ counts how many values of x at timestamp $t'' \in T_x$ aggregate over a window that contains the value of y at timestamp $t' \in T_y$.

The aggregations of x that include the value y at timestamp t' are in the real-time window between $p(t)$ and $p(t) + W$. Because p is well-typed (and the aggregation e is contained in the stream-expression for x), it holds that

$$T_x = \{t' \in \text{Time} \mid \exists k. p(Y)(t') = k \cdot \delta_x\}.$$

Because the timestamps in T_x are spaced by δ_x in real time, any interval of length W can contain at most

$$\left\lfloor \frac{W}{\delta_x} \right\rfloor + 1$$

timestamps of T_x . Therefore, for every $t' \in T_y$, we have

$$m(t') = |\{t'' \in T_x \mid t' \in wtimes(y, t'', p, W)\}| \leq \left\lfloor \frac{W}{\delta_x} \right\rfloor + 1,$$

as required.

– **Window Aggregation (last):** $e = \text{aggr}(y, W, \text{last})$. Then we set $b_e = \left(\left\lfloor \frac{W}{\delta_x} \right\rfloor + 1 \right) \cdot b_y$ and it holds that:

$$\begin{aligned} \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e) &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \text{last})) \\ &\leq \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(\text{aggr}(y, W, \Sigma)) && \Sigma \text{ contains last} \\ &\leq \left(\left\lfloor \frac{W}{\delta_x} \right\rfloor + 1 \right) \cdot b_y = b_e && \text{Proof for } \Sigma \end{aligned}$$

- **Bounded hold:** $e = \text{hold}_{\text{bound}}(y)$. Then we set $b_e = b_y \cdot \text{bound}$ and it holds that:

$$\begin{aligned}
\sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e) &= \sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(\text{hold}_{\text{bound}}(y)) \\
&\leq \sum_{t' \in T_x} \begin{cases} \Delta_{t,p,s}^{\text{last}(y,t',0,p)}(y) & \text{if } \text{holdn}(x,y,t') \leq \text{bound} \\ 0 & \text{otherwise} \end{cases} \\
&\leq \sum_{t' \in T_y} \Delta_{t,p,s}^{t'}(y) \cdot m(t') \\
&\leq b_y \cdot m(t') && \text{Induction hypothesis} \\
&\leq b_y \cdot \text{bound} && m(t') \leq \text{bound}
\end{aligned}$$

where

$$m(t) = |\{t' \in T_x \mid \text{last}(y,t',0,p) = t \wedge \text{holdn}(x,y,t',p) \leq \text{bound}\}|$$

is bounded by bound , because there can be at most bound distinct timepoints $t' \in T_x$ that map to the same t before the hold counter $\text{holdn}(x,y,t',p)$ exceeds bound .

For the extension presented in Section 4.5, we have shown in Appendix E that

$$\exists n_x. \forall p \in \mathbb{P}_\varphi. \forall t \in \text{Time}. |\{t' \in T_e \mid t \rightsquigarrow_p^x t'\}| \leq n_x.$$

Then,

$$\sum_{t' \in T_x} \Delta_{t,p,s}^{t'}(e) = \sum_{t' \in T'} ub(x) - lb(x)$$

where $T' = \{t' \in T_x \mid t \rightsquigarrow_p^x t'\}$

$$\begin{aligned}
&= |T'| \cdot (ub(x) - lb(x)) \\
&\leq n_x \cdot (ub(x) - lb(x))
\end{aligned}$$

G Proof for Differential Privacy

Proof. Let $\omega, \omega' \in \mathbb{W}$ be s -distant, event-level adjacent evaluation models, differing at timestamp t . By Theorem 2, for every timestamp t' ,

$$|\omega(x)(t') - \omega'(x)(t')| \leq \Delta_{t,p,s}^{t'}(x),$$

where p is the common pacing model.

Summing over all timestamps at which x is evaluated and applying Theorem 3 yields

$$\sum_{t' \in T_{p,x}} |\omega(x)(t') - \omega'(x)(t')| \leq \sum_{t' \in T_{p,x}} \Delta_{t,p,s}^{t'}(x) \leq b_{x,s}.$$

Thus, the total L_1 -sensitivity of the released stream is bounded by $b_{x,s}$. The result now follows directly from Theorem 1: adding independent Laplace noise with scale $b_{x,s}/\varepsilon$ to each output value ensures s -distant, ε -event-level differential privacy.

H Tree-Based Aggregation

Remark 1. For this section we assume that we aggregate over a stream x which has a value at every timepoint. If the original stream does not, we introduce an auxiliary stream by deterministically replacing each $\perp$ with the neutral element of the aggregation function. This transformation is independent of the data and does not affect differential privacy.

The nodes of the tree partition the timepoints into intervals. Let $\mathcal{I}_k$ be a partition of the timepoints into the nodes in level k of the tree:

$$\mathcal{I}_k = \{I_{k,1}, I_{k,2}, \dots\}.$$

For each level k of the tree, we define a separate mechanism, which returns all the aggregates for each interval on level k given a stream $x \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$ in an evaluation model $\omega \in \mathbb{W}$:

$$M_{k,x}(\omega) = \left(\sum_{t \in I_{k,j}} \omega(x)(t) \right)_{j=1}^{\infty}$$

and we define $\tilde{M}_k(x) = M_k(x) + \eta_k$ as a perturbed version where $\eta_{k,j} \sim \text{Lap}(\frac{b_{x,s}}{\varepsilon_k})$ and $b_{x,s}$ is the sensitivity bound obtained for the stream x through Theorem 3.

This mechanism releases all nodes at level k at once.

Theorem 6. *Given a valid evaluation model ω the mechanism $\tilde{M}_{k,x}(\omega)$ satisfies s -distant, ε_k -event-level differential privacy.*

Proof. We compute an overapproximation of the L1-sensitivity of $M_{k,x}$. Given two s -distant, event-level adjacent evaluation models ω and ω' :

$$\begin{aligned} \|M_{k,x}(\omega) - M_{k,x}(\omega')\|_1 &= \sum_{j=1}^{\infty} |M_{k,x}(\omega)_j - M_{k,x}(\omega')_j| \\ &= \sum_{j=1}^{\infty} \left| \sum_{t \in I_{k,j}} \omega(x)(t) - \omega'(x)(t) \right| \\ &\leq \sum_{j=1}^{\infty} \sum_{t \in I_{k,j}} |\omega(x)(t) - \omega'(x)(t)| \\ &\leq \sum_{t=1}^{\infty} |\omega(x)(t) - \omega'(x)(t)| \cdot |\{j \mid t \in I_{k,j}\}| \end{aligned}$$

I_k is a partition of all time points and therefore $|\{j \mid t \in I_{k,j}\}| = 1$.

$$\begin{aligned} &= \sum_{t=1}^{\infty} |\omega(x)(t) - \omega'(x)(t)| \\ &\leq \sum_{t=1}^{\infty} \Delta_{t',p,s}^t \quad \text{Theorem 2} \end{aligned}$$

where t' is the timepoint where ω and ω' differ.

$$\leq b_{x,s} \quad \text{Theorem 3}$$

As we have shown that $b_{x,s}$ is an overapproximation of the sensitivity of $M_{k,x}$, it follows from Theorem 1 that $\tilde{M}_{k,x}$ satisfies ε_k -differential privacy w.r.t the event-level, s -distance adjacency relation.

Theorem 7. *The mechanism $\tilde{M}(x) = (\tilde{M}_1(x), \tilde{M}_2(x), \dots)$ satisfies s -distant, ε -private event-level differential privacy when $\sum_{k=1}^{\infty} \varepsilon_k \leq \varepsilon$.*

This follows directly from the composition theorem of differential privacy. One example for ε_k is using the geometric series with $\varepsilon_k = \frac{6}{\pi^2 \cdot k^2} \cdot \varepsilon$.

The mechanism releases all perturbed tree nodes. For each timepoint, the reported aggregate is obtained by combining a finite subset of these nodes whose intervals partition the interval from the start up to this timepoint. This combination is pure post-processing.

I RTLola Extension Constructs

To build these constructs, we make use of RTLola features not covered in this paper. For more information, we refer to the RTLola Tutorial [5].

I.1 Bounded Hold

Given a bounded hold access from x to y :

```
1 | output x @px := ...
2 | output y @py := x.hold(for_discrete: b).defaults(to: dft)
```

we can replace the bounded hold access by utilizing the annotated (or derived) pacing types and introducing a counter stream:

```
1 | output x @px := ...
2 | output hold_counter
3 |   eval @px with 0
4 |   eval @py with hold_counter.offset(by:-1).defaults(to: 0) + 1
5 | output y @py := if hold_counter > b then dft else x.hold(or: dft)
```

I.2 Tree Based Aggregation

As an example, consider the following over-all sum aggregation:

```
1 | input x : UInt64
2 | output y := x.aggregate(over_discrete: all, using: sum)
```

we can express this aggregation using the tree-based approach:

```

1 input x : UInt64
2 output time : UInt64 @x := time.last(or: 0) + 1
3 output height := log_2(time)
4 output layers(level)
5   spawn with height
6   eval @x when level == 0 with a
7   eval @x when (time - 1) % (2**level) == 0 with a
8   eval @x when (time - 1) % (2**level) != 0 with
      layers(level).last(or: 0) + x
9
10 output private_layer(level)
11   spawn with height
12   eval @x when time % (2**level) == 0 with layers(level).hold(or: 0)
      + laplace(1/(level+1))
13
14 output layer_contribution(level)
15   spawn with height
16   eval @x when ((time >> level) & 1) == 1 with
      private_layer(level).hold(or: 0)
17
18 output y := layer_contribution.aggregate(over_instances: fresh,
      using: sum)

```

A sliding window output `b @1s := a.aggregate(over: 3s, using: sum)` can be expressed as

```

1 output time @1s := time.last(or: 0) + 1
2 output time_m @1s := time % 6
3 output l0 @1s := a.aggregate(over: 1s, using: sum)
4 output l0_n @1s := l0 + laplace(1/2)
5 output l1 @1s := if (time + 1) % 2 == 0 then l0 +
      l0.offset(by:-1).defaults(to: 0.0) else 0.0
6 output l1_n @1s := l1 + laplace(1/2)
7 output b @1s :=
8   if time_m == 0 || time_m == 2 || time_m == 4 then
9     l1_n + l0_n.offset(by: -2).defaults(to: 0.0)
10  else if time_m == 1 || time_m == 3 || time_m == 5 then
11    l1_n + l0_n.offset(by: -2).defaults(to: 0.0)
12  else 0.0

```

J Specifications

J.1 Fraud Detection

```

1 #[sensitivity="unbounded"]
2 input target: Float64
3 #[range_from="0", range_to="1000"]
4 input amount : Float64
5

```

```

6  output total_amount_per_month @30d := amount.aggregate(over: 30d,
   using: sum)
7  output average_amount_per_month 30d := amount.aggregate(over: 30d,
   using: average).defaults(to: 0.0)
8
9  #[public]
10 output t0 := total_amount_per_month > 50000.0
11 #[public]
12 output t1 := average_amount_per_month > 700.0
13
14 output amount_per_target(t)
15   spawn with target
16   eval with if t == target then amount else 0.0
17
18 output total_amount_per_target_per_month(t)
19   spawn with target
20   eval @Global(30d) with amount_per_target(t).aggregate(over: 30d,
   using: sum)
21
22 output max_amount_to_single_target_per_month
23   eval @30d with
   total_amount_per_target_per_month.aggregate(over_instances:
   all, using: max).defaults(to: 0.0)
24
25 output total_max_ratio @30d := max_amount_to_single_target_per_month
   / total_amount_per_month
26
27 #[public]
28 output t2 := max_amount_to_single_target_per_month > 10000.0
29 #[public]
30 output t3 := total_max_ratio > 0.5

```

J.2 Public Transportation

```

1  #[sensitivity="unbounded"]
2  input line : UInt64
3  #[range_from="1", range_to="10"]
4  input crowdedness : Float64
5
6  output crowdedness_per(l)
7   spawn with line
8   eval with if line == 1 then crowdedness else 0.0
9
10 output one_if(l)
11   spawn with line
12   eval with if line == 1 then 1.0 else 0.0
13
14 output sum_crowdedness(l)

```

```

15   spawn with line
16   eval @1h with crowdedness_per(1).aggregate(over: 1h, using: sum)
17
18   output count_crowdedness(1)
19   spawn with line
20   eval @1h with one_if(1).aggregate(over: 1h, using: sum)
21
22   output avg_crowdedness(1)
23   spawn with line
24   eval @1h with sum_crowdedness(1) / count_crowdedness(1)
25
26   output avg_crowdedness_clipped(1)
27   spawn with line
28   eval @1h with if avg_crowdedness(1) > 10.0 then 10.0 else if
      avg_crowdedness(1) < 0.0 then 0.0 else avg_crowdedness(1)
29
30   #[public]
31   output avg_124
32   eval @1h when count_crowdedness(1.0).hold(or: 0.0) > 5.0 with
      avg_crowdedness_clipped(1.0).hold(or: 0.0)
33   #[public]
34   output avg_101
35   eval @1h when count_crowdedness(2.0).hold(or: 0.0) > 5.0 with
      avg_crowdedness_clipped(2.0).hold(or: 0.0)
36   #[public]
37   output avg_party
38   eval @1h when count_crowdedness(3.0).hold(or: 0.0) > 5.0 with
      avg_crowdedness_clipped(3.0).hold(or: 0.0)

```