Safe Validation of Pricing Agreements

John Kolesar
Yale University[§]
john.kolesar@yale.edu

Tancrède Lepoint
Amazon Web Services
tlepoint@amazon.com

Martin Schäf
Amazon Web Services
schaef@amazon.com

Willem Visser
Amazon Web Services
vissie@amazon.com

***Abstract*—Pricing agreements at AWS define how customers are billed for usage of services and resources. A pricing agreement consists of a complex sequence of terms that can include free tiers, savings plans, credits, volume discounts, and other similar features. To ensure that pricing agreements reflect the customers' intentions, we employ a protocol that runs a set of validations that check all pricing agreements for irregularities.**

Since pricing agreements are sensitive, we want to limit the amount of information available to actors involved in the validation protocol. Actors who repair validation errors should not be able to obtain information that is not necessary for the repair. Personal information for individual customers should remain private, and we also want to hide statistical properties of the set of pricing agreements, such as the average number of credits awarded to customers.

We introduce Parrot, a protocol for validation of pricing agreements that prevents information leakage. Parrot involves three categories of actors: the auditor who writes a query to check a validity condition, the fixers who repair anonymized invalid pricing agreements, and the owners who have permission to view non-anonymized pricing agreements. The protocol includes safeguards to prohibit parties other than the owners from learning sensitive information about individual customers or statistical properties of the full set of pricing agreements.

I. Introduction

A pricing agreement is a legal document between a customer and AWS that defines how the customer is billed for usage of services and resources. The official record of a customer's usage over a given billing period takes the form of a list of *line items*. A line item contains the product code for the service used, such as a one-hour rental of an EC2 `r8g.2xlarge` instance in the Virginia region, and the time period when the usage occurred.

The pricing agreement determines the price of the usage in a line item. For instance, a pricing agreement could contain a *pricing term* that assigns the product code for a one-hour rental of an `r8g.2xlarge` instance in Virginia to a price of $0.47128. Because rates can change over time, pricing terms can also specify a period during which they are valid. Additionally, pricing can be conditional: for example, in the case of EC2, customers can purchase Savings Plans that offer a lower hourly rate in exchange for an up-front payment, or prices can be tiered to offer discounted rates above a certain usage. Services can also offer free tiers for a period of time, or a certain amount of free usage every month. Further, customers can receive credits that are applied to their usage.

Intuitively, a pricing agreement is a loop-free program of pricing terms that takes usage data in the form of line items for a billing period as input and assigns a price to each line item. Like any other program, pricing agreements can contain bugs. A bug in a pricing agreement is anything that results in an unexpected price for a line item. Possible bugs include data-entry errors, unexpected interactions between different terms in a pricing agreement (such as credits and savings plans being applied in an unexpected order), and unsupported combinations of pricing terms (not all products are eligible for all types of discounts). To detect pricing agreement bugs before they materialize in a faulty bill to the customer, we employ static analysis to evaluate all agreements periodically.

In this paper, we discuss our protocol for validating pricing agreements and for fixing errors that we identify during validation. Our main goal for the design of the protocol is to preserve the privacy of customers' pricing agreements. Pricing agreements contain sensitive information, such as account IDs, spend commitments, and available credits, and we do not want to grant unnecessary access to actors who do not need it during validation. Furthermore, we want to ensure that our protocol cannot leak statistical information, such as the average discount across all pricing agreements or the total committed usage across agreements.

To protect sensitive information during validation, we introduce *Parrot* (Pricing Agreement Repair with Randomized Obscured Tasks), a new protocol for pricing agreement validation. Parrot divides the responsibilities for validation between three categories of actors: the auditor, the fixers, and the owners. The auditor writes a *query* that checks whether any pricing agreements contain validation errors, the fixers repair any flawed agreements that the query identifies, and the owners possess full permissions for viewing agreements.

We need to balance two objectives for the design of Parrot: we want to avoid granting any form of bulk access to agreements to actors other than the owners, but we also want to ensure that all actors have the information they need to perform their tasks effectively. More specifically, we want to ensure that the auditor receives sufficient information about the performance of individual validation queries to produce business metrics, such as the fix rate or the number of spurious results, without being able to learn sensitive information about individual agreements or the set of agreements as a whole. Additionally, we want to ensure that the fixers see enough details of the pricing agreements to fix validation errors without accumulating unnecessary information over time. We specify a list of formal safety guarantees for our prevention of information leakage about both individual agreements and aggregate properties of the whole set of pricing agreements.

[§]Work performed while at Amazon Web Services

In this paper, we focus specifically on the domain of pricing agreements, but Parrot applies equally well in any other domain where specialists with limited permissions need to review large amounts of sensitive data, such as security audits of unreleased source code, audits of medical data in a hospital, or billing audits in large companies.

We begin by giving a high-level overview of pricing agreements and how we validate them in Section II. In Section III, we describe the existing validation protocol together with the privacy properties that we want to establish. Then, in Section IV, we propose our new protocol Parrot. In Section V, we show that Parrot upholds our desired privacy properties.

II. Semantics and Analysis of Pricing Agreements

This section introduces our representation of pricing agreements and the procedure we follow to validate them. We provide only a high-level description, as the focus of the paper is on the protocol that uses this analysis and not the analysis itself. In fact, our protocol is agnostic to the representation of agreements or the technology used for validation.

A pricing agreement is a program that operates on sets of *line items* and assigns a price to each line item. A line item is a tuple in $\texttt{Code} \times \texttt{Period}$ of a product code and a usage period. A product code $c \in \texttt{Code}$ is a value that uniquely identifies a usage of a service. The period in `Period` is the date range during which the usage occurred, such as $2024/01 - 2024/12$.

Pricing agreements assign prices to line items. The state of the execution of an agreement is a tuple in $\texttt{Use} \times \texttt{Rate} \times \texttt{Credit}$ of maps that assign each product code in `Code` to its current usage (`Use`), the current billing rate (`Rate`), and the available credit or accumulated charges for the product code (`Credit`). Structurally, a pricing agreement $p = T_0; T_1; \ldots$, is a sequence of terms. Terms all follow the same pattern:

$$T = \textbf{during}(\texttt{Period})\ \textbf{if}(\texttt{Condition})\ \textbf{do}(\texttt{Effect})$$

Each term consists of a period during which it is active (`Period`), a condition under which it applies (`Condition`), and an effect (`Effect`). The `Condition` field is a Boolean expression over the state of the execution, and the `Effect` field is an assignment update to the maps in the current state. As an example of a pricing agreement, let p_X be an agreement with a tiered discount for a service X for a user u:

$\textbf{during}(2024/2025)$
  $\textbf{if}(\textit{true})\ \textbf{do}(\texttt{Rate}(u, X) \leftarrow \$5);$
$\textbf{during}(2024/2025)$
  $\textbf{if}(100 < \texttt{Use}(u, X) < 200)\ \textbf{do}(\texttt{Rate}(u, X) \leftarrow \$4);$
$\textbf{during}(2024/2025)$
  $\textbf{if}(200 < \texttt{Use}(u, X))\ \textbf{do}(\texttt{Rate}(u, X) \leftarrow \$3);$

The pricing agreement p_X states that, for a given set of line items, usage of X by u in 2024 and 2025 will be billed at \$5 up to a usage of 100 GB, usage above 100 and below 200 GB will be billed at \$4, and any usage above 200 GB at \$3.

All terms in an agreement whose `Condition` evaluates to *true* have their `Effect` fields executed, not just the first one. Consequently, the `Condition` of *true* in the first term does not make the other two terms unreachable. The assignments $\texttt{Rate}(u, X) \leftarrow \$4$ and $\texttt{Rate}(u, X) \leftarrow \$3$ in the `Effect` fields of the second and third terms can overwrite the assignment from the first term.

In practice, line items are more complex and can include quantities for usage codes and other features. For this paper, we assume that some pre-processing exists that simplifies line items to ensure that each product code can be assigned a price and that the usage period of the line item is fully included in the period of our pricing agreements.

Since the focus of the paper is on the validation of pricing agreements, we omit further details of the semantics.

A. Analysis of Pricing Agreements

To analyze pricing agreements, we develop a static analysis tool. *Queries* are the main component of our procedure for static analysis. A query is a program that takes a pricing agreement as input and returns a Boolean result. For the sake of brevity, since the focus of the paper is on the task of triaging the results of validation queries, we treat the analysis tool as an opaque box. The analysis tool takes a list of pricing agreements and a validation query as input, runs the query on the pricing agreements, and returns a possibly empty list of query results. The query results consist of the pricing agreements that violate the query and also a human-readable message that describes the issue.

Validation queries can encode a variety of properties. For instance, they can encode data sanitizing requirements such as the requirement that every date range's start comes before its end or the requirement that percentage discounts fall within the range $[0, 100]$. A single query can examine relationships between multiple terms: for example, a query can check that tiered discount terms in a pricing agreement are non-overlapping and do not have gaps. Here is a query that we can run on p_X, the tiered discount agreement for service X:

$$\forall u, \texttt{Use}, \texttt{Use}', \texttt{Rate}, \texttt{Rate}' :$$
$$\texttt{Use}(u, X) < \texttt{Use}'(u, X) \Rightarrow \texttt{Rate}(u, X) \geq \texttt{Rate}'(u, X)$$

This query checks that billing rates are monotonic. The query will produce one validation error that shows us that $\texttt{Rate}(u, X)$ is \$5 and thus non-monotonic for $\texttt{Use}(u, X) = 200$. The agreement will need to be amended to include 200 in the second or third discount term. Designing queries requires a certain amount of technical expertise, as it can require one to reason about expressions with multiple nested quantifiers.

Again, we omit the specifics of the query language to keep the focus on the protocol for running queries and triaging results.

III. Privacy Requirements for Validation Protocol

Before we introduce Parrot, we describe our current protocol for pricing agreement validation. In the first half of this

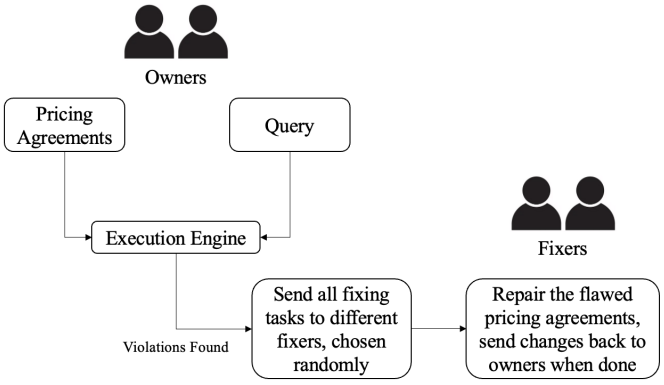

Fig. 1: Our existing protocol for pricing agreement validation. The owners execute validation queries on the pricing agreements on their own without relying on a central actor.

section, we explain the details of the existing protocol, the different actors involved in the protocol, and the privacy requirements that we want to uphold. In the second half of the section, we discuss how we want to extend the existing protocol and the additional privacy requirements that come with the changes.

A. Existing Distributed Validation Protocol

Before we implemented Parrot, we employed a protocol based on distributed validation. A flowchart for the existing protocol appears in Figure 1. The existing protocol involves two categories of actors: the *owners* and the *fixers*. Each pricing agreement has a unique owner who is responsible for the quality of the agreement and for communication with the customer about errors in the agreement if the need arises. Individual owners have full access to the agreements assigned to them. They also have the ability to run the validation queries from Section II on the agreements that they own and then to assign a random fixer to repair any violations that the queries identify.

An individual fixer has access to a set of agreements. A fixer is not tied to a specific owner: fixers can receive assignments from multiple different owners. The fixer triages the findings from the owners' queries and decides which validation errors need to be addressed. Importantly, fixers can mark validation errors as spurious rather than repairing them. The fixer suggests changes for invalid pricing agreements and sends the suggestions back to the pricing agreements' respective owners for approval.

Each fixer has the technical expertise necessary to modify agreements represented as programs as described in Section II. Additionally, fixers can re-run queries on individual pricing agreements to ensure that the proposed changes fix the validation errors identified by the queries.

The owner does not need any technical expertise but has the business background necessary to understand whether a change to an agreement is acceptable. In theory, a fixer can also be an owner, but both the existing protocol and Parrot would be unnecessary if the individuals with full permissions also had the technical expertise necessary to repair pricing agreements. Furthermore, in practice, the existing protocol requires more owners than fixers. Consequently, it is more cost-effective to have individual fixers collaborate with multiple owners than to have a distinct fixer for every owner.

In keeping with our goals from Section I, we do not want to grant unnecessary permissions to actors involved in our protocol for pricing agreement validation. In the existing protocol, the fixers bear the responsibility of repairing errors in pricing agreements, but they do not need to understand an agreement as a whole, and they never interact with customers. This leads us to our first privacy requirement:

> **Requirement 1**: Only the owners are able to view information about customers' identities.

The existing protocol has protective measures in place to limit the information available to the fixers. When a pricing agreement violates a query, the fixer chosen to repair the agreement receives only a *pseudonymized* version of the pricing agreement from the owner. Each value that is not necessary for understanding and repairing validation errors is replaced by a pseudonym. More specifically, if a query does not include any references to a specific field, occurrences of that field will be pseudonymized in the versions of the pricing agreements that the fixers see. We can hide the fields that a query never uses because those fields are irrelevant for the fixers' repairs. Names, account IDs, and regions are always pseudonymized as well, even if the query depends on those fields. The pseudonymization process [4], [3], [6] consistently uses the same pseudonym for different copies of the same value across all agreements. This ensures that the fixers can still understand the relationships between agreement terms without having access to the real values.

Requirement 1 ensures that the fixer cannot view unnecessary data from a specific agreement, but we also need to ensure that a fixer cannot learn about changes to a specific agreement. A fixer could correlate changes to an agreement with other data sources, such as public announcements about companies expanding their usage of our services, to identify a specific customer. This leads us to our second privacy requirement:

> **Requirement 2**: Only the owners can track changes that happen to individual pricing agreements over time.

A fixer has access only to a snapshot of an agreement at the time when the owner ran the validation, and a different fixer may be assigned when the agreement's owner validates the same agreement again. In addition, our protocol uses a fresh seed during pseudonymization every time an agreement is passed to a fixer to ensure that the fixer does not accumulate extra information about the same agreement across multiple assignments of it. The use of a fresh seed ensures that the fixer cannot observe changes to a specific agreement over

time. If an individual fixer ever sees the same agreement twice during the repair process, pseudonymization prevents the fixer from learning that the snapshots are two copies of the same agreement. The fixer can see that the two versions of the agreement have the same overall structure, but, from the fixer's perspective, there is still a possibility that the two agreements are distinct and merely follow the same pattern.

The existing protocol upholds Requirements 1 and 2, but it has two main limitations. The first is that there is no central actor who can design and run new validation queries. The existing protocol relies on individual owners to run the same set of validation queries repeatedly on their respective pricing agreements. This is analogous to relying on individual software developers to run tests on their code instead of using a CI/CD system. The second limitation is that there is no mechanism for collecting business metrics about the performance of individual validation queries. Since the owners do not coordinate with each other, they cannot learn the collective impact of the protocol.

B. New Centralized Validation Protocol

To ensure that all pricing agreements undergo the same validations consistently, we add a centralized scanning service that periodically scans all agreements with a common set of queries. For the centralized service, we need to introduce a new actor to the protocol: the *auditor*. The auditor designs and runs validation queries on the entire set of pricing agreements so that individual owners do not need to bear the responsibility of running validation queries on their respective agreements.

When we introduce a central auditor, we need to take new forms of information leakage into consideration. An individual with bulk access to large numbers of pricing agreements can learn about general trends within the set of agreements, and we do not want to expose that information to actors who do not have full permissions. This leads us to our next privacy requirement:

> **Requirement 3**: No actors other than the owners can learn the maximum, minimum, average, or median for any numerical fields stored in pricing agreements.

We enforce Requirement 3 because we do not want fixers or the auditor to learn about the business metrics of AWS as a whole. Aggregate information about customers' pricing agreements qualifies as non-public business information for AWS itself, even if the underlying individual pricing agreements are anonymized. For instance, a fixer who sees that the number of customers with high-volume discounts is increasing over time could infer that AWS is growing. The fixer could make investment decisions based on that information, so the SEC's trading regulations come into effect. Ordinarily, Amazon employees can purchase Amazon stock, but employees with access to non-public information about the business trends of Amazon need to observe limitations on trading Amazon stock, just as they need to observe trading limitations for businesses other than Amazon when they possess non-public information about them [22]. Limiting sensitive information about the business trends of AWS to the individuals who need to know it reduces the need to restrict AWS employees' trading activity.

Requirement 3 does not apply to the existing protocol since no actors in it have bulk access to pricing agreements. The only individuals who make queries about agreements are the owners, so there is no risk of exposing aggregate information to the fixers. On the other hand, in our new centralized protocol, we need to include safeguards against leakage of aggregate information since the auditor is not an owner.

Because of requirements 1, 2, and 3, we cannot give the auditor direct access to pricing agreements. The auditor can only run queries, and the findings of those queries need to be passed to the fixers instead of the auditor. If we allowed the auditor to view the full results of queries, the queries would effectively serve as a search tool and would grant implicit access to arbitrary contents of the agreements. For the opposite extreme, we could satisfy all of our privacy requirements for the auditor's side trivially by showing nothing about the results of a query to the auditor. The trivial solution addresses the first of the two limitations of the existing protocol that we discuss in Section III-A, but it leaves the second limitation unaddressed: there is still no way for any actors to learn the business impact of the protocol. This leads us to our final requirement for our new protocol:

> **Requirement 4**: The auditor can see whether a query identifies any violations and whether the fixers have repaired the violations.

Requirements 1 and 2 are comparatively straightforward to satisfy when we introduce the auditor: we can simply reuse the mechanisms for pseudonymization from the existing protocol. Requirement 3 requires a different kind of protection. Even if the auditor does not have access to agreements or pseudonymized versions of agreements, like the owners or the fixers, the auditor may still learn information about the set of pricing agreements from the number of violations that a query has, or the fact that a query has violations at all, if the protocol does not have proper safeguards in place. Without seeing any agreements directly, the auditor could learn broader trends about average discounts or the total amount of awarded credits by crafting queries accordingly. We want to design our protocol so that no actors other than the owners can learn aggregate information about the entire set of agreements. Since aggregate information about pricing agreements can reveal business trends, it qualifies as sensitive information. Consequently, as is the case for the agreements themselves, access to aggregate information needs to be limited.

To uphold Requirement 4, we need to balance privacy and productivity: the auditor should have enough information to monitor queries effectively but also should not learn anything about the actual agreements. To ensure the quality and accuracy of the queries, the auditor needs to know whether

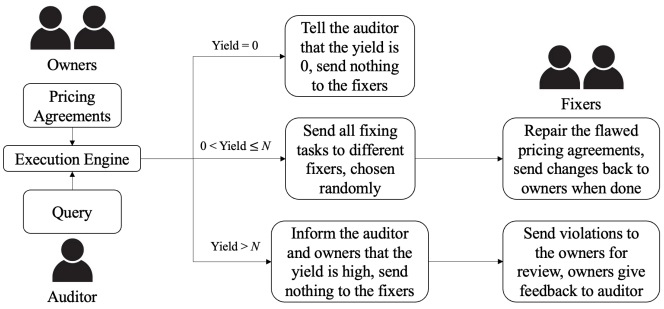

Fig. 2: The new protocol Parrot. The auditor executes queries on the entire set of pricing agreements on the owners' behalf. The fixers receive pricing agreements to repair only if the query's yield does not exceed the upper limit N.

any fixer takes action on the query results or flags them as spurious. Our safeguards for the other three requirements cannot interfere with the auditor's monitoring.

Now that we have defined our privacy requirements and the actor types of owner, fixer, and auditor, we can introduce our new protocol. We will define Parrot in Section IV and show that it satisfies our requirements in Section V.

IV. Parrot Protocol

A flowchart for Parrot appears in Figure 2. For Parrot, we have only one auditor, but there are multiple owners and multiple fixers. There are three main phases in the execution of Parrot: the auditing phase, the repair phase, and the approval phase. We will examine each of the three phases in detail.

A. Auditing Phase

The execution of Parrot starts when the auditor submits a query. Let $\mathcal{P}$ be the set of all pricing agreements. The protocol runs the query on every entry in $\mathcal{P}$ on the auditor's behalf. We do not allow the auditor to select a more specific subset of $\mathcal{P}$ to examine, except by refining the conditions in the query.

The next step that Parrot takes depends on the *yield* of the auditor's query. The yield of a query is the number of pricing agreements that violate the condition that it checks. We define a constant N that is the maximum yield that we allow for a query. N cannot be greater than the total number of fixers.

If the yield is zero, the auditor learns that the yield is zero, and the protocol ends because there is nothing that needs to be repaired. If the yield is nonzero but not greater than N, Parrot proceeds with its normal execution. Let $\mathcal{F}$ be the set of all fixers, and let Y be the yield of the query. The protocol chooses Y random fixers from $\mathcal{F}$ to repair the violations. Each selected fixer receives one pricing agreement to repair. At this point, the auditor learns that the query's yield was within the allowable range but does not receive a specific number. The protocol proceeds to the second phase: the repair phase.

If the yield is greater than N, then Parrot sends the violations to the owners to review. The protocol terminates prematurely, and the owners bear the responsibility of giving a summary of the query's outcome to the auditor. The auditor learns that the query's yield is greater than N but does not learn anything more specific about the yield. Importantly, all of the owners coordinate to give feedback to the auditor, even the ones who do not own any pricing agreements that violate the auditor's query. If we required only the owners who own pricing agreements that violate the query to provide feedback, the auditor could gain extra information about the query's results by seeing which owners provide the feedback.

There are two main reasons why we send the repairs to the owners in the event that a query's yield is high. First, showing more than N pricing agreements to the fixers would be a problematic source of information leakage. We would not want an individual fixer to see dozens or hundreds of pricing agreements all at once, even in an anonymized form. The fixer could obtain an accurate estimate of the average value of a numerical field within the pricing agreements from a sample that large. Second, if a query's yield is high, the auditor's query probably contains a mistake. The auditor may have made an error in writing the query, or the business requirements for pricing agreements may not align exactly with the auditor's own understanding of the meaning of validity. To address the problem, the owners need to give feedback to the auditor directly. The auditor may need to refine the query to reduce the number of pricing agreements that violate it.

Furthermore, when $Y > N$, showing the results to the fixers would be redundant because the owners need to intervene. We could give N violations to the fixers and $Y - N$ to the owners, but that would still require the owners to take action.

B. Repair Phase

The repair phase happens only when the auditor's query has a nonzero yield that is not greater than N. During the repair phase, the fixers see anonymized versions of the pricing agreements that violate the auditor's query. We use the same techniques for pseudonymization that the existing distributed protocol for validation uses. The fixers cannot collaborate with each other during the repair phase: they all operate independently. To repair a pricing agreement, a fixer can make arbitrary modifications to parts of the agreement that are not pseudonymized. Alternatively, a fixer can choose to mark a violation as spurious rather than repairing it. A *spurious violation* is a pricing agreement that violates a query but contains no genuine problem corresponding to the condition that the query checks. A spurious violation is not a false positive in the ordinary sense: the pricing agreement really does violate the condition specified by the query. Instead, a spurious violation is a sign that a query is not sufficiently precise in specifying a problem that can exist within pricing agreements. Spurious violations can happen if the auditor made a mistake in writing the query or if the query fails to take a special case into account for certain pricing agreements. For instance, pricing agreements that are no longer active do not need to be repaired, even if they contain flaws.

When the fixers have addressed every violation, either by repairing them or by marking them as spurious, Parrot begins

the approval phase: it sends the fixers' changes and markings to the pricing agreements' respective owners for approval.

C. Approval Phase

When the fixers finish, the owners can approve the fixers' actions during the repair phase or request further adjustments from the same fixers for the same pricing agreements. If an owner rejects a fixer's actions for a pricing agreement, the repair phase effectively restarts for that fixer with the same pricing agreement.

The protocol does not end until the owners approve every fixer's changes. Parrot deploys the newly modified pricing agreements within $\mathcal{P}$ when the owners approve all the changes. At the end, after the owners approve all of the fixers' changes, the auditor receives a limited summary of the protocol's results. The summary includes the total number of violations and the number of violations that the fixers marked as spurious, so the auditor learns the yield of the query at the end. We can disclose the yield of the query to the auditor safely at the end of the approval phase because the yield cannot be greater than N. Since we have an upper limit on the yield of queries that can pass the auditing phase, the auditor cannot learn arbitrarily precise yield information at the end of the protocol.

The auditor never learns the identities of the fixers who repaired the pricing agreements. Also, throughout the entire protocol, the auditor never sees any of the agreements directly.

D. Query Restrictions

Along with the main structure of the protocol itself, there are some additional restrictions that we enforce for queries within Parrot to prevent the leakage of sensitive information. We limit both the content of auditors' queries and their frequency.

1) Query Content Restrictions: We impose strict limitations on the ways that the auditor's queries can use account names, IDs, or regions. Queries can ask whether different fields in a pricing agreement for account names, account IDs, or regions are equal to each other, but they cannot use names, IDs, or regions in any other way. In particular, a query cannot check whether a name, ID, or region is equal to a specific constant. The validity conditions for pricing agreements are not tailored to specific customers, so there is no need to examine names, IDs, and regions except by comparing them to each other. Names, IDs, and regions are sensitive information that neither the auditor nor the fixers should be able to view directly, but we still want the auditor to be able to check certain validity conditions involving names, IDs, and regions. For example, we do not want pricing agreements to give a discount for a specific product to the same account twice over. With only the ability to compare IDs to each other, a query from the auditor can check whether two discount assignments have the same customer ID. Also, a fixer can repair pricing agreements with this specific problem without ever viewing the IDs directly by removing one of the duplicate discount terms.

Additionally, queries can ask only about the properties of individual pricing agreements in isolation. Queries cannot check aggregate properties of the entire collection of pricing agreements. We forbid queries about aggregate properties of $\mathcal{P}$ because they could leak a large amount of sensitive information. With only a single multi-agreement query, an auditor could ask directly for the average of a numerical field. For instance, a multi-agreement query could define a variable d that is the sum of all discount values across all agreements and then ask for all pricing agreements with discount values greater than d/P, where P is the total number of pricing agreements. Furthermore, because we allow equality checks between name, ID, and region fields, we forbid multi-agreement queries to prevent the auditor from comparing name, ID, and region fields between different pricing agreements.

2) Query Rate Limit: The auditor can make only one query per day. Imposing a rate limit reduces the risk of information leakage. If a piece of sensitive information requires Q queries to expose, the series of queries takes Q days to execute. Increasing the time needed to execute a series of queries decreases the likelihood of sensitive information being exposed. The rate of change of the contents of $\mathcal{P}$ is not fast enough to provide effective protection on its own, but the rate limit prevents the auditor from making many queries in rapid succession that jointly reveal an aggregate property of $\mathcal{P}$.

The rate limit of one query per day does not impede the productivity of the actors in the protocol. In practice, the expected response time for repairs from the fixers is approximately one day, so the throughput of the protocol would not be significantly higher without an explicit rate limit.

V. Privacy Guarantees

Now we will examine the privacy guarantees that Parrot provides relative to its size parameters. There are three main size parameters for Parrot: the total number of pricing agreements (P), the maximum yield allowed for queries (N), and the total number of fixers (F).

Throughout our discussion of Parrot's privacy guarantees, we use discount rates as a proxy for all sensitive information within pricing agreements. Most pieces of information within an agreement are numbers. Additionally, there are fixed upper and lower bounds for discount rates that are publicly known.

A. Assumptions

The purpose of Parrot is to prevent the exposure of sensitive information. We work in the honest-but-curious security model. The actors in the protocol are not aiming to perform denial-of-service attacks, to rewrite pricing agreements maliciously, or to obstruct the execution of the protocol in any other manner. Also, most fixers are not consciously attempting to learn sensitive information, but we consider the possibility that an individual fixer acting in isolation attempts to estimate statistical properties of the set of pricing agreements.

We assume that all owners are trustworthy. An owner who intends to leak sensitive information would not need to use Parrot: the owner could simply give the information directly to the auditor or fixers.

There is at least one pricing agreement for every account ID, and most of the pricing agreements contain no mistakes.

Parrot is safe to execute only if $\mathcal{P}$ is a large set. $\mathcal{P}$ needs to be large enough that no individual fixer can expect to view the entire set over the course of multiple executions of Parrot.

Although the collection of pricing agreements is large, its contents do not change at a rapid rate. Most agreements remain in effect for multiple years once signed. For most purposes, the turnover rate of the set of agreements is not a significant safeguard against information leakage on its own.

The auditor knows the general structure that pricing agreements follow. Also, the auditor has access to a number of fake example agreements for testing queries. The queries that an auditor submits are not sensitive information. When the auditor makes a query, all of the owners can see it, even the owners who do not own agreements that violate the query. Only the agreements and their statistical properties need to be hidden. Additionally, at any time, the owners can intervene to stop the auditor from making queries if the results of the queries begin to look problematic in terms of information leakage.

B. Expected Size Parameters

For this paper, we assume that $P > 5000$, N is 10, and F is also 10. In practice, the size parameters may be larger, but we want the privacy guarantees of Parrot to be independent of the specifics of our own situation. We allow N and F to be equal, but, as we state in Section IV, N cannot be greater than the total number of fixers. Having a larger number of fixers makes information leakage easier to prevent, but the number of individuals who have the technical expertise necessary to repair pricing agreements is limited. The fact that the pool of fixers is small requires us to take types of information leakage into consideration that we would not need to consider if we had hundreds of fixers. We need to prevent individual fixers from accumulating significant information about the set of agreements from the flawed agreements that they view across multiple executions of Parrot.

C. Requirements 1 and 2: Personal Information

Parrot upholds Requirements 1 and 2 because of our procedures for pseudonymization and the fact that the auditor never views any pricing agreements. The versions of the pricing agreements that the fixers see are anonymized. Fixers cannot use information from an external source to identify specific customers' agreements because they can view only the fields in an agreement that are relevant for the requested repair.

Fixers never have direct access to names, IDs, or regions within pricing agreements. Strictly speaking, IDs do not count as personal information since they are arbitrary labels with no external meaning. We choose to hide IDs anyway because a fixer could learn that two pricing agreements belong to the same individual by seeing the same ID in both.

D. Requirement 3: Aggregate Information

To establish that Parrot upholds Requirement 3, we need to perform a more thorough analysis of the protective measures that the protocol includes. For protection of personal information, we only need to consider individual queries in isolation, but for aggregate information we need to consider the effects of combinations of queries. Neither the auditor nor the fixers can learn the maximum, minimum, average, or median of a numerical field from a single query, but combinations of queries can leak more information than individual queries can. We consider three kinds of information leakage that can come from groups of queries: exhaustive observation, information from query yields, and sampling.

1) Exhaustive Observation: An individual fixer cannot observe the entire set of pricing agreements in a reasonable amount of time. Because $N \leq F$, an individual fixer can view at most one pricing agreement per day. Even if a fixer never sees any repeats, it would take more than 13 years to view every single pricing agreement if $P > 5000$. Pricing agreements last for only a few years each: for instance, EC2 savings plans can have a duration of one year or three years [13]. Consequently, the turnover rate of $\mathcal{P}$ makes an exhaustive survey of the set impossible.

2) Information from Query Yields: Without sufficient protections in place, the auditor can learn sensitive information from the yield of a query without ever seeing any agreements. In fact, the auditor can learn sensitive information from only a Boolean indication of whether a query's yield is nonzero. Discount values all lie in the range $[0, 100]$, and the auditor can take advantage of that fact. To find the maximum discount rate, the auditor can start by making a query for discounts that are at least 100 percent. If there are none, the auditor moves the margin downward by one percentage point to ask for discounts that are at least 99 percent. If the yield is still zero, the auditor can continue to move the margin downward with successive queries until the yield is nonzero. When a query asking for discounts that are at least x percent has a nonzero yield, the auditor learns that x is the maximum discount rate.

The gradual descent search is slow and can require up to 100 queries in the worst case, but faster alternatives exist. From only a Boolean result, the auditor can learn the maximum discount rate for a service up to the accuracy of a single percentage point using only seven queries. More specifically, the auditor can perform a binary search to find the maximum discount rate. For the binary search, the auditor starts by asking whether $\mathcal{P}$ contains any agreements with discount rates greater than 50 percent. If the query's yield is nonzero, the auditor moves the discount margin upward for the next query by 25 percentage points. Otherwise, the auditor moves the margin downward by the same amount. The auditor can repeat the process six more times, halving the movement distance each time, to complete the binary search.

Safeguards exist within Parrot to prevent the auditor from learning the maximum discount rate from the yields of different queries. The main safeguard is the rate limit of one query per day that we impose. Since the auditor can make only one query per day, the owners have a window of multiple days to intervene to stop the auditor from making problematic queries. Additionally, the fact that Parrot will notify the owners if the auditor ever makes a query with a yield higher than N serves as a deterrent against the auditor attempting to gain sensitive

information from queries.

3) Sampling: A fixer who observes the contents of a large number of pricing agreements could estimate the average value for discount rates or some other numerical field. As the number of samples increases, the fixer can obtain an increasingly accurate estimate of the average. The first safeguard against sampling-based estimation of averages is the fact that the fixers cannot control the queries that the auditor makes. The fixers have no guarantee of receiving a representative sample of the full collection of pricing agreements during the ordinary execution of Parrot. Additionally, there is no guarantee that the underlying values for a specific field will follow a normal distribution or some other predictable pattern. Nevertheless, we want to maintain our privacy guarantees even when the auditor's queries provide a representative sample of the pricing agreements and the values follow a predictable distribution.

In the worst-case scenario, the results of the auditor's queries act as a uniform random sampler of pricing agreements, the queries always have a yield of N, and the queries always require the fixers to see the same numerical field. If the underlying distribution of the values for the field is a normal distribution, then an individual fixer can obtain a relatively accurate estimate of the average with 20 samples [5]. If we assume that discount rates follow a normal distribution and have a standard deviation of 5 percentage points, then, with 20 samples, a fixer can attain 95 percent confidence that the average discount is within 2.19 percentage points of the sample average [11], [26]. With a smaller number of samples, the fixer can obtain an estimate with a wider margin of error.

As we explain in Section IV, when we assign agreements to fixers for a query, every agreement goes to a different fixer. Since we also impose a rate limit of one query per day, an individual fixer can see at most one sample per day. Consequently, the auditor would need to make at least 20 queries about the same field for an individual fixer to receive 20 samples. As in the case where we prevent information leakage on the auditor's side based on the yield of a query, the owners can intervene to stop the auditor from making problematic queries. The queries for revealing the maximum discount rate take seven days to execute, but the owners have a larger window of twenty days to intervene in this situation. The owners can intervene before the fixer even accumulates half the number of samples needed to make an accurate estimate.

The fact that we assign all pricing agreements for a single query to different fixers is critical for our privacy guarantees. Since $N \leq F$, we can enforce the rule that no fixer sees more than one pricing agreement per day. Alternative methods for assigning pricing agreements to fixers do not enforce the same hard limit. If we assign all violations for an individual query to the same fixer, then an individual fixer can view up to N pricing agreements in a single day, and the risk of information leakage becomes significantly higher. Suppose that the auditor makes multiple consecutive queries about the same field, namely the discount rate, and suppose that all of the queries have a yield of N. Let Francis be an individual fixer who wants to compute the average discount rate. If two queries' results go to Francis, then Francis can obtain an accurate estimate of the average discount rate from the 20 samples [5]. The probability of a query's results going to Francis is $1/F$, so, with our expected size parameters, the probability of Francis receiving both queries' results for two days is 0.01, which is comparatively unlikely but still feasible. More generally, the probability of Francis receiving at least m queries' results over k days is $F^{-k} \cdot \sum_{i=m}^{k} \binom{k}{i} (F-1)^{k-i}$. The probability of Francis receiving at least two queries' results is approximately 0.028 for three days, 0.052 for four days, 0.081 for five days, 0.114 for six days, and 0.150 for seven days. There is a significant chance of Francis receiving 20 samples in less time than is necessary for the auditor to compute the maximum discount rate with a binary search, so the risk of information leakage is too high relative to the number of days when we assign all violations to the same fixer.

A different option for assignment of pricing agreement to fixers is independent distribution. With independent distribution, we choose a fixer for each pricing agreement at random separately, ignoring the assignments of other pricing agreements. Under independent distribution, it is possible for one fixer to receive multiple pricing agreements. The expected number of samples that an individual fixer can receive per day is still at most one, but the maximum is N.

Independent distribution makes the risk of rapid information leakage substantially lower than it would be if we always assigned all pricing agreements to the same fixer, but the risk is still present nevertheless. Once again, let Francis be an individual fixer who wants to compute the average discount rate. For k queries, there are kN samples in total, and there are F^{kN} ways to distribute them among the fixers. The likelihood of Francis receiving m or more samples across k queries is $F^{-kN} \cdot \sum_{i=m}^{kN} \binom{kN}{i} (F-1)^{kN-i}$. With our expected size parameters, the worst-case scenario for independent distribution is to have all 20 samples for two queries sent to Francis. The probability of the worst-case scenario is 10^{-20}, which is very low, but outcomes that are close to the worst-case scenario have higher probabilities. With independent distribution, the probability of Francis receiving at least 20 samples is approximately 1.106×10^{-13} for three queries, 1.872×10^{-10} for four queries, 2.369×10^{-8} for five queries, 7.823×10^{-7} for six queries, and 1.126×10^{-5} for seven queries. The probability of Francis receiving 20 samples in seven queries or fewer is low but non-trivial when we distribute pricing agreements independently. Our real method for pricing agreement distribution imposes a hard limit of one pricing agreement per day for fixers, so we eliminate the possibility of information leakage in this situation.

There are two trade-offs for our decision to forbid duplicate assignments. The first is that, when we assign every violation to a different fixer, Francis is guaranteed to receive 20 samples from 20 queries if every query has a yield of N. Under the other two approaches, there is a chance that Francis will not receive 20 samples from 20 queries. The probability of Francis receiving 20 samples from 20 queries when we assign all results from a query to the same fixer

is $10^{-20} \cdot \sum_{i=2}^{20} \binom{20}{i}(9^{20-i})$, which is approximately 0.608. Under independent distribution, the probability is $10^{-200} \cdot \sum_{i=20}^{200} \binom{200}{i}(9^{200-i})$, which is approximately 0.534. Under the two alternate schemes, there is no guarantee that Francis will receive 20 samples from 20 queries, but the likelihood of that outcome is still greater than half for each. Consequently, for Parrot, we choose to eliminate the risk of information leakage for small numbers of queries rather than reducing the risk for large numbers of queries.

The second trade-off for our assignment scheme is the fact that the number of fixers who learn that the yield of a query is nonzero is higher than it would be otherwise. If we allow no fixer to receive more than one pricing agreement, the probability of Francis receiving no pricing agreements is $\frac{F-Y}{F}$. For the situation where $F = Y = N = 10$, the probability is zero: every fixer learns that the query has at least one violation. Alternatively, if we assign all agreements to a single fixer, the likelihood of Francis receiving no agreements to repair stays constant at $\frac{F-1}{F} = \frac{9}{10}$ and does not depend on Y.

Under independent distribution, the probability of an individual fixer learning that the yield is nonzero converges to 1 as the yield increases but never reaches 1. If $Y \leq N$, then the likelihood of Francis receiving no pricing agreements to repair is $(\frac{F-1}{F})^Y$ if we distribute the pricing agreements independently. This value approaches 0 at an exponential rate as Y increases. If $F = 10$ and $Y = N = 10$, then the probability is $(\frac{9}{10})^{10} \approx 0.349$. In this event, it is more likely than not that Francis will receive at least one pricing agreement, but it is still possible that Francis will receive nothing. Moreover, the inequality $\frac{F-Y}{F} \leq (\frac{F-1}{F})^Y$ always holds when $Y \leq N \leq F$.

We do not consider it a problem that our method for assigning pricing agreements has a higher probability of individual fixers learning that the yield of a query is nonzero. Our safeguards against information leakage for the auditor double as safeguards for the information leakage that comes from fixers learning that a query's yield is nonzero. Furthermore, if the fixers receive the same business metrics that the auditor does, they will learn whether the yield is nonzero anyway.

E. Requirement 4: Productivity Guarantees

All that remains to be confirmed is that Parrot upholds Requirement 4. Within the protocol, the auditor learns high-level information about the results of a query that can be useful for business metrics. In the ordinary case of the protocol's execution, the auditor learns the query's yield, the number of genuine violations, and the number of spurious violations. For the two abnormal cases, the auditor can learn whether the query's yield is zero or greater than N.

The information that we provide about the results allows the auditor to learn the business impact of specific queries. If some queries consistently have nonzero yields and other queries consistently find no violations at all, the auditor can direct the owners to prioritize the issues identified by the high-yield queries when creating new pricing agreements in the future. Also, if some queries have spurious violations, the auditor can refine the queries for later use.

VI. Related Work

A. Similar Problems

a) Static and Dynamic Analysis: Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools, like Google's Tricorder [21] and Error-Prone [20], Amazon's CodeGuru [16], and Meta's Infer [12], need to address privacy issues similar to the ones that Parrot addresses. Ideally, when a SAST/DAST tool detects a security vulnerability, the tool should limit the visibility of the vulnerability to as few people as possible. However, the tool still needs to track basic business metrics to ensure that its security checks perform as expected. Another tension between privacy and effectiveness for SAST/DAST tools arises when the tools scan code from unreleased products. The tool should scan all code inside a company, including code for unreleased products, but the team that operates the SAST/DAST tool should not learn about the unreleased products from repository names, file names, or other auxiliary information in the tool's findings.

The use of large language models for detection and repair of security issues makes the problem of privacy even more significant for security vulnerability detection, especially when the LLM belongs to a third party. A careful design of the validation life-cycle is necessary to ensure that the the auditors do not teach sensitive information to the LLM inadvertently.

b) Private Information Retrieval: Private information retrieval is the task of making a query about a public database without revealing the contents of the query to the database owner who provides the results [17]. Private information retrieval is the opposite of the problem that we want to address: in our case, the database is private and the query is public.

c) Bug Bounties: Managers of large-scale code repositories often offer *bug bounties*: payments for third-party programmers who identify bugs in the code [25]. There are standardized procedures for the disclosure of bugs for bug bounties: for instance, Apple requires individuals who claim a bug bounty to provide a guide for exploiting the vulnerability and not to disclose the vulnerability publicly beforehand [19].

A system like bug bounty reporting is not appropriate for pricing agreement validation. The safeguards in bug bounty reporting systems exist to ensure that third-party programmers cannot falsely claim to have discovered a bug and to disincentivize them from exploiting bugs for their own gain [2]. We cannot outsource pricing agreement validation to third-party programmers because the pricing agreements contain sensitive information. Also, in pricing agreement validation, protection against false claims of errors is not a significant problem because the owners need to approve all edits that fixers make to pricing agreements. Additionally, fixers cannot exploit flawed pricing agreements for their own gain because they cannot influence specific customers' usage patterns.

B. Other Methods for Information Leakage Prevention

a) Privacy Metrics: The concepts of differential privacy and k-anonymity exist to quantify guarantees about the privacy that individuals have when their personal information appears in a database. A database is differentially private if the presence or absence of a specific individual's entry does not have a significant effect on the externally observable features of the database [7]. The property of k-anonymity holds for a database or a view of a database if every individual in it is indistinguishable from at least $k - 1$ others [23].

On their own, differential privacy and k-anonymity cannot capture the privacy guarantees that we want to uphold with Parrot. Individual information is not the only type of information that we want to protect: we also want to prevent the leakage of aggregate properties.

b) Query Similarity: The problem of quantifying the similarity of different SQL queries has received some attention in prior research [14], [15]. Some of the examples of information leakage that we cover in this paper involve multiple similar queries with slight variations, but query similarity on its own is not an all-encompassing indicator of whether a query is problematic. Multiple queries can leak sensitive information jointly even if they are not similar to each other, and the leakage can depend on factors other than the definitions of the queries themselves. Suppose that the auditor makes three queries, q_1, q_2, and q_3. The query q_1 looks for pricing agreements that start on March 21, 2024, q_2 looks for invalid pricing agreements whose sign dates come after their start dates, and q_3 looks for invalid pricing agreements whose end dates come before their start dates. Suppose that all three queries have a yield of one, and the same pricing agreement p is the only agreement that matches each query. If the results from all three queries go to the same fixer Francis, then Francis learns more than what the three queries convey on their own. The start date will be visible in the results for all three queries, and q_1 reveals that p is the only agreement that starts on March 21, 2024, so the fixer can learn that p is the pricing agreement matched by all three. Consequently, Francis learns the sign date, start date, and end date of p, even though no query asks about all three fields together. All three queries involve the start date, but the full set of fields used is different for each one. For comparison, the binary-search queries for finding the maximum discount that we discuss in Section V-D2 all involve only one field, namely the discount rate. The binary search queries are more similar to each other than q_1, q_2, and q_3 are, but q_1, q_2, and q_3 still leak information jointly.

c) Security Type Systems: Type systems exist that confirm that programs uphold safety properties, including restrictions on information leakage [18], [24]. Similar language-based information leakage prevention systems have been designed for database query languages [10]. Security type systems are not appropriate for pricing agreement validation. We need to consider information leakage that can occur across multiple queries, but type systems consider only individual programs in isolation. Also, in some cases, it is not possible to quantify the information leakage of a query in a meaningful way without running it. Returning to the example of q_1, q_2, and q_3 from before, the joint information leakage from q_1, q_2, and q_3 is possible only because p exists. Static analysis of the queries themselves cannot take the contents of the set of pricing agreements into consideration.

d) Zero-Knowledge Proofs: A zero-knowledge protocol is a method of communication between two parties, known as the prover and the verifier. The prover possesses a private witness w that satisfies a public predicate Φ. The verifier learns that the prover knows the value of w and that w satisfies Φ, but the prover does not share the value of w with the verifier [8].

Zero-knowledge proofs are not appropriate for pricing agreement validation. The owners do not need to provide any evidence for the auditor or fixers that a query was executed correctly. Also, the fixers need to be able to view fields within pricing agreements when they repair them.

e) Homomorphic Encryption: Homomorphic encryption allows programs to operate on encrypted data without knowing the true values of the underlying data [1]. Homomorphic encryption is inappropriate for pricing agreement validation for the same reason that zero-knowledge proofs are inappropriate: the fixers need to know the real underlying values inside pricing agreements when they repair the agreements.

f) Multi-Party Computation: Cryptographic protocols for secure multi-party computation allow two or more mutually distrusting parties to compute a result jointly without leaking sensitive information during the computation or relying on a trusted central authority [9]. Distributed consensus protocols are multi-party computation protocols for coordination within systems that run continuously [27]. Blockchain systems are the principal application of distributed consensus protocols.

Multi-party computation is not an appropriate model for pricing agreement validation because we have a trusted central authority, namely the group of owners. There are no secrets that the auditor and fixers need to hide from the owners. Also, multi-party computation protocols aim to prevent information leakage during the computation of a result, but, for Parrot, the results of the queries themselves are what we want to hide.

VII. Conclusion

We have introduced Parrot, a protocol for validation and repair of pricing agreements that prevents leakage of both sensitive personal information and statistical properties of the whole set of agreements. While our protocol specifically targets pricing agreements in this paper, we believe that similar protocols can find applications in other domains, such as static application security testing (SAST). In particular, for large enterprises, it may be beneficial to employ a protocol similar to Parrot to manage the finding life-cycle of their SAST solutions so that security engineers do not gain access to unnecessary information about unreleased products from identifiers, such as file or variable names, that could be included in a SAST tool's findings. Moreover, since Parrot is agnostic to the type of analysis or findings, it can be integrated with existing SAST tools without requiring changes to the SAST system itself.

References

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[2] Omer Akgul, Taha Eghtesad, Amit Elazari, Omprakash Gnawali, Jens Grossklags, Michelle L Mazurek, Daniel Votipka, and Aron Laszka. Bug hunters' perspectives on the challenges and benefits of the bug bounty ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2275–2291, 2023.

[3] Mihir Bellare, Phillip Rogaway, and Terence Spies. Addendum to "The FFX mode of operation for format-preserving encryption". Technical report, National Institute of Standards and Technology, 2010.

[4] Mihir Bellare, Phillip Rogaway, and Terence Spies. The FFX mode of operation for format-preserving encryption. Technical report, National Institute of Standards and Technology, 2010.

[5] H.B. Berman. Sampling distributions. https://stattrek.com/sampling/sampling-distribution. Accessed: 2024-09-20.

[6] John Black and Philip Rogaway. Ciphers with arbitrary domains. In *Proceedings RSA-CT*, pages 114–130, 2002.

[7] Cynthia Dwork. Differential privacy. In *International colloquium on automata, languages, and programming*, pages 1–12. Springer, 2006.

[8] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[9] Shafi Goldwasser. Multi party computations: past and present. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Cistributed Computing*, pages 1–6, 1997.

[10] Raju Halder, Matteo Zanioli, and Agostino Cortesi. Information leakage analysis of database query languages. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 813–820, 2014.

[11] Brayton Hall. The normal distribution, confidence intervals, and their deceptive simplicity. https://medium.com/swlh/a-simple-refresher-on-confidence-intervals-1e29a8580697. Accessed: 2024-09-26.

[12] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23, 2018.

[13] Amazon Web Services Inc. Compute savings plans – amazon web services. https://aws.amazon.com/savingsplans/compute-pricing/. Accessed: 2024-09-30.

[14] Leo Köberlein, Dominik Probst, and Richard Lenz. Quantifying semantic query similarity for automated linear sql grading: A graph-based approach. *arXiv preprint arXiv:2403.14441*, 2024.

[15] Gokhan Kul, Duc Thanh Anh Luong, Ting Xie, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. Similarity metrics for sql query clustering. *IEEE Transactions on Knowledge and Data Engineering*, 30(12):2408–2420, 2018.

[16] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. Static analysis for AWS best practices in python code. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[17] Femi Olumofin and Ian Goldberg. Privacy-preserving queries over relational databases. In *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings 10*, pages 75–92. Springer, 2010.

[18] Kevin R O'Neill, Michael R Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 12–pp. IEEE, 2006.

[19] Apple Security Research. Bounty - apple security research. https://security.apple.com/bounty/. Accessed: 2024-09-19.

[20] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.

[21] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 598–608. IEEE Press, 2015.

[22] Kenneth E Scott. Insider trading: rule 10b-5, disclosure and corporate privacy. *The Journal of Legal Studies*, 9(4):801–818, 1980.

[23] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[24] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, 2000.

[25] Thomas Walshe and Andrew Simpson. An empirical study of bug bounty programs. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 35–44. IEEE, 2020.

[26] Valerie Watts. 7.5 calculating the sample size for a confidence interval. https://ecampusontario.pressbooks.pub/introstats/chapter/7-5-calculating-the-sample-size-for-a-confidence-interval/. Accessed: 2024-09-26.

[27] Yang Xiao, Ning Zhang, Jin Li, Wenjing Lou, and Y Thomas Hou. Distributed consensus protocols and algorithms. *Blockchain for Distributed Systems Security*, 25:40, 2019.