Side-channel attacks on query-based data anonymization

Franziska Boenisch
franziska.boenisch@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, Germany

Reinhard Munz
munz@mpi-sws.org
Max Planck Institute for Software Systems
Saarbruecken, Germany

Marcel Tiepelt
marcel.tiepelt@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Simon Hanisch
simon.hanisch@tu-dresden.de
Center for Tactile Internet (CeTI), TU Dresden
Dresden, Germany

Christiane Kuhn
christiane.kuhn@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Paul Francis
francis@mpi-sws.org
Max Planck Institute for Software Systems
Saarbruecken, Germany

ABSTRACT

A longstanding problem in computer privacy is that of data anonymization. One common approach is to present a query interface to analysts, and anonymize on a query-by-query basis. In practice, this approach often uses a standard database back end, and presents the query semantics of the database to the analyst.

This paper presents a class of novel side-channel attacks that work against any query-based anonymization system that uses a standard database back end. The attacks exploit the implicit conditional logic of database runtime optimizations. They manipulate this logic to trigger timing and exception-throwing side-channels based on the contents of the data.

We demonstrate the attacks on the implementation of the CHORUS Differential Privacy system released by Uber as an open source project. We obtain perfect reconstruction of millions of data values even with a Differential Privacy budget smaller than epsilon = 1.0 and no prior knowledge.

The paper also presents the design of a general defense to the runtime-optimization attacks, and a concrete implementation of the defense in the latest version of Diffix. The defense works without modifications to the back end database, and operates by modifying SQL to eliminate the runtime optimization or disable the side-channels.

In addition, two other attacks that exploit specific flaws in Diffix and CHORUS are reported. These have been fixed in the respective implementations.

CCS CONCEPTS

• **Security and privacy → Data anonymization and sanitization**; Pseudonymity, anonymity and untraceability; Privacy-preserving protocols; *Software security engineering*; • **Information systems** → *Query optimization.*

KEYWORDS

side-channels, anonymization, privacy, databases

ACM Reference Format:
Franziska Boenisch, Reinhard Munz, Marcel Tiepelt, Simon Hanisch, Christiane Kuhn, and Paul Francis. 2021. Side-channel attacks on query-based data anonymization. In *Proceedings of ACM CCS 2021 (ACM CCS '21).* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

One of the oldest problems in computer privacy is that of data anonymity: exposing statistical information about data without exposing information about individuals. The first papers were written in the early 1970's [4], and thousands of papers have been published since.

There are two broad approaches to data anonymity, table-based and query-based. In table-based approaches, a data set is modified to produce a new, anonymized data set. The anonymized data set is made available to analysts, who are free to run any queries they wish. The majority of practical data anonymization is table-based, for example national census data releases and medical data releases like HCUP [5].

In query-based approaches, the anonymization mechanisms operate over individual queries. While query-based anonymization offers more analytic flexibility than table-based, it also exposes a larger attack surface. The problem was actively researched in the late 70's, but the challenges appeared insurmountable, leading to Denning in 1981 saying *"It seems unlikely that we will ever be able to efficiently decide at the time of a query whether a requested statistic could lead to the disclosure of a sensitive statistic."* [2]

Interest in query-based anonymization died down until the development of Differential Privacy (DP) in 2006 [3] and its formal guarantees of privacy. In 2009 and 2010, two query-based anonymization tools based on DP, PINQ [17] and Airavat [28] respectively, were released. Both gave the analyst the ability to compose queries as code, allowing for arbitrary queries and effectively giving the analyst a Turing machine as an attack surface. Both were developed primarily as academic tools: the authors acknowledged the possibility of side-channel attacks on the tools, and indeed side-channel attacks on both systems were demonstrated by Haeberlen et al. in

2011 [11]. Their attacks exploited the underlying Turing machine to manipulate the observable behavior of the systems.

In 2017 and 2018, two more query-based anonymization tools were released, Diffix (version Birch) [8] and CHORUS (Uber release) [12, 30] respectively. Unlike PINQ and Airavat, these tools were designed to be used in commercial settings to protect real data. Diffix was built by Aircloak GmbH and sold commercially[1]. CHORUS was adapted by Uber [20, 26] for internal use. As of this writing, Uber states that they no longer use CHORUS. The Uber GitHub repository [30] was archived in December of 2019 and is no longer actively developed. The last code update was in March 2018. Note that a second open-source version of CHORUS was released in 2020 and is still actively maintained [14]. We refer to the Uber implementation as CHORUS-v2, and the currently active implementation as CHORUS-v3. The attacks were demonstrated on CHORUS-v2 only.

Although CHORUS is based on DP while Diffix is not, the two have strong similarities. Both use SQL as the query interface, and both operate as proxies in front of standard SQL databases (Figure 1). Both impose limitations on the SQL, and neither allows arbitrary code, thus offering a smaller attack surface compared to PINQ and Airavat. Indeed none of the Haeberlen et al. attacks work on CHORUS or Diffix.

Nevertheless, in this paper we show that even this smaller attack surface is vulnerable to a variety of powerful side-channel attacks which an adversary can reconstruct any data with 100% accuracy. Specifically, this paper reports on a novel class of side-channel attacks that exploit standard SQL mechanisms in the back end database.

The attacks can in principle work against *any* query-based anonymization system that uses a standard database in the back end. These attacks exploit run-time query optimizations where SQL clauses may or may not be executed depending on the data itself. One class of attacks uses exception reporting as the side-channel, while the other uses timing. The attacks are 100% accurate, and allow perfect reconstruction of millions of database records even, in the case of CHORUS-v2, with a Differential Privacy budget smaller than epsilon = 1.0. Our defense (Section 5) is completely effective and is implemented in Diffix Dogwood.

In addition, we report on several attacks that exploit specific flaws in the Diffix and CHORUS-v2 proxy implementations. While not as general as the above-mentioned database attacks, these attacks serve to illustrate the kind of vulnerabilities that can emerge when building practical anonymization systems. The attacks against the Diffix proxy exploit a design flaw stemming from how the Aircloak implementation reported query progress. One of the attacks used a timing side-channel, while the other used the query status reports themselves as the side-channel. The attack against the CHORUS-v2 proxy exploits a flaw that was accidentally discovered while testing the system. The flaw allows the attacker to literally read out any column value. All of the proxy-targeted attacks allowed perfect reconstruction of millions of records, and all have been fixed in the latest respective versions.

The main contributions of this paper are:

- A new class of side-channel attacks on query-based anonymization systems that exploit SQL run-time query optimizations, use timing and exception reporting as the side-channels, and can work against *any* system that uses a standard database back end.
- Demonstration of the attacks on CHORUS-v2 running with PostgreSQL as the back end database.
- Novel defenses against the side-channel attacks that work without modifications to the back end databases.
- A description of how the defenses were implemented in Diffix Dogwoood.
- Novel side-channel attacks on Diffix Dogwood that exploit a (now fixed) design flaw and use timing and query progress status reporting as the side-channels.
- A side-channel attack that exploits a flaw in CHORUS-v2 (which is reported not to exist in CHORUS-v3).

Overall, this paper shows that query-based anonymization systems, including those that place restrictions on SQL, nevertheless still leave a substantial attack surface that can be exploited with side-channels unless extreme care is taken. While this paper shows how to defend against a new class of side-channel attacks, it seems likely that there are still more attacks to be discovered. Organizations deploying query-based anonymization systems must be cognizant of this, and control access to the system interface accordingly.

Ethical Vulnerability Notification. The database-based attacks of Section 3 were discovered by Aircloak and MPI-SWS as part of the Diffix design process, and was demonstrated to work against Diffix Cedar in early 2020. The defenses described in this paper were implemented before the attacks were made known publicly. These same attacks, as well as the attack in Section 6.3, were demonstrated to work against CHORUS-v2 in early 2021. Both Uber and the developers of CHORUS were promptly notified. Since at the time of reporting there were no production uses of either CHORUS-v2 or CHORUS-v3, neither party requested that publication be delayed.

The Diffix proxy attack of Section 6.1 was demonstrated to work in late 2020 as part of a bounty program operated by MPI-SWS. The Diffix proxy attack of Section 6.2 was discovered in early 2021 by MPI-SWS and Aircloak. Publication of the attacks were delayed until Aircloak added a defense.

Outline. Section 2 describes CHORUS-v2 and Diffix Dogwood, the two query-based anonymization systems targeted in this paper. Section 3 describes the database attacks in general terms. Section 4 reports on the effectiveness of the attacks against CHORUS-v2, while section 5 describes how Diffix Dogwood defends against the attacks. Section 6 describes attacks against the Diffix and CHORUS-v2 proxy implementations themselves. Section 7 describes related work, and Section 8 gives some additional observations and conclusions.

2 OVERVIEW OF TARGET SYSTEMS

Figure 1 shows the basic operation of Diffix and our port of CHORUS-v2. Both systems are deployed as a proxy in front of a standard backend database. At a high level, both systems offer similar interfaces to an analyst. The analyst submits an SQL query. The system

[1]One of the authors, Paul Francis, is a co-founder of Aircloak. Aircloak and MPI-SWS are research partners working on the development of Diffix.

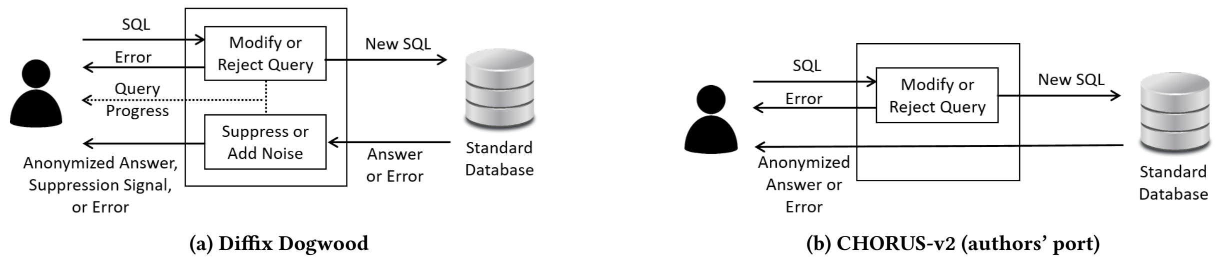


Figure 1: Operation of Diffix Dogwood and CHORUS-v2 systems. Both systems operate as a proxy in front of a standard back end database.

inspects the query to determine if it adheres to the SQL limitations imposed by the system. If not, the query is rejected and an appropriate error message is returned.

If allowed, the SQL query is modified and submitted to the database. Although both systems are designed to send only error-free queries to the database, it is always possible that the database returns an error (and in fact some of the attacks exploit this). If so, the error is passed on to the analyst.

Both systems add noise to the answer's aggregate values. The amount of noise is sensitive to the amount that individuals in the database can contribute to the aggregate. For instance, if the query computes the sum of salaries, then the amount of noise is proportional to the largest possible salary. CHORUS-v2 adds the noise by building noise addition into the SQL query itself. Diffix adds noise to the aggregate returned by the database.

If the database returns an answer, CHORUS-v2 forwards the answer to the analyst. Diffix either suppresses or perturbs the answer, and returns either an implicit suppression signal or the perturbed answer.

All of the attacks were executed via the standard interfaces provided by the two systems from remote locations over the internet.

2.1 Diffix Dogwood

The implementation of Diffix [22, 24] that was attacked in this paper is the production version Dogwood released by Aircloak in 2020. The deployment that was attacked was made publicly available for the anonymization bounty program run in 2020 [23].

Diffix Dogwood[2] perturbs answers in two ways. First, it adds noise to aggregate values like count or sum. Second, it suppresses answers that pertain to too few users. The added noise is not relevant to the attack described in this paper, so nothing more is said about it here.

For suppression to work, Diffix must know how many distinct users are associated with each aggregate in the answer. For instance, if the query is

```
SELECT age, count(*)
FROM table GROUP BY age
```

Diffix would need to know that there are for instance 12 distinct users associated with age=1, 28 distinct users associated with age=2, and so on.

[2]Dogwood is the current version of Diffix. Unless otherwise stated, references to Diffix presume version Dogwood.

One of the several reasons that Diffix modifies the analyst SQL is so that it may obtain this information from the database. Though this is a gross simplification, conceptually Diffix changes the above SQL to

```
SELECT age, count(*), count(DISTINCT users)
FROM table GROUP BY age
```

Diffix uses a *noisy threshold* to determine if a given aggregate should be suppressed or not. If the number of users is zero or one, the aggregate is suppressed. If the number of users is greater than six, then the aggregate is not suppressed. Otherwise Diffix selects a random threshold between 2 and 6 (Gaussian with mean of 4). If the number of users is below the threshold, the aggregate is suppressed. Substantial detail is omitted here, but may be found in [7, 24].

Diffix provides a number of features that aid the analyst in writing queries. One of these is to report the status of the query execution progress at various points in the workflow, in particular the total time spent by the database and the total time spent by the various anonymizing components. This helps the analyst optimize long-running queries, but also adds another side-channel to the attack surface.

Specific workflow reports included *database query time*, *time ingesting data* (pulling the results from the database), and *time processing data*. In particular, if the database query results were empty, then the *ingesting data* report was suppressed. This was a serious design flaw, as it reveals direct information about the results of the query. The two side-channel attacks on Diffix in this paper exploit this design flaw.

2.2 CHORUS-v2

Our port of CHORUS-v2 is based on the archived Uber SQL Differential Privacy GitHub repo [30]. The Uber repo does not provide a complete working query-based anonymization tool. Rather, it supplies a library of basic DP primitives; Elastic Sensitivity [13] and Query Rewriting [12]. It is up to the user to package the primitives into a working anonymization tool. The Uber repo also does not provide any budget management primitives.

To integrate the CHORUS-v2 library into our tool, we needed to make a number of design decisions. We believe that the decisions we made are sensible given the available documentation (the Uber repo itself [30] and the original CHORUS paper [12]). Figure 2a shows the architecture of the CHORUS system as published in [12]. Figure 2b shows the setup of our tool. The code for our tool is available on GitHub [27] and the tool itself is used as part of the GDA-Score[3] suite of anonymization tools [25].

Our tool implements a web interface to receive SQL queries from an analyst and return answers or error messages. It has a simple budget manager which decrements the epsilon supplied with the query from the budget. If the query would exhaust the budget, our tool returns a budget-expired error. If not, our tool queries the CHORUS-v2 re-write library for the modified SQL. If the library rejects the analyst SQL, the error message from the library is returned to the analyst. If not, the modified SQL is submitted to the database.

Our tool takes the result returned from the database, including any error reported by the database, and sends it directly to the analyst. Some of our attacks depend on the analyst (attacker) being able to detect that the database produced an error. Obviously, returning the error itself makes this easy for the attacker, but in general it is non-trivial to hide that a database error occurred: an artificially constructed answer that appears realistic in all cases is difficult to produce, or the timing change incurred by the error can be detected.

In any event, given the available documentation, passing the database error on to the analyst would seem a sensible choice in the absence of knowledge of our attack. The CHORUS paper itself [12] suggests this (Figure 2a), and there is no mention of how to handle database errors, or for that matter any side-channel, one way or the other. It would be reasonable for a user to assume that the CHORUS-v2 rewriter ensures that any SQL it produces is safe to send to the database.

It is important to stress that Uber built and internally deployed its own tool derived from the Uber repo, with potentially a very different set of design decisions and a different back end database. The attack results presented in this paper do not necessarily apply to Uber's deployment.

3 GENERAL SIDE-CHANNEL ATTACKS AGAINST DATABASE SYSTEMS

This section describes attacks that can potentially be executed against *any* query-based anonymizing system using standard back end databases. Subsequent sections drill down on the specific details of the attacks as applied to CHORUS-v2 and on our defense design and implementation in Diffix Dogwood.

The attacks against the database are of the following form:

IF ***isolating_criteria*** **[AND** ***target_attributes*** **]**
THEN ***trigger side-channel***

The *isolating criteria* is one or more conditions that match one and only one individual (the *victim*). The isolating criteria might be a single SQL condition like:

```
WHERE ssn = '123-45-6789'
```

(the social security number of the victim), or a quasi-identifier consisting of a set of SQL clauses like:

[3]The General Data Anonymimity (GDA) Score is an open-source project to measure the strength and utility of data anonymization mechanisms.

Mechanism	Component	Fixed	
		Diffix	CHORUS-v2
WHERE optimizer	Database	(Side)	not-fixed
JOIN optimizer	Database	2020	not-fixed
CASE statement	Database	disallowed	not-fixed
Empty query result	Diffix	(Side)	N/A

Table 1: Summary of If/Then mechanisms, showing in which system component is the mechanism executes, and whether or when a defense was built. ***(Side)*** **means that the defense is handled at the side-channel.** ***Disallowed*** **means that the corresponding SQL is not allowed. N/A means that the attack doesn't apply.**

```
WHERE birth_day = '1990-01-01' AND
      zip = 12345 AND gender = 'M'
```

The *target attributes* are what the attacker wishes to learn about the victim. The SQL clause

```
WHERE disease = 'condition'
```

targets the attribute `disease` and determines if the victim has the given condition or not, and

```
WHERE salary BETWEEN 70000 and 80000
```

targets the attribute `salary` and determines if the victim's salary falls in the specified range.

The target attributes are optional. If not included, then the attack is a *membership inference* attack, which simply determines if the victim is in the database or not. If included, then the attack is an *attribute inference* attack, whereby attributes of a victim known to be in the database are learned.

To run an attack, the attacker must therefore be able to do the following two things:

(1) Execute If/Then logic.
(2) Execute a side-channel.

This section overviews all of the If/Then logic mechanisms and side channels described in this paper. They are summarized in tables 1 and 2 respectively.

3.1 If/Then Mechanisms

The If/Then mechanisms described in this section work on virtually all standard databases.

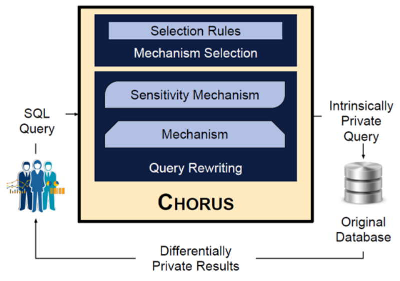
3.1.1 *WHERE Optimization.*

If/Then can be executed by exploiting optimizations in the database query engine for processing conditions in a `WHERE` clause. The optimization is that the query engine will stop processing conditions once the outcome of the `WHERE` clause is certain.

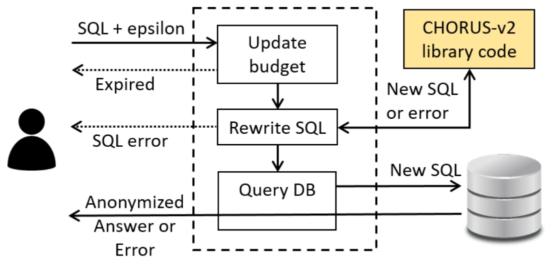
For example, consider the `WHERE` clause

```
WHERE ssn = '123-45-6789'
      AND disease = 'condition'
      AND SQRT(age-1000) = 1
```

Here `ssn` is the isolating criteria, `disease` is the target attribute, and the `SQRT` expression is the side-channel. If the first or second



(a) CHORUS architecture taken from [12]



(b) Authors' port of CHORUS-v2 (our code in dashed box)

Figure 2: Details of authors' port of CHORUS-v2 (b) as patterned after the original CHORUS architecture (a)

	Mechanism	Example	Component	Fixed	
				Diffix	CHORUS-v2
Exception Reporting	Divide-by-zero	1/(col*0) = 1	Database	2020	not fixed
	SQRT negative	SQRT(col-10000) = 1	Database	2020	not fixed
	Numeric overflow	POWER(2,(10000.01 * col)) = 1	Database	2020	not fixed
	Date overflow	col + INTERVAL 'P1000Y' = '1995-06-22'	Database	2020	disallowed
Timing	Invoke delay	SELECT col FROM table	Database	(If/Then)	not fixed
	Query status report	(Native to Diffix)	Diffix	2021	N/A
Other	Query status report	(Native to Diffix)	Diffix	2021	N/A
	SELECT flaw	(Native to CHORUS-v2)	CHORUS-v2	N/A	2020

Table 2: Summary of side-channels, giving an example of each channel, the system component in which the side-channel executes, whether or when a defense was built, and the accuracy of the side-channel detection. *Disallowed* means that the corresponding SQL is not allowed. *(If/Then)* means that the defense is handled in the If/Then mechanism. N/A means that the attack doesn't apply.

conditions are `False`, then the entire `WHERE` expression must be `False`, so the query engine will not bother to execute the final condition. If on the other hand the first and second conditions are `True`, then the final condition must also be evaluated. This therefore acts as an If/Then mechanism.

3.1.2 JOIN Optimization. Similar to the `WHERE` optimization, the If/Then mechanism can be executed by exploiting optimizations in the database query engine for processing `JOIN` . The optimization is that if one of the `JOIN` expressions returns nothing, then there is no need for the query engine to evaluate the remaining `JOIN` expressions because there is nothing for them to `JOIN` with.

For example, consider the following `JOIN`:

```
SELECT count(*) FROM
(  SELECT col FROM table
   WHERE ssn = '123-45-6789' AND
   disease = 'condition' ) j1
JOIN
(  SELECT col FROM table ) j2
ON j1.col = j2.col
```

If there is no user with that `ssn` in the table, or if there is such a user but that user does not have the indicated disease, then the `JOIN` expression `j1` will return nothing, and the query engine will not bother to evaluate `JOIN` expression `j2`. By contrast, if there is such a user and the user has the disease, then `j1` will return something and `j2` will be evaluated. Therefore this acts as an If/Then mechanism.

3.1.3 CASE Statement. The `CASE` statement in and of itself directly implements If/Then logic:

```
CASE WHEN ssn = '123-45-6789' AND
          disease = 'condition'
THEN SQRT(age-1000) = 1
```

3.2 Side-Channels

This section describes the side-channels that operate on the database. Not all of the side-channels work on all databases. We don't make an exhaustive survey of which side-channels work on which databases, but instead give a few examples of databases where the side-channels work or don't work.

3.2.1 Exception Reporting. Databases execute mathematical operations, and can therefore have the arithmetic errors of divide-by-zero and square root of a negative number. Numeric and date columns in databases are often limited in size (i.e. 32 bits, year 9999), and are therefore subject to overflow or underflow errors. Some databases handle the arithmetic errors silently by evaluating the math expression as `NULL` and continuing execution. MySQL and SQLite are two examples. PostgreSQL, on the other hand, halts query execution and reports an exception for the arithmetic errors. All three report exceptions for overflow/underflow errors.

The example exception throwing SQL expressions from Table 2 are repeated here for convenience:

```
1/(col*0) = 1
SQRT(col-10000) = 1
POWER(2,(10000.01 * col)) = 1
col + INTERVAL 'P1000Y' = '1995-06-22'
```

Each expression must include a column value (here `col`) to trigger the side channel. If no column value is included (i.e. `SQRT(-1)`), then the database tries to pre-compute the expression and returns an error before query execution begins. The right-hand sides of the expressions are irrelevant: the error takes place before any comparison can be made.

The first three require a numeric column, and the last a date or time column. Since the expression executes on rows matching the victim, the attacker must pick an expression that is sure to trigger the side-channel for the victim's corresponding column value. These are just four of many possible ways of triggering the error. In this specific example for `SQRT`, the constant (10000 in this example) needs to be larger than the largest possible column value. It is not necessary for the attacker to know the victim's value per se. Likewise for the overflow examples (`POWER` and `interval`), the attacker can use a constant that produces an overflow for even the smallest column value, and again doesn't need to know the victim's value.

3.2.2 Invoke delay. The second side-channel on databases is to invoke an action that takes long enough to be measurable over a network with high confidence. As demonstrated in Section 6.1, a few 10s of milliseconds delay is enough to detect the side-channel with 100% accuracy. The example in Table 2 gives an expression that causes a scan of the entire database table:

```
SELECT col FROM table
```

An example of how this table scan can be used along with a `JOIN` If/Then mechanism is shown in the `JOIN` query of Section 3.1.2. An SQL `IN` clause can be used to invoke a table scan along with a `WHERE` If/Then mechanism like this:

```
WHERE ssn = '123-45-6789'
      AND disease = 'condition'
      AND col1 IN (SELECT col2 FROM table)
```

An example with the `CASE` If/Then is:

```
CASE WHEN ssn = '123-45-6789' AND
          disease = 'condition'
THEN (SELECT sum(col) FROM table)
```

Note that an aggregate (`sum(col)`) is being selected rather than a full column because the `THEN` clause expects a scalar rather than an array.

There are no doubt many ways to invoke a measurable delay. For the purposes of this paper, it is sufficient to give working examples and is out of scope to try to survey them all.

3.3 Attacker Assumptions

We assume that the attacker has access to the full API offered by the system over a shared network such as the Internet. The attacker does not require physical access to the machines themselves, nor the ability to modify code on the machines.

Both membership inference and attribute inference attacks require that the attacker is able to isolate the victim with high confidence. This implies having specific knowledge of certain victim attributes, as well as high confidence that only the victim has those attributes. This does not necessarily mean, however, that the attacker must have *prior knowledge* of the victim. In general the attacker can use the attacks themselves to discover isolating criteria.

For instance, if the attacker knows that a given column contains unique identifiers, like `ssn`, then the attacker can query for `ssn` values until one is found that has an associated user. A search tree may be used for efficiency, for instance by selecting ranges for numeric columns, or substrings for text columns. If there are no identifying columns, the attacker can search multiple columns that can serve as a pseudo-identifier with high probability.

4 DATABASE ATTACKS DEMONSTRATED ON CHORUS-V2

All of the side-channels from the prior section work on CHORUS-v2 in one form or another except for the overflow on dates: CHORUS-v2 does not allow *INTERVAL* or math with dates. While CHORUS-v2 does limit SQL, it does so only for the purpose of ensuring that it applies noise properly, not to defend against side-channels.

Our CHORUS-v2 port does not allow `sum(column)`, which is used in the invoke delay side-channel in conjunction with the `CASE` If/Then mechanism. In this case we can use the `IN (SELECT ... )` expression within the `THEN (SELECT ...)` to invoke delay:

```
CASE WHEN ...
THEN ( SELECT COUNT(*) FROM table
       WHERE col IN(SELECT ...) )
```

Note that the `COUNT(*)` itself does not necessarily produce a table scan, which is why the additional `IN` is needed.

4.1 Number of Queries

Differential Privacy systems impose a total privacy budget ϵ_{total}. This effectively limits the total number of queries relative to the amount of noise ϵ in each query. More noise allows more queries. It is common for the database owner to set ϵ_{total}, but to allow the analyst to set ϵ per-query. We follow this convention in our CHORUS-v2 port. In addition, we don't impose a lower limit on the value. This allows the analyst to increase the number of queries by setting a very large noise value (very small epsilon). For instance, by selecting $\epsilon = 0.0000001$ against a total budget of $\epsilon_{total} = 1.0$, we get 10 million queries before exhausting the budget.

Such a tiny ϵ is not needed for any reasonable analytic purpose, and so a sensible mitigation would be to impose a lower limit on ϵ. While this would severely limit the amount of information an analyst could infer, it doesn't prevent the attack per se.

The tiny ϵ approach worked as expected with all combinations of side-channel and If/Then mechanisms except those using `JOIN`. It so happens that the length of time it takes CHORUS-v2 to compute the noise parameter for `JOIN` queries increases with decreasing ϵ. A query that takes under one second for $\epsilon = 0.1$, increases to

14 seconds with $\epsilon = 0.001$ and 284 seconds for $\epsilon = 0.0000053$ and appears to grow linearly with decrease in ϵ. This delay puts a practical limit on the number of queries an attacker can execute using `JOIN`; an ϵ value that allows 10000 queries would take two weeks to execute all of the queries.

4.2 Demonstration of Arbitrary Data Reconstruction

The fact that a variety of If/Then mechanisms and side-channels can be run against CHORUS-v2 implies that virtually any information can be extracted from the database. We demonstrated this by extracting full last names from a database with no prior knowledge other than knowing that a certain given column (the user ID column) contains unique identifiers. The database, deployed on our premises, consisted of a combination of public and synthetic data. No private data was a risk in this or any of our attacks.

For this demonstration, we used the `CASE` If/Then mechanism and timing side-channel. Our attack wasn't particularly sophisticated in that we didn't implement efficient search algorithms to discover user IDs, but nevertheless the attack worked.

We started by running membership inference attacks with random user IDs. Upon finding an ID, we used it as the isolation criteria to attack the names. We did this by first attacking the length of the name using `CHARLEN`, and then using `SUBSTRING` to attack each character in the name one by one until the name was reproduced.

Note that the purpose of this demonstration is not to measure how efficiently data can be extracted, but rather to simply show that it can be done. A study of how efficiently data can be extracted would entail search heuristics optimized for the structure of the specific data being attacked, and possibly using dictionaries (i.e. a dictionary of common last names). Such a study is out of scope for this paper.

5 DEFENSES AGAINST THE DATABASE ATTACKS

There are broadly three ways to defend against the general database side-channel attacks of Section 3. They are listed here in order of preference:

(1) Disable or mask the side-channel itself.
(2) Disable the query optimizations that lead to If/Then logic.
(3) Limit the SQL functionality.

We prefer to avoid limiting SQL for obvious reasons, and do so only as a last resort. Disabling query optimizations is unattractive in part because it leads to slower queries, but also because in most cases it is not clear how to do so without modifying the database implementation itself. Since a core design goal of both CHORUS and Diffix is to accommodate unmodified back end databases, this is not an option.

Disabling the exception side-channel by returning NULL rather than throwing an exception is an attractive approach. The same thing, however, cannot be done for the timing side-channel. Masking the timing side-channel requires adding substantial amounts of artificial delay to queries [11], which in our experience with real users leads to unacceptable performance.

Ultimately we were forced to use all three approaches. We modify the SQL transmitted to the database to disable the exception side channel. Where possible, we disable query optimizations to prevent the timing side-channel, but limit SQL functionality otherwise.

These approaches are described in detail in the following sections.

5.1 Defense against exception reporting

A general approach to defending against exception reporting is to install user defined functions (UDF) into the database which inserts NULL rather than throws exceptions, and then rewrite the SQL to invoke the UDFs instead of the standard equivalent native function. So for instance `POWER()` could be replaced with a UDF `UDF_POWER`, and `col + 1` could be replaced with `UDF_SUM(col,1)`.

With some databases, UDFs are quite performant because they can be written for instance in C. With other databases, however, the UDFs must be written in SQL and can be quite slow. Since execution time is an important performance metric, we developed the following strategy for avoiding the use of UDFs where possible. (For readability we limit the following to overflow, but equivalent mechanisms are used for underflow as well.)

(1) In the background, periodically determine an anonymized upper bound on values in each numeric and datetime column.
(2) At query time, cast numeric columns to a larger native type (i.e. 8 bits to 64 bits) by re-writing the SQL.
(3) Statically evaluate the re-cast SQL expressions using the bound to determine if an overflow would occur.
(4) If yes, then replace the expression with its safe equivalent UDF.
(5) If no, leave the expression as is but modify the SQL inline to enforce the upper bound (values exceeding the bound are replaced by `NULL`).
(6) Determine if a *SQRT* or divide-by-zero exception is possible. If yes, then modify the SQL inline to prevent the exception and return NULL instead.

To determine the upper (and lower) bound for each numeric column, Diffix periodically queries for the highest column values and (re)computes the bound.

The bound needs to be anonymized because an attacker can potentially learn the bound through a timing attack that detects whether a (slow) UDF was installed or not. In addition, the Aircloak system makes the rewritten SQL available to analysts for query debugging purposes and so can be read directly. The bound should be high enough that actual column values almost always fall within the bound, but low enough that it is rarely necessary to invoke the UDF.

Our design for anonymizing the bound is to first select a per-column *rank* value R randomly between 10 and 20. Working in order of highest to lowest column value, the column value V of the R^{th} distinct user is selected, and then increased by a factor of 10. The resulting value is then rounded up to the next higher number with one significant digit (i.e. 23043 is rounded to 100000). The resulting anonymized upper bound is usually higher than the actual highest column value, but not necessarily. For instance it can be lower if there are extreme outliers in the data or if new data was inserted after the last bound computation.

At query time, we statically evaluate the SQL expression by using the upper bound as the column value, and determine if the upper bound would cause an overflow. In this evaluation, we use the largest native numeric type. This type is enforced by re-writing the SQL to CAST the column to the largest type.

The following SQL snippet is an example of this. The original WHERE condition col * 2 = 1 is replaced in the SQL by the following expression:

```
...
   (SELECT CASE
       WHEN (col < 0) THEN 0
       WHEN (col > 1000) THEN 1000
       ELSE col END AS bounded_col
      FROM table) AS t
WHERE ((CAST(t.bounded_col AS BIGINT)*2) = 1)
```

Here 0 and 1000 are the anonymized lower and upper bounds (on a column col whose real lower and upper bounds are 0 and 75). An inner SELECT is created with a CASE statement to force the column value to be within the bounds during query execution. The bounded column bounded_col is CAST up to the largest column type that fits within the native machine integer type (BIGINT). In this example, a UDF was not needed.

If SQRT or division operations are in the SQL, then we modify the SQL to test for the error condition in the SQRT operand (negative value) or division denominator (zero), and if so replace the entire expression with NULL using CASE. For example, if a WHERE expression contains 1/(col*0)=1, we replace the entire left hand side with this CASE statement:

```
CASE WHEN col*0 < 1.0e-100 THEN NULL
     ELSE 1/(col*0)
END
```

This effectively mimics how many databases handle these errors (by inserting NULL rather than throwing an exception).

5.1.1 Performance. Despite our attempts to avoid performance loss, re-writing SQL to prevent exceptions as described above does lead to increased query times. As an example, we compared the execution time of the following two queries on the Aircloak implementation of Diffix:

```
SELECT count(*) FROM jan01
WHERE pow(trip_time_in_secs,2) = 100

SELECT count(*) FROM jan01
WHERE trip_time_in_secs = 100
```

The first query, which invokes the SQL re-writing described above, completed in roughly 550ms on average. The second query completed in roughly 300ms on average. These queries were run on a database of taxi rides consisting of roughly 400K rows.

5.2 Disabling If/Then with JOIN

As mentioned earlier, defending against timing side-channels by artificially adding delay to query answers leads to unacceptable performance. Disabling If/Then logic in WHERE and CASE statements by re-writing SQL appears extremely difficult if even possible. We did not seriously attempt to do so.

Instead we chose to limit the SQL to prevent the combination of timing side-channels with WHERE and CASE statements. Specifically, we prevent the use of SELECT sub-queries within IN and THEN statements. These SQL constructs are in any event not that commonly used.

On the other hand, re-writing SQL to disable If/Then logic in JOIN statements is relatively straight-forward. This is fortunate, because preventing the use of SELECT sub-queries with JOIN makes no sense, and preventing JOIN altogether is too limiting.

To disable If/Then with JOIN, we need to ensure that every JOIN expression is executed by the query engine, which in turn means that every JOIN expression must return at least one row. To do this, we use UNION ALL to artificially append an arbitrary "chaff" row into the JOIN expression.

For example, if the original analyst query is this:

```
SELECT ... FROM
   ( SELECT ... ) join1
JOIN
   ( SELECT ... ) join2
ON join1.col = join2.col
```

We force an additional chaff row to the first JOIN expression join1 like this:

```
SELECT ... FROM
   (( SELECT ... )
     UNION ALL
    ( SELECT rare_value1, rare_value2, ...)
   ) join1
JOIN
   ( SELECT ... ) join2
ON join1.col = join2.col
```

As a result, even if the original join1 expression returns zero rows, the chaff row from the UNION ALL is returned, thus forcing the execution of the second JOIN expression in all cases. If there are multiple JOIN expressions, a chaff row is appended to all but the last.

The SELECT for the chaff row is literally as shown above: a SELECT followed by a series of constants, one for each column given by the original analyst query. A value very unlikely to exist in the original data is used. In this way, the probability that the chaff row joins with anything is extremely low and so the chaff row doesn't effect the query result. In the Diffix implementation, these are hard-coded values, literally -2147483648 for numeric values and '__ac_invalid_value' for text. These values can be read from the rewritten queries that the Aircloak system makes available to analysts. In the rare event that the chaff row does get included in the JOIN, it is not a disaster since in any event the query result is noisy.

6 ATTACKS AGAINST THE PROXY IMPLEMENTATIONS

In contrast to the attacks described in prior sections that exploit general database mechanisms, this section describes novel attacks against mechanisms specific to the Diffix and CHORUS proxies themselves. Although these attacks are less broadly applicable and therefore less interesting academically, we include them here because both attacks stem from adding features that on one hand make the systems more usable, but on the other fall outside of the core privacy principles of the system. For Diffix these core privacy principles are answer suppression and perturbation. For CHORUS the core privacy principle is Differential Privacy. As such, the attacks in this section serve as cautionary tales of how features can creep into practical query-based anonymization systems, and the extent to which designers need to guard against vulnerabilities that result from these features.

6.1 Timing Attack Against the Diffix Proxy

The timing attack described in this section differs from those of Section 3.2.2 in two ways. First, the timing difference is not explicitly invoked by the SQL, but rather exists natively in the implementation of Diffix. Second, the timing difference is quite small, and so considerable sophistication is required to detect it. Nevertheless, we were able to achieve 100% accuracy in a membership inference attack.

Interestingly, it was only after the attack was demonstrated that the root cause of the attack, Diffix query status reporting, was discovered. Our original concept for the attack came from the observation that Diffix executes different processing steps depending on whether the database returns *some data* to Diffix or returns *no data* to Diffix. In the former case, Diffix must undergo a number of processing steps that are not necessary in the latter case. It must read in the data, determine the number of distinct user identifiers associated with the data, and if the number is greater than one, seed a random number, generate a random number, and then use that value as a noisy threshold to decide whether or not to suppress the answer. The original goal of the attack, then, was to determine if the delay incurred by this extra processing could be detected, thus, revealing if zero or more than zero users match the query.

If the extra processing can be detected, then the isolating criteria alone can be used in a membership inference attack, and the isolating criteria and target attributes together can be used in an attribute inference attack (Section 3). There is no need to trigger a side-channel per se.

6.1.1 Attack Overview. Our attack was designed and executed as part of an anonymization bounty program offered by the Max Planck Institute for Software Systems [6]. The bounty program awards prizes for both membership and attribute inference attacks. For our attack we chose the former, and that is what we report on in this paper. The attack can be easily extended to an attribute inference attack by adding the appropriate isolating criteria to the queries.

In the setup for the bounty program's membership inference attack, there is a *public database* whose contents are fully known, and a *target database* protected by Diffix. The two databases have the same columns. The records for roughly half of the users in the public database are also in the target database (randomly selected and fully replicated), and the goal of the attack is to determine which public database users are in the target database.

The basic procedure is to establish a ground-truth timing distribution for queries that return zero rows (non-existing users), and a separate ground-truth for queries that return one or more rows (existing users). The former requires no prior knowledge, because it is easy to generate queries with conditions that match no users with very high probability. The latter requires prior knowledge of at least one user certain to be in the target database. To determine if a given user (the victim) is in the target database, we establish timing distribution using queries whose conditions isolates the victim. By comparing the victim's timing distribution with the two ground-truth distributions, we can deduce whether the victim is a member of the target database or not.

6.1.2 Attack Details. We chose the user-unique social security number (ssn) column as the isolating criteria using the following query:

```
SELECT count(DISTINCT ssn)
FROM accounts
WHERE ssn = '123-45-6789'
```

Let C^{PUBLIC} be a set of victims from the public database each identified by the ssn column. Let $C^{\text{NON-EXISTENT}}$ be a set of non-existing users, each identified by a social security number generated from a random 15-character strings such that the number does not exist with very high probability. Finally, let C^{PRIOR} be a (small) set of users known a-priori to exist in the target database.

The attack then consists of the following steps:

(1) For each user in the set $C^{\text{NON-EXISTENT}}$ (respectively C^{PRIOR}) query the target database $\mathcal{R}$ times, each resulting in a timing. Merge all timings into a single list $\mathcal{T}^{\text{NON-EXISTENT}}$ (respectively $\mathcal{T}^{\text{PRIOR}}$).
(2) Compute the average distance between the non-existing and the existing (prior knowledge) timings to define a threshold used during decision making.
(3) For each victim $C_i^{\text{PUBLIC}} \in C^{\text{PUBLIC}}$ query the target database $\mathcal{R}$ times, resulting in a list of timings $\mathcal{T}_i^{\text{PUBLIC}}$.
(4) Perform a statistical test on the two data sets $\mathcal{T}^{\text{NON-EXISTENT}}$ and $\mathcal{T}_i^{\text{PUBLIC}}$ for each victim, resulting in value d which quantifies the distance.
(5) Finally, decide the membership for each victim based on the d-value and the threshold.

Decision Functions. To assess the success of our attack we borrowed the corresponding measure of *confidence improvement* (*CI*) from the GDA-Score [25] as used in the bounty program: The *CI* measures the success of the attack if a claim is made as the normalized difference between the confidence (C), i.e. the probability of the attack making a correct claim, and the statistical confidence (S), i.e. the probability that a simple statistical guess is correct: $CI = {}^{C-S}/_{1-S}$. For example, a $CI = 0$ means that the claim is not better than guessing a value, whereas $CI = 1$ means that all claims were correct. This metric is in line with the privacy property Diffix aims to achieve: *The confidence the analyst has in a guess is not substantially higher than a guess the analyst could have made purely with external knowledge.*

To implement our attack we used the two-sided *Kolmogorov-Smirnov* (KS) [15, pp. 392-394] from the *Python 3* library *SciPy* [31]. The test outputs a d-value, representing the distance between the empirical distribution function of the first and the cumulative distribution of the second data set. Figure 3 depicts an example of the distributions for our query timings of non-existing entries

#queries	CI		min [ms]	max [ms]	avg [ms]
5	0.13	NE	177	225	198
		E	188	253	216
50	0.96	NE	167	280	200
		E	182	315	228
100	1.0	NE	167	300	202
		E	182	328	229

Table 3: Detailed timings for queries of existing (E) and non-existing (NE) entries in milliseconds. Non-existing values refer to the random strings for the ground truth. Existing queries refer to the timings resulting from the single prior knowledge value. A Confidence Improvement *CI* of 0 corresponds to a 50% success rate (equivalent to a statistical guess). In the final attack we used $\mathcal{R} = 100$ queries resulting in a timing difference of about 27ms.

compared to those of the existing entries to visualize the distance. In general, $d \rightsquigarrow 0$ means that the distributions are equal, and $d \rightsquigarrow 1$ that the distributions are different.

We defined our threshold for the d-value based on the average distance of timings, i.e., $t = 1 - (1 - avg(\mathcal{T}^{\text{NON-EXISTENT}})/avg(\mathcal{T}^{\text{PRIOR}})) \cdot 1/3$. To decide the membership, we compared the d-value d_i for a specific victim C_i^{TARGET} to the average over all d-values and claimed that the user exists in the target database, if this value is greater than the average d-value of all victims: $d_i \cdot t \overset{?}{>} avg(\{d_j\}_j)$

Establishing Attack Parameters. To achieve perfect *CI* we evaluated different numbers of users for the ground truth and the prior knowledge. With no prior knowledge users (and therefore no ground truth timing distribution for existing users), we were able to achieve a *CI* of about 0.7 by fixing the threshold to 0.9 of the average of all victim timing measures. In other words, if a d-value is larger than 90% of the average across all victims we claim membership in the target database database.

After experimenting with different attack parameters, we determined that $\mathcal{R} = 100$ is enough to average out most timing differences. Moreover, we required prior knowledge of only a single user from the target database to get 100% accuracy.

The attack was performed from a virtual machine with a round-trip delay of about 15 ms between the attack machine and the Diffix machine. Table 3 compares the timings of queries for non-existent values to the timings for existing values for number $\mathcal{R}$ of queries. The table shows that the average timing difference converges, such that a larger number of queries allows a better estimation. As a result, the average timing for non-existing rows is 202 ms and for existing rows about 229 ms. To decide the membership of 200 victims we performed 300 queries for the ground truth, 300 queries for the threshold, and 20000 queries for the victims, resulting in an overall attack time of about 100 minutes.

6.1.3 Mitigation. In debugging the root cause of the timing attack reported in the previous section, the Aircloak team determined that the timing difference is caused by differences in the way Diffix reports query status. Prior to being fixed, Diffix would report a query status of *ingesting data* if the query answer is not empty, and would not report that query status otherwise. Reporting this status required a number of additional steps, such as updating the logging system and making the status message available over the API. The additional delays in handling the status message accounted for the extra 23.5ms delay.

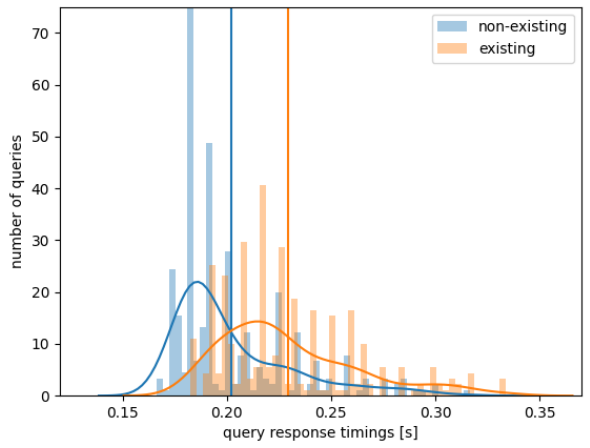


Figure 3: Probability density function of timings for non-existing values vs existing values after $\mathcal{R} = 100$ queries. The vertical lines denote the average timings. The curves show a clear offset for the timing of the existing entries, however, they also show that multiple queries might be necessary to identify membership of a target correctly.

Aircloak modified the system so that the *ingesting data* status was disabled altogether. After the fix, our attack could detect no difference in the timing between queries that return zero rows and queries that return multiple rows.

6.2 Diffix Query Status Reporting

Once the design flaw with reporting *ingesting data* was discovered, we realized that the mere existence or non-existence of the status report itself could be used as the side-channel rather than timing. This leads to 100% accuracy using only a single query. The If/Then logic is then:

IF *non-empty query answer*
THEN *report ingesting data*
and the database can be completely reconstructed.

6.3 CHORUS-v2 SELECT flaw

While experimenting with CHORUS-v2, we happened to notice that queries that simply select a column return the column values. For example, the query

```
SELECT col1, col2
FROM table
```

is re-written by the CHORUS-v2 implementation to

```
SELECT "col1", "col2"
FROM "public"."table"
```

which returns the resulting noiseless column values. We refer to this as the *SELECT flaw* in Table 2.

It might seem odd that there is any code path at all that can output a column value in a Differential Privacy system. After all Differential Privacy in its pure form can only output the noisy results of aggregates over numeric functions like `count()` or `sum()`. To make Differential Privacy more practical, however, it is useful to output column values when doing so does not violate some notion of privacy. For example, the `department` column of a database may have values like 'sales' and 'engineering'. These values are in any event publicly known and can be released without violating privacy per se.

CHORUS-v2 has a feature that allows individual columns as well as whole tables to be configured as 'public'. This allows the individual column values to be released with `SELECT column`. In our port of CHORUS-v2, no column or table was configured as 'public'. Nevertheless, the mere possibility of allowing such a configuration creates a code path that can lead to this flaw.

It so happens that in our port of CHORUS-v2, we expected a single noisy real value as an answer, and that is all we return (or an error message if the value is not a real). Therefore strictly speaking we cannot simply request a dump of the data using our port, although the underlying CHORUS-v2 system allows it.

The developer of CHORUS-v3 [14] reports that this flaw does not exist in CHORUS-v3.

7 PRIOR WORK

Several prior attacks on query-based anonymization systems have been published. Haeberlen et al. [11] demonstrated three different classes of side-channels against two query-based DP systems, PINQ [17] and Airavat [28]. The If/Then mechanisms of Haeberlen et al. differ from ours in that Haeberlen et al. could exploit the language features of C# (PINQ) and Java (Airavat), whereas our attacks are limited to a subset of SQL. In this regard the attacks of Haeberlen et al. do not work against CHORUS or Diffix. Haeberlen et al. exploited three side channels: timing, the value of the answer, and the DP budget. We likewise use a timing side channel, but our other side channels differ.

To defend against the timing side channel attacks, Haeberlen et al. proposes FUZZ [11], which eliminates the side channel itself by forcing a fixed timing on the query. The DP-based GUPT system [19] likewise forces a fixed timing. This approach leads to massively increased query times. By contrast, Diffix eliminates the corresponding If/Then mechanism, which in turn leads to a different trade-off; limiting query expressiveness.

Mironov [18] identified a different type of attack against DP-based systems that is based on floating point irregularities in the implementation of the underlying Laplacian mechanism. These irregularities result from finite precision and rounding effects of floating-point operations. The author demonstrated the attack against both PINQ and Airavat. Other prior work showed that CHORUS can also be successfully attacked by the Mironov attack [29]. Although one participant in the Diffix bounty program tried and failed to execute this class of attack (unfortunately no details are available), it is of course unknown whether another variant of the attack could work against Diffix.

PSI [9] achieves robustness against the Haeberlen et al. [11] attacks as well as the attacks in this paper by only allowing built-in differentially private data analyses. To protect against the floating point attack by Mirnov [18], the system includes an implementation of the snapping algorithm [18].

In prior work on attacks against Diffix, Cohen and Nissim [1] showed that an earlier version of Diffix (Cedar) is vulnerable to linear reconstruction attacks. Such attacks rely on queries with dummy conditions to obtain noise samples that can be used to infer individual data points. Gadotti et al. [10] proposed an attack on Diffix Birch to cancel out the noise generated by Diffix to learn private attributes of the underlying data. These attacks exploited design weaknesses in the corresponding versions of Diffix itself, and are not side channel attacks per se. Diffix Dogwood defends against these specific attacks.

Zheng and Shen [32] propose using machine learning to detect and block privacy-disclosing queries. Although they demonstrated this approach against specific Diffix attacks, it is unclear whether such an approach can work in general.

Kiefer et al. [16] developed general guidelines for the implementation and audit of Differential Privacy systems to minimize the attack surface.

8 DISCUSSION AND CONCLUSIONS

There was a period of time when both CHORUS and Diffix were deployed in production environments to analyze real user data during which the vulnerabilities described in this paper existed. In spite of this, we believe that it is very unlikely that the vulnerabilities were ever exploited maliciously. This is in part because the analysts working on these systems were employees of the companies responsible for the data, and were therefore unlikely to be motivated to violate privacy. The number of analysts was also relatively small (a few 10s at most in the case of Aircloak). Even if one of them were motivated to violate privacy, he or she would have had to have independently discovered the vulnerability (and having done so, would probably benefit more from reporting the vulnerability to his or her employers than by attacking the users!) Finally, at least in the case of Aircloak, a log of queries is maintained, adding an additional deterrent.

It would be tempting at this point to declare that the defense mechanisms presented in this paper are fully effective, and that CHORUS could simply adopt them. There are, however, some unattractive trade-offs associated with our defense mechanisms. First of all, they are complex. This not only makes them brittle, but it makes it hard to add new SQL features to the system. Second, some of the mechanisms are database-specific, especially the UDFs. Adding a new database to the system is a substantial work effort. Third, correct operation of the overflow mechanisms requires that the system makes periodic out-of-band measurements of the data; yet another complexity and overhead.

Instead, CHORUS could if possible limit itself to databases where all exceptions can be disabled (i.e. always converted to NULL in the database). In this case all of the exception reporting attacks would be eliminated, leaving just the timing attacks. In our experience with Diffix, `SELECT` within `IN` and `CASE` are relatively low priority SQL features, and so these could be disallowed in CHORUS as they are in Diffix. This would leave `JOIN`, which is in fact a critical feature. For this, CHORUS could implement the `UNION ALL` mechanism, which is relatively simple. Alternatively, CHORUS could disable `JOIN` in CHORUS itself, and require instead that the needed `JOIN`s be implemented as `VIEW`s by a trusted administrator.

Speaking now as Aircloak, we believe that it may have been a mistake to design the system as a proxy that can interface with multiple back end databases. Doing so has led to tremendous complexity, to the point where it is very difficult to add new features. An alternative approach would be to select a single database and tightly integrate to that database. This not only makes it easier to manage errors and query engine optimizations, but simplifies many other aspects of Diffix. Indeed we have started a project to do just this, the Open Diffix project [21]. The goal is to integrate Diffix with PostgreSQL using PostgreSQL extensions, and to release it under and open source or partial open source license.

A query-based data anonymization system with strong anonymization and good utility is a kind of holy grail in computer science. An advantage of query-based anonymization systems is that they promise more analytic flexibility: different analysts can slice and dice the data in different ways. By contrast, table-based approaches invariably require that the analytic purpose is known in advance. On the other hand, query-based approaches expose a larger attack surface, and experience shows that it is hard to defend against everything.

While the defenses presented in this paper indeed defend against the attacks presented herein, it would be naive to assume that there are no more attacks, side-channel or otherwise. It is important that organizations deploying query-based anonymization systems nevertheless maintain good security practices such as authorization and access control, contractual oversight, and query logging.

ACKNOWLEDGMENTS

Funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden. Funded by the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL), topic "46.23.01 Engineering Secure Systems"

REFERENCES

[1] Aloni Cohen and Kobbi Nissim. 2018. Linear program reconstruction in practice. *arXiv preprint arXiv:1810.05692* (2018).
[2] Dorothy E. Denning. 1981. Restricting Queries that Might Lead to Compromise. In *1981 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 27-29, 1981*. 33–40. https://doi.org/10.1109/SP.1981.10000
[3] Cynthia Dwork. 2006. Differential Privacy. In *ICALP*.
[4] Ivan P Fellegi. 1972. On the question of statistical confidentiality. *J. Amer. Statist. Assoc.* 67, 337 (1972), 7–18.
[5] Agency for Healthcare Research and Quality. 2013. Healthcare Cost and Utilization Project (HCUP) . http://www.ahrq.gov/research/data/hcup/index.html. Last Accessed April 10, 2021.
[6] Paul Francis. 2021. *Procedures and Rules for the 2020 Diffix Bounty Program* . Technical Report MPI-SWS-2021-002. MPI-SWS. http://www.mpi-sws.org/tr/2021-002.pdf
[7] Paul Francis, Sebastian Probst Eide, Pawel Obrok, Cristian Berneanu, Sasa Juric, and Reinhard Munz. 2018. Diffix-Birch: Extending Diffix-Aspen. *CoRR* abs/1806.02075 (2018). arXiv:1806.02075 http://arxiv.org/abs/1806.02075
[8] Paul Francis, Sebastian Probst Eide, and Reinhard Munz. [n.d.]. Diffix: High-Utility Database Anonymization. In *Privacy Technologies and Policy*, Erich Schweighofer, Herbert Leitold, Andreas Mitrakas, and Kai Rannenberg (Eds.). Lecture Notes in Computer Science, Vol. 10518. Springer International Publishing, 141–158. https://doi.org/10.1007/978-3-319-67280-9_8
[9] Marco Gaboardi, James Honaker, Gary King, Jack Murtagh, Kobbi Nissim, Jonathan Ullman, and Salil Vadhan. 2018. *PSI ({\Psi}): A Private Data Sharing Interface.* arXiv:1609.04340 [cs, stat] http://arxiv.org/abs/1609.04340
[10] Andrea Gadotti, Florimond Houssiau, Luc Rocher, Benjamin Livshits, and Yves-Alexandre de Montjoye. 2019. When the Signal is in the Noise: Exploiting Diffix's Sticky Noise. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1081–1098. https://www.usenix.org/conference/usenixsecurity19/presentation/gadotti
[11] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. 33 (2011).
[12] Noah Johnson, Joseph P. Near, Joseph M. Hellerstein, and Dawn Song. 2018. *Chorus: Differential Privacy via Query Rewriting.* arXiv:1809.07750 [cs] http://arxiv.org/abs/1809.07750
[13] Noah M. Johnson, Joseph P. Near, and Dawn Song. 2018. Towards Practical Differential Privacy for SQL Queries. *Proc. VLDB Endow.* 11, 5 (2018), 526–539. https://doi.org/10.1145/3187009.3177733
[14] Joseph Near. 2020. Github Repo uvm-plaid/chorus. https://github.com/uvm-plaid/chorus. Last Accessed April 10, 2021.
[15] Marvin Karson. 1968. Handbook of Methods of Applied Statistics. Volume I: Techniques of Computation Descriptive Methods, and Statistical Inference. Volume II: Planning of Surveys and Experiments. I. M. Chakravarti, R. G. Laha, and J. Roy, New York, John Wiley; 1967, $9.00. *J. Amer. Statist. Assoc.* 63 (1968), 1047–1049.
[16] Daniel Kifer, Solomon Messing, Aaron Roth, Abhradeep Thakurta, and Danfeng Zhang. 2020. *Guidelines for Implementing and Auditing Differentially Private Systems.* arXiv:2002.04049 [cs] http://arxiv.org/abs/2002.04049
[17] Frank McSherry. 2009. Privacy Integrated Queries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. ACM, 19–30.
[18] Ilya Mironov. 2012. On Significance of the Least Significant Bits for Differential Privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, 650–661. https://doi.org/10.1145/2382196.2382264
[19] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. 2012. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings of the 2012 International Conference on Management of Data - SIGMOD '12* (Scottsdale, Arizona, USA). ACM Press, 349. https://doi.org/10.1145/2213836.2213876
[20] Joe Near. 2018. Differential Privacy at Scale: Uber and Berkeley Collaboration. In *Enigma 2018 (Enigma 2018)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/node/208168
[21] Open Diffix Project. 2020. Open Diffix. http://open-diffix.org. Last Accessed April 10, 2021.
[22] Paul Francis. 2021. Customer Documentation for Aircloak's Diffix Dogwood . http://www.mpi-sws.org/tr/2021-003.pdf. Last Accessed April 10, 2021.
[23] Paul Francis. 2021. Procedures and Rules for the 2020 Diffix Bounty Program . http://www.mpi-sws.org/tr/2021-002.pdf. Last Accessed April 10, 2021.
[24] Paul Francis. 2021. Specification of Diffix Dogwood . http://www.mpi-sws.org/tr/2021-001.pdf. Last Accessed April 10, 2021.
[25] Paul Francis. 2021. GDA Score Overview. https://www.gda-score.org/what-is-a-gda-score/. Last Accessed April 10, 2021.
[26] Uber Privacy and Security. 2017. Uber Releases Open Source Project for Differential Privacy . https://medium.com/uber-security-privacy/differential-privacy-open-source-7892c82c42b6. Last Accessed April 10, 2021.
[27] Reinhard Munz. 2019. Github code branch for the GDA-Score port of Uber sql-differential-privacy. https://github.com/gda-score/anonymization-mechanisms/tree/master/uber. Last Accessed April 10, 2021.
[28] Indrajit Roy, Hany Ramadan, Srinath Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. 10 (2010), 297–312.
[29] Amaresh Ankit Siva. 2020. CHORUS Is Porous: Attacking Implementations of Differential Privacy. (2020).
[30] Uber and UC Berkeley. 2017. Github Repo sql-differential-privacy. https://github.com/uber-archive/sql-differential-privacy.
[31] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Reddy et al., and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2
[32] Jianguo Zheng and Xinyu Shen. 2021. Pattern Mining and Detection of Malicious SQL Queries on Anonymization Mechanism. *IEEE Access* 9 (2021), 15015–15027.