USENIX Security '26 Artifact Appendix: Plaintext Recovery Against Post-Filtering Access Control

Zachary Espiritu
MongoDB Research

David Cash
University of Chicago

A Artifact Appendix

A.1 Abstract

In this paper, we show how to amplify side-channels in *fine-grained access control* database mechanisms such as *row-level security* (RLS) or *document-level security* (DLS) to expressive record reconstruction by using rich query interfaces (e.g., ranges, prefixes). We implement a family of CLI utilities called `unfilter-*` which implement these attacks:

- **`unfilter-rls` (RLS).** In Section 3 of the paper, we exploit a timing side-channel and expressive SQL queries (e.g., ranges, conjunctions) to enumerate unknown attribute values in *relational databases* like PostgreSQL. We then show how to recover full records via binary search over large domains. Our tool implements these attacks.
- **`unfilter-dls` (DLS).** In Section 4 of the paper, we exploit scoring and prefix-expansion side-channels to recover indexed terms from documents in *full-text search databases* like Elasticsearch and OpenSearch. Under certain index configurations, we can also extract n-grams in the corpus to recover approximate text.

Both artifacts are designed to be extensible and usable for not just future research projects, but also for industry practitioners who are interested in seeing how applicable these attacks may be in their system. Each artifact contains documentation on how to add new evaluation system functionality (in addition to regression tests and strong code typing and formatting requirements) as well as how to extend it to other scenarios.

It is helpful to think of this artifact as two independent artifacts. We distinguish between the **RLS** and **DLS** sections in the Artifact Appendix using highlighted colors.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact runs all attack experiments on *isolated* systems, though in theory one could adapt our scripts to attack live systems. We discuss more details about our ethical analysis in the Ethical Considerations section of our paper.

For evaluators, the **RLS attacks** require a large number of network requests between the target database host and the attacker host. This will introduce a high amount of network load, so evaluators should be careful with the network environment in which they choose to reproduce our results. They also require a reasonably high degree of parallelism (up to 16 workers) and memory (up to 64 GB) which can potentially overwhelm system resources and cause crashes.

The **DLS attacks** can either be run between two host machines (over a network) or locally in Docker containers, though they also require a high amount of memory (up to 64 GB). See Section A.2.3 for details.

A.2.2 How to access

Our artifacts are accessible for permanent access on Zenodo at `https://doi.org/10.5281/zenodo.20370280`.

A.2.3 Hardware dependencies

The **RLS** experiments exploit timing side-channels whose accuracy is sensitive to noise and hardware. Evaluators are welcome to run the **RLS** experiments on their own setup, but they may get substantially different results (due to noise) unless they use equivalent architecture to the VMs described below. On the other hand, the **DLS** experiments can be run on a single host, though the largest experiments require ≈64 GB of RAM.

While the **RLS** experiments can run on any hardware, replicating our specific accuracy results requires *dedicated-core* machines like the `c4` family of Google Compute Engine (GCE) VMs that we use in the paper. Specifically:

- *Database servers* run on a `c4-standard-16` VM with 16 vCPUs, 60 GB RAM, and a 200 GB local SSD.
- The *attacker* is located on a `c4-highmem-16` VM with 16 vCPUs, 128 GB RAM, and a 100 GB disk.
- Some experiments use a *noise* `c4-standard-96` VM with 96 vCPUs, 128 GB RAM, and a 100 GB disk.

The **DLS** experiments are not sensitive to noise and can be run on any hardware that meets the memory requirements. The primary focus for the **DLS** claims is on query counts and accuracy which should be essentially the same on all setups.

Nevertheless, for consistency (and to capture reasonable wall-clock time numbers), the paper's **DLS** results were run on the following GCE VMs:

- *Database servers* run on a `c4-standard-16` VM with 16 vCPUs, 60 GB RAM, and a 200 GB local SSD.
- The *attacker* is located on a `c4-standard-16` VM with 16 vCPUs, 60 GB RAM, and a 200 GB local SSD.

A.2.4 Software dependencies

The artifact provides scripts which automatically install all necessary dependencies. These scripts have been primarily tested on clean installs of **Debian 13** and **Ubuntu 24.04**. See Section A.3.1 for installation instructions.

A.2.5 Benchmarks

The **RLS** experiments require the use of a synthetic electronic medical record dataset which is provided in the artifact. The **DLS** experiments require the use of truncated Enron datasets which are provided in the artifact. Scripts for generating these datasets are in the artifact.

A.3 Set-up

A.3.1 Installation

Installing RLS. Start by downloading the artifact from Zenodo. Then, unzip the artifact and enter the `rls` directory:

```
cd fgac-main/rls
```

Then, install all **RLS** dependencies via:

```
bash setup.sh
```

You will need to either run the `source` command printed at the end of the `setup.sh` script or restart your terminal. Finally, authenticate the `gcloud` CLI to your Google Cloud credentials:

```
gcloud auth login
```

While we recommend the integrated Google Cloud orchestration features in `unfilter-rls` for easier reproducibility, see Section A.6 if you want to run the **RLS** artifact on machines that do not involve Google Cloud.

Installing DLS. Start by downloading the artifact from Zenodo. Then, unzip the artifact and enter the `dls` directory:

```
cd fgac-main/dls
```

Then, install all **DLS** dependencies via:

```
bash setup.sh
```

You will need to either run the `source` command printed at the end of the `setup.sh` script or restart your terminal.

A.3.2 Basic Test

Kick-the-tires on RLS. To make sure your setup is working and that your account has the ability to provision Google Compute Engine VMs, first check that your environment has been properly setup:

```
unfilter-rls doctor
```

If all rows print "OK", then you can run a quick experiment (**E-R9**):

```
unfilter-rls claims run C-R9
# also supports 'unfilter-rls claims run 9'
```

This will take about 20–25 compute-minutes to run (most of this time is spent provisioning and tearing down Google Cloud VMs). After this runs, it should write this file:

```
rls/results/dbsize/summary.txt
```

Verify that the numbers align with the statistics reported in the **"Schema and data" paragraph in Section 3.3** of the paper. If this works, you should be ready.

Kick-the-tires on DLS. To make sure your setup is working, first check that your environment has been properly setup:

```
unfilter-dls doctor
```

If all rows print "OK", then you can start running experiments. For convenience, we provide a one-shot script which runs all of the DLS experiments in sequence. To quickly test that it works, you can run the DLS attack pipeline on only the smallest dataset ($|D| = 1$):

```
bash dls/run.sh --datasets 1
```

This will take about 5 compute-minutes to run. This should produce the following files:

```
dls/results/reviewer/opensearch_table.tex
dls/results/reviewer/elasticsearch_table.tex
```

The rows should match the first $|D| = 1$ row of **Table 6** in the paper. If this works, you should be ready.

A.4 Evaluation workflow

For readability, we organize our claims and experiments by whether they are related to the **RLS (R)** or **DLS (D)** attack.

A.4.1 Major Claims

(C-R1): Authorized and unauthorized queries are reliably longer than non-existent queries under join-based policies. This is proven by the experiment (**E-R1**) described in Section 3.1 whose results are illustrated in **Figure 2**.

(C-R2): The RLS timing side-channel is preserved under expressive queries. This is proven by the experiment (**E-R2**) described in Section 3.2 whose results are illustrated in **Figure 3**.

(C-R3): For **join**-based policies, $k = 1,2,3,4$ suffices for $> 99.9\%$ accuracy at CPU $\leq 95\%$. **Inline**-based policies are significantly noisier. When load decreases, fewer queries suffice. This is proven by the experiment (**E-R3**) described in Section 3.3.1 whose results are in **Table 1**.

(C-R4): Cross-region latency impacts accuracy but can be mitigated by increasing k. This is proven by the experiment (**E-R4**) described in the "Physical distance" paragraph in Section 3.3.1 whose results are in **Table 2**.

(C-R5): *Transparent data encryption* does not change the timing differences we exploit. This is proven by the experiment (**E-R5**) described in the "Transparent data encryption" paragraph in Section 3.3.1. There are no figures in the paper directly associated with this experiment, but the experiment should produce figures that look very similar to **Figure 2** and **Figure 3**.

(C-R6): When $|D| \gg |S|$ (e.g., `ssn`), binary search yields large speedups over linear probing. When $|D| \approx |S|$ (e.g., `zip_code`, `age`), linear probing is already practical and binary search slows the attack. Parallelism speeds up the attack, but too many threads decrease performance due to increased (self-imposed) DB load. This is proven by the experiment (**E-R6**) described in Section 3.3.2 whose results are in **Table 3**.

(C-R7): Attack performs better when starting with most selective attributes first. The low selectivity of `age` and `zip` introduces timing variance which results in loss of accuracy. This is proven by the experiment (**E-R7**) described in Section 3.3.2 whose results are in **Table 4**.

(C-R8): The *subquery + composite index* approach closes the side-channel for our scenario. Adding just a composite index *or* only changing the policy to a subquery is not sufficient to mitigate the attack. This is proven by the experiment (**E-R8**) described in Section 3.4 whose results are in **Table 5**.

(C-R9): The full `patients` and `doctors` database occupies 185 MB on disk (79 MB heap, 37 MB `ssn` index, 11 MB `zip_code` index, 7 MB `age` index, 51 MB other indexes). This is proven by the experiment (**E-R9**) whose results are in the **"Schema and data" paragraph in Section 3.3**.

(C-D1): On OpenSearch, the DLS attacks recover the indexed n-gram set exactly across all tested corpus sizes ($|D| \in \{1, 10, 100, 1000\}$). Injections and queries scale linearly with the # of trie nodes. This is proven by the experiment (**E-D1**) described in Section 4.5 whose results are in **Table 6**.

(C-D2): The attack produces identical results on Elasticsearch as on OpenSearch. This is proven by the experiment (**E-D2**) described in Section 4.5. There are no figures in the paper directly associated with this experiment, but the experiment should produce results very similar to **Table 6**.

(C-D3): The de Bruijn text traversal on the 4-grams recovers approximate document text. While text is not complete, the overall meaning is preserved. This is supported by the experiment (**E-D3**) described in the "Document reconstruction" paragraph in Section 4.5 from which sample results are in **Figure 5**.

A.4.2 Experiments

(E-R_): Run all RLS experiments *[20 human-minutes + 36 compute-hours]*

For evaluator convenience, our **RLS** artifact provides a command which runs all of the **RLS** experiments in a single pipeline.

Execution: This runs the full experiment pipeline:

```
unfilter-rls claims run 1,2,3,4,5,6,7,8,9
```

The script displays frequent progress updates in the terminal. We recommend the use of a terminal detacher like `screen` if you wish to close your terminal while the script is running. (See the top-level `README.md` in the artifact for a guide on using `screen`.)

This script runs the following experiments (the per-experiment time estimates below exclude VM provisioning and teardown time, as those provisioning steps only happen once when running all experiments in the same `unfilter-rls` command):

(E-R1): Equality microbenchmark (Figure 2) *[5 compute-minutes]*

This experiment verifies that the timing side-channel is preserved with *equality queries* under the *join*-based policy described in Section 3.3.

Results: Results are written as a rendered TEX figure. The TEX source code is:

```
rls/results/existence/existence_kde.pgf
rls/results/existence/
existence_kde_figure.tex
```

Compare this to **Figure 2** in the paper.

(E-R2): Range query microbenchmark (Figure 3) *[5 compute-minutes]*

This experiment verifies that the timing side-channel is preserved with *range queries* under the *join*-based policy described in Section 3.3.

Results: Results are written to:

```
rls/results/range/existence_range_kde.pgf
rls/results/range/
existence_range_kde_figure.tex
```

Compare this to **Figure 3** in the paper.

(E-R3): CPU load vs accuracy (Table 1) *[4 compute-hours]*

This experiment checks how the accuracy of the timing oracle changes based on CPU load introduced by a 3rd *noise* VM.
Results: Results are written to:

```
rls/results/table1/table1.pdf
```

Compare this to **Table 1** in the paper—please be aware that due to rate limiting that we cannot control in this *specific* GCE environment, less load than anticipated may occur on higher CPU loads (e.g., 75%, 85%, and 95%) for the *inline* policy which may result in higher accuracy than expected. (The claim's main focus is the *join* experiments.)

Optional (E-R3): increase # of samples. By default, `unfilter-rls claims run C-R3` only takes 10k samples for each cell. In the paper, we use 100k samples (which tightens the confidence intervals). You can change number of samples by overriding the default `table1.probes` option:

```
unfilter-rls claims run C-R3 \
  --set table1.probes 100000
# Warning: this will 10x the
# compute-hour estimate
```

(E-R4): Cross-region latency vs accuracy (Table 2) *[12 compute-hours]*
This experiment performs the cross-region experiments in Table 2, with an additional factor of CPU load introduced by a 3rd *noise* VM.
Results: Results are written to:

```
rls/results/table2/table2.pdf
```

Compare this to **Table 2** in the paper.

Optional (E-R4): increase # of samples. By default, `unfilter-rls claims run C-R4` only takes 1k samples for each cell. In the paper, we use 10k samples (which tightens the confidence intervals).[a] You can change number of samples by overriding the default `crosszone.probes` option:

```
unfilter-rls claims run C-R4 \
  --set crosszone.probes 10000
# Warning: this will 10x the
# compute-hour estimate
```

[a] The number of samples for the cross-zone experiments is 10× less than the same-zone experiments as the round-trip time is about 10× higher due to the network latency.

(E-R5): Transparent data encryption (no figures) *[10 compute-minutes]*
This experiment validates that TDE does not change the efficacy of the timing attack. It is in a separate script as it requires reprovisioning the VMs to install LUKS.
Results: Results are written to:

```
rls/results/existence/
existence_tde_figure.pgf
rls/results/existence/
existence_tde_figure.tex
```

Verify that the two figures look visually similar to **Figure 2** and **Figure 3** in the paper.

(E-R6): Binary search / parallelism for attribute domain recovery (Table 3) *[8 compute-hours]*
This validates that the binary search attack (with optional parallelism) outperforms the prior linear attacks on large, sparse domains. It shows it performs slightly worse on dense domains. It also shows that too much parallelism can lower the accuracy due to self-imposed CPU load on the database.
Results: Results are written as a LaTeX file to:

```
rls/results/table3/table3.tex
```

Compare this to **Table 3** in the paper. For the higher parallelism experiments (workers > 4), the additional variance and binary search nature of the attack means that accuracy may be occasionally lower than the averages reported in the paper (e.g. if the attack got unlucky at a higher tree level and mistakenly pruned much of the search space). (This is why we run each row 10 times in the paper.)

Optional (E-R6): increase # of repetitions. By default, `unfilter-rls claims run C-R6` runs each experiment row 1 time. In the paper, we use 10 repetitions (which tightens the confidence intervals). You can change number of repetitions by overriding the default `singleattr.reps` option:

```
unfilter-rls claims run C-R6 \
  --set singleattr.reps 10
# Warning: this will 10x the
# compute-hour estimate
```

(E-R7): Selectivity ordering vs tuple reconstruction accuracy (Table 4) *[12 compute-hours]*
This experiment validates our theoretical analysis of how the accuracy / performance of the attack changes based on the order of the attributes used in the tuple reconstruction algorithm.
Results: Results are written as a LaTeX file to:

```
rls/results/table4/table4.tex
```

Compare this to **Table 4** in the paper. For the reconstruction orderings starting with `zip` and `age`, you sometimes will get significantly higher *Recall* and *Precision* than what is in the paper due to inherent

variance in the binary search and oracle accuracy, but this should come at the cost of significantly higher *Queries* and *Time* because the lack of accuracy results in more false positives, resulting in unnecessary trie traversals. (This is why we run each row 3 times in the paper.) The most important factor to check for is that *ssn*-leading orderings should outperform the *zip*- or *age*-leading orderings in either Recall / Precision, or Queries / Time.

Optional (E-R7): increase # of repetitions. By default, `unfilter-rls claims run C-R7` runs each experiment row 1 time. In the paper, we use 3 repetitions (which tightens the confidence intervals). You can change the number of repetitions by overriding the default `tuplext.reps` option:

```
unfilter-rls claims run C-R7 \
  --set tuplext.reps 3
# Warning: this will 3x the
# compute-hour estimate
```

(E-R8): Mitigation ablation experiments (Table 5) *[5 compute-minutes]*
This experiment validates our analysis of the necessary components needed to mitigate the attack.
Results: Results are written to:

```
rls/results/table5/table5.pdf
```

Compare this to **Table 5** in the paper.

(E-R9): Dataset statistics ("Schema and data" paragraph) *[1 compute-minute]*
This experiment validates some summary statistics about the datasets used in the experiments.
Results: Results are written to:

```
rls/results/dbsize/summary.txt
rls/results/dbsize/db_sizes.json
```

Verify the statistics align with the **"Schema and data" paragraph in Section 3.3** of the paper.

(E-D_): Run all DLS experiments *[5 human-minutes + 16–18 compute-hours + 64GB disk + 60GB memory]*
For evaluator convenience, we provide a script which runs all of the **DLS** experiments in a single pipeline.
Execution: We provide a one-shot script which runs all of the DLS experiments in sequence. This command runs the full experiment pipeline,[1] which will take 16–18 compute-hours:

```
bash dls/run.sh
```

[1] For convenience, the script will cache results for dataset runs after they have successfully completed (e.g., in case you accidentally stop the pipeline). If you would like to rerun existing results, add the `--destructive-rerun-existing-results` flag.

Optional (E-D_): shorter / less memory experiments. To reduce time to 2–3 compute-hours and the required RAM (< 16 GB RAM), you can skip the longer $|D| = 1000$ run by only picking the $|D| \in \{1, 10, 100\}$ datasets:

```
bash dls/run.sh --datasets 1,10,100
```

Optional (E-D_): increase # of reps. By default, the `run.sh` script runs 1 run of each attack. In the paper, we ran each attack 3 times to take wall-clock time averages. You can specify the # of reps using the `--trials` flag:

```
bash dls/run.sh --trials 3
# Warning: this will 3x the compute-
hour estimate
```

This script runs the following experiments:

(E-D1): Run attack on OpenSearch (Table 6)
This experiment verifies that the DLS attacks recover the indexed n-gram set exactly across all tested corpus sizes ($|D| \in \{1, 10, 100, 1000\}$) on OpenSearch 3.6.0.
Results: Results are aggregated and written to

```
dls/results/reviewer/opensearch_table.tex
```

Verify that all columns are similar to **Table 6** in the paper. In particular, "Logical Queries" should be identical.

Optional (E-D1): raw data. If you want to see the raw statistics (not yet rendered into TEX), see the `dls/results/reviewer/stats` folder. If you want to see the full attack logging output, see the `dls/results/reviewer/logs` folder.

(E-D2): Run attack on Elasticsearch (Table 6)
This experiment verifies that the DLS attacks recover the indexed n-gram set exactly across all tested corpus sizes ($|D| \in \{1, 10, 100, 1000\}$) on Elasticsearch 9.3.3.
Results: Results are aggregated and written to

```
dls/results/reviewer/elasticsearch_table.
tex
```

Verify that all columns are similar to **Table 6** in the paper. In particular, "Logical Queries" should be identical.

Optional (E-D2): raw data. If you want to see the raw statistics (not yet rendered into TEX), see the `dls/results/reviewer/stats` folder. If you want to see the full attack logging output, see the `dls/results/reviewer/logs` folder.

(**E-D3**): **Validate de Bruijn reconstruction produces "reasonable" results (Figure 5)**
This experiment demonstrates the ability of the greedy de Bruijn reconstruction to reconstruct "reasonable" approximate document text. It specifically operates against the recovered 4-grams from the individual dataset runs done as part of (**E-D1**) and (**E-D2**).
Results: Results are written to

```
dls/results/reviewer/reconstruction/
opensearch-d10-r1_debruijn_greedy.txt
```

Verify that `contig_2` and `contig_6` match the contents of **Figure 5**. If interested, you can view other reconstructions from the $|D| = 10$ attacks in the same file, or view reconstructions from other attacks in the other files in `dls/results/reviewer/reconstruction`.

A.5 Troubleshooting Guide

- **RLS** **`ZONE_RESOURCE_POOL_EXHAUSTED` or `STOCKOUT`:** This error occurs when Google Cloud is out of `c4-standard-*` VMs in a particular zone (likely the default zone of `us-central1-a` used in the experiments). You can change the zone by the `--set zone` configuration option. For example:

```
unfilter-rls claims run C-R9 \
  --set zone us-east1-b
```

- **DLS** **`UNEXPECTED_EOF_WHILE_READING`:** This error is usually caused by the database server crashing due to an out-of-memory (OOM) error. Make sure that your system meets the minimum memory requirements specified for the experiment.
- **DLS** **`index_create_block_exception`:** This error is usually caused by the database server preventing additional uploads due to insufficient disk space. Make sure that your system meets the minimum storage requirements specified for the experiment.

A.6 Notes on Reusability

For **RLS**:

- Please see the "Bring Your Own Machines" in the `README.md` if you would like to run the **RLS** experiments on different (non-GCE) machines. It involves creating a new configuration file that points to your chosen machines.

For **DLS**:

- `dls/setup.sh`, `dls/run.sh`, and `dls/teardown.sh` helper scripts have an optional `--help` flag which provides additional options for running the scripts.
- `dls/run.sh` invokes the `python -m enumerate` module, which itself has several command-line options for changing the datasets to attack or different attack options. For more details, see the documentation in `dls/enumerate/README.md`.
- `dls/dataset/util/parse_hf_dataset.py` can be used to generate more dataset files from other Hugging Face datasets for testing the attack on other datasets. See the `dls/dataset/README.md` for more details.
- `dls/config/config.yml` contains an `attack` block which controls parameters used in the experiments. For example, the `chars` field controls the set of characters used to explore the term trie. You can change the `chars` field to see how that affects the runtime of the attack.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.