Design and Implementation of a Privacy-Preserving Database on PDC

Andrey Ermolinskiy

1 Introduction

Regulating access to electronically stored personal information is an increasingly challenging task and significant concerns about the privacy of personal data have emerged. These concerns are fueled, in part, by the ever-growing number of highly-publicized security incidents involving data theft and privacy violations.

As the most recent example, UC Berkeley has discovered that foreign attackers breached the databases maintained by the University Health Services and gained unauthorized access to large volumes of personal data including names, birth dates, and social security numbers [5]. As a consequence of this security breach, approximately 160,000 current and former University students have been exposed to the risk of identity theft.

One of the defining principles of information privacy is the notion of *limited disclosure*, which stipulates that individuals and organizations should have control over who is allowed to see their private information and for what purpose. Such information may not be revealed for purposes other than those for which there is consent from the owner of the information.

The issues of privacy and information disclosure in database management systems have received a significant amount of research attention. In the context of relational databases, the principle of limited disclosure implies that the personal information stored in a database may be revealed to database users through queries only in accordance with the privacy preferences specified by the owner of this information. These preferences are essentially a set of rules that describe to whom data may be disclosed (*recipients*) and how it may be used (*purposes*).

[2] proposes re-architecting our database management systems to include responsibility for the privacy of data as a fundamental tenet. This paper outlines the ten principles for *privacy-preserving* (*Hippocratic*) databases which include, among others, *purpose specification*, *consent*, and *limited disclosure*. In [4], the authors demonstrate how the limited disclosure principle can be realized within the confines of a traditional RDBMS architecture. In the *table semantics* model of limited disclosure, each $\langle Purpose, Recipient\rangle$ pair is conceptually assigned a unique view over the entire database, in which prohibited attributes are masked with the NULL value. To implement these semantics, the authors propose storing the privacy preferences (and other policy metadata) in relational form and modifying incoming queries with *CASE* statements to enforce the rules and conditions expressed in the privacy metadata.

While this approach is undoubtedly attractive due to its simplicity and ease of implementation, a purely database-level solution would only partially address the major security challenges. An unscrupulous user with administrative privileges can easily disable or circumvent the query rewriting mechanisms, for example by reading the raw (unfiltered) table contents directly from the database files in the underlying filesystem. Furthermore, the output of a query submitted on behalf

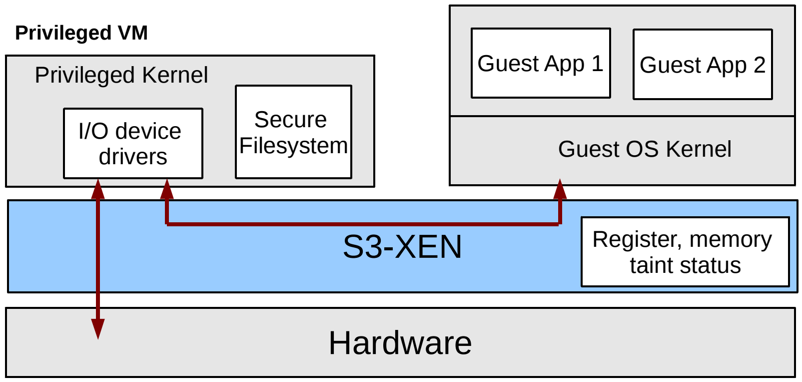


Figure 1: The high-level organization of PDC.

of an authorized user may get retained and stored in an external format (such as a text file or a spreadsheet document). In this case, enforcing end-to-end privacy guarantees would necessitate additional application- and format-specific mechanisms for associating the content of the document with sensitive records in the database from which this content was derived, as well as for enforcing the relevant privacy policies.

In this report, we explore an alternate approach that leverages the PDC platform [3] to enforce policies on private data *below* the database, effectively at the hardware level. PDC relies on fine-grained taint tracking to monitor the propagation of sensitive data and enforces policies at the time of externalization.

The rest of this report is structured as follows. Section 2 provides a high-level overview of the PDC architecture and describes the central tradeoffs. Section 3 outlines the general design for a privacy-preserving relational database atop the PDC platform. Section 4 describes our proof-of-concept prototype based on Postgres [1] - an open-source RDBMS implementation. Finally, Section 5 evaluates our prototype and Section 6 concludes.

2 Overview of PDC

2.1 High-Level Design

Practical Data Confinement (PDC) is a hypervisor-based platform that seeks to provide end-to-end guarantees of security and the ability to enforce user-defined policies on the movement and exposure of sensitive data. Our platform can prevent unauthorized exchanges of data between pairs of network endpoints or, more broadly, between organizations or between an individual and an organization.

Figure 1 illustrates the high-level architecture of PDC. In broad terms, we interpose a thin functional component between the hardware and the operating systems, which enables us to track the movement of sensitive data thorough the system and prevent unauthorized externalization through I/O devices. PDC's architecture is similar to hypervisors such as Xen (and indeed, our current prototype implementation is based on Xen). The OS and applications with which the user interacts operate within a guest virtual machine (VM). In addition each PDC-enabled machine runs a minimal privileged operating system, which offers limited services to the hypervisor and the guest VMs, including access to I/O devices. Our platform is defined by two core components:

1. **Policy specification**: Each abstract user-level data object is named with a unique identifier (denoted by *DataID*). These identifiers carry no inherent meaning, but they conceptually associate an object with some category of privacy. Each data object O has an *owner* (denoted $O.owner$) who can specify how the information derived from O can be handled by attaching *policies* to O. In our current design, a policy is conceptually a 3-dimensional array that associates an $\{ALLOW/DENY\}$ action with 3-tuples of the form $\langle Recipient, Purpose, Channel\rangle$. The *Channel* component indicates the I/O channel and typically corresponds to an I/O device class (e.g., screen, USB-attached storage, printer, etc). For instance the policy

   ```
   ALLOW <Alice, *, <NIC, Screen, Printer>>
   ```

   on data object O would permit Alice to receive sensitive data derived from O from the network and externalize it via her screen and/or her local printer. As another example, a user can send e-mail to someone and apply a "no-forward" policy that would prevent the recipient from sending any e-mail that contains any data derived from the sensitive attachment.

2. **Fine-grained information flow tracking**: The PDC hypervisor implements a fine-grained instruction-level taint tracking mechanisms, which monitors the flow of sensitive data inside the guest VM and tracks its movement between registers, memory, and local disk with byte-level granularity. Each byte b of physical memory and disk addressable by the guest VM is conceptually associated with a *TaintTag*, which stores a list of *DataID*s corresponding to data objects from which the value of b may have been derived.

   When an application opens a file with sensitive content, PDC lifts the corresponding *TaintTag*s from disk and applies them to memory buffers associated with the open file. When an application touches a piece of physical memory associated with sensitive content, the hypervisor traps the access (using paging hardware), examines the next instruction, and updates the tainting metadata accordingly.

   When the guest VM makes an attempt to generate externally-observable output (e.g., showing data on the screen, writing data to a USB key, sending a packet), the hypervisor inspects the set of *TaintTag*s associated with the contents of output buffers and checks them for policy compliance.

2.2 Central Tradeoffs

The architecture described above has several attractive properties. First, since the taint-tracking and policy enforcement mechanisms reside below the operating system, our scheme permits the guest VM to execute untrusted and possibly malicious code. Furthermore, PDC requires no changes to existing operating systems and application and thus, faces a relatively low barrier to adoption.

At the same time, our scheme faces several significant challenges. First and perhaps most importantly, fine-grained taint-tracking imposes a considerable performance overhead. To keep the overhead at a manageable level, we expect to be able to leverage two techniques: speculative execution and parallelized passive taint tracking. The key insight is that we need access to byte-level taint information only at the time of externalization. Hence, it is sufficient to asynchronously track byte-level information flow and allow the guest VM to speculatively execute at native speed.

We also expect to be able to leverage extra computing capacity available due to the presence of multiple CPU cores.

The second hurdle pertains to the storage overhead of policy metadata and *TaintTag*s. As noted above, our basic unit of granularity for taint tracking purposes is a byte - the smallest individually addressable unit of memory. Conceptually, PDC associates a *TaintTag* with each byte of physical memory addressable from the guest machine, each byte in its virtual CPU registers, and each byte on its virtualized disk. Our current design seeks to achieve a balance between speed (efficiency of lookups and modifications) and the storage overhead.

2.3 Key Data Structures

2.3.1 Maintaining the memory taint status

PDC uses two distinct data structures to maintain the taint status of 'physical virtual' memory seen by the guest VM. For each page of memory, PDC maintains a small 64-bit data structure (*PageTaintSummary*), which holds a concise summary of taint labels in the corresponding page. This data structure enables us to efficiently answer queries from the shadow paging code (i.e., "is page X tainted with sensitive data?").

We use a 4-level tree data structure to resolve a 32-bit machine page number into the corresponding *PageTaintSummary* instance. Each tree node occupies a single memory page (4KB). A non-leaf (index) node stores an array of 32-bit page numbers that point to children (1024 items per node). A leaf tree node stores an array of *PageTaintSummary* structures (512 items per node).

The *PageTaintSummary* data structure, in turn, stores a pointer to a *page taint descriptor* (*PageTaintDescr*), which enables us to determine the *TaintTag* for each individual byte in the corresponding page. These descriptors are represented in one of several formats, which trade off storage overhead and lookup efficiency. These formats include:

1. **Uniform format**: The page taint descriptor consists of a single *TaintTag*. This format enables us to represent uniformly-tainted pages in a space-efficient manner.

2. **RLE format**: The page taint descriptor represents byte-level taints using run-length encoding (essentially a list of $\langle Length, TaintTag \rangle$ pairs). This format is most appropriate for pages that exhibit a low-to-moderate degree of taint fragmentation, as well as spatial locality.

3. **Taint array format**: The page taint descriptor holds a fixed-length array of *TaintTag*s with one entry for each individual byte within the page. This format is suitable for pages that exhibit a high degree of taint fragmentation.

2.3.2 Maintaining the disk taint status

We have implemented a specialized taint-aware filesystem (based on ext3) that enables us to retain associations between data regions and *TaintTag*s when data objects move between memory and disk. Analogously to memory region tainting, we maintain a *BlockTaintSummary* data structure for each file data block and this data structure, in turn, holds a pointer to a *block taint descriptor* (*BlockTaintDescr*), which enables us to resolve taint values with byte-level granularity. The *BlockTaintSummary* data structures are maintained as part of filesystem metadata inside ext3 indirect blocks, along with pointers to the corresponding data blocks. The block taint descriptors are implemented using the formats introduced above (i.e., *Uniform*, *RLE*, and *Taint array*) and are

ID	Name	Age	Address	Phone
1	Alice A.	10	1 April Ave.	111-1111
2	Bob B..	20	2 Brooks Blvd.	222-2222
3	Charles C.	30	3 Cricket Ct.	333-3333
4	David D.	40	4 Dogwood Dr.	444-4444

Table 1: Complete data table of patient information.

ID	D_ID	D_Name	D_Age	D_Address	D_Phone
1	yes	yes	yes	yes	yes
2	yes	yes	no	yes	yes
3	yes	no	no	no	yes
4	yes	yes	no	no	no

Table 2: Patients' information disclosure preferences for $Recipient = Charity$, $Purpose = Solicitation$.

maintained on a separate block device or a separate disk partition. The taint-aware filesystem is deployed in the privileged virtualization domain and has direct access to local storage device(s). The guest (unprivileged) VM communicates with the taint-aware filesystem using the NFS protocol and a shared memory ring transport.

3 Design of a Privacy-Preserving Database

In this section, we present the high-level design for a privacy-preserving database that leverages the PDC platform. In this design, PDC is responsible for storing and evaluating the disclosure policies associated with sensitive database records. Each sensitive attribute value corresponds to an PDC *data object* and is assigned a unique *DataID* by its owner.

We begin by formalizing and illustrating the semantics of limited disclosure that we wish to provide. Subsection 3.2 discusses the mechanisms for data entry and policy specification. Finally, Subsection 3.3 describes query evaluation.

3.1 Model of Limited Disclosure

Our design implements the *table semantics* model of cell-level limited disclosure enforcement, as defined in [4]. In this model, each $\langle Purpose, Recipient\rangle$ pair is conceptually assigned a view over each data table based on the disclosure constraints specified by privacy preferences. These views combine to produce a coherent version of the original database for each $\langle Purpose, Recipient\rangle$ pair, whereby prohibited attributes are masked using the NULL value. We refer the reader to [4] for a formal specification.

To illustrate this definition, consider Table 1, which shows a hypothetical patient information

ID	Name	Age	Address	Phone
1	Alice A.	10	1 April Ave.	111-1111
2	Bob B..	NULL	2 Brooks Blvd.	222-2222
3	NULL	NULL	NULL	333-3333
4	David D.	NULL	NULL	NULL

Table 3: Privacy-preserving version of the patient information table.

dataset in a hospital database. Table 2 lists the patients' choices for disclosure of private information to charities for solicitation purposes. The contents of this table indicate that Alice has agreed to disclose every attribute of her record, Bob has chosen to disclose every attribute with the exception of his age, and so forth. Table 3 shows the privacy-preserving patient dataset according to table semantics that results when the privacy preferences are applied to the original data in Table 1.

3.2 Data Insertion

When a new record with sensitive attributes is inserted into a database, we must tag the appropriate locations in memory and on disk (holding the sensitive attribute values) with the corresponding *DataID*s. (As we explain in Section 2, these identifiers establish the association between a sensitive data item and an externalization policy).

In the typical mode of operation, the memory buffer holding the contents of an SQL INSERT statement would already be tainted with the corresponding *DataID*s at the time of its arrival to the database backend process. The initial tainting occurs upon the entry of sensitive information into the system from a user-facing input device (such as a keyboard). For instance, during the patient registration process, the new patient would be asked to enter her information into an electronic registration form. As the patient proceeds with the registration, she would indicate the attribute sensitivity identifiers via the PDC hypervisor (i.e., *"the data I am about to enter via the keyboard constitutes my home address. This data object should be labeled with the MyAddress identifier"*).

The taint-tracking mechanisms in PDC assume the responsibility for propagating these tags into the body of an SQL INSERT statement. During the execution of this statement by the database backend process, the sensitive attribute values (and the associated *DataID*s) propagate through the various implementation-specific data structures and eventually make their way into the buffer cache and persistent data structures on disk (e.g., relation heap pages, index pages, the write-ahead log).

It is important to note that in the above scheme, the propagation of sensitivity identifiers into persistent DBMS-level data structures occurs without any involvement from the database backend process and is handled entirely by the PDC hypervisor.

Another approach, which we found useful for testing and debugging, involves extending the SQL language syntax in a manner that enables database clients to explicitly annotate the body of an INSERT statement with PDC data identifiers, for example as follows:

```
INSERT INTO Patients VALUES (1              {AliceID},
                             'Alice'        {AliceName},
                             20             {AliceAge},
                             '1 April Ave.' {AliceAddress},
                             '111-1111'     {AlicePhone}
                            );
```

In this scheme, the database backend process is responsible for tagging the memory regions holding sensitive attribute values with the corresponding *DataID*s during the query parsing stage.

3.3 Query Evaluation

Recall that the table semantics model of limited disclosure enforcement requires the database to expose an alternate view of each data table to each $\langle Purpose, Recipient \rangle$ pair, in which disallowed attribute values are replaced with a NULL value. These semantics can be implemented via a conceptually straightforward extension to the query execution engine. More concretely, we modify the scan operators at the leaves of the execution plan tree (e.g., *SequentialHeapScan*, *IndexScan*, ...) , which are responsible for scanning input relations and returning a stream of raw tuples. Assuming that these operators expose the conventional *iterator* interface, we modify their *GetNext()* method to perform the filtering of prohibited attribute values prior to returning a tuple. For each potentially sensitive attribute *A*, the operator determines the virtual address of the memory region holding its value (typically a pointer into the buffer cache or one of the auxiliary data structures) and issues a request to PDC to obtain the $TaintTag$ associated with that memory address. Another request to PDC resolves the $TaintTag$ into a $DataID$ and the associated externalization policy specified by the owner of *A*. We evaluate this policy for the *recipient* (on whose behalf the query was issued) and the intended *purpose* (as specified by the recipient). If the policy prohibits revealing the value of *A* to the specified recipient and for the specified purpose, we mask the contents of *A* with a NULL value prior to returning the tuple.

It is crucial to note that while our design requires augmenting the database engine with the attribute filtering mechanism described above, the responsibility for actually enforcing policy compliance remains with the PDC hypervisor. The scan operators query the PDC platform to determine which of the attributes in the next tuple *must* be filtered out in order to be permitted to externalize the query output. In the event that the database process fails to issue the appropriate set of calls to PDC or fails to correctly replace the prohibited attribute value in its memory buffers with NULL, the hypervisor would detect tainted data in the output and preclude the guest VM from exposing it.

4 Implementation

We have implemented a proof-of-concept prototype based on the open-source Postgres [1] database engine and in this section, we briefly describe our modifications and extensions to Postgres. It is important to note that some of the central elements of the PDC architecture are still under active development and a fully-featured platform capable of running arbitrary guest applications is not yet available. Our current prototype runs on top of a partial PDC implementation, which includes the following elements:

- The taint-aware filesystem (s3ext3), which keeps track of taint information in data files on disk.
- The memory taint management module, which keeps track of taint information in memory pages using the tree data structure described in Section 3.

An important feature still missing from our partial PDC prototype is the ability to emulate guest instructions and track the propagation of taints between memory locations and CPU registers. For the purposes of this project, we devised a workaround solution which aims to simulate this functionality: we intercept all calls to library functions that manipulate the contents of memory buffers (e.g., routines such as $memset()$ and $memcpy()$), examine their arguments, and manually update the memory taint data structures to reflect the effects of these memory manipulations. As a consequence, the evaluation results presented in the following section do not fully reflect the performance overhead of instruction-level taint tracking that will be observed when we deploy our privacy-preserving Postgres prototype on top of the fully-functional PDC platform.

Recipient and purpose specification: We extended the frontend/backend Postgres protocol, as well as the frontend tools, to provide support for purpose specification. Database users are required to specify the purpose on a per-session basis using a command-line argument, e.g.:

```
> psql -U accounting --purpose billing
```

In our current implementation, the notion of a *recipient* is synonymous with a Postgres user account, which enables us to leverage existing mechanisms for user specification and authentication.

Data insertion: We extended the SQL language grammar and the Postgres parser in a manner that enables users to annotate constant elements in the $VALUES$ clause of an $INSERT$ statement with PDC $DataID$s. These identifiers are specified in curly braces following the actual values and the full set of grammar extensions is provided in Appendix A.

Query evaluation: As we described in the previous section, enforcing the table semantics model of limited disclosure requires extending the leaf operators that return raw tuples from input relations. Prior to returning or operating on the contents of a tuple, the operator calls the PDC runtime library to determine the $TaintTag$ for each of the attributes and the associated policy. We evaluate these policies with respect to the session's recipient and purpose and replace the contents of prohibited attributes with NULL values.

A sample code fragment, which illustrates the requisite set of changes to $slot\_deform\_tuple$ (a function invoked during sequential relation scans) is shown in Appendix B. Similar changes were made to $heap\_deform\_tuple$, $index\_getattr$, $hash\_next$, and $\_bt\_next$ to support access via B-Tree and hash-based indices. Our prototype does not yet provide support for the GiST and GIN index types.

5 Preliminary Evaluation

In this section, we present the results of our preliminary performance study. The primary objective of these experiments is to determine the magnitude of the performance overhead associated with policy enforcement. We measure the query execution time for a modified Postgres backend that implements PDC-driven policy enforcement, as described in Section 4. There are two obvious points of comparison:

1. An unmodified Postgres backend, which executes queries and externalizes results without regard for privacy policies.

2. A modified Postgres backend, which implements the table semantics model of limited disclosure using query modification mechanisms, as described in [4].

5.1 Experimental Setup

We measure the performance of our Postgres-based prototype using a synthetically generated dataset, whose parameters are reported in Tables 4 and 5. The dataset consists of two tables ($DataTable1$ and $DataTable2$) with 10 and 2 attributes, respectively. $DataTable1.IntAttr1$ serves as a foreign key into $DataTable2$ and we construct a B-Tree index on $DataTable2.IntAttr1$.

We generate two sets of privacy policies on the contents of $DataTable1$ for a hypothetical $\langle Purpose, Recipient \rangle$ pair with the parameters shown in Table 6. $PolicySet1$ corresponds to the scenario of full disclosure, whereby every attribute of every record is revealed. In contrast, $PolicySet2$

Attribute	Type	Contents
Id	INT	Unique values in sequential order
IntAttr1	INT	Uniform random values chosen from $[0-999]$
IntAttr2	INT	Uniform random values chosen from $[0-99]$
IntAttr3	INT	Uniform random values chosen from $[0-9]$
StrAttr1	VARCHAR(40)	Random 40-byte text string
StrAttr2	VARCHAR(40)	Random 40-byte text string
StrAttr3	VARCHAR(40)	Random 40-byte text string
BoolAttr1	BOOL	Random boolean value, 10% $= TRUE$
BoolAttr2	BOOL	Random boolean value, 50% $= TRUE$
BoolAttr3	BOOL	Random boolean value, 90% $= TRUE$

Table 4: Components of the benchmark dataset: $DataTable1$ schema.

Attribute	Type	Contents
IntAttr1	INT	Unique values from $[0-999]$ in sequential order
StrAttr1	VARCHAR(40)	Random 40-byte text string

Table 5: Components of the benchmark dataset: $DataTable2$ schema.

represents a much more restrictive privacy configuration, in which the value of a particular attribute is revealed for only a fraction of records.

Our initial evaluation efforts focus on measuring the cost of executing the following queries:

```
Q1:   SELECT * FROM DataTable1;

Q2:   SELECT DataTable1.Id, DataTable2.StrAttr1
          FROM DataTable1, DataTable2
          WHERE DataTable1.IntAttr1 = DataTable2.IntAttr1;
```

Q1 simply returns the full contents of $DataTable1$, subject to partial filtering in accordance with the privacy preferences. Q2 is a somewhat more complex query, which exercises projection and join facilities. We explore the scalability characteristics by running these queries on three distinct instances of $DataTable1$, which contain 10000, 100000, and 500000 records, while keeping the contents of $DataTable2$ fixed.

Our experiments were run on a desktop-grade machine possessing a quad-core AMD Phenom 1.2Ghz processor, three GB of memory, and one ATA hard drive. Our system ran Linux Fedora Core 9 with kernel version 2.6.24-14. Unless otherwise stated, all Postgres configuration parameters were left at their default values. We measure the query execution time using the standard *time* Linux system utility and report the average over 5 iterations. The standard deviation does not exceed 8% of the reported mean value in any of the experiments. We clear out the Postgres buffer cache and the Linux page cache after each iteration.

5.2 Experimental Results

Figures 2(a) and 2(b) report the average query response time for Q1 with $PolicySet1$ and $PolicySet2$, respectively. In these figures, *no enforcement* denotes the behavior of the standard Postgres implementation that does not interact with PDC and externalizes query results without accounting for privacy preferences. *Database-level enforcement* denotes the strategy of enforcing disclosure policies using a database-level mechanism analogous to the one presented in [4]. In this scheme,

Option	PolicySet1	PolicySet2
Reveal ID?	YES	YES for 50% of records
Reveal IntAttr1?	YES	YES for 1% of records
Reveal IntAttr2?	YES	YES for 5% of records
Reveal IntAttr3?	YES	YES for 10% of records
Reveal StrAttr1?	YES	NO
Reveal StrAttr2?	YES	Yes for 50% of records
Reveal StrAttr3?	YES	YES
Reveal BoolAttr1?	YES	Yes for 10% of records
Reveal BoolAttr2?	YES	Yes for 50% of records
Reveal BoolAttr3?	YES	Yes for 90% of records

Table 6: Privacy policies on the contents of $DataTable1$.

the privacy preferences for the contents of $DataTable1$ are explicitly maintained in relational form (i.e., tables $PolicySet1$ and $PolicySet2$)[1]. Incoming queries are modified to perform lookups in the privacy preferences table and selectively filter out prohibited values using the CASE statement. For instance, Q1 (a simple query that returns all records in $DataTable1$) would be rewritten for $PolicySet1$ as follows in this scheme [2]:

```
CREATE VIEW PolicySet1View AS
SELECT
CASE WHEN EXISTS
    (SELECT reveal_Id FROM PolicySet1
     WHERE PolicySet1.Id = DataTable1.Id
     AND PolicySet1.reveal_Id = TRUE)
     THEN Id ELSE NULL END AS id ,
CASE WHEN EXISTS
    (SELECT reveal_IntAttr1 FROM PolicySet1
     WHERE PolicySet1.Id = DataTable1.Id
     and PolicySet1.reveal_IntAttr1 = TRUE)
     THEN IntAttr1 ELSE NULL END AS IntAttr1 ,
...
FROM DataTable1;

SELECT * FROM PolicySet1View;
```

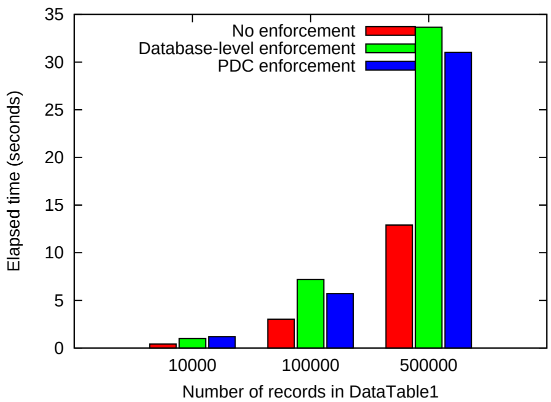
Finally, *PDC enforcement* represents the PDC-driven scheme, which relies on a modified Postgres implementation and the mechanisms presented in Sections 3 and 4.

We observe a significant (and expected) increase in query response time for both privacy enforcement schemes, but the modified Postgres implementation that leverages the PDC platform for policy storage and enforcement offers competitive performance.

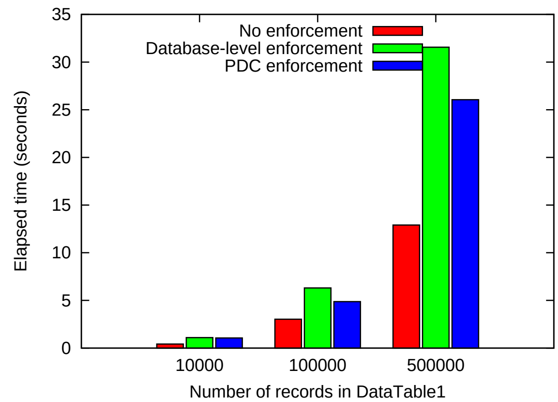
Figures 3(a) and 3(b) show the response times for Q2 with $PolicySet1$ and $PolicySet2$, respectively. This query joins $DataTable1$ with $DataTable2$ on $IntAttr1$ and projects a pair of attributes. Analogously to the previous result, enforcing privacy policies incurs a noticeable performance penalty and the database-level scheme appears to outperform the PDC-driven design by a significant margin. We are currently investigating the likely causes of this performance discrepancy.

[1]For efficiency, we constructed a B-Tree index on the *Id* field for both policy relations.

[2]Defining a view simplifies presentation, but is not strictly necessary.

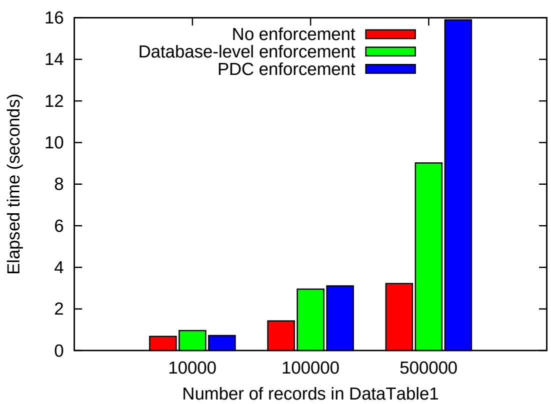


(a) Query response time with *PolicySet*1.

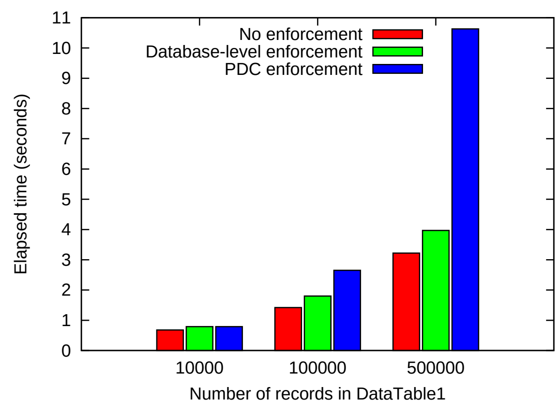


(b) Query response time with *PolicySet*2.

Figure 2: Performance and scalability results for Q1 - a simple query that returns the full contents of *DataTable*1.



(a) Query response time with *PolicySet*1.



(b) Query response time with *PolicySet*2.

Figure 3: Performance and scalability results for Q2. This query joins *DataTable*1 and *DataTable*2 on *IntAttr*1 and projects two of the resulting attributes.

6 Summary and Future Work

Information privacy is an increasingly growing concern for organizations that collect and maintain large amounts of personal data. In this report, we presented an initial design for a privacy-preserving relational database that provides the table semantics model of limited disclosure. Our approach is influenced to a significant extent by earlier work [2, 4], but relies on the PDC platform for the storage and enforcement of privacy policies. We are currently implementing the remaining components of PDC and making improvements to our Postgres-based prototype. Future work will include conducting an extensive performance evaluation on top of the fully-functional PDC platform.

References

[1] Postgres. `http://www.postgresql.org`.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, 2002.

[3] S. Katti, A. Ermolinskiy, M. Casado, S. Shenker, and H. Balakrishnan. S3: Securing sensitive stuff. In *USENIX OSDI Work-in-Progress (WiP) report*, 2008.

[4] K. Lefevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, 2004.

[5] M. Meyers. UC Berkeley computers hacked, 160,000 at risk. `http://news.cnet.com/8301-1009_3-10236793-83.html"`, 2009.

APPENDIX

A SQL Grammar Extensions

Below, we list our extensions to the SQL language grammar (defined in *src/backend/parser/gram.y*). The apparent complexity is due to the fact that the annotated version of the VALUES clause is meaningful *only* in the context of an INSERT statement and should not be allowed to appear in other cases permitted by the original grammar (e.g., inside *SelectStmt*).

```
insert_rest:
    tainted_values_clause:
        {
            $$ = makeNode(InsertStmt);
            $$->cols = NIL;
            $$->selectStmt = $1;
        }

    | '(' insert_column_list ')' tainted_values_clause
        {
            $$ = makeNode(InsertStmt);
            $$->cols = $2;
            $$->selectStmt = $4;
        }
    | SelectStmtNoValues
        {
            $$ = makeNode(InsertStmt);
            $$->cols = NIL;
            $$->selectStmt = $1;
        }
    | '(' insert_column_list ')' SelectStmtNoValues
        {
            $$ = makeNode(InsertStmt);
            $$->cols = $2;
            $$->selectStmt = $4;
        }
    | DEFAULT VALUES
        {
            $$ = makeNode(InsertStmt);
            $$->cols = NIL;
            $$->selectStmt = NULL;
        }
    ;

tainted_values_clause:
    VALUES tainted_ctext_row
        {
            SelectStmt *n = makeNode(SelectStmt);
            n->valuesLists = list_make1($2);
            $$ = (Node *) n;
        }
    | tainted_values_clause ',' tainted_ctext_row
        {
            SelectStmt *n = (SelectStmt *) $1;
```

```
            n->valuesLists = lappend(n->valuesLists, $3);
            $$ = (Node *) n;
        }
    ;

tainted_ctext_row: '(' tainted_ctext_expr_list ')'    { $$ = $2; }
    ;

tainted_ctext_expr_list:
    tainted_ctext_expr                                { $$ = list_make1($1); }
    | tainted_ctext_expr_list ',' tainted_ctext_expr  { $$ = lappend($1, $3); }
    ;

taint_label:   '{' Iconst '}'
    {
        TaintLabel *l = makeNode(TaintLabel);
        l->label = (Node *)makeInteger($2);
        $$ = (Node *)l;
    }
    ;

tainted_ctext_expr:
    a_expr                   { $$ = (Node *) $1; }
    | a_expr taint_label
        {
            Tainted_A_Expr *e = makeNode(Tainted_A_Expr);
            e->a_expr = $1;
            e->taint = $2;
            $$ = (Node *)e;
        }
    | DEFAULT         { $$ = (Node *) makeNode(SetToDefault); }
    ;

SelectStmtNoValues: select_no_values_no_parens %prec UMINUS
    | select_no_values_with_parens %prec UMINUS
    ;

select_no_values_with_parens:
    '(' select_no_values_no_parens ')'      { $$ = $2; }
    | '(' select_no_values_with_parens ')'{ $$ = $2; }
    ;

select_no_values_no_parens:
    simple_select_no_values        { $$ = $1; }
    | select_clause_no_values sort_clause
        {
            insertSelectOptions((SelectStmt *) $1, $2, NIL,
                NULL, NULL);
            $$ = $1;
        }
    | select_clause_no_values opt_sort_clause for_locking_clause opt_select_limit
        {
            insertSelectOptions((SelectStmt *) $1, $2, $3,
                list_nth($4, 0), list_nth($4, 1));
```

```
            $$ = $1;
        }
    | select_clause_no_values opt_sort_clause select_limit opt_for_locking_clause
        {
            insertSelectOptions((SelectStmt *) $1, $2, $4,
                list_nth($3, 0), list_nth($3, 1));
            $$ = $1;
        }
    ;

select_clause_no_values:
    simple_select_no_values        { $$ = $1; }
    | select_no_values_with_parens { $$ = $1; }
    ;

simple_select_no_values:
    SELECT opt_distinct target_list
    into_clause from_clause where_clause
    group_clause having_clause
        {
            SelectStmt *n = makeNode(SelectStmt);
            n->distinctClause = $2;
            n->targetList = $3;
            n->intoClause = $4;
            n->fromClause = $5;
            n->whereClause = $6;
            n->groupClause = $7;
            n->havingClause = $8;
            $$ = (Node *)n;
        }
    | select_clause_no_values UNION opt_all select_clause
        {
            $$ = makeSetOp(SETOP_UNION, $3, $1, $4);
        }
    | select_clause_no_values INTERSECT opt_all select_clause
        {
            $$ = makeSetOp(SETOP_INTERSECT, $3, $1, $4);
        }
    | select_clause_no_values EXCEPT opt_all select_clause
        {
            $$ = makeSetOp(SETOP_EXCEPT, $3, $1, $4);
        }
    ;
```

B Masking Prohibited Values

```
/* slot_deform_tuple
 * Given a TupleTableSlot, extract data from the slot's physical tuple
 * into its Datum/isnull arrays.  */
static void slot_deform_tuple(TupleTableSlot  *slot, int natts) {
        HeapTuple tuple = slot->tts_tuple;
        Datum *values = slot->tts_values;
        bool *isnull = slot->tts_isnull;
        TupleDesc tupleDesc = slot->tts_tupleDescriptor;
```

```
        Form_pg_attribute *att = tupleDesc->attrs;
        HeapTupleHeader tup = tuple->t_data;
        s3_tainttag_t attr_tainttag;
        s3_policy_t attr_policy;
        bool allow;
        ...

        tp = (char *) tup + tup->t_hoff;

        for (; attnum < natts; attnum++) {
                Form_pg_attribute thisatt = att[attnum];
                ...
                s3_get_vaddr_tainttag(tp + off, &attr_tainttag);
                s3_get_tainttag_policy(&attr_tainttag, &attr_policy);

                allow = s3_evaluate_policy(&attr_policy,
                  GetSessionPurpose(),
                  GetSessionRecipient(),
                  GetSessionChannel());
                if (allow) {
                        values[attnum] = fetchatt(thisatt, tp + off);
                        isnull[attnum] = false;
                } else {
                        values[attnum] = (Datum)0;
                        isnull[attnum] = true;
                        slow = true;
                }

                off = att_addlength_pointer(off, thisatt->attlen, tp + off);
                ...
        }
        ...
}
```