Equivalence-based Security for Querying Encrypted Databases: Theory and Application to Privacy Policy Audits

Omar Chowdhury
Purdue University
West Lafayette, Indiana
ochowdhu@purdue.edu

Deepak Garg
MPI-SWS
Germany
dg@mpi-sws.org

Limin Jia, Anupam Datta
Carnegie Mellon University
Pittsburgh, Pennsylvania
{liminjia,danupam}@cmu.edu

ABSTRACT

To reduce costs, organizations may outsource data storage and data processing to third-party clouds. This raises confidentiality concerns, since the outsourced data may have sensitive information. Although semantically secure encryption of the data prior to outsourcing alleviates these concerns, it also renders the outsourced data useless for any relational processing. Motivated by this problem, we present two database encryption schemes that reveal just enough information about structured data to support a wide-range of relational queries. Our main contribution is a definition and proof of security for the two schemes. This definition captures confidentiality offered by the schemes using a novel notion of equivalence of databases from the adversary's perspective. As a specific application, we adapt an existing algorithm for finding violations of a rich class of privacy policies to run on logs encrypted under our schemes and observe low to moderate overheads.

Categories and Subject Descriptors

H.2.0 [**DATABASE MANAGEMENT**]: General—*Security, integrity, and protection*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy, Regulation*

Keywords

Privacy Policy Audit; HIPAA; GLBA; Querying Encrypted Databases

1. INTRODUCTION

To reduce infrastructure costs, small- and medium-sized businesses may outsource their databases and database applications to third-party clouds. However, such data is often private, so storing it in a cloud raises confidentiality concerns. Semantically secure encryption of databases prior to outsourcing alleviates confidentiality concerns, but it also makes it impossible to run any relational queries on the cloud without client interaction. Several prior research projects have investigated encryption schemes that trade-off perfect data confidentiality for the ability to run relational queries [39, 6, 21]. However, these schemes either require client-side processing [21], or require additional hardware support [6], or support a very restrictive set of queries [39]. Our long-term goal is to develop database encryption schemes that can (1) be readily deployed on commercial off-the-shelf (COTS) cloud infrastructure without any special hardware or any kernel modifications, (2) support a broad range of (non-update) relational queries on the encrypted database without interaction with the client, (3) be implemented with low or moderate overhead, and (4) provide provable end-to-end security and a precise characterization of what information encryption leaks in exchange for supporting a given set of queries. Both in objective and in method, our goal is similar to that of CryptDB [34], which attains properties (1)–(3), but not (4).

As a step towards our goal, in this paper, we design two database encryption schemes, $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$, with properties (1)–(4). Our design is guided by, and partly specific to, a single application, namely, audit of data-use logs for violations of privacy policies. This application represents a real-world problem. Organizations are subject to privacy legislation. For example, in the US, the healthcare and finance industry must handle client data in accordance with the federal acts HIPAA [1] and GLBA [2] respectively. To remain compliant with privacy legislation, organizations record logs of privacy-relevant day-to-day operations such as data access/use and employee role changes, and audit these logs for violations of privacy policies, either routinely or on a case-by-case basis. Logs can be fairly large and are often organized in commodity databases. Audit consists of a sequence of policy-guided queries. Audit is computationally expensive but highly parallelizable, so there is significant merit in outsourcing the storage of logs and the execution of audit algorithms to third-party clouds.

Security. We characterize formally what information about an encrypted log (database) our schemes may leak to an adversary with direct access to the encrypted store (modeling a completely adversarial cloud). We prove that by looking at a log encrypted with either of our schemes, an adversary can learn (with non-negligible probability) only that the plaintext log lies within a certain, precisely defined *equivalence class of logs*. This class of logs characterizes the uncertainty of the adversary and, therefore, the confidentiality of the

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10.
http://dx.doi.org/10.1145/2810103.2813638.

encrypted log [5]. Prior work like CryptDB lacks such a theorem. CryptDB uses a trusted proxy server to dynamically choose the most secure encryption scheme for every database column (from a pre-determined set of schemes), based on the queries being run on that column. While each scheme is known to be secure in isolation and it is shown that at any time, a column is encrypted with the weakest scheme that supports all past queries on the column [32, Theorem 2], there is no end-to-end characterization of information leaked after a sequence of queries. (In return, CryptDB supports all SQL queries, including aggregation queries, which we do not support.)

Functionality. To demonstrate that our proposed encryption schemes support nontrivial applications, we adapt an audit algorithm called **reduce** from our prior work [19] to execute on logs encrypted with either $\mathsf{Eunomia}^{\mathsf{DET}}$ scheme or $\mathsf{Eunomia}^{\mathsf{KH}}$ scheme. We implement and test the adapted algorithm, **ereduce**, on both schemes and show formally that the algorithm runs correctly on both schemes (except with negligible probability). The algorithm **ereduce** can audit all policies that **reduce** can, including most clauses of the HIPAA and GLBA Privacy Rules [19].

Audit with **ereduce** is a challenging application for encryption schemes because it requires almost all standard relational query operations on logs. These operations include selection, projection, join, comparison of fields, and what we call *displaced comparison* (is the difference between two timestamps less than a given constant?). Both our encryption schemes support all these query operations. The only standard query operation not commonly required by privacy audit (and not supported by our schemes) is aggregation (sums and averages; counting queries are supported). Any application that requires only the query operations listed above can be adapted to run on $\mathsf{Eunomia}^{\mathsf{DET}}$ or $\mathsf{Eunomia}^{\mathsf{KH}}$, even though this paper focuses on the audit application only.

$\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ trade efficiency and flexibility differently. $\mathsf{Eunomia}^{\mathsf{DET}}$ uses deterministic encryption and has very low overhead (3% to 9% over a no-encryption baseline in our audit application), but requires anticipating which pairs of columns will be join-ed in audit queries prior to encryption. $\mathsf{Eunomia}^{\mathsf{KH}}$ uses Popa *et al.*'s adjustable key hash scheme [35, 34] for equality tests and has higher overhead (63% to 406% in our audit application), but the columns that will be join-ed during audit do not have to be determined prior to encryption. To determine which columns will be join-ed during audit for violations of a given policy, we develop a new static analysis of policies, which we call the EQ mode check.

To support displaced comparisons, which privacy audit often requires, we design and prove the security of a new cryptographic sub-scheme dubbed **mOPED** (short for, mutable order-preserving encoding with displacement). This scheme extends the **mOPE** scheme of Popa *et al.* [33], which does not support displacements, and may be of independent interest.

Deployability. Both $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ can be deployed on commodity database systems with some additional metadata. In both schemes, a client encrypts the individual data cells locally and store the ciphertexts in a commodity database system in the cloud (possibly incrementally). Audit (**ereduce**) runs on the cloud without interaction with the client and returns encrypted results to the client, who decrypts them to recover policy violations. Both $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ use basic, widely-available cryptographic operations only.

Contributions. We make the following technical contributions:

- We introduce two database encryption schemes, namely $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$, that support selection, projection, join, comparison of fields, and displaced comparison queries. The schemes trade efficiency for the need to predict expected pairs of join-ed columns before encryption. As a building block, we develop the sub-scheme **mOPED**, that allows displaced comparison of encrypted values.
- We characterize the confidentiality attained by our schemes as equivalence classes of plaintext logs and prove that both our schemes are secure.
- We adapt an existing privacy policy audit algorithm to execute on our schemes. We prove the functional correctness of the execution of the algorithm on both our schemes.
- We implement both our schemes and the adapted audit algorithm, observing low overheads on $\mathsf{Eunomia}^{\mathsf{DET}}$ and moderate overheads on $\mathsf{Eunomia}^{\mathsf{KH}}$.

Proofs of theorems omitted from this paper can be found in an accompanying technical report [15].

Notation. This paper is written in the context of the privacy audit application and our encryption schemes are presented within this context. We sometimes use the term "log" or "audit log" when the more general term "database" could have been used and, similarly, use the term "policy" or "privacy policy" when the more general term "query" would fit as well.

2. OVERVIEW OF EUNOMIA

We first present the architecture of Eunomia. Then, we motivate our choice of encryption schemes through examples and discuss policy audit in Eunomia in more detail. Finally, we discuss our goals, assumptions, and adversary model.

2.1 Architecture of Eunomia

We consider the scenario where an organization, called the client or Cl, with sensitive data and audit requirements wishes to outsource its log (organized as a relational database) and audit process (implemented as a sequence of policy-dependent queries) to a potentially compromisable third-party cloud server, denoted CS. Cl generates the log from its day-to-day operations. Cl then encrypts the log and transfers the encrypted log to the CS. Cl initiates the audit process by choosing a policy. The auditing algorithm runs on the CS infrastructure and the audit result containing encrypted values is sent to Cl, which can decrypt the values in the result.

The mechanism of log generation is irrelevant for us. From our perspective, a log is a pre-generated database with a public schema, where each table defines a certain privacy-relevant predicate. For example, the table Roles may contain columns Name and Role, and may define the mapping of

CI's employees to CI's organizational roles. Similarly, the table Sensitive_accesses may contain columns Name, File_name, and Time, recording who accessed which sensitive file at what time. Several tables like Sensitive_accesses may contain columns with timestamps, which are integers.

2.2 Encryption Schemes

An organization may naïvely encrypt the entire log with a strong encryption scheme before transferring it to a cloud, but this renders the stored log ineffective for audit, as audit (like most other database computations) must *relate* different parts of the log. For example, suppose the log contains two tables T_1 and T_2. T_1 lists the names of individuals who accessed patients' prescriptions. T_2 lists the roles of all individuals in the organization. Consider the privacy policy:

Policy 1: *Every individual accessing patients' prescriptions must be in the role of Doctor.*

The audit process of the above policy must read names from T_1 and test them for equality against the list of names under the role Doctor in T_2. This forces the use of an encryption method that allows equality tests (or equi-joins). Unsurprisingly, this compromises the confidentiality of the log, as an adversary (e.g., the cloud host, which observes the audit process) can detect equality between encrypted fields (e.g., equality of names in T_1 and T_2). However, not all is lost: for instance, if per-cell deterministic encryption is used, the adversary cannot learn the concrete names themselves.

A second form of data correlation necessary for audit is the order between time points. Consider the following policy:

Policy 2: *If an outpatient's medical record is accessed by an employee of the Billing Department, then the outpatient must have visited the medical facility in the last one month.*

Auditing this policy requires checking whether the distance between the timestamps in an access table and the timestamps in a patient visit table is shorter than a month. In this case, the encryption scheme must reveal not just the relative order of two timestamps but also the order between a timestamp and another timestamp displaced by one month. Similar to Policy 1, the encryption scheme must reveal equality between patient names in the two tables.

To strike a balance between functional (audit) and confidentiality requirements, we investigate two cryptographic schemes, namely Eunomia$^{\mathsf{DET}}$ and Eunomia$^{\mathsf{KH}}$, to encrypt logs. Each cell in the database tables is encrypted individually. All cells in a column are encrypted using the same key. Eunomia$^{\mathsf{DET}}$ uses deterministic encryption to support equality tests; two columns that might be tested for equality by subsequent queries are encrypted with the same key. Eunomia$^{\mathsf{DET}}$ requires that log columns that might be tested for equality during audit are known prior to the encryption. Audit under Eunomia$^{\mathsf{DET}}$ is quite efficient. However, adapting encrypted logs to audit different policies that require different column equality tests requires log re-encryption, which is costly. Our second scheme Eunomia$^{\mathsf{KH}}$ handles frequent policy updates efficiently. Eunomia$^{\mathsf{KH}}$ relies on the adjustable key hash scheme [35, 34] for equality tests. A transfer token is generated for each pair of columns needed to be tested for equality prior to audit. Eunomia$^{\mathsf{KH}}$ additionally stores keyed hashes of all cells. Audit under Eunomia$^{\mathsf{KH}}$ requires the audit algorithm to track the provenance of the ciphertext (i.e., from which table, which column the ciphertext originated) and is less efficient than audit under Eunomia$^{\mathsf{DET}}$.

Displaced comparison (needed for Policy 2) is supported using a new sub-scheme **mOPED**, which is described in Section 4. Both Eunomia$^{\mathsf{DET}}$ and Eunomia$^{\mathsf{KH}}$ use **mOPED**. Like its predecessor, **mOPE** [33], the scheme adds a additional search tree (additional metadata) to the encrypted database on CS. (Supporting displacements is necessary for a practical audit system because privacy regulations use displacements to express obligation deadlines. Out of 84 HIPAA privacy clauses, 7 use displacements. Cignet Health of Prince George's County, Maryland was fined $1.3 million for violating one of these clauses, §164.524 [29].

The encrypted database has a schema derived from the schema of the plaintext database and may be stored on CS using any standard database management systems (DBMS). The DBMS may be used to index the encrypted cells. As shown in [19], database indexing plays a key role in improving the efficiency of the audit process. Hence, we develop our encryption scheme in such a way that it is possible to leverage database indexing supported by commodity DBMS.

2.3 Policies and audit

Privacy policies may be extracted from privacy legislation like HIPAA [1] and GLBA [2], or from internal company requirements. Technically, a privacy policy specifies a *constraint* on the log. For example, Policy 1 of Section 1 requires that any name appearing in table T_1 appears in table T_2 with role Doctor. Generally, policies can be complex and may mention various entities, roles, time, and subjective beliefs. For instance, DeYoung *et al.*'s formalization of the HIPAA and GLBA Privacy Rules span over 80 and 10 pages, respectively [18]. We represent policies as formulas of first-order logic (FOL) because we find it technically convenient and because FOL has been demonstrated in prior work to be adequate for representing policies derived from existing privacy legislation (DeYoung *et al.*, mentioned above, use the same representation). We describe this logic-based representation of policies in Section 3.

Our audit algorithm adapts our prior algorithm, **reduce** [19], that works on policies represented in FOL. This algorithm takes as input a policy and a log and *reduces* the policy by checking the policy constraints on the log. It outputs constraints that cannot be checked due to lack of information (missing log tables, references to future time points, or need for human intervention) in the form of a *residual policy*. Similar to **reduce**, our adapted algorithm, **ereduce**, uses database queries as the basic building block. Our encryption schemes permit queries with selection, projection, join, comparison and displaced comparison operations. Our schemes do not support queries like aggregation (which would require an underlying homomorphic encryption scheme and completely new security proofs).

To run **reduce** on Eunomia$^{\mathsf{DET}}$, we need to identify columns that are tested for equality. This information is needed prior to encryption for Eunomia$^{\mathsf{DET}}$ and prior to audit for Eunomia$^{\mathsf{KH}}$, as explained in Section 4. We develop a static analysis of policies represented in FOL, which we call the *EQ mode check*, defined in Section 7, to determine which columns may need to be compared for equality when the policy is audited.

2.4 Adversary model and Security Goals

Assumptions and threat model. In our threat model, CI is trusted but CS is an *honest but curious* adversary with the following capability: CS can run any polynomial time algorithm on the stored (encrypted) log, including the audit over any policy. We assume that CI generates keys and encrypts the log with our encryption schemes before uploading it to CS. Audit runs on the CS infrastructure but (by design) it does not perform decryption. Hence, CS never sees plaintext data or the keys, but CS can glean some information about the log, e.g., the order of two fields or the equality of two fields. The output of audit may contain encrypted values indicating policy violations, but these values are decrypted only at CI.

We assume that privacy policies are known to the adversary. This assumption may not be true for an organization's internal policies, but relaxing this assumption only simplifies our formal development. To audit over logs encrypted with Eunomia$^{\mathsf{DET}}$, any constants appearing in the policy (like "Doctor" in Policy 1 of Section 1) must be encrypted before the audit process starts, so CS can recover the association between ciphertext and plaintexts of constants that appear in the (publicly known) privacy policy. Similarly, in Eunomia$^{\mathsf{KH}}$, the hashes of constants in policies must be revealed to the adversary. in a set

Security and functionality goals. (Confidentiality) Our primary goal is to protect the confidentiality of the log's content, despite any compromise of CS, including its infrastructure, employees, and the audit process running on it. (Expressiveness) Our system should be expressive enough to represent and audit privacy policies derived from real legislation. In our evaluation, we work with privacy rules derived from HIPAA and GLBA.

Log equivalence. Central to the definition of the end-to-end security property that we prove of our Eunomia$^{\mathsf{DET}}$ and Eunomia$^{\mathsf{KH}}$ is the notion of log equivalence. It characterizes what information about the database *remains confidential* despite a complete compromise of CS. Our security definition states that the adversary can only learn that the log belongs to a stipulated equivalence class of logs. The coarser our equivalence, the stronger our security theorem.

For semantically secure encryption, we could say that two logs are equivalent if they are the same length. When the encryption permits join, selection, comparison and displaced comparison queries, this definition is too strong. For example, the attacker must be allowed to learn that two constants on the log (e.g., Doctor and Nurse) are not equal if they lie in different columns that the attacker can try to join. Hence, we need a refined notion of log equivalence, which we formalize in Section 5.2.

3. POLICY AND LOG SPECIFICATIONS

We review the logic that we use to represent privacy policies and give a formal definition of logs (databases). These definitions are later used in the definition and analysis of our encryption schemes and the **ereduce** audit algorithm.

Policy logic. We use the guarded-fragment of first-order logic introduced in [3] to represent privacy policies. The syntax of the logic is shown in Figure 1. Policies or formulas are denoted φ. Terms t are either constants c, d drawn from a domain $\mathcal{D}$ or variables x drawn from a set Var. (Function symbols are disallowed.) $\vec{t}$ denotes a list of terms. The basic building block of formulas is *atoms*, which represent relations between terms. We allow three kinds of atoms.

$$\begin{array}{rrcl}
\mathsf{Atoms} & \mathcal{P} & ::= & \mathsf{p}(t_1,\ldots,t_n) \mid \mathsf{timeOrder}(t_1,d_1,t_2,d_2) \mid \\
 & & & t_1 = t_2 \\
\mathsf{Guard} & g & ::= & \mathcal{P} \mid \top \mid \bot \mid g_1 \wedge g_2 \mid g_1 \vee g_2 \mid \exists x.g \\
\mathsf{Formula} & \varphi & ::= & \mathcal{P} \mid \top \mid \bot \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \\
 & & & \forall \vec{x}.(g \rightarrow \varphi) \mid \exists \vec{x}.(g \wedge \varphi)
\end{array}$$

Figure 1: Policy specification logic syntax

First, $\mathsf{p}(t_1,\ldots,t_n)$ represents a relation which is established through a table named p in the audit log. The symbol p is called a predicate (or, interchangeably, a table). The set of all predicate symbols is denoted by $\mathbb{P}$. An arity function $\alpha : \mathbb{P} \rightarrow \mathbb{N}$ specifies how many arguments each predicate takes (i.e., how many columns each table has). Second, for numerical terms, we allow comparison after displacement with constants, written $\mathsf{timeOrder}(t_1,d_1,t_2,d_2)$. This relation means that $t_1 + d_1 \leq t_2 + d_2$. Here, d_1, d_2 must be constants. Third, we allow term equality, written $t_1 = t_2$. Although we restrict atoms of the logic to these three categories only, the resulting fragment is still very expressive. All the HIPAA- and GLBA-based policies tested in prior work [19] and all but one clause of the entire HIPAA and GLBA privacy rules formalized by DeYoung *et al.* [18] lie within this fragment.

Formulas or policies, denoted φ, contain standard logical connectives $\top$ ("true"), $\bot$ ("false"), $\wedge$ ("and"), $\vee$ ("or"), $\forall x$ ("for every x") and $\exists x$ ("for some x"). Saliently, the form of quantifiers $\forall x$ and $\exists x$ is restricted: Each quantifier must include a *guard*, g. As shown in [19], this restriction, together with the mode check described in Section 7, ensures that audit terminates (in general, the domain $\mathcal{D}$ may be infinite). Intuitively, one may think of a policy φ as enforcing a constraint on the predicates it mentions, i.e., on the tables of the log. A guard g may be thought of as a query on the log (indeed, the syntax of guards generalizes Datalog, a well-known database query language). The policy $\forall \vec{x}.(g \rightarrow \varphi)$ may be read as "for every result $\vec{x}$ of the query g, the constraint φ must hold." Dually, $\exists \vec{x}.(g \wedge \varphi)$ may be read as "some result $\vec{x}$ of the query g must satisfy the constraint φ."

Example 1. Consider the following policy, based on §6802(a) of the GLBA privacy law:

$$\begin{array}{l}
\forall p_1, p_2, m, q, a, t. \quad (\mathsf{send}(p_1, p_2, m, t) \wedge \\
\mathsf{tagged}(m, q, a) \wedge \mathsf{activeRole}(p_1, \mathit{institution}) \wedge \\
\mathsf{notAffiliateOf}(p_2, p_1, t) \wedge \mathsf{customerOf}(q, p_1, t) \wedge \mathsf{attr}(a, \mathit{npi})) \\
\quad \rightarrow \Big((\exists t_1, m_1.\mathsf{send}(p_1, q, m_1, t_1) \wedge \mathsf{timeOrder}(t_1, 0, t, 0) \wedge \\
\qquad \mathsf{timeOrder}(t, 0, t_1, 30) \wedge \mathsf{discNotice}(m_1, p_1, p_2, q, a, t)) \\
\qquad\qquad \bigvee \\
\quad (\exists t_2, m_2.\mathsf{send}(p_1, q, m_2, t_2) \wedge \mathsf{timeOrder}(t, 0, t_2, 0) \wedge \\
\qquad \mathsf{timeOrder}(t_2, 0, t, 30) \wedge \mathsf{discNotice}(m_2, p_1, p_2, q, a, t)) \Big)
\end{array}$$

The policy states that principal p_1 can *send* a message m to principal p_2 at time t where the message m contains principal q's attribute a (*e.g.*, account number) and (i) p_1 is in the role of a financial institution, (ii) p_2 is *not* a third-party affiliate of p_1 at time t, (iii) q is a customer of p_1 at time t, (iv) the attribute a is non-public personal information (npi, *e.g.*, a social security number) only if any one of the two conditions separated by $\vee$ holds. The first condition says that the institution has already sent a notification of this disclosure in the past 30 days to the customer q (*i.e.*,

$0 \leq (t - t_1) \leq 30$). The second condition says that the institution will send a notification of this disclosure within the next 30 days (*i.e.*, $0 \leq (t_2 - t) \leq 30$).

Logs and schemas. An audit log or log, denoted $\mathcal{L}$, is a database with a given schema. A schema $\mathcal{S}$ is a set of pairs of the form $\langle \mathsf{tableName}, \mathsf{columnNames} \rangle$ where columnNames is an ordered list of all the column names in the table (predicate) tableName. A schema $\mathcal{S}$ corresponds to a policy φ if $\mathcal{S}$ contains all predicates mentioned in the policies φ, and the number of columns in predicate p is $\alpha(\mathsf{p})$.

Semantically, we may view a log $\mathcal{L}$ as a function that given as argument a variable-free atom $\mathsf{p}(\vec{t})$ returns either $\top$ (the entry $\vec{t}$ exists in table p in $\mathcal{L}$) or $\bot$ (the entry does not exist). To model the possibility that a log table may be incomplete, we allow for a third possible response uu (unknown). In our implementation, the difference between uu and $\bot$ arises from an additional bit on the table p indicating whether or not the table may be extended in future. Formally, we say that log $\mathcal{L}_1$ extends log $\mathcal{L}_2$, written $\mathcal{L}_1 \geq \mathcal{L}_2$ when for every p and $\vec{t}$, if $\mathcal{L}_2(\mathsf{p}(\vec{t})) \neq \mathsf{uu}$, then $\mathcal{L}_1(\mathsf{p}(\vec{t})) = \mathcal{L}_2(\mathsf{p}(\vec{t}))$. Thus, the extended log $\mathcal{L}_1$ may determinize some unknown entries from $\mathcal{L}_2$, but cannot change existing entries in $\mathcal{L}_2$.

Our logic uses standard semantics of first-order logic, treating logs as models. The semantics, written $\mathcal{L} \models \varphi$, take into account the possibility of unknown relations; we refer the reader to [19] for details (these details are not important for understanding this paper). Intuitively, if $\mathcal{L} \models \varphi$, then the policy φ is satisfied on the log $\mathcal{L}$; if $\mathcal{L} \not\models \varphi$, then the policy is violated; and if neither holds then the log does not have enough information to determine whether or not the policy has been violated.

Example 2. The policy in Example 1 can be checked for violations on a log whose schema contains tables send, tagged, activeRole, notAffiliateOf, customerOf, attr and discNotice with 4, 3, 2, 3, 3, 2 and 6 columns respectively. In this audit, values in several columns may have to be compared for equality. For example, the values in the first columns of tables send and activeRole must be compared because, in the policy, they contain the same variable p_1. Similarly, timestamps must be compared after displacement with constants 0 and 30. The log encryption schemes we define next support these operations.

4. ENCRYPTION SCHEMES

We present our two log encryption schemes, $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ in Section 4.2 and Section 4.3 respectively. Both schemes use (as a black-box) a new sub-scheme called **mOPED**, for comparing timestamps after displacement, which we present in Section 4.4.

4.1 Preliminaries

We introduce common constructs used through out the rest of this section.

Equality scheme. To support policy audit, we determine, through a static analysis of the policies to be audited, which pairs of columns in the log schema may be tested for equality or joined. We defer the details of this policy analysis to Section 7. For now, we just assume that the result of this analysis is available. This result, called an *equality scheme*, denoted δ, is a set of pairs of the form $\langle \mathsf{p}_1.\mathsf{a}_1, \mathsf{p}_2.\mathsf{a}_2 \rangle$. The key property of δ is that if, during audit, column a_1 of table p_1 is tested for equality against column a_2 of table p_2, then $\langle \mathsf{p}_1.\mathsf{a}_1, \mathsf{p}_2.\mathsf{a}_2 \rangle \in \delta$.

Policy constants. Policies may contain constants. For instance, the policy of Example 1 contains the constants *npi*, *institution*, 0 and 30. Before running our audit algorithm over encrypted logs, a new version of the policy containing these constants in either encrypted (for $\mathsf{Eunomia}^{\mathsf{DET}}$) or keyed hash (for $\mathsf{Eunomia}^{\mathsf{KH}}$) form must be created. Consequently, the adversary, who observes the audit and knows the plaintext policy, can learn the encryption or hash of these constants. Hence, these constants play an important role in our security definitions. The set of all these policy constants is denoted C.

Displacement constants. Constants which feature in the 2nd and 4th argument positions of the predicate timeOrder() play a significant role in construction of the **mOPED** encoding and our security definition. These constants are called *displacements*, denoted D. For instance, in Example 1, $D = \{0, 30\}$. For any policy, $D \subseteq C$.

Encrypting timestamps. We assume (conservatively) that all timestamps in the plaintext log may be compared to each other, so all timestamps are encrypted (in $\mathsf{Eunomia}^{\mathsf{DET}}$) or hashed (in $\mathsf{Eunomia}^{\mathsf{KH}}$) with the same key K_{time}. This key is also used to protect values in the **mOPED** sub-scheme. The assumption of all timestamps may be compared with each other, can be restricted substantially (for both schemes) if the audit policy is fixed ahead of time.

4.2 $\mathsf{Eunomia}^{\mathsf{DET}}$

The log encryption scheme $\mathsf{Eunomia}^{\mathsf{DET}}$ encrypts each cell individually using deterministic encryption. All cells in a column are encrypted with the same key. Importantly, if cells in two columns may be compared during audit (as determined by the equality scheme δ), then the two columns also share the same key. Hence, cells can be tested for equality simply by comparing their ciphertexts. To allow timestamp comparison after displacement, the encrypted log is paired with a **mOPED** encoding of timestamps that we explain later. Note that it is possible to replace deterministic encryption with a cryptographically secure keyed hash and a semantically secure ciphertext to achieve the same functionality (the keyed hash value could be used to check for equality). However, this design incurs higher space overhead than our design with deterministic encryption.

Technically, $\mathsf{Eunomia}^{\mathsf{DET}}$ contains the following three algorithms: $\mathsf{KeyGen}^{\mathsf{DET}}(1^{\kappa}, \mathcal{S}, \delta)$, $\mathsf{EncryptLog}^{\mathsf{DET}}(\mathcal{L}, \mathcal{S}, \mathcal{K})$, and $\mathsf{EncryptPolicyConstants}^{\mathsf{DET}}(\varphi, \mathcal{K})$.

Key generation. The probabilistic algorithm $\mathsf{KeyGen}^{\mathsf{DET}}(\cdot, \cdot, \cdot)$ takes as input the security parameter κ, the plaintext log schema $\mathcal{S}$, and an equality scheme δ. It returns a *key set* $\mathcal{K}$. The *key set* $\mathcal{K}$ is a set of triples of the form $\langle \mathsf{p}, \mathsf{a}, k \rangle$. The triple means that all cells in column a of table p must be encrypted (deterministically) with key k. The constraints on $\mathcal{K}$ are that (a) if $\mathsf{p}.\mathsf{a}$ contains timestamps, then $k = K_{\mathsf{time}}$, and (b) if $\langle \mathsf{p}_1.\mathsf{a}_1, \mathsf{p}_2.\mathsf{a}_2 \rangle \in \delta$, $\langle \mathsf{p}_1, \mathsf{a}_1, k_1 \rangle \in \mathcal{K}$ and $\langle \mathsf{p}_2, \mathsf{a}_2, k_2 \rangle \in \mathcal{K}$, then $k_1 = k_2$.

Encrypting the log. The algorithm $\mathsf{EncryptLog}^{\mathsf{DET}}(\cdot, \cdot, \cdot)$ takes as input a plaintext log $\mathcal{L}$, its schema $\mathcal{S}$, and the key set $\mathcal{K}$ generated by KeyGen(). It returns a pair $e\mathcal{L} = \langle e\mathsf{DB}, e\mathcal{T} \rangle$

where, $e\mathsf{DB}$ is the cell-wise encryption of $\mathcal{L}$ with appropriate keys from $\mathcal{K}$ and $e\mathcal{T}$ is the **mOPED** encoding.

Encrypting constants in the policy. To audit over logs encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$, constants in the policy must be encrypted too (else, we cannot check whether or not an atom mentioning the constant appears in the encrypted log). The algorithm $\mathsf{EncryptPolicyConstants}^{\mathsf{DET}}(\cdot,\cdot)$ takes as input a plaintext policy φ, and a key set $\mathcal{K}$, and returns a policy φ' in which constants have been encrypted with appropriate keys. The function works as follows: If, in φ, the constant c appears in the ith position of predicate p, then in φ', the ith position of p is c deterministically encrypted with the key of the ith column of p (as obtained from $\mathcal{K}$). Other than this, φ and φ' are identical.

Remarks. The process of audit on a log encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$ requires no cryptographic operations. Compared to an unencrypted log, we only pay the overhead of having to compare longer ciphertexts and some cost for looking up the **mOPED** encoding to compare timestamps. However, auditing for a policy that requires equality tests beyond those prescribed by an equality scheme δ is impossible on a log encrypted for δ. To do so, we would have to re-encrypt parts of the log, which is a slow operation. Our second log encryption scheme, $\mathsf{Eunomia}^{\mathsf{KH}}$, represents a different trade-off.

4.3 Eunomia$^{\mathsf{KH}}$

$\mathsf{Eunomia}^{\mathsf{KH}}$ relies on the adjustable keyed hash (AKH) scheme [35, 34] to support equality tests. We review AKH and then describe how we build $\mathsf{Eunomia}^{\mathsf{KH}}$ on it.

Abstractly, AKH provides three functions: $\mathsf{Hash}(k,v) = \mathsf{P} \times \mathsf{DET}(k_{\mathsf{master}}, v) \times k$ (P is a point on an elliptic curve, $\mathsf{DET}(\cdot,\cdot)$ is the deterministic encryption function, and k_{master} is a master encryption key), $\mathsf{Token}(k_1,k_2) = k_2 \times k_1^{-1}$ and $\mathsf{Adjust}(w,\Delta) = w \times \Delta$. $\mathsf{Hash}(k,v)$ returns a keyed hash of v with key k on a pre-determined elliptic curve with public parameters. $\mathsf{Token}(k_1,k_2)$ returns a token $\Delta_{k_1 \mapsto k_2}$, which allows *transforming* hashes created with key k_1 to corresponding hashes created with k_2. The function $\mathsf{Adjust}(w,\Delta)$ performs this transformation: If $w = \mathsf{Hash}(k_1,v)$ and $\Delta = \Delta_{k_1 \mapsto k_2}$, then $\mathsf{Adjust}(w,\Delta)$ returns the value $\mathsf{Hash}(k_2,v)$. The AKH scheme allows the adversary to compare two values hashed with keys k_1 and k_2 for equality only when it knows either $\Delta_{k_1 \mapsto k_2}$ or $\Delta_{k_2 \mapsto k_1}$. Popa *et al.* prove this security property, reducing it to the elliptic-curve decisional Diffie-Hellman assumption [35].

To encrypt a log in $\mathsf{Eunomia}^{\mathsf{KH}}$, we generate two keys k_h, k_e for each column. These are called the hash key and the encryption key, respectively. Each cell v in the column is transformed into a pair $\langle \mathsf{Hash}(k_h,v), \mathsf{Encrypt}(k_e,v)\rangle$. Here, $\mathsf{Hash}(k_h,v)$ is the AKH hash of v with key k_h and $\mathsf{Encrypt}(k_e,v)$ is a standard probabilistic encryption of v with key k_e.[1] Columns do not share any keys. If audit on a policy requires testing columns $\mathsf{t_1.a_1}$ and $\mathsf{t_2.a_2}$ for equality and these columns have hash keys k_{h_1} and k_{h_2}, then the audit algorithm is given one of the tokens $\Delta_{k_{h_1} \mapsto k_{h_2}}$ and $\Delta_{k_{h_2} \mapsto k_{h_1}}$. The algorithm can then transform hashes to test for equality. Each execution of the audit process can be given a different set of tokens depending on the policy being audited and, hence, unlike $\mathsf{Eunomia}^{\mathsf{DET}}$, the same encrypted log supports audit over any policy. However, equality testing is more expensive now as it invokes the $\mathsf{Adjust}()$ function. This increases the runtime overhead of audit.

Formally, $\mathsf{Eunomia}^{\mathsf{KH}}$ contains the following four algorithms: $\mathsf{KeyGen}^{\mathsf{KH}}(1^\kappa,\mathcal{S})$, $\mathsf{EncryptLog}^{\mathsf{KH}}(\mathcal{L},\mathcal{S},\mathcal{K})$, $\mathsf{EncryptPolicyConstants}^{\mathsf{KH}}(\varphi,\mathcal{K})$, and $\mathsf{GenerateToken}(\mathcal{S},\delta,\mathcal{K})$.

Key generation. The probabilistic key generation algorithm $\mathsf{KeyGen}^{\mathsf{KH}}(\cdot,\cdot)$ takes as input a security parameter and a log schema $\mathcal{S}$ and returns a key set $\mathcal{K}$. $\mathcal{K}$ contains tuples of the form $\langle \mathsf{p}, \mathsf{a}, k_h, k_e\rangle$, meaning that column $\mathsf{p.a}$ has hash key k_h and encryption key k_e. The only constraint is that if $\mathsf{p.a}$ contains timestamps, then $k_h = K_{\mathsf{time}}$.

Encrypting the log. The algorithm $\mathsf{EncryptLog}^{\mathsf{KH}}(\cdot,\cdot,\cdot)$ takes as arguments a plaintext log $\mathcal{L}$, its schema $\mathcal{S}$ and a key set $\mathcal{K}$. It returns a pair $e\mathcal{L} = \langle e\mathsf{DB}, e\mathcal{T}\rangle$ where, $e\mathsf{DB}$ is the cell-wise encryption of $\mathcal{L}$ with appropriate keys from $\mathcal{K}$ and $e\mathcal{T}$ is the **mOPED** encoding. Because each cell maps to a pair, each table has twice as many columns in $e\mathsf{DB}$ as in $\mathcal{L}$.

Encrypting policy constants. To audit over $\mathsf{Eunomia}^{\mathsf{KH}}$ encrypted logs, constants in the policy must be encrypted. The algorithm $\mathsf{EncryptPolicyConstants}^{\mathsf{KH}}(\cdot,\cdot)$ takes as input a plaintext policy φ, and a key set $\mathcal{K}$, and returns a policy φ' in which constants have been encrypted with appropriate keys taken from $\mathcal{K}$: If constant c appears in the ith position of predicate p in φ and the hash and encryption keys of the ith column of p in $\mathcal{K}$ are k_h and k_e, respectively, then the constant c is replaced by $\langle \mathsf{Hash}(k_h,v), \mathsf{Encrypt}(k_e,v)\rangle$ in φ'.

Generating tokens. $\mathsf{GenerateToken}(\cdot,\cdot,\cdot)$ is used to generate tokens that are given to the audit algorithm to enable it to test for equality on the encrypted log. For each tuple $\langle \mathsf{p.a_1}, \mathsf{q.a_2}\rangle$ in δ, the algorithm $\mathsf{GenerateToken}(\mathcal{S},\delta,\mathcal{K})$ returns the tuple $\langle \mathsf{p.a_1}, \mathsf{q.a_2}, \Delta_{k_1 \mapsto k_2}\rangle$, where $\langle \mathsf{p}, \mathsf{a_1}, k_1, \_\rangle \in \mathcal{K}$ and $\langle \mathsf{q}, \mathsf{a_2}, k_2, \_\rangle \in \mathcal{K}$.

Remarks. From the perspective of confidentiality, the same amount of information is revealed irrespective of whether the audit algorithm (which may be compromised by the adversary) is given $\Delta_{k_{h_1} \mapsto k_{h_2}}$ or $\Delta_{k_{h_2} \mapsto k_{h_1}}$, because each token can be computed from the other. However, the actual token used for comparison by the audit algorithm can have a significant impact on its performance. Consider Policy 1 from Section 1, which stipulates that each name appearing in table T_1 appears in T_2 with the role Doctor. The audit process will iterate over the names in T_1 and look up those names in T_2. Consequently, for performance, it makes sense to index the hashes of the names in T_2 and for the audit algorithm to use the token $\Delta_{k_1 \mapsto k_2}$, where k_1 and k_2 are the hash keys of names in T_1 and T_2, respectively. If, instead, the algorithm uses $\Delta_{k_2 \mapsto k_1}$, then indexing is ineffective and performance suffers. The bottom line is that *directionality* of information flow during equality testing matters for $\mathsf{Eunomia}^{\mathsf{KH}}$. Our policy analysis, which determines the columns that may be tested for equality during audit (Section 7) takes this directionality into account. The equality scheme δ returned

[1] The $\mathsf{Encrypt}(k_e,v)$ component of the ciphertext is returned to the client Cl as part of the audit output. Cl then decrypts it to obtain concrete policy violations.

by this analysis is directional (even though the use of δ in $\mathsf{Eunomia}^{\mathsf{DET}}$ ignored this directionality): if $\langle \mathsf{p}_1.\mathsf{a}_1, \mathsf{p}_2.\mathsf{a}_2 \rangle \in \delta$, and $\mathsf{p}_1.\mathsf{a}_1$ and $\mathsf{p}_2.\mathsf{a}_2$ have hash keys k_1 and k_2, then the audit algorithm uses the token $\Delta_{k_1 \mapsto k_2}$, not $\Delta_{k_2 \mapsto k_1}$.

4.4 Mutable Order Preserving Encoding with Displacements (mOPED)

We now discuss the **mOPED** scheme which produces a data structure, $e\mathcal{T}$, that allows computation of the boolean value $t_1 + d_1 \leq t_2 + d_2$ on the cloud, given only $\mathsf{Enc}(t_1)$, $\mathsf{Enc}(t_2)$, $\mathsf{Enc}(d_1)$ and $\mathsf{Enc}(d_2)$. Here, $\mathsf{Enc}(t)$ denotes the deterministic encryption of t (in the case of $\mathsf{Eunomia}^{\mathsf{DET}}$) or the AHK hash of t (in the case of $\mathsf{Eunomia}^{\mathsf{KH}}$) with the fixed key K_{time}. The scheme **mOPED** extends a prior scheme **mOPE** [33], which is a special case $d_1 = d_2 = 0$ of our scheme.

Consider first the simple case where the log $\mathcal{L}$ and the policy φ are fixed. This means that the set T of values of the form $t + d$ that the audit process may compare to each other is also fixed and finite (because t is a timestamp on the finite log $\mathcal{L}$ and $d \in D$ is a displacement occurring in the finite policy φ). Suppose that the set T has size N (note that $N \in O(|D| \cdot |\mathcal{L}|)$. Then, the client can store on the cloud a map $e\mathcal{T} : \mathsf{EncTimeStamp} \times \mathsf{EncD} \rightarrow \{1, \ldots, N\}$, which maps each encrypted timestamp $\mathsf{Enc}(t)$ and each encrypted displacement $\mathsf{Enc}(d)$ to the relative order of $t + d$ among the elements of T. To compute $t_1 + d_1 \leq t_2 + d_2$, the audit process can instead compute $e\mathcal{T}(\mathsf{Enc}(t_1), \mathsf{Enc}(d_1)) \leq e\mathcal{T}(\mathsf{Enc}(t_2), \mathsf{Enc}(d_2))$. The map $e\mathcal{T}$ can be represented in many different ways. In our implementation, we use nested hash tables, where the outer table maps $\mathsf{Enc}(t)$ to an inner hash table and the inner table maps $\mathsf{Enc}(d)$ to the relative order of $t+d$. For audit applications where the log and policy are fixed upfront, this simple data structure $e\mathcal{T}$ suffices.

The scheme **mOPED** is more general and allows the client to incrementally update $e\mathcal{T}$ on the cloud. This is relevant when either the policy or the log changes often. A single addition or deletion of t or d can cause the map $e\mathcal{T}$ to change for potentially all other elements and, hence, a naive implementation of $e\mathcal{T}$ may incur cost linear in the current size of T for single updates. Popa *et al.* show how this cost can be made logarithmic by interactively maintaining a balanced binary search tree over encrypted values $\mathsf{Enc}(t)$ and using *paths* in this search tree as the co-domain of $e\mathcal{T}$. We extend this approach by maintaining a binary search tree over pairs $(\mathsf{Enc}(t), \mathsf{Enc}(d))$, where the search order reflects the natural order over $t + d$. Since the cloud never sees plaintext data, the update of this binary search tree and the map $e\mathcal{T}$ *must* be interactive with the client. We omit the details of this interactive update and refer the reader to [33] for details. As the cloud may be compromised, the security property we prove of **mOPED** (Section 5.1) holds despite the adversary observing every interaction with the client. We note that an audit algorithm never updates $e\mathcal{T}$, so its execution remains non-interactive.

5. SECURITY ANALYSIS

We now prove that our schemes $\mathsf{Eunomia}^{\mathsf{DET}}$, $\mathsf{Eunomia}^{\mathsf{KH}}$ and **mOPED** are secure. We start with **mOPED**, because $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ rely on it.

5.1 Security of mOPED

We formalize the security of **mOPED** as an indistinguishability game in which the adversary provides two sequences of timestamps and a set of displacements D, then observes the client and server construct the **mOPED** data structure $e\mathcal{T}$ on one of these sequences chosen randomly and then tries to guess which sequence it is. We call this game IND-CDDA (indistinguishability under chosen distances with displacement attack). This definition is directly based on the IND-OCPA (indistinguishability under ordered chosen plaintext attack) definition by Boldyreva *et al.* [10] and the LoR security definition by Pandey and Rouselakis [31]. Because $e\mathcal{T}$ intentionally reveals the relative order of all timestamps after displacement with constants in D, we need to impose a constraint on the two sequences chosen by the adversary. Let $\vec{u}[i]$ denote the ith element of the sequence $\vec{u}$. We say that two sequences of timestamps $\vec{u}$ and $\vec{v}$ are equal up to distances with displacements D, written $\mathsf{EDD}(\vec{u}, \vec{v}, D)$ iff $|\vec{u}| = |\vec{v}|$ and $\forall d, d' \in D, i, j.\ (\vec{u}[i]+d \geq \vec{u}[j]+d') \Leftrightarrow (\vec{v}[i]+d \geq \vec{v}[j] + d')$. We describe here the IND-CDDA game and the security proof for **mOPED** with deterministic encryption; the case of **mOPED** with AKH hashes is similar.

IND-CDDA **game.** The IND-CDDA security game between a client or challenger Cl (*i.e.*, owner of the audit log) and an adaptive, probabilistic polynomial time (ppt) adversary Adv for the security parameter κ proceeds as follows:

1. Cl generates a secret key K_{time} using the probabilistic key generation algorithm KeyGen. $K_{\mathsf{time}} \xleftarrow{\$} \mathsf{KeyGen}(1^{\kappa})$.
2. Cl chooses a random bit b. $b \xleftarrow{\$} \{0, 1\}$.
3. Cl creates an empty $e\mathcal{T}$ on the cloud.
4. Adv chooses a set of distances $D = \{d_1, \ldots, d_n\}$ and sends it to Cl.
5. Cl and Adv engage in a polynomial of κ number of rounds of interactions. In each round j:
   (a) Adv selects two values v_j^0 and v_j^1 and sends them to Cl.
   (b) Cl deterministically encrypts the following $n + 1$ values v_j^b, $v_j^b + d_1, v_j^b + d_2, \ldots, v_j^b + d_n$ using K_{time}.
   (c) Cl interacts with the cloud to insert $\mathsf{DET}(K_{\mathsf{time}}, v_j^b)$ and $\{\mathsf{DET}(K_{\mathsf{time}}, v_j^b + d_i)\}_{i=1}^n$ into $e\mathcal{T}$. The adversary observes this interaction and the cloud's complete state, but not Cl's local computation.
6. Adv outputs his guess b' of b.

Adv wins the IND-CDDA security game iff:

1. Adv guesses b correctly (*i.e.*, $b = b'$);
2. $\mathsf{EDD}([v_0^0, \ldots, v_m^0], [v_0^1, \ldots, v_m^1], D)$ holds, where m is the number of rounds played in the game.

Let $\mathsf{win}^{\mathsf{Adv},\kappa}$ be a random variable which is 1 if the Adv wins and 0 if Adv loses. Recall that a function $f : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible* with respect to its argument κ, if for every $c \in \mathbb{N}$ there exists another integer K such that for all $\kappa > K$, $f(\kappa) < x^{-c}$. We write $\mathsf{negl}(\kappa)$ to denote some negligible function of κ.

Theorem 1 (Security of mOPED with deterministic encryption) *Assuming that deterministic encryption is a* *pseudorandom function, our* **mOPED** *scheme is* IND-CDDA *secure, i.e.,* $\mathsf{Pr}[\mathsf{win}^{\mathsf{Adv},\kappa} = 1] \le \frac{1}{2} + \mathsf{negl}(\kappa)$ *where the probability is taken over the random coins used by* Adv *as well as the random coins used in choosing the key and the bit* b.

PROOF. By a hybrid argument. We augment a similar proof of security for the **mOPE** scheme [33] to also take displacements D into account. □

mOPED's security degrades with the size of the constants' set D occurring in the audited policies. If a system's policies do not use constants, **mOPED** is as secure as its predecessor **mOPE**.

Security of mOPED with AKH hash. The security game for **mOPED** with AKH hashes is very similar to IND-CDDA. We replace $\mathsf{DET}(\cdot,\cdot)$ with $\mathsf{Hash}(\cdot,\cdot)$ in the game. The proof is in the standard model and reduces to the security of AKH [35, Definition 4] and finally to the elliptic-curve decisional Diffie-Hellman (ECDDH) assumption.

5.2 Security of Eunomia$^{\mathbf{DET}}$

We prove security for Eunomia$^{\mathsf{DET}}$, formalized as an indistinguishability game. We first define a notion of log equivalence that characterizes the confidentiality achieved by Eunomia$^{\mathsf{DET}}$ (and, as we explain later, by Eunomia$^{\mathsf{KH}}$). This notion is a central contribution of our work. The security theorem in this section shows that by looking at the Eunomia$^{\mathsf{DET}}$ encryption of a log, an adversary can learn only that the log belongs to its equivalence class (with non-negligible probability). Hence, the equivalence class of the log represents the uncertainty of the adversary about the log's contents and, therefore, characterizes what confidentiality the scheme provides.

Definition 1 (Plaintext log equivalence) *Given any two plaintext audit logs* $\mathcal{L}_1$ *and* $\mathcal{L}_2$*, an equality scheme* δ*, a set of constants* C *and a set of displacements* $D \subseteq C$*,* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are equivalent, denoted by* $\mathcal{L}_1 \equiv_{(\delta,C,D)} \mathcal{L}_2$*, if and only if all of the followings hold:*

1. $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *have the same schema and tables of the same name in* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *have the same number of records (rows).*
2. *For each equivalence class of columns defined by* δ*, there is a bijection from values of* $\mathcal{L}_1$ *to values of* $\mathcal{L}_2$*. (By equivalence class of columns defined by* δ*, we mean an equivalence class of columns defined by the reflexive, symmetric, transitive closure of* δ*.) For a table* t *and a column* a*, let* $\mathcal{M}_{\mathsf{t},\mathsf{a}}$ *denote the bijection corresponding to the equivalence class of* δ *in which* (t,a) *lies. Let* v *be the value in some row* i *of the table* t*, column* a *in* $\mathcal{L}_1$*. Then,*
   (a) *The value in the ith row of table* t*, column* a *in* $\mathcal{L}_2$ *is* $\mathcal{M}_{\mathsf{t},\mathsf{a}}(v)$*.*
   (b) *If* $v \in C$*, then* $\mathcal{M}_{\mathsf{t},\mathsf{a}}(c) = c$*.*
   (c) $|v| = |\mathcal{M}_{\mathsf{t},\mathsf{a}}(v)|$*.*
3. *Let* $\mathsf{timeStamps}(\mathcal{L}_1)$ *be the sequence of timestamps in* $\mathcal{L}_1$ *obtained by traversing the tables of* $\mathcal{L}_1$ *in any order and the timestamps within each table in row order. Let* $\mathsf{timeStamps}(\mathcal{L}_2)$ *be the timestamps in* $\mathcal{L}_2$ *obtained similarly, traversing tables in the same order. Then,* $\mathsf{EDD}(\mathsf{timeStamps}(\mathcal{L}_1), \mathsf{timeStamps}(\mathcal{L}_2), D)$ *holds.*

Intuitively, each clause of the above definition specifies a property of logs that a log's encryption under either of our schemes may reveal to an adversary. Everything else about the encrypted log remains hidden. We list below the revealed properties to which each clause of the above definition corresponds:

1. Schema of the log and the number of records in each of its tables.

2a. If two columns can be joined (according to δ) then for two values, one from each column, whether the values are equal.

2b. The encryption(s) of any constant that appears in the policy.

2c. The length of each value in the log.

3. The relative order of all timestamps in the log, displaced by constants in D and by 0.

No other information about the log can be recovered by an adversary looking at the log's encryption. In particular, a log's encryption reveals neither the actual values on the log (other than constants occurring in the policy), nor the equality between values in non-joinable columns. Note that all revealed information is either necessary to execute audit queries or it cannot be hidden even with (cell-granularity) semantically-secure encryption.

We now define the IND-CPLA$^{\mathsf{DET}}$ game, which formalizes when an adversary Adv breaks the security of Eunomia$^{\mathsf{DET}}$. IND-CPLA$^{\mathsf{DET}}$ stands for indistinguishability under the chosen plaintext log attack.

IND-CPLA$^{\mathsf{DET}}$ game. The IND-CPLA$^{\mathsf{DET}}$ game is played between a client or challenger Cl and an adversary Adv for all large enough security parameters κ.

1. Adv picks a log schema $\mathcal{S}$, the sets C, D and an equality scheme δ and gives these to Cl.
2. Cl probabilistically generates a set of secret keys $\mathcal{K}$ based on the sufficiently large security parameter κ, the log schema $\mathcal{S}$, and the equality scheme δ. $\mathcal{K} \xleftarrow{\$} \mathsf{KeyGen}^{\mathsf{DET}}(1^\kappa, \mathcal{S}, \delta)$.
3. Cl randomly selects a bit b. $b \xleftarrow{\$} \{0,1\}$.
4. Adv chooses two plaintext audit logs $\mathcal{L}_0$ and $\mathcal{L}_1$ such that both $\mathcal{L}_0, \mathcal{L}_1$ have schema $\mathcal{S}$, $\mathcal{L}_0 \equiv_{(\delta,C,D)} \mathcal{L}_1$, $\mathcal{L}_0 \neq \mathcal{L}_1$, and sends $\mathcal{L}_0$, $\mathcal{L}_1$ to Cl.
5. Following the scheme Eunomia$^{\mathsf{DET}}$, Cl deterministically encrypts $\mathcal{L}_b$ according to the key set $\mathcal{K}$ to obtain the encrypted audit log $e\mathsf{DB}_b$.

   $\langle e\mathsf{DB}_b, e\mathcal{T}_b\rangle \leftarrow \mathsf{EncryptLog}^{\mathsf{DET}}(\mathcal{L}_b, \mathcal{S}, \mathcal{K})$.

   It then constructs the **mOPED** data structure $e\mathcal{T}_b$. Adv may observe the construction of the **mOPED** data-structure $e\mathcal{T}_b$ passively. Cl then sends $\langle e\mathsf{DB}_b, e\mathcal{T}_b\rangle$ to Adv.
6. For any constant $c \in C$, if c appears in table t, column a of $\mathcal{L}_b$, then Cl gives Adv the encryption of c with the encryption key of column a.
7. Adv runs a probabilistic algorithm that may invoke the encryption oracle on keys from $\mathcal{K}$ but never asks for the encryption of any value in $\mathcal{L}_0$ or $\mathcal{L}_1$.
8. Adv outputs its guess b' of b.

Adv wins the $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ game *iff* $b = b'$. Let the random variable $\mathsf{win}^{\mathsf{Adv}}_{\mathsf{DET}}$ be 1 if the Adv wins and 0 otherwise.

Theorem 2 (Security of $\mathsf{Eunomia}^{\mathsf{DET}}$) *If deterministic encryption is a pseudorandom function, then the encryption scheme* $\mathsf{Eunomia}^{\mathsf{DET}}$ *is* $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ *secure, i.e., for any ppt adversary* Adv *and sufficiently large* κ, $\mathsf{Pr}[\mathsf{win}^{\mathsf{Adv}}_{\mathsf{DET}} = 1] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$ *where the probability is taken over the random coins used by* Adv *as well as the random coins used in choosing keys and the random bit* b.

Proof. By hybrid argument. We successively replace uses of deterministic encryption with a random oracle. If the Adv can distinguish two consecutive hybrids with non-negligible probability, it can also distinguish a random oracle from a pseudorandom function, which is a contradiction. □

Intuitively, this theorem says that any adversary cannot distinguish two equivalent logs if they are encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$, except with negligible probability. An immediate corollary of this theorem is that a *passive* cloud that observes the execution of our audit algorithm **ereduce** on a log encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$ learns only the $\equiv_{(\delta,C,D)}$ class of the log and, hence, only the log properties listed earlier in this section. This follows because **ereduce** can be simulated by the adversary.

5.3 Security of Eunomia$^{\mathrm{KH}}$

We now define and prove security for the log encryption scheme $\mathsf{Eunomia}^{\mathsf{KH}}$. The security game, $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$, is similar $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ and uses the same definition of log equivalence. The proof of security for $\mathsf{Eunomia}^{\mathsf{KH}}$ is in the standard model and reduces to the ECDDH assumption.

$\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$ game. The $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$ game is played between a challenger Cl and a PPT adversary Adv for all large enough security parameters κ. It is very similar to the $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ security game but has the following differences. All the encryption is done using the $\mathsf{Eunomia}^{\mathsf{KH}}$ scheme. Additionally, after step 5, Cl generates the token list $\vec{\Delta}$ according to δ and send its to Adv. Adv wins the $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$ game *iff* $b = b'$. Let the random variable $\mathsf{win}^{\mathsf{Adv}}_{\mathsf{KH}}$ be 1 if the Adv wins and 0 otherwise.

Theorem 3 (Security of $\mathsf{Eunomia}^{\mathsf{KH}}$) *If the ECDDH assumption holds and the encryption scheme used in* $\mathsf{Eunomia}^{\mathsf{KH}}$ *is IND-CPA secure, then* $\mathsf{Eunomia}^{\mathsf{KH}}$ *is* $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$ *secure, i.e., for any ppt adversary* Adv *and sufficiently large* κ*, the following holds:* $\mathsf{Pr}[\mathsf{win}^{\mathsf{Adv}}_{\mathsf{KH}} = 1] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$*, where the probability is taken over the random coins used by* Adv *as well as the random coins used in choosing keys and the random bit* b.

Proof. By hybrid argument, we reduce to the IND-CPA security of encryption and the security of AKH [35, Definition 4]. The latter relies on the ECDDH assumption. □

Generalizing the security definitions. $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ and $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$ security games cover only a single round of interaction between the adversary and the challenger. It is possible to extend the games to a polynomial number of interactions and maintain the security theorems. However, we must assume that the adversary chooses more precise logs in each interaction, *i.e.*, if the adversary chooses logs $\mathcal{L}^i_0$, $\mathcal{L}^i_1$ in the ith interaction and $\mathcal{L}^{i+1}_0$, $\mathcal{L}^{i+1}_1$ in the $(i+1)$-th interaction, then $\mathcal{L}^{i+1}_0 \geq \mathcal{L}^i_0$ and $\mathcal{L}^{i+1}_1 \geq \mathcal{L}^i_1$. Recall that, $\mathcal{L}_1 \geq \mathcal{L}_2$ means the log $\mathcal{L}_1$ extends the log $\mathcal{L}_2$ with additional information, possibly, replacing unknown values in $\mathcal{L}_1$ with either true or false.

Attacks using frequency analysis. Similar to prior work based on deterministic encryption, our security games (*i.e.*, $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ and $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$) and security theorems (*i.e.*, Theorems 2 and 3) implicitly assume that all plaintext logs within an equivalence class are equally likely *a priori.* This is because in both games, the value b is chosen without bias. If the plaintexts are not uniformly distributed and some auxiliary information about this distribution is known to the adversary, then the security theorems may not apply. In fact, concurrent work by Naveed *et al.* [28] shows that the association between ciphertexts and plaintexts for deterministically encrypted databases can be recovered using frequency analysis, when the adversary knows the distribution of the frequencies of data values in the columns of the plaintext database. However, guessing such distributions for columns containing sensitive information (*e.g.*, SSNs, names) is usually very difficult for an adversary. Naveed *et al.*'s evaluation is based on publicly available plaintext patient record databases containing only non-personally identifiable columns like race, gender, and duration of stay at a hospital.

6. AUDITING ALGORITHM

We now present our auditing algorithm **ereduce**, which adapts the prior algorithm **reduce** [19] to run on logs encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$. Our choice of **reduce** as the basis is motivated by the fact that **reduce** is general enough to capture rich policies, including most privacy clauses of HIPAA and GLBA. The algorithm **ereduce** has two very similar versions that execute on logs encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$. We call these versions **ereduce**$^{\mathsf{DET}}$ and **ereduce**$^{\mathsf{KH}}$, respectively. The principal difference between **reduce** and **ereduce**$^{\mathsf{KH/DET}}$ is that the **ereduce**$^{\mathsf{KH/DET}}$ uses the special **mOPED** data structure to evaluate displaced comparisons. In the following we first describe **ereduce**$^{\mathsf{KH}}$ in detail and then describe how to simplify it to obtain **ereduce**$^{\mathsf{DET}}$.

6.1 Auxiliary Definitions

A *substitution* σ is a finite map from variables to value, *provenance* pairs. Each element in the range of a substitution is of the form $\langle v, \ell \rangle$, where v is the value that the variable is mapped to and ℓ is called the provenance of v. The provenance ℓ indicates which table and which column the value v originated from. ℓ has the form $\mathsf{p.a}$. We often write a substitution σ as a finite list of elements, each of the form $\langle x, v^h, v^e, \ell \rangle$. For any variable x in σ's domain, we use $\sigma(x).\mathsf{hash}$, $\sigma(x).\mathsf{cipher}$, and $\sigma(x).\ell$ to select the hash value (*i.e.*, v^h), the ciphertext value (*i.e.*, v^e), and the provenance (*i.e.*, ℓ), respectively.

We say substitution σ_1 extends σ_2 (denoted $\sigma_1 \geq \sigma_2$) if σ_1's domain contains σ_2's domain and σ_1 agrees with σ_2 on all variables in σ_2's domain. Given a substitution σ, we define $[\sigma] = \{\langle x, \mathsf{p.a} \rangle \mid \exists v.\sigma(x) = \langle v, \mathsf{p.a} \rangle\}$. We use $\sigma \downarrow X$, where $X \subseteq \mathsf{domain}(\sigma)$, to denote the substitution σ' such that $\sigma \geq \sigma'$ and the domain of σ' contains variables from the set X only. We lift the $\downarrow$ operation to a set of substitutions

Σ pointwise. We use $\bullet$ to denote the identity substitution. We say that a substitution σ *satisfies* a formula g on the ($\mathsf{Eunomia}^{\mathsf{KH}}$-)encrypted log $e\mathcal{L}$ if replacing each free variable x in g with the concrete value $\sigma(x).\mathsf{hash}$ results in a formula that is true on $e\mathcal{L}$.

6.2 Algorithm $\mathbf{ereduce}^{\mathsf{KH}}$

Like its basis **reduce**, the algorithm $\mathbf{ereduce}^{\mathsf{KH}}$ is defined as a recursive function that operates on the logical representation of the policy being audited. It uses two auxiliary functions, $\widehat{\mathsf{esat}}^{\mathsf{KH}}$ and $\mathtt{esat}^{\mathsf{KH}}$. We describe these functions below. To simplify notation, we drop the superscript KH from $\mathbf{ereduce}^{\mathsf{KH}}$, $\widehat{\mathsf{esat}}^{\mathsf{KH}}$ and $\mathtt{esat}^{\mathsf{KH}}$ in the rest of this section and write **ereduce**, $\widehat{\mathsf{esat}}$ and $\mathtt{esat}$ instead.

$\mathbf{ereduce}(e\mathcal{L}, \varphi, \vec{\Delta}, \sigma)$ is the top-level function that takes as input a $\mathsf{Eunomia}^{\mathsf{KH}}$ encrypted audit log $e\mathcal{L}$, a constant encrypted policy φ, a set of tokens $\vec{\Delta}$, and an input substitution σ, and returns a residual policy ψ. ψ represents a part of the original policy φ that cannot be evaluated due to lack of information in $e\mathcal{L}$. We use $\bullet$ as the input substitution to the initial call to **ereduce**.

$\widehat{\mathsf{esat}}(e\mathcal{L}, g, \vec{\Delta}, \sigma)$ is an auxiliary function used by **ereduce** while evaluating quantifiers to get all finite substitutions that satisfy the quantifier's guard formula. It takes as input a $\mathsf{Eunomia}^{\mathsf{KH}}$ encrypted audit log $e\mathcal{L}$, a constant encrypted formula g, a set of tokens $\vec{\Delta}$, and an input substitution σ, and returns all finite substitutions for free variables of g that extend σ and satisfy g with respect to $e\mathcal{L}$.

$\mathtt{esat}(e\mathcal{L}, \mathsf{p}(\vec{t}), \vec{\Delta}, \sigma)$ is an auxiliary function used by $\widehat{\mathsf{esat}}$ for evaluating all finite substitutions that satisfy a given predicate (with an input substitution applied). The inputs $e\mathcal{L}$, $\vec{\Delta}$, and σ have their usual meaning. $\mathsf{p}(\vec{t})$ is a constant encrypted predicate. This function returns all finite substitutions for free variables of $\mathsf{p}(\vec{t})$ that extend the input substitution σ and satisfy $\mathsf{p}(\vec{t})$ on $e\mathcal{L}$. The implementation of $\mathtt{esat}$ is log-representation dependent. Evaluation of the predicate $\mathsf{timeOrder}$ uses the **mOPED** data structure.

ereduce eagerly evaluates as much of the input policy φ as it can; in case it cannot evaluate a portion of φ due to $e\mathcal{L}$'s incompleteness, it returns that portion of φ as part of the result. The return value of **ereduce** is thus a logical formula ψ (called the *residual formula*). Auditing with **ereduce** is an iterative process. When the current log $e\mathcal{L}$ is extended with additional information (thus removing some incompleteness) resulting in the new log $e\mathcal{L}_1$ ($e\mathcal{L}_1 \geq e\mathcal{L}$), we can invoke **ereduce** again with the residual formula ψ as the input policy and $e\mathcal{L}_1$ as the input log.

We present selected cases of **ereduce** in Figure 2. We use the notation $\mathsf{f}(\vec{a}) \Downarrow \psi$ to mean that function f returns ψ when applied to arguments $\vec{a}$. When the formula input to **ereduce** is a predicate $\mathsf{p}(\vec{t})$ (rule R-P), **ereduce** uses σ and $\vec{\Delta}$ to replace all variables in $\mathsf{p}(\vec{t})$ with concrete values (with proper hash adjustments) to obtain a new ground predicate $\mathsf{p}(\vec{t'})$. (A *ground* predicate only has constants as arguments.) Then it consults $e\mathcal{L}$ to check whether $\mathsf{p}(\vec{t'})$ exists. If $e\mathcal{L}(\mathsf{p}(\vec{t'})) = \mathsf{uu}$, indicating the log doesn't have enough information, then **ereduce** returns $\mathsf{p}(\vec{t'})$. Otherwise, it returns either true or false depending on whether there is a row in table p with hash values matching $\vec{t'}$. For example, let us assume that **ereduce** is called with the input substitution $\sigma = [\langle p_1, v^h_{k_1}, *, t.\mathsf{cl}\rangle, \ldots]$ and the input predicate $\mathsf{activeRole}(p_1, \langle \mathsf{doctor}^h_{k_3}, *\rangle)$ (p_1 is a variable and doctor is a constant). $*$ represents a ciphertext that is not important for this example. Let us assume that column 1 of the $\mathsf{activeRole}$ table uses the keys $(k_2, \_)$ whereas in σ, the hash value mapped to p_1 is generated using k_1. Hence, we have to change the value $v^h_{k_1}$ to $v^h_{k_2}$ using the adjustment key $\Delta_{k_1 \mapsto k_2} \in \vec{\Delta}$. Then, using the following SQL query we check whether a row with the appropriate hash values exists: "select * from $\mathsf{activeRole}$ where column1Hash=$v^h_{k_2}$ and column2Hash=$\mathsf{doctor}^h_{k_3}$". If such a row exists, then $\top$ is returned; otherwise, $\bot$ is returned. When **ereduce** is called for $\mathsf{timeOrder}$, the same hash adjustment applies before the **mOPED** data structure is consulted.

$$\frac{\begin{array}{c}\mathsf{p}(\vec{t'}) \leftarrow \forall t_i \in \mathsf{Var}.\ \mathsf{p}(\vec{t})[t_i \mapsto \langle \mathsf{Adjust}(\sigma(t_i).\mathsf{hash}, \\ \Delta_{\sigma(t_i).\ell \to p.i}), \sigma(t_i).\mathsf{cipher}\rangle] \\ P \leftarrow e\mathcal{L}(\mathsf{p}(\vec{t'}))\end{array}}{\mathbf{ereduce}(e\mathcal{L}, \mathsf{p}(\vec{t}), \vec{\Delta}, \sigma) \Downarrow P}\ \text{R-P}$$

$$\frac{\begin{array}{c}\mathbf{ereduce}(e\mathcal{L}, \varphi_1, \vec{\Delta}, \sigma) \Downarrow \varphi'_1 \\ \mathbf{ereduce}(e\mathcal{L}, \varphi_2, \vec{\Delta}, \sigma) \Downarrow \varphi'_2 \quad \psi \leftarrow \varphi'_1 \vee \varphi'_2\end{array}}{\mathbf{ereduce}(e\mathcal{L}, \varphi_1 \vee \varphi_2, \vec{\Delta}, \sigma) \Downarrow \psi}\ \text{R-}\bigvee$$

$$\frac{\begin{array}{c}\widehat{\mathsf{esat}}(e\mathcal{L}, g, \vec{\Delta}, \sigma) \Downarrow \Sigma' \\ \forall \sigma_i \in \Sigma'.\ \mathbf{ereduce}(e\mathcal{L}, \varphi, \vec{\Delta}, \sigma_i) \Downarrow \varphi_i \\ \varphi' \leftarrow \forall \vec{x}.\ (g \wedge \vec{x} \notin [\Sigma' \downarrow \vec{x}] \to \varphi) \\ \psi \leftarrow \bigwedge_i \varphi_i \wedge \varphi'\end{array}}{\mathbf{ereduce}(e\mathcal{L}, \forall \vec{x}.(g \to \varphi), \vec{\Delta}, \sigma) \Downarrow \psi}\ \text{R-}\forall$$

Figure 2: ereduce description

In rule R-$\bigvee$, **ereduce** is recursively called for the two subformulas of the disjunction. The returned residual formula is the disjunction of the residual formulas returned from the two recursive calls.

When the input formula is of the form $\forall \vec{x}.\ (g \to \varphi)$ (rule R-$\forall$), we first use the function $\widehat{\mathsf{esat}}$ (described below) to get all substitutions Σ' for $\vec{x}$ that extend σ and satisfy g on $e\mathcal{L}$. Our EQ mode check (described later), ensures there are only a finite number of such substitutions. For each of these substitutions $\sigma_i \in \Sigma'$, we recursively call **ereduce** for φ to obtain a residual formula φ_i. Then the returned residual formula is $\bigwedge_i \varphi_i \wedge \varphi'$ where φ' ensures that the same substitutions σ_i for $\vec{x}$ are not checked again when $e\mathcal{L}$ is extended.

Next, we explain selected rules for $\widehat{\mathsf{esat}}$ (presented below) with an example.

$$\frac{\Sigma \leftarrow \mathtt{esat}(e\mathcal{L}, \mathsf{p}(\vec{t}), \vec{\Delta}, \sigma)}{\widehat{\mathsf{esat}}(e\mathcal{L}, \mathsf{p}(\vec{t}), \vec{\Delta}, \sigma) \Downarrow \Sigma}\ \text{S-P}$$

$$\frac{\begin{array}{c}\widehat{\mathsf{esat}}(e\mathcal{L}, g_1, \vec{\Delta}, \sigma) \Downarrow \Sigma' \\ \forall \sigma_i \in \Sigma'.\ \widehat{\mathsf{esat}}(e\mathcal{L}, g_2, \vec{\Delta}, \sigma_i) \Downarrow \Sigma_i\end{array}}{\widehat{\mathsf{esat}}(e\mathcal{L}, g_1 \wedge g_2, \vec{\Delta}, \sigma) \Downarrow \bigcup_i \Sigma_i}\ \text{S-}\bigwedge$$

Let us assume $\widehat{\mathsf{esat}}$ is called with the formula $g \equiv p(x) \wedge q(x, y)$ and substitution $\sigma = \emptyset$ (empty) as input. The S-$\wedge$ rule applies and first $\widehat{\mathsf{esat}}$ is recursive called on $p(x)$ and $\sigma = \emptyset$. Now, the rule S-P applies. Here, x is not in the domain of σ, so the $\mathtt{esat}$ function consults $e\mathcal{L}$ (*i.e.*, using SQL query like: "select * from p") to find concrete values of x to make $p(x)$ true. Let us assume that we get $\langle v^h_{k_1}, *\rangle$ (*i.e.*, k_1 is used to hash the column 1 of table p). Then, $\mathtt{esat}$ returns the substitution $\sigma_1 = [\langle x, v^h_{k_1}, *, p.1\rangle]$ as output. Going back to the S-$\wedge$ rule, now the second premise of S-$\wedge$ calls $\widehat{\mathsf{esat}}$ for $q(x, y)$ with each substitution obtained after evaluating $p(x)$, in our case, σ_1. Let us assume that columns 1 and 2 of table q are hashed with keys k_2 and k_3, respectively. While evaluating, $q(x, y)$ with σ_1, S-P rule is used. σ_1 maps variable x with key k_1, so $\mathtt{esat}$ converts $v^h_{k_1}$ to $v^h_{k_2}$ using the token $\Delta_{p.1\to q.1}$. It then tries to get concrete values for y (with respect to given value of x) by consulting table q in $e\mathcal{L}$ using the following SQL query: "select column2Hash, column2Cipher from q where column1Hash=$v^h_{k_2}$". Assuming that the SQL query returns $\langle w^h_{k_3}, *\rangle$ for column 2 (*i.e.*, y), $\mathtt{esat}$ returns the substitution $[\langle x, v^h_{k_1}, *, p.1\rangle, \langle y, w^h_{k_3}, *, q.2\rangle]$.

Obtaining ereduce$^{\mathsf{DET}}$ from ereduce$^{\mathsf{KH}}$. As described above, **ereduce**$^{\mathsf{KH}}$ tracks the provenance of the encrypted data value required for audit. This is not required when logs are encrypted using $\mathsf{Eunomia}^{\mathsf{DET}}$ in place of $\mathsf{Eunomia}^{\mathsf{KH}}$. Therefore, **ereduce**$^{\mathsf{DET}}$ is a simplification of **ereduce**$^{\mathsf{KH}}$. In **ereduce**$^{\mathsf{DET}}$, the substitution σ maps variables to deterministic ciphertexts. Further, in the rules R-P and S-P, no adjustment is needed.

6.3 Properties

We have proved the functional correctness of both algorithms, **ereduce**$^{\mathsf{DET}}$ and **ereduce**$^{\mathsf{KH}}$. We show the correctness theorem for **ereduce**$^{\mathsf{KH}}$ below. The theorem states that the result of decrypting the output (residual policy) of **ereduce** on a $\mathsf{Eunomia}^{\mathsf{KH}}$-encrypted log and the output of **reduce** on the corresponding plaintext log are equal with high probability. A low probability exception exists because hash collisions are possible in $\mathsf{Eunomia}^{\mathsf{KH}}$ (but very unlikely). The function $\mathsf{EncryptSubstitution}^{\mathsf{KH}}$ encrypts a plaintext substitution with provenance. It is very similar to the function $\mathsf{EncryptPolicyConstants}^{\mathsf{KH}}$ (Section 4.3). The notation $\chi_I \vdash \varphi_{\mathsf{P}} : \delta$ refers to the EQ mode check, which is described in Section 7.

Theorem 4 (Correctness of ereduce$^{\mathsf{KH}}$) *For all plaintext policies φ_{P} and ψ_{P}, for all constant encrypted policies φ_{E} and ψ_{E}, for all database schema $\mathcal{S}$, for all plaintext audit logs $\mathcal{L} = \langle \mathsf{DB}, \mathcal{T}\rangle$, for all encrypted audit logs $e\mathcal{L} = \langle e\mathsf{DB}, e\mathcal{T}\rangle$, for all plaintext substitutions σ_{P}, for all encrypted substitutions σ_{E}, for all χ_I, for all equality schemes δ, for all security parameters κ, for all encryption keys $\mathcal{K}$, for all token lists $\vec{\Delta}$, if all of the following hold: (1) $\chi_I \vdash \varphi_{\mathsf{P}} : \delta$, (2) $[\sigma_{\mathsf{P}}] \supseteq \chi_I$, (3) $\mathcal{K} = \mathsf{KeyGen}^{\mathsf{KH}}(\kappa, \mathcal{S})$, $\vec{\Delta} = \mathsf{GenerateToken}(\mathcal{S}, \delta, \mathcal{K})$, (4) $e\mathcal{L} = \mathsf{EncryptLog}^{\mathsf{KH}}(\mathcal{L}, \mathcal{S}, \mathcal{K})$, (5) $\varphi_{\mathsf{E}} = \mathsf{EncryptPolicyConstants}^{\mathsf{KH}}(\varphi_{\mathsf{P}}, \mathcal{K})$, (6) AKH key adjustment is correct, (7) $\sigma_{\mathsf{E}} = \mathsf{Encrypt Substitution}^{\mathsf{KH}}(\sigma_{\mathsf{P}}, \mathcal{K})$, (8) $\psi_{\mathsf{P}} = \mathbf{reduce}(\mathcal{L}, \sigma_{\mathsf{P}}, \varphi_{\mathsf{P}})$, (9) $\psi_{\mathsf{E}} = \mathbf{ereduce}^{\mathsf{KH}}(e\mathcal{L}, \varphi_{\mathsf{E}}, \vec{\Delta}, \sigma_{\mathsf{E}})$, and (10) $\psi'_{\mathsf{P}} = \mathsf{DecryptPolicyConstants}^{\mathsf{KH}}(\psi_{\mathsf{E}}, \mathcal{K})$, then $\psi_{\mathsf{p}} = \psi'_{\mathsf{P}}$ with high probability.*

The correctness theorem for **ereduce**$^{\mathsf{DET}}$ is similar.

7. EQ MODE CHECK

We now present the EQ mode check, which is a static analysis of policies that serves two purposes: (i) It ensures that **ereduce** terminates for any policy that passes the check and (ii) It outputs *the equality scheme* δ of the policy, which is needed for both $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ (see Section 4). The EQ mode check runs time linear in the size of the policy. The EQ mode check extends the mode check described in [19] by additionally carrying provenance and key-adjustment information, which are necessary for **ereduce**$^{\mathsf{KH}}$.

$$\frac{\begin{array}{c}\forall k \in I(\mathsf{p}).\ t_k \in \mathsf{Var} \to t_k \in \mathsf{FE}(\chi_I)\\ \chi_O = \chi_I \cup \left(\bigcup_{j \in O(\mathsf{p}) \wedge t_j \in \mathsf{Var} \wedge t_j \notin \mathsf{FE}(\chi_I)} \langle t_j, \mathsf{p}.j\rangle\right)\\ \delta = \{\langle \mathsf{p}'.i, \mathsf{p}.l\rangle \mid 0 < l \leq \alpha(\mathsf{p}) \wedge t_l \in \mathsf{Var}\\ \wedge \langle t_l, \mathsf{p}'.i\rangle \in \chi_I\}\end{array}}{\chi_I \vdash_{\mathbf{g}} \mathsf{p}(t_1, \ldots, t_n) : \langle \chi_O, \delta\rangle}\ \textsc{g-Pred}$$

$$\frac{\chi_I \vdash_{\mathbf{g}} g_1 : \langle \chi, \delta_1\rangle \qquad \chi \vdash_{\mathbf{g}} g_2 : \langle \chi_O, \delta_2\rangle}{\chi_I \vdash_{\mathbf{g}} g_1 \wedge g_2 : \langle \chi_O, \delta_1 \cup \delta_2\rangle}\ \textsc{g-Conj}$$

$$\frac{\chi_I \vdash_{\mathbf{g}} g_1 : \langle \chi_1, \delta_1\rangle \qquad \chi_I \vdash_{\mathbf{g}} g_2 : \langle \chi_2, \delta_2\rangle}{\chi_I \vdash_{\mathbf{g}} g_1 \vee g_2 : \langle \chi_1 \Cap \chi_2, \delta_1 \cup \delta_2\rangle}\ \textsc{g-Disj}$$

Figure 3: Selected $\chi_I \vdash_{\mathbf{g}} g : \langle \chi_O, \delta\rangle$ judgements

Mode specification. The concept of "*modes*" comes from logic programming [4]. Consider the following example: Predicate $\mathsf{tagged}(m, q, a)$ is true when the message m is tagged with principal q's attribute a. Assuming that the number of possible messages in English language is infinite, the number of concrete values for variables m, q, and a for which tagged holds is also infinite. However, if we are given a concrete message (*i.e.*, concrete value for the variable m), then the number of concrete values for q and a for which tagged holds is finite. Hence, we say the predicate tagged's argument position 1 is the input position (denoted by "+") whereas the argument positions 2 and 3 are output argument positions (denoted by "$-$"). We call such a description of inputs and outputs of a predicate its *mode specification*. The mode specification of a predicate means that given concrete values for variables in the input positions, the number of concrete values for variables in the output position that satisfy the predicate is finite. Hence, $\mathsf{tagged}(m^+, q^-, a^-)$ is a valid mode specification whereas $\mathsf{tagged}(m^-, q^-, a^+)$ is not.

EQ mode checking. EQ mode check uses the mode specification of predicates to check whether a formula is well-moded. EQ mode check has two types of judgements: $\chi_I \vdash_{\mathbf{g}} g : \langle \chi_O, \delta\rangle$ for guards, and $\chi_I \vdash \varphi : \delta$ for policy formulas. Each element of the sets χ_I, χ_O is a pair of form $\langle x, \mathsf{p}.a\rangle$ which signifies that when g or φ is evaluated, a concrete value for the variable x will exist with provenance $\mathsf{p}.a$.

The top level judgement $\chi_I \vdash \varphi : \delta$ states that given ground values for variables in set χ_I, the formula φ is *well-moded* and that audit φ would require the equality checking for column pairs given by δ. We call a given policy φ well-moded if there exists a δ for which we can prove $\{\} \vdash \varphi : \delta$. The judgement $\vdash$ uses $\vdash_{\mathbf{g}}$ as a sub-judgement in the quantifier case. We explain $\vdash_{\mathbf{g}}$ first. The judgement $\chi_I \vdash_{\mathbf{g}} g : \langle \chi_O, \delta\rangle$ states that given concrete values for variables in the set χ_I, the number of concrete values for variables in the set χ_O (χ_O is a subset of the free variables of g) for which the formula g holds true is finite. It also outputs the column pairs which may be checked for equality during the evaluation of g.

Selected mode checking rules for guards are listed in Figure 3. We explain these rules using an example. We show how to check the formula $g = (p(x^-) \vee q(x^-, z^-)) \wedge r(x^+, y^-)$ with $\chi_I = \{\}$. The function I (resp., O) takes as input a predicate p and returns all input (resp., output) argument positions of p. For instance, $I(r) = \{1\}$ and $O(r) = \{2\}$.

First, the rule G-CONJ applies. The first premise of G-CONJ requires that $g_1^c = (p(x^-) \vee q(x^-, z^-))$ is well-moded with $\chi_I = \{\}$. The rule G-DISJ can be used to check the well-modedness of g_1^c with $\chi_I = \{\}$. The first and second premise require $g_1^d = p(x^-)$ and $g_2^d = q(x^-, z^-)$ to be independently well-moded with the input $\chi_I = \{\}$. While checking $p(x^-)$ with $\chi_I = \{\}$ we see that the rule G-PRED applies. The first premise of G-PRED checks whether all input variables of p, none in this case, are included in χ_I; this is trivially satisfied here. We use an auxiliary function FE for checking this, defined as follows: $\mathsf{FE}(\chi_I) = \{x \mid \exists \mathsf{p}, i. \langle x, \mathsf{p}.i \rangle \in \chi_I\}$. When p is evaluated, we will get concrete values for variable(s) in output positions of p (*i.e.*, x in this case with provenance $p.1$), hence $\chi_O = \{\langle x, p.1 \rangle\}$. This is formalized in premise 2. Finally, because $\chi_I = \{\}$, so we will not need any equality comparisons in evaluating p, so $\delta = \{\}$ (premise 3). Similarly, we can derive, $\{\} \vdash_{\mathbf{g}} q(x^-, z^-) : \langle \{\langle x, q.1 \rangle, \langle z, q.2 \rangle\}, \{\} \rangle$. Once we have established that both g_1^d and g_2^d are well-moded, we see that we are only guaranteed to have a concrete value for variable x after $g_1^d \vee g_2^d$ has evaluated (if g_1^d is true we will not get any concrete value for z, which appears only in g_2^d), but x can have provenance $p.1$ or $q.1$. We have to keep track of both, which is captured using the $\Cap$ operator defined as follows: $\chi_1 \Cap \chi_2 = \{\langle x, p_1.a_1 \rangle \mid \exists p_2, a_2.((\langle x, p_1.a_1 \rangle \in \chi_1 \wedge \langle x, p_2.a_2 \rangle \in \chi_2) \bigvee (\langle x, p_1.a_1 \rangle \in \chi_2 \wedge \langle x, p_2.a_2 \rangle \in \chi_1)).\}$. So we have, $\{\} \vdash_{\mathbf{g}} g_1^c : \langle \{\langle x, p.1 \rangle, \langle x, q.1 \rangle\}, \{\} \rangle$.

Next, we return to the second premise of G-CONJ, which requires that $r(x^+, y^-)$ be well-moded with respect to $\chi = \{\langle x, p.1 \rangle, \langle x, q.1 \rangle\}$. The rule G-PRED applies again. Its first premise, which requires that variables in input argument position (x in this case) be included in χ_I, is satisfied. According to the second premise, we will get concrete values for y with provenance $r.2$ when the predicate is evaluated, so $\chi_O = \{\langle x, p.1 \rangle, \langle x, q.1 \rangle, \langle y, r.2 \rangle\}$. Finally, a concrete value for x (with provenance $p.1$ or $q.1$) is needed while evaluating r (x an input argument of r), hence we need to check for equality between the following column pairs, $p.1$, $r.1$ and $q.1$, $r.1$. Therefore, $\delta = \{\langle p.1, r.1 \rangle, \langle q.1, r.1 \rangle\}$.

Top-level mode checking rules for policy formulas are very similar to those for guards, except that formulas do not ground variables. We show the rule for universal quantification below. Recall that the audit algorithm **ereduce** checks formulas of the form $\forall \vec{x}.\ (g \to \varphi)$ by first obtaining all substitutions for $\vec{x}$ that satisfy g and then checking whether φ holds for each of these substitution.

$$\frac{\begin{array}{cc} \chi_I \vdash_{\mathbf{g}} g : \langle \chi_O, \delta_g \rangle & \vec{x} \subseteq \mathsf{FE}(\chi_O) \\ fv(g) \subseteq \mathsf{FE}(\chi_I) \cup \{\vec{x}\} & \chi_O \vdash \varphi : \delta_c \end{array}}{\chi_I \vdash \forall \vec{x}.\ (g \to \varphi) : \delta_g \cup \delta_c} \ \textsc{Univ}$$

The first premise of UNIV checks that we have only a finite number of substitutions for $\vec{x}$ that satisfy g with respect to χ_I and with equality scheme δ_g. This is necessary for termination of **ereduce**. The second and third premises check that g yields concrete substitutions for all quantified variables $\vec{x}$ and its own free variables. The last premise checks that φ is well-moded.

8. IMPLEMENTATION AND EVALUATION

We have implemented $\mathsf{Eunomia}^{\mathsf{DET}}$, $\mathsf{Eunomia}^{\mathsf{KH}}$, $\mathbf{ereduce}^{\mathsf{DET}}$ and $\mathbf{ereduce}^{\mathsf{KH}}$ using OpenSSL version 1.0.1e. For deterministic encryption, we use AES with a variation of the CMC mode [22] with a fixed IV and a 16-byte block size. We use 256-bit keys. For the AKH scheme, we use the library by Popa *et al.* [34]. The underlying elliptic curve is the NIST-approved NID_X9_62_prime192v1.

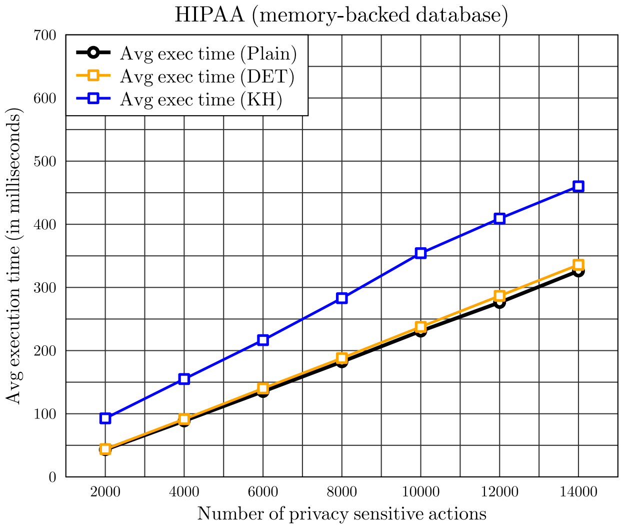

Figure 4: Experimental results on HIPAA policies

We report our empirical evaluation of $\mathbf{ereduce}^{\mathsf{DET}}$ and $\mathbf{ereduce}^{\mathsf{KH}}$. We run experiments on a 2.67GHz Intel Xeon X5650 CPU with Debian Linux 7.6 and 50GB of RAM, of which no more than 3.0 GB is used in any experiment. SQLite version 3.8.7.1 is used to store the plaintext and encrypted logs. We aggressively index all database columns in input argument positions. In $\mathsf{Eunomia}^{\mathsf{DET}}$, the index is built over deterministically encrypted values; in $\mathsf{Eunomia}^{\mathsf{KH}}$, the index is built over hashed values. We use privacy policies derived from the GLBA and HIPAA privacy rules and cover 4 and 13 representative clauses of these rules, respectively.

We use synthetically generated plaintext audit logs. Given an input policy and a desired number of privacy sensitive actions, our audit log generation algorithm randomly decides whether each action will be policy compliant or not. To generate log entries for a compliant action, the algorithm traverses the abstract syntax tree of the policy and generates instances of atoms that together satisfy the policy. For the non-compliant actions, we randomly choose atoms to falsify a necessary condition. Our synthetic log generator also outputs the **mOPED** data structure but with plaintext values for timestamps. We generate logs containing between 2000 and 14000 privacy-sensitive actions. Each plaintext log is separately encrypted with the $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ schemes. The maximum plaintext audit log size we considered is 17 MB. The corresponding maximum encrypted log sizes in $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$ are 67.3MB and 267MB, respectively. Most of the size overhead of the $\mathsf{Eunomia}^{\mathsf{KH}}$-encrypted log comes from the keyed hashes.

We measure the relative overhead of running **ereduce** on logs encrypted with $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$, choosing **reduce** on plaintext audit log as the baseline. We experiment with both RAM-backed and disk-backed versions of SQLite. We report here only the memory-backed results (the disk-backed results are similar). Figure 4 shows the average execution time per privacy-sensitive action for the HIPAA policy in all three configurations (GLBA results are similar). The number of privacy-sensitive actions (and, hence, the log size) varies on the x-axis. The overhead of $\mathsf{Eunomia}^{\mathsf{DET}}$ is very low, between 3% and 9%. This is unsurprising, because no cryptographic operations are needed during audit. The overhead comes from the need to read and compare longer (encrypted) fields in the log and from having to use the **mOPED** data structure. With $\mathsf{Eunomia}^{\mathsf{KH}}$, overheads are much higher, ranging from 63% to 406%. These overheads come entirely from two sources: the cost of reading a much larger database and the cost of performing hash adjustments to check equality of values in different columns. We observe that the overhead due to the increased database size is more than that due to hash adjustment. For the policies we experimented with, the per-action overhead due to database size grows linearly, but the overhead due to hash adjustments is relatively constant. About 30% of **ereduce**'s overhead when running on $\mathsf{Eunomia}^{\mathsf{KH}}$ comes from key-adjustments. Hence, there is room for substantial improvement by caching previous key-adjustments, which we do not do currently.

9. RELATED WORK

Functional and predicate encryption. Functional encryption [13, 20, 27, 30] allows the declassification of any stipulated function of data, given only the ciphertext of the data and a decryption key. Functional encryption can be used to implement audit over encrypted logs: The declassification function can perform the audit and return the outcome. However, existing functional encryption schemes are not efficient enough to be practically usable for audit. Property-preserving encryption [31] and predicate encryption [38, 14, 24] are a special case of functional encryption where the function returns a boolean value. Predicate encryption can also be used to implement audit when the goal is only to find whether or not there is a violation (which is a boolean outcome). However, this is usually insufficient for audit in practice. Pandey and Rouselakis [31] describe several notions of security for symmetric predicate encryption. Our security definition $\mathsf{IND\text{-}CPLA}^{\mathsf{DET}}$ (resp., $\mathsf{IND\text{-}CPLA}^{\mathsf{KH}}$) is inspired by their LoR security definition.

Searchable audit log. Waters *et al.* [41] present a framework for log confidentiality and integrity with the ability to search based on keywords. They use hash chains for integrity and identity-based encryption [12] with extracted keywords to provide confidentiality and search [11]. In our work, we consider confidentiality of the data, but not integrity. Complementary techniques can ensure integrity of the audit log [36, 37, 25, 23]. Our framework supports more expressive policies than that of Waters *et al.* Additionally, audit requires timestamp comparison, which their framework does not support.

Order-preserving encryption. A symmetric encryption scheme that maintains the order of the plaintext data is proposed by Boldyreva *et al.* [10]. This scheme does not satisfy the ideal $\mathsf{IND\text{-}OCPA}$ security definition. Popa *et al.* present the **mOPE** scheme, which we enhance to support timestamp comparison with displacements [33]. Recently, Kerschbaum and Schröpfer present a keyless order-preserving encryption scheme for outsourced data [26]. In their approach, the owner of the plaintext data must keep a dictionary mapping plaintexts to ciphertexts, which would be undesirable in our setting, where the objective is to outsource storage to a cloud.

Querying outsourced database. Hacigümüş *et al.* [21] develop a system that allows querying over encrypted data by asking the client to decrypt data. In contrast, our schemes require no interaction with the client for read-only queries. Tu *et al.* [40] introduce split client/server query execution for processing analytical queries on encrypted databases. Our schemes do not require any query processing on the client-side. Damiani *et al.* [17] develop a secure indexing approach for querying an encrypted database. In contrast, we do not require modification to the indexing algorithm of the DBMS.

CryptDB [34] uses a trusted proxy to dynamically choose an encryption scheme for each database column, based on the query operations being performed on the column. Moreover, CryptDB does not provide a complete, rigorous characterization of its confidentiality properties, which we do. However, CryptDB supports all SQL queries, whereas we cannot support aggregation queries.

Privacy policy compliance checking. Prior work on logic-based compliance checking algorithms focuses on plaintext logs [8, 7, 9, 16, 19]. In particular, this paper adapts our prior work [19] to execute on encrypted logs. The key addition is the EQ mode check, which provides additional information about predicate arguments that may be compared for equality during the audit of a policy.

10. SUMMARY

In this paper, we have presented two database encryption schemes, $\mathsf{Eunomia}^{\mathsf{DET}}$ and $\mathsf{Eunomia}^{\mathsf{KH}}$, that reveal just enough information to allow projection, selection, join, comparison and displaced comparison queries. We present a novel definition of database equivalence, which characterizes the confidentiality properties provided by our schemes. We prove that our schemes are secure. As a concrete application, we show how to execute audit for privacy policy violations over logs that have been encrypted using either of our schemes. This requires a new static analysis of policies, which tracks pairs of columns that may be joined during audit.

Acknowledgements. This work was partially supported by the NSF grants CNS 1064688, CNS 1116991, CNS 1314688, and CCF 042442 and the AFOSR MURI grant FA9550-11-1-0137. The authors thank the anonymous reviewers for their helpful comments.

11. REFERENCES

[1] Health Insurance Portability and Accountability Act, 1996. U.S. Public Law 104-191.

[2] Gramm-Leach-Bliley Act, 1999. U.S. Public Law 106-102.

[3] H. Andréka, I. Németi, and J. van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[4] K. Apt and E. Marchiori. Reasoning about prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 1994.

[5] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption, and key release policies. In *IEEE S&P*, 2007.

[6] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, 2011.

[7] D. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring of temporal first-order properties with aggregations. In *RV*, 2013.

[8] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring compliance policies over incomplete and disagreeing logs. In *RV*, 2012.

[9] A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *RV*. 2013.

[10] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.

[11] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. Cryptology ePrint Archive, Report 2003/195.

[12] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, 2001.

[13] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *TCC*, 2011.

[14] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *TCC*, 2007.

[15] O. Chowdhury, D. Garg, L. Jia, and A. Datta. Equivalence-based Security for Querying Encrypted Databases: Theory and Application to Privacy Policy Audits. Technical Report CMU-CyLab-15-003, Cylab, Carnegie Mellon University, 2015. Available at `http://arxiv.org/abs/1508.02448`.

[16] O. Chowdhury, L. Jia, D. Garg, and A. Datta. Temporal mode-checking for runtime monitoring of privacy policies. In *CAV*, 2014.

[17] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS*, 2003.

[18] H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Datta. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In *WPES*, 2010.

[19] D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *CCS*, 2011.

[20] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, 2006.

[21] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.

[22] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *CRYPTO*, 2003.

[23] J. E. Holt. Logcrypt: Forward security and public verification for secure audit logs. In *ACSW Frontiers*, 2006.

[24] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*, 2008.

[25] J. Kelsey and B. Schneier. Minimizing bandwidth for remote access to cryptographically protected audit logs. In *Recent Advances in Intrusion Detection*, 1999.

[26] F. Kerschbaum and A. Schroepfer. Optimal average-complexity ideal-security order-preserving encryption. In *CCS*, 2014.

[27] A. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *EUROCRYPT*, 2010.

[28] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks against property-preserving encrypted databases. In *CCS*, 2015.

[29] U. D. of Health & Human Services. Cignet Health Fined a $4.3M Civil Money Penalty for HIPAA Privacy Rule Violations. Available at `http://www.hhs.gov/ocr/privacy/hipaa/enforcement/examples/cignetcmp.html`.

[30] A. O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010. `http://eprint.iacr.org/2010/556`.

[31] O. Pandey and Y. Rouselakis. Property preserving symmetric encryption. In *EUROCRYPT*, 2012.

[32] R. A. Popa. *Building practical systems that compute on encrypted data*. PhD thesis, MIT, 2014.

[33] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *IEEE S&P*, 2013.

[34] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[35] R. A. Popa and N. Zeldovich. Cryptographic treatment of CryptDB's adjustable join. Technical Report MIT-CSAIL-TR-2012-006, 2012.

[36] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security Symposium*, 1998.

[37] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM TISSEC*, 2(2):159–176, 1999.

[38] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In *TCC*, 2009.

[39] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE S & P*, 2000.

[40] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *PVLDB*, 2013.

[41] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *NDSS*, 2004.