Confidentiality and Tamper--Resistance of Embedded Databases

Yanli Guo

UNIVERSITE DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Ecole Doctorale Sciences et Technologies de Versailles - STV

Laboratoire PRiSM UMR CNRS 8144

THESE DE DOCTORAT

DE L'UNIVERSITE DE VERSAILLES SAINT–QUENTIN–EN–YVELINES

présentée par : **Yanli GUO**

Pour obtenir le grade de Docteur de l'Université de Versailles Saint-Quentin-en-Yvelines

CONFIDENTIALITE ET INTEGRITE DE BASES DE DONNEES EMBARQUEES

(Confidentiality and Tamper-Resistance of Embedded Databases)

Soutenue le 6 décembre 2011

———

Rapporteurs :

Didier DONSEZ, *Professeur, Université Joseph Fourier, Grenoble*
Patrick VALDURIEZ, *Directeur de recherche, INRIA, Sophia Antipolis*

Examinateurs :

Nicolas ANCIAUX, *Chargé de recherche, INRIA–Paris Rocquencourt, Co-encadrant de thèse*
Luc BOUGANIM, *Directeur de recherche, INRIA–Paris Rocquencourt, Directeur de thèse*
Anne CANTEAUT, *Directeur de recherche, INRIA–Paris Rocquencourt*
Stéphane GANCARSKI, *Maître de Conférences, LIP6 – UPMC, Paris*
Philippe PUCHERAL, *Professeur, Université de Versailles Saint-Quentin*
Patricia SERRANO-ALVARADO, *Maître de Conférences, Université de Nantes*

Table of Content

List of Figures

List of Tables

Abstract

As the amount of personal data collected by governments, institutions and companies increase, centralized data suffer from privacy violations arising from negligence, abusive use, internal and external attacks. Therefore, we envision a radically different, fully decentralized vision of personal data managed within a Personal Data Server (PDS) where the data is let under the control of its owner. The PDS approach resorts to Secure Portable Token (SPT), a new device which combines the tamper resistance of a smart card microcontroller with the mass storage capacity of NAND Flash. The data will be stored, accessed and its access rights controlled using such devices. To support powerful PDS application requirements, a full-fledged DBMS engine is embedded in the device. In order to cope with the SPT's hardware constraints, new database techniques have been introduced, based on pure sequential data structures to store the raw data, and also indexes, buffers, transaction logs, etc. To reach high scalability limits, the data structures must timely be stratified, i.e., reorganized into more efficient (but still sequential) data structures.

In this novel context, the current thesis addresses the security techniques required to take full benefits of such a decentralized vision. Indeed, only the microcontroller of the SPT is secured. The database stored on the NAND Flash linked to the controller by a bus, remains outside the security perimeter of the microcontroller. As such, it may still suffer from confidentiality and integrity attacks. Moreover, PDS also relies on external supporting servers to provide data durability, availability, or other global processing functionalities. In the study, we consider the supporting servers as Honest-but-Curious, i.e., they correctly provide the services that are expected from them (typically serve store, retrieve, and delete data requests), but they may try to breach the confidentiality of any data that is stored locally. Therefore, appropriate secure protocols must be devised to achieve the services delegated to the supporting servers without breach.

This thesis aims at providing security solutions to ensure the security of personal data by relying on cryptography techniques, without incurring large overhead to the existing system. More precisely, (i) we propose a set of light-weight, secure crypto-protection building blocks by considering hardware constraints and PDS engine's features, to fight against confidentiality and integrity attacks on Flash. (ii) We propose a preliminary design for the communication protocols between PDSs and supporting servers, such that the latter can serve requests correctly and

securely (store, retrieve or delete data). (iii) Finally, we do a set of experiments with synthetic data and SPT simulator to evaluate the impact of the designed building blocks on the performance of PDS engine. The results show the effectiveness of our proposals.

The proposed solutions can be reusable in other (more general) context or provide more opportunities for the coming works combining database security and cryptography. For example, smart selection enable selections on encrypted data by using traditional encryption algorithms in a novel way, thus it opens a different direction for executing queries on encrypted database (e.g., traditional centralized DBMS).

Résumé en Français

La quantité de données personnelles collectées par les gouvernements, les institutions et les entreprises augmente considérablement. Ces données sont stockées sur des serveurs et sont la cible de violations de confidentialité provenant de négligence, utilisation abusive, d'attaques internes ou externes. Nous envisageons une approche radicalement différente, une vision totalement décentralisée dans laquelle les données personnelles sont gérées au sein de serveurs personnel de données (PDS) et laissées sous le contrôle de son propriétaire. L'approche PDS s'appuie sur un nouveau composant matériel, le Secure Portable Token (SPT), qui combine un microcontrôleur de carte à puce protégé matériellement contre toute attaque avec une mémoire de masse de type flash NAND. Les données personnelles sont stockées, rendues disponibles et les droits d'accès sont contrôlés à l'aide de tels dispositifs. Un moteur de base de données est intégré dans ce dispositif afin de permettre le développement d'applications manipulant ces données personnelles. Afin de faire face aux contraintes matérielles du SPT, de nouvelles techniques de stockage, d'indexation et d'exécution de requêtes basées uniquement sur des structures de données séquentielles ont été proposées. Ces structures doivent être régulièrement stratifiées, i.e., réorganisées en structures plus efficaces, toujours séquentielles, afin de passer à l'échelle.

Dans ce nouveau contexte, cette thèse étudie les techniques de protection nécessaires pour sécuriser cette vision décentralisée. En effet, seul le microcontrôleur du SPT est matériellement sécurisé. La base de données, stockées sur la Flash NAND, liée au contrôleur par un bus, est donc en dehors du périmètre de sécurité et peut être la cible d'attaques sur la confidentialité ou l'intégrité des données. Par ailleurs, le PDS s'appuie également sur des serveurs externes de support pour offrir la durabilité et la disponibilité des données, ou d'autres traitements globaux. Dans cette étude, nous considérons les serveurs de support comme honnêtes-mais-curieux, c'est à dire qu'ils fournissent correctement les services attendus (stocker, retrouver ou supprimer des données ou des messages), mais qu'ils peuvent tenter d'attaquer les données stockées localement. Par conséquent, des protocoles sûrs doivent être conçus.

Cette thèse vise donc à fournir des solutions pour assurer la sécurité des données personnelles en s'appuyant sur des techniques cryptographiques et en limitant les surcoûts générés. Plus précisément, (i) nous proposons un ensemble de briques de bases prenant en compte les contraintes matérielles et les fonctionnalités du moteur PDS, pour assurer efficacement la

confidentialité et l'intégrité des données stockées en mémoire Flash. (ii) Nous proposons des protocoles de communication entre les PDS et les serveurs de support permettant d'assurer leurs fonctions en toute sécurité ; (iii) Enfin, nous validons les techniques proposées par des évaluations avec des données synthétiques et un simulateur de SPT. L'objectif est ici d'évaluer l'impact des techniques de protection proposées sur la performance du moteur de PDS. Les résultats montrent l'efficacité de nos propositions.

Les solutions proposées peuvent, sans doute, être réutilisées dans d'autres contextes plus généraux ou inspirer de nouvelles études combinant sécurité des bases de données et cryptographie. Par exemple, l'algorithme « smart sélection » permet de réaliser des sélections directement sur les données chiffrées à l'aide d'algorithmes de chiffrement traditionnels. Il ouvre donc une autre direction pour exécuter des requêtes directement sur les données cryptées (dans le contexte plus traditionnel d'un SGBD centralisé).

Acknowledgement

First of all, I would like to thank all the jury members, who spent their precious time to review my work, gave me valuable comments on the manuscript, and helpful advices for my future work.

I would like to thank the permanent members of the SMIS team, who gave me the opportunity to do this thesis. It has been a great experience for me.

I thank you all, SMIS members, for your nice help on work and on administrative things: Philippe Pucheral, Luc Bouganim, Nicolas Anciaux, Shaoyi Yin, Alexei Troussov, Lionel Le Folgoc and Elisabeth Baque. Without your help, I would not have reached the step where I am standing now.

I also thank the SMIS directors who gave me many training opportunities to broaden my view. When attending major international conferences like VLDB and SIGMOD, I have learned the professionals' enthusiasm, persistence and excellent works of the international database community. When attending BDA conference and BDA summer school, I have learned more about the French database community, I have improved my French and discovered more about this beautiful country. The nice French way of life and amazing landscape have become unforgettable memories.

I thank all the colleagues who gave me so much happiness in my daily life: lively lunch discussions (P. Pucheral, L. Bouganim, N. Anciaux, B. Nguyen, etc.), exciting hiking at Houches (A. Troussov, Tristan Allard, S. Yin, L. Le Folgoc), nice week-end yoga trips (E. Baque), funny French expressions and vocabularies (L. Bouganim, L. Le Folgoc, etc.), and excellent team parties and dining hall's games. All this made me spend a happy, colorful and substantial PhD student life.

I thank all my Chinese compatriots in Paris. In the shuttle, we have shared happy moments after hard day's work (Yan-Hui Qu, Peipei Shang, Chun Liu, S. Yin, Hao Li, Rong Guan, Xiong Jin etc). At home, we have discussed the daily events and enjoyed relief in life (LinHuizi Li, Guoyan Qu, Hongda Gao, Ying Zhang, Guisong Xia, Haijian Zhang, Lei Yu, etc). During European travels, we have explored the scenic spots and have spent a wonderful time together

(Xin Zhang, Hao Wang, etc). I also thank many anonymous Chinese people who offered me some nice assistance.

Thanks to my parents and to my sister, for their endless love. Thanks to the family of Jacqueline Sasson, who has provided me with a perfect living and working environment during more than two years.

Yanli GUO

2011-08-12

Chapter 1
Introduction

An increasing amount of personal data is automatically gathered and stored on central servers by administrations, hospitals, insurance companies, etc. Citizens have no way to opt-out of these applications because governments, public agencies or companies that regulate our daily life require them. In the meantime, citizen themselves often count on internet companies to store their data (salary forms, invoices, banking statements, etc) making them reliable and highly available through internet.

The benefits of centralizing personal data are unquestionable in terms of data completeness, failure resiliency, high availability and even consistency of security polices. But these benefits must be weighted carefully against the privacy risks of centralization. There are many examples of privacy violations arising from negligence, abusive use, internal attacks, external attacks [CSI11], and even the most secured servers are not spared [DBLoss, Wes06]. Intensified concerns about security and privacy of data have caused the establishment of new legislations and industry standards such as Health Insurance Portability and Accountability Act (HIPAA), Sarbanes-Oxley Act (SOX), Gramm-Leach-Bliley Act (GLBA) and Cardholder Information Security Program (CISP).

We envision a radically different, fully decentralized vision, where personal data is managed by a *Personal Data Server (PDS)*. The PDS approach builds upon the emergence of new hardware devices called *Secure Portable Tokens (SPT)*, as shown in Figure 1. Whatever their form factor (SIM card, secure USB stick, wireless secure dongle), SPTs combine the tamper resistance of smart card microcontrollers with the storage capacity of NAND Flash chips. The secure microcontroller is equipped with a 32-bit RISC processor (clocked at about 50 MHz today), a tiny static RAM (about 64 KB today) and a small internal stable storage (about 1 MB today). The internal stable storage inherits the tamper resistance of the microcontroller and can store sensitive metadata (securely). However, the mass NAND storage is outside the security perimeter provided by the microcontroller and connects the latter via bus. This unprecedented conjunction of portability, secure processing and Gigabytes-sized storage holds the promise of a real breakthrough in the secure management of personal data. Moreover, a SPT capable of acquiring, storing, organizing, querying and sharing personal data under the holder's control would be a step forward in translating the PDS vision into a reality.

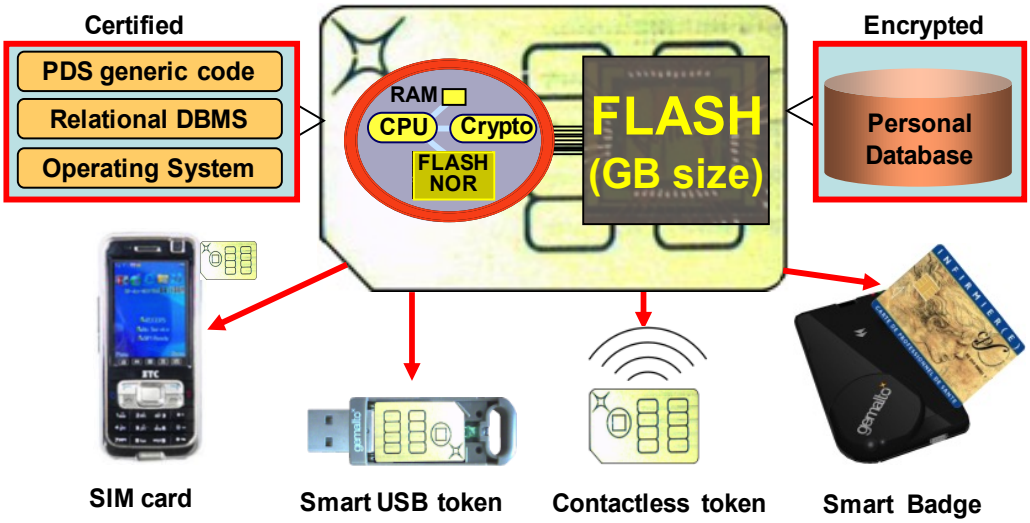


Figure 1. Secure Portable Token

Privacy and security being the ground of that vision, we have to endow the global PDS architecture with appropriate security features. This manuscript precisely focuses on the design of crypto-protection schemes on PDS and secure protocols to enforce the security of externalized data. Although the proposed solutions are suitable in the particular context of the SPT, endowed with a tamper-resistance microcontroller, we consider that some of the proposals can be extended to more general contexts (e.g. traditional DBMS), and provide more opportunities for the coming works combining database security and cryptography.

In this chapter, we first sketch the relevant aspects of the PDS environment, for our security study. Then we give the precise objectives of the thesis. Third, we present the main difficulties that must be faced to provide a secure implementation of those PDS processes. Finally, we give the main contributions of this thesis and the outline of this manuscript.

1 Context of the study: the PDS environment

The objective of a PDS is to provide powerful data management capabilities (e.g. acquiring, organizing, querying and sharing etc) to support versatile applications (e.g. medical health care, epidemiological study, financial help and budget optimization), hence full-fledged embedded DBMS engine (called PDS engine hereafter) are required as well.

Based on the SPT performance parameters provided by its manufacturer (see Chapter 6), IO cost is dominant over cryptography (e.g., AES algorithm is implemented in hardware and is highly efficient compared to the cost for read/write IOs). Therefore, we first design an IO

optimized PDS engine, and then consider adding cryptography on the existing engine without changing its IO patterns. In the following, we introduce the design principles followed by the PDS engine and postpone the discussion on the crypto-protection design to Section 2.

The intrinsic hardware constraints exhibited by the SPT, especially the limited RAM size and particularities of NAND Flash, pose great challenges for the PDS engine design. Indeed, to enable computing SQL queries on GBs of data with tiny RAM, the PDS engine has to rely on a massively indexed model. Such a model generates fine-grain random writes at insertion time (typically, to update all the indexes). Unfortunately, random writes badly suits the NAND Flash features. To break the implication between massively indexed databases and fine-grain random writes, and make the global design consistent, we have proposed the *database serialization* paradigm. As its name suggests, this paradigm consists of organizing the complete database (data, indexes, logs and buffers) sequentially. As a result, random writes and their negative side effects on Flash write cost are simply precluded.

However, a purely sequential database organization scales badly. To cope with this issue, we have proposed the *database stratification* paradigm. The objective is to timely reorganize the database, notably to transform non clustered sequential indexes into more efficient clustered ones, this without abandoning the benefit of database serialization in terms of random writes.

In the PDS vision, other components, called supporting servers, are required, to restore the common database features that a pure SPT based solution could not offer. Typically, a SPT can be easily lost, stolen or destructed, and has an unpredictable connection activity. The PDS approach resorts to the external supporting servers to provide traditional DBMS features like availability, durability and global processing functionalities. However, in the PDS context, we consider that the supporting servers are Honest-but-Curious, and as such correctly implement the service they are supposed to offer, but may try to obtain personal data or infer participant (SPT) identities. Consequently, cryptographic techniques are also needed to ensure the supporting servers function correctly and securely.

2 Objectives: providing security for PDS data and communication

The first objective of the thesis is to protect the personal data stored and managed within the SPT. While the SPT is endowed with a secure, tamper resistant microcontroller, the NAND Flash memory used to store the database is external to the microcontroller and as such does not

inherit from its tamper resistance. Hence, Flash resident data may suffer from confidentiality and integrity attacks. In particular, we consider that a SPT holder may become an attacker (e.g., SPT can be stolen or owners may corrupt their own data, for example to self prescribe drugs). The following attacks may be conducted: (1) spy the Flash memory and deduce unauthorized information (i.e., snooping attack), (2) replay old-version of the data to accomplish undue objectives (e.g., get refunded several times), (3) substitute valid data items with other valid ones (e.g., change the name of a valid drug with another one), (4) modify the data content (e.g., forge new prescriptions). Cryptographic techniques have to be used to detect any tampering of the data.

The second objective is to build secure communication protocols between PDSs and supporting servers. Since PDS have to externalize some personal data to curious supporting servers, thus cryptographic techniques have to be used to ensure the security of externalized data and communications between them, such that (a) supporting servers could serve data store, retrieve and delete request correctly, (b) supporting servers have no knowledge about the content of externalized data nor participant (PDS) identities.

3 Challenges

The main security advantage of the SPT is the tamper resistance of the embedded microcontroller. Indeed, it makes the computing environment highly secure, meaning that attackers have no way to spy or tamper the data resident in the microcontroller's secure memory (RAM or NOR Flash), easing key management which is considered as a tricky task in many contexts. Moreover, internal attacks are precluded as well, given that the PDS engine can be certified and is auto-administered.

Many difficulties arise when trying to extend the security perimeter of the microcontroller, mainly due to the intrinsic hardware constraints of the SPT and to the specific hypotheses of the PDS environment, listed below:

(1) *Few secure non-volatile storage (i.e., NOR) on chip.* This constraint mainly causes challenges for version number storage: in a PDS, to fight against replay attacks, we have to maintain the association between each data item and its version number, stored in secure memory. Due to this constraint, the number of versions must to be maintained must be kept small.

(2) *Database on NAND Flash memory.* NAND Flash constraints, and especially its poor support for random write patterns, strongly impact security management. For instance, considering version storage, to reduce the version storage space, the state-of-the-art methods usually structure versions as a tree and only store the root of tree in the secure memory [Mer89] [ECL+07]. However, these methods generate fine-grain random writes which are very costly in NAND Flash (versus sequential writes) [BJB+09]. Moreover, at the time of retrieving versions to check integrity, the tree has to be traversed from the leaf level up to the root, hence incurring access latency and degrading query processing performance.

(3) *Fine-grain data random access*. This access pattern comes from the fact that the embedded database design relies on a massive indexing scheme. The size of accessed data can thus be very small (e.g., in the order of pointer size, such as 4 bytes), and guaranteeing their integrity efficiently becomes a challenging problem. Indeed, traditional integrity methods based on cryptographic Hash functions operating at the granularity of large chunks, typically 512 bits. In addition, such methods also require storing an additional fixed size tag (e.g. 16 bytes hash value) along with the data item causing a storage penalty.

(4) *Honest-but-Curious supporting servers*. Unlike in the traditional centralized approaches, we consider that supporting servers can threaten the confidentiality of the data and the anonymity of the participant PDSs. Therefore, we have to guarantee that: (a) the communications between PDSs and supporting servers are done in anonymous way, and (b) the supporting servers answer store, retrieve and delete requests without being able to learn any personal information nor the identity of any participant PDS.

(5) *Limited set of cryptographic primitives*. SPTs (and more generally secure microcontrollers) typically include hardware implementation of an encryption algorithm and software implementations of cryptographic hash functions. For example, only AES algorithm is available in current platforms, due to the fact that it performs well on variety of settings, including 8-bit smart card [NES03]. Hash functions, implemented in software are extremely slow. Even hash functions, implemented in hardware, may still perform worse than AES-based constructions, according the numbers reported in [DHV03, NES03], because of its radical different implementation way. We thus consider that these cryptographic limitations will remain valid in the foreseeable future. Consequently, the design scope of crypto-protection schemes is limited and some state-of-the-art cryptographic techniques are precluded (e.g., exotic encryption algorithms, privacy homomorphism).

(6) *Limited computing power on SPT*. This intrinsic resource limitation challenges the design and enforcement of security solutions. The proposed solutions should take advantage of the IO access patterns exhibited by the PDS engine to reduce the incurred cryptography overhead as much as possible, such that they can cope with the limited computing power of SPTs.

4 Contributions

The objective of this thesis is to solve security problems lying in the PDS approach by resorting to cryptography, and cope with the challenges mentioned above. In summary, the contributions made in this thesis are the following:

(1) We propose a set of lightweight crypto-protection building blocks, considering limited resources and hardware constraints. These building blocks can be combined and used together. They can achieve expected security levels, such that any attack considered in our threat model can be detected. In addition, we expect these building blocks to be reusable in other (more general) contexts and provide more opportunities for the coming works combining database security and cryptography.

(2) We propose the preliminary designs for main communication protocols between PDSs and supporting servers, to ensure the latter could serve data store, retrieve and delete request correctly and securely (i.e., without leaking any information about the exchanged data nor participant identities), and restore availability, durability and global processing functionalities which are expected from PDS approach. More precisely, we give the protocols for exchanging messages and secure deletion, which are the basis to implement expected functionalities. In this manuscript, we do not go into the implementation levels on this aspect, and only provide sound solutions to validate the feasibility of PDS approach in global view. As a result, further work is required in this direction.

(3) We enforce designed crypto-protection building blocks on existing PDS engine. Considering the features of engine data, we choose adequate crypto-protection building blocks for each of them, avoiding incurring large cryptography overhead and to change existing PDS engine designs. To avoid the redundancy, in this thesis, we only focus on some representative data structures to illustrate such enforcement.

(4) We conduct broad experiments both with synthetic data and SPT simulator, compare our security solutions with other traditional methods, and show their effectiveness and analyze their impact to the performance of existing PDS engine. According to the experiments, our

proposals have significant advantages considering crypto overhead, which gives promising prospects for extending our solutions to more general contexts. Moreover, the performance numbers obtained from SPT simulator indicate that the incurred cryptography overhead to the system is acceptable, and does not violate PDS user patience expectations.

5 Outline

This study is at the crossroad of two domains: cryptography and databases. Chapter 2 gives the necessary background knowledge about cryptography, to make the contributions done in this manuscript understandable. Chapter 3 introduces the basic technologies and related works done for the database encryption and integrity, which is partially based on our published book chapter [BoG09]. Chapter 4 focuses on the introduction on the PDS approach and on the design of the embedded PDS engine. It presents the serialization and stratification paradigms, the indexing and query processing techniques, and highlights the specificity of the PDS context in terms of security. This chapter is based on the published paper at VLDB'10 [AAB+10] and the working paper [ABG+11]. In Chapter 5, we address the security problems lying in the PDS approach. We give our design of crypto-protection building blocks and exemplify their enforcement on the data structures of the embedded PDS engine. In Chapter 6, through a set of experiments, we compare our solutions with traditional methods and show their significant advantages. Moreover, we measure the system performance of the whole PDS engine and analyze the impact of cryptography globally. This chapter is partially based on the working paper [ABG+11]. Finally, Chapter 7 concludes the manuscript and opens some new research perspectives.

Chapter 2
Cryptography Background

In this chapter, we provide the required background knowledge to help making our proposal understandable. In the PDS approach, both symmetric and asymmetric cryptography are used. Their characteristics and main techniques are addressed in the first two sections respectively. In Section 3, we address data integrity techniques, used to detect malicious tampering on data and guarantee its correctness. Note that those techniques share similarities with symmetric cryptography. In the same section, we also introduce a specific encryption scheme called *authenticated encryption*, which provides confidentiality and integrity guarantees simultaneously.

1 Symmetric Key Cryptography

1.1 Introduction

Symmetric key cryptosystems, also known as single key, shared key or secret key cryptosystems, use identical (or trivially related, i.e., related in an easily computable way) cryptographic keys for both encryption and decryption. Figure 2 shows the model for such cryptosystems. The secret keys K_1 and K_2 can be hold by two distinct parties (e.g., Bob and Alice), or hold by a unique party. In our context, symmetric key cryptography is mainly used for protecting embedded data stored on PDS. The keys are confined in the SPT, and are never exposed to the external world. Hence, even the SPT's owner may not have access to encryption keys.

Symmetric key ciphers include block ciphers, stream ciphers, cryptographic hash functions and *Message Authentication Codes (MAC)*. In this section, we focus on block ciphers and stream ciphers, and postpone the discussion on the last two (used for integrity purpose) to Section 3.

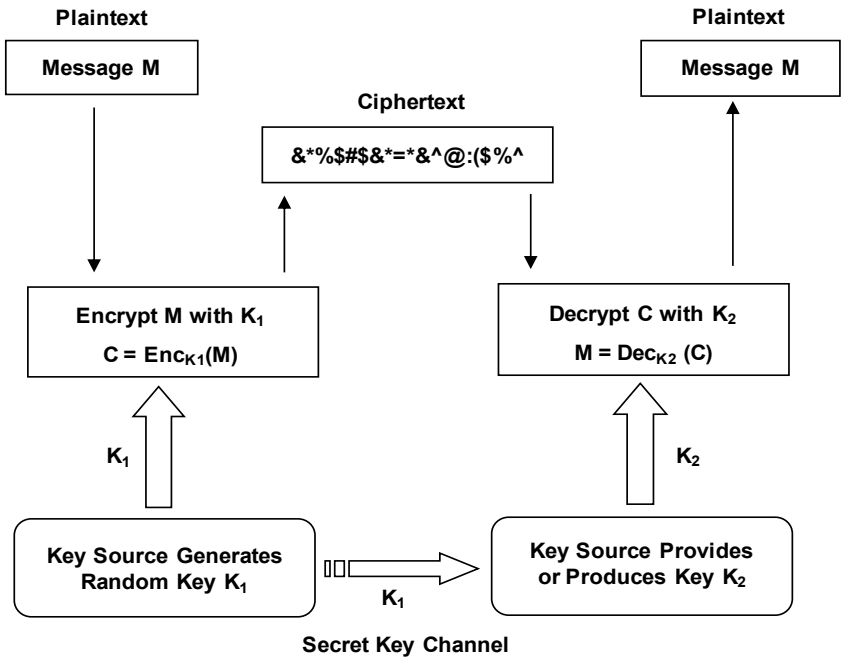


Figure 2. Secret Key Cryptography

1.2 Stream Cipher vs. Block Cipher

In a stream cipher, an arbitrary length of key stream is created to match the size of a plaintext for encryption. It is combined with the plaintext (e.g., usually use XOR) on byte or bit granularity, as shown in Figure 3. The decryption process (not shown in the figure) is the opposite one.

A block cipher operates on fixed-length groups of digits, called blocks. When targeting longer size of data, the data will be divided into individual blocks for encryption. If the last block is smaller than the defined block length (e.g., 64 bits or 128 bits, depending on the algorithm), the block needs to be padded before encryption to make its length up to a multiple of the block size. In addition, the encryption function must be one-to-one (i.e., invertible) to allow unique decryption. In block ciphers, such unvarying transformation depends on the cryptographic key and encryption algorithm. A block cipher acts as a fundamental building block, which can be used to construct more versatile cryptographic primitives, such as pseudorandom number generators, MACs and hash functions [MOV97]. Moreover, using specific modes of operation (i.e., the CTR mode), a block cipher can be used to implement a stream cipher (see Section 1.3).

Stream ciphers are usually faster in hardware and have less hardware complexity than block ciphers. Moreover, they do not have error propagation (or very limited) since each character or digit is processed individually. This gives advantages when processing corrupted data (e.g., transmission errors) since corrupted parts does not influence uncorrupted ones [MOV97]. In addition, stream ciphers do not require padding data to make its length satisfy certain conditions (as a consequence they do not cause storage expansion after encryption).

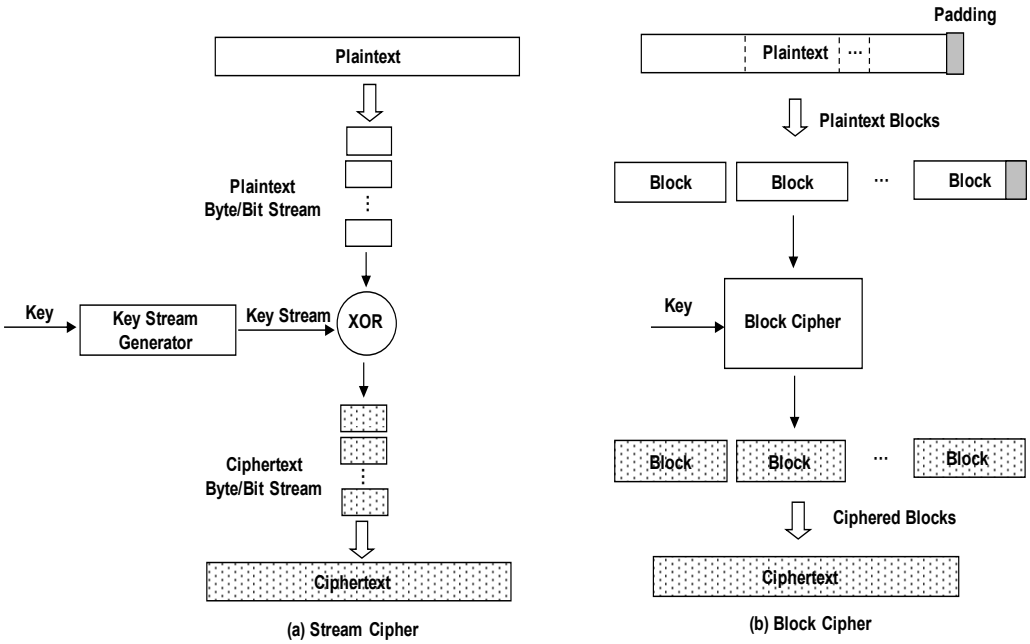


Figure 3. Stream Cipher and Block Cipher

For stream ciphers, the key stream is generated independently from the plaintext, which is only used for the final exclusive-or operation. Hence, the stream cipher operations can be performed in advance, allowing the final step to be performed in parallel once the plaintext becomes available (similar for decryption) [MOV97].

In contrast to stream ciphers, block ciphers operate at block granularity, and data often need to be padded before encryption. Moreover, the encryption of one block may be affected by the encryption of other blocks depending on the mode of operation which is chosen. This enables avoiding some security weakness due to repetitive patterns (see Section 1.3). Thus any bits change in a certain plaintext block may affect the subsequent ciphertext after encryption (e.g., CBC mode).

Numerous block ciphers have been proposed and published, and some have been standardized like DES [DES], Triple-DES [TDES] and AES [NIS01] algorithms. In our context, only AES algorithm is available on SPT, which constrains the proposal made in this thesis. Therefore, we only introduce the AES algorithm in this manuscript. We refer the reader to the literature [MOV97] for other algorithms.

In 1997, the *National Institute of Standards and Technology* (NIST) abandoned their support for DES and launched a competition to choose its successor known as *Advanced Encryption Standard* (AES). In 2001, Rijndael algorithm was selected as AES, among other four AES finalists including RC6, Serpent, MARS and Twofish. AES standard includes three block ciphers. Each cipher has a 128-bit block size, with key sizes of 128 bits, 192 bits and 256 bits respectively. AES has been extensively analyzed and widely used worldwide, and is considered computationally secure. Moreover, AES performs well on a wide variety of settings, from 8-bit smart cards to high performance computers. This important fact leads the SPT manufacturers (e.g., Gemalto) to choose AES to be implemented within secure microcontrollers.

1.3 Modes of Operation for Block Ciphers

As mentioned before, for block ciphers, if the message is longer than a single block, it is divided into a set of blocks which are then encrypted one by one. The way to chain successive blocks together (called mode of operation) has an impact on the encryption security. For instance, encrypting identical plaintext blocks with a same key may lead to identical ciphertext blocks. Thus the mode of operation needs to be carefully chosen when using a block cipher in a cryptosystem.

The classical modes of operation can be classified into two categories: block modes and stream modes. Block modes process the message by blocks, such as *Electronic Codebook Mode* (ECB) and *Cipher Block Chaining Mode* (CBC), while stream modes process the message by bit or byte stream like *Cipher FeedBack Mode* (CFB), *Counter Mode* (CTR) and *Output FeedBack Mode* (OFB).

In this section, we introduce some modes of operation which are available on SPTs (including ECB, CBC and CTR). We analyze their characteristics and potential weaknesses. Note that we only address the encryption process in the following since decryption can be easily deduced.

1.3.1 ECB mode

Electronic Codebook (called ECB) is the simplest mode of operation. The message is divided into blocks and each block is processed separately, such that each block is independent of all others (see Figure 4).

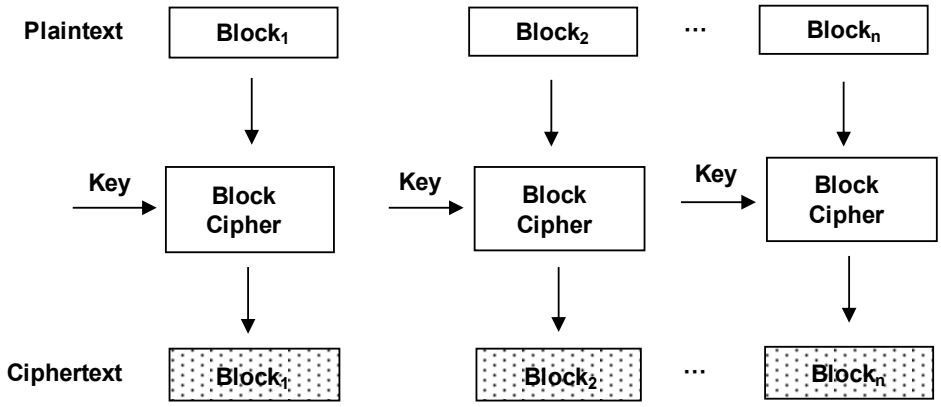


Figure 4. Electronic Codebook Mode (ECB)

The main drawback of ECB mode is that repetitions in the plaintext can be reflected in the ciphertext, which would open a door to statistical attacks. This problem becomes even more serious for specific kind of data like particular images [Sch] or messages that change very little, since code-book analysis becomes feasible. Consequently, ECB mode is useful in the cases where each block is unique (without repetition).

1.3.2 CBC mode

In the CBC mode, cipher blocks are chained with plaintext during encryption (see Figure 5). Consequently, each ciphertext block is dependent on all the blocks encrypted before it. Thus, any change in the plaintext blocks affects the corresponding ciphertext blocks but also the subsequent ciphertext blocks.

In addition, an *Initialization Vector* (IV) is required to encrypt the first block. The IV can be public without hurting the security of the encryption, but it should be unique if several messages are encrypted using the same key (to avoid generating identical ciphertext blocks when encrypting identical plaintext blocks).

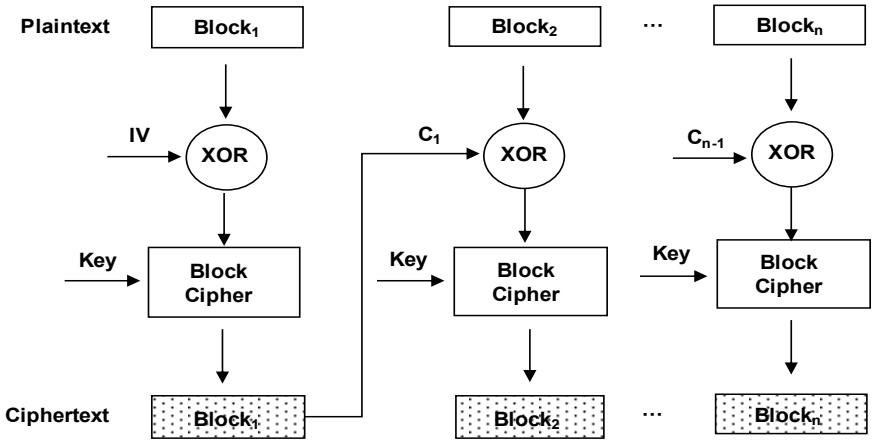


Figure 5. Cipher Block Chaining Mode (CBC)

1.3.3 CTR mode

With stream modes like CTR, block ciphers can be used as stream ciphers. As shown in Figure 6, plaintext blocks are only used at final step to perform an exclusive-or with the ciphered counter blocks. Thus, the counter blocks can be encrypted in advance or in parallel.

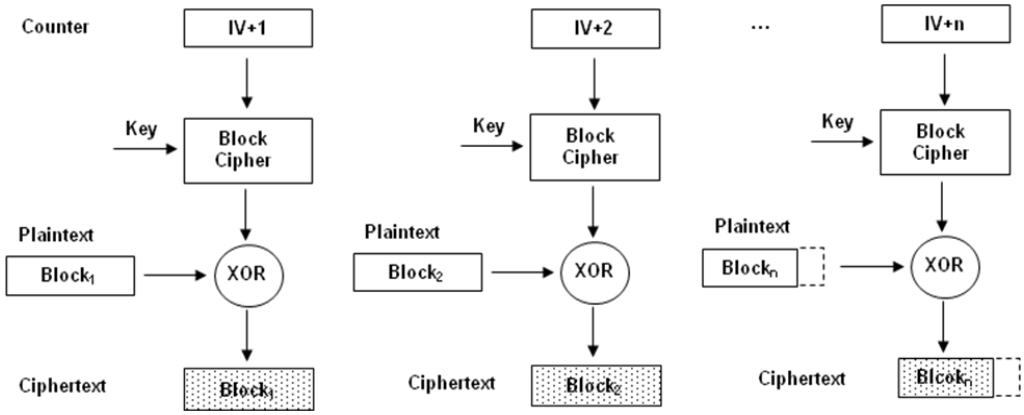


Figure 6. Counter Mode (CTR)

In CTR mode, each counter block must be unique, to avoid exhibiting security weaknesses which could be exploited by attackers. For example, if we consider identical counters, two plaintext blocks M and M' would be encrypted into two different ciphertext blocks C and C', but the whole would verify C ⊕ C' = M ⊕ M'. This could be used by attackers to deduce information about plaintext.

2 Public Key Cryptography

2.1 Introduction

A significant disadvantage of symmetric key cryptography lies in the difficulty of managing the keys. Typically, each communicating party exchanging data must share a common key. Therefore, each party needs to maintain a huge number of keys to communicate with others. This number of keys increases with the number of participants. In addition, since two parties have the same knowledge about the key, there is no way to prevent one party (e.g., receiver) to forge a message and claim that the message was sent by another party (e.g., sender). In order to solve such issues, in 1976, Whitfield Diffie and Martin Hellman proposed the notion of public key (called asymmetric key as well) cryptography [DiH76], in which each entity has two different but mathematically related keys: a public key and a private key. However, it is computationally infeasible to compute the private key given the public one. The public key defines an encryption transformation, while the private key defines the associated decryption transformation [MOV97].

Figure 7 shows the model of public key cryptography. Alice intends to send a message to Bob. She obtains an authentic copy of Bob's public key which is publicly known. She uses this key to encrypt the message and sends it to Bob. Once Bob receives the encrypted message from Alice, he can obtain the original message using his private key.

Public key algorithms can be divided into three important classes: Public Key Distribution Schemes (PKDS), Public Key Encryption Schemes (PKS) and Digital Signatures. In PKDS, public key algorithms are used to exchange securely a key. This (secret) key can then be used (as a session key) for starting a communication using the secret-key scheme (e.g., the Diffie–Hellman key exchange protocol, see next section). The above example (with Alice and Bob) explains the PKS scheme, where public key algorithms are used to encrypt messages. Regarding digital signatures, public key algorithms are used to create a digital signature. More precisely, the private key creates signatures while the public key verifies signatures.

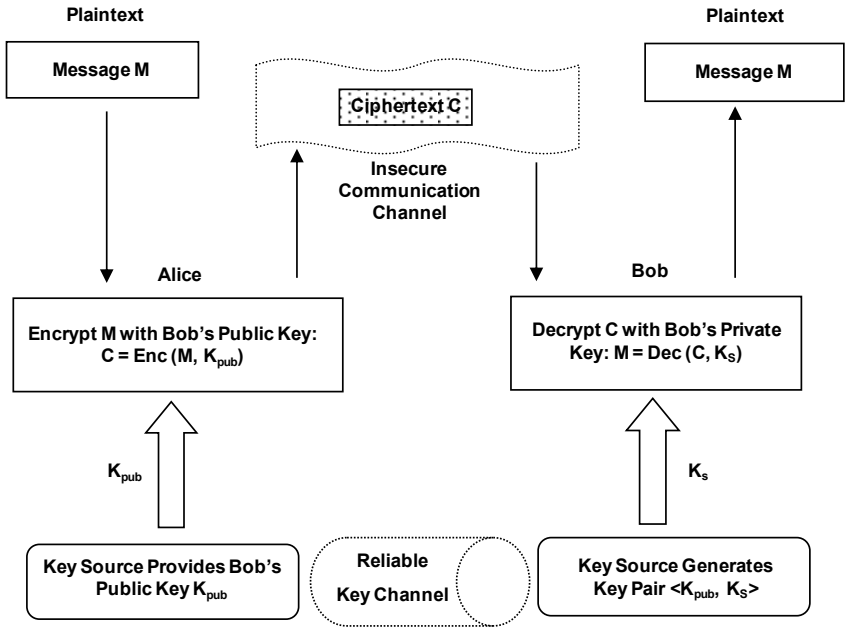


Figure 7. Model of Public Key Cryptography

Public key cryptography eases the key management problem, and provides versatile applications as mentioned above. However, since public key algorithms are usually based on the computational complexity of "hard" problems which are often from number theory (e.g., RSA [RSA78] and elliptic curve cryptography [ECC]), they involve computationally expensive techniques such as modular multiplication and exponentiation. As a result, public key algorithms are slower than symmetric key algorithms. Thus, they are commonly used with hybrid cryptosystems, in which a symmetric key is generated by one party and used for encrypting the message, while a public-key algorithm is used to encrypt the symmetric key which is transmitted to the receiver along with the message. As a result, such cryptosystems inherit the advantages from both schemes.

In the following, we will introduce two techniques based on public key cryptography, including Diffie–Hellman key exchange protocol and onion routing protocol, which are very relevant to our problems.

2.2 Diffie-Hellman Key Exchange

As a typical key distribution scheme, Diffie-Hellman key exchange protocol [DiH76] allows two parties that have no prior knowledge of each other to establish a shared secret key over an

un-trusted network, and the shared key could be used to ensure the security for subsequent communications.

Figure 8 illustrates this protocol. Let assume two parties, Alice and Bob, who intend to exchange a secret key K_{AB} over an insecure channel. This requires two system-wide constants named ρ and α, ρ being a very large prime number (e.g., about 200 digits), and α being a small integer. Note that these two numbers are only used after obtaining agreements from both two parties.

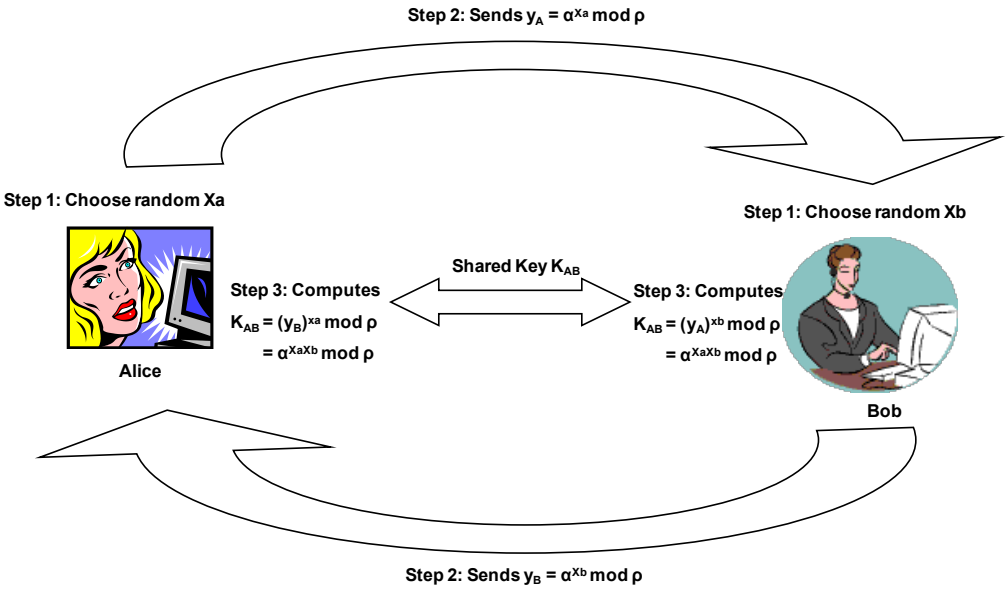


Figure 8. Diffie-Hellman Key Exchange Protocol

The protocol works in three main steps. (1) Alice and Bob choose a secret random number X_a and X_b respectively. (2) Alice sends Bob $y_A = \alpha^{Xa} \bmod \rho$, while Bob sends Alice $y_B = \alpha^{Xb} \bmod \rho$. Note that y_A and y_B are not encrypted, thus they can be observed by eavesdroppers. (3) Once Bob receives y_A, he computes $K_{AB} = (y_A)^{Xb} \bmod \rho$, which equals to $\alpha^{XaXb} \bmod \rho$. On the other side, once Alice receives y_B, she computes $K_{AB} = (y_B)^{Xa} \bmod \rho = \alpha^{XaXb} \bmod \rho$. As a result, a shared key K_{AB} is established between Alice and Bob, while attackers can not deduce it since they have no knowledge about X_a and X_b, and because the discrete $\log_\rho b$ is hard.

2.3 Onion Routing

The idea of Onion Routing is to preserve the anonymity for senders and recipients of an encrypted message while the message traverses over an un-trusted network. Onion Routing

follows the principle of Chaum's mix cascades [Cha81]. Indeed, the messages are sent via a sequence of Onion Routers (OR) from sender to receiver, and each intermediary router re-routes the message in an unpredictable path.

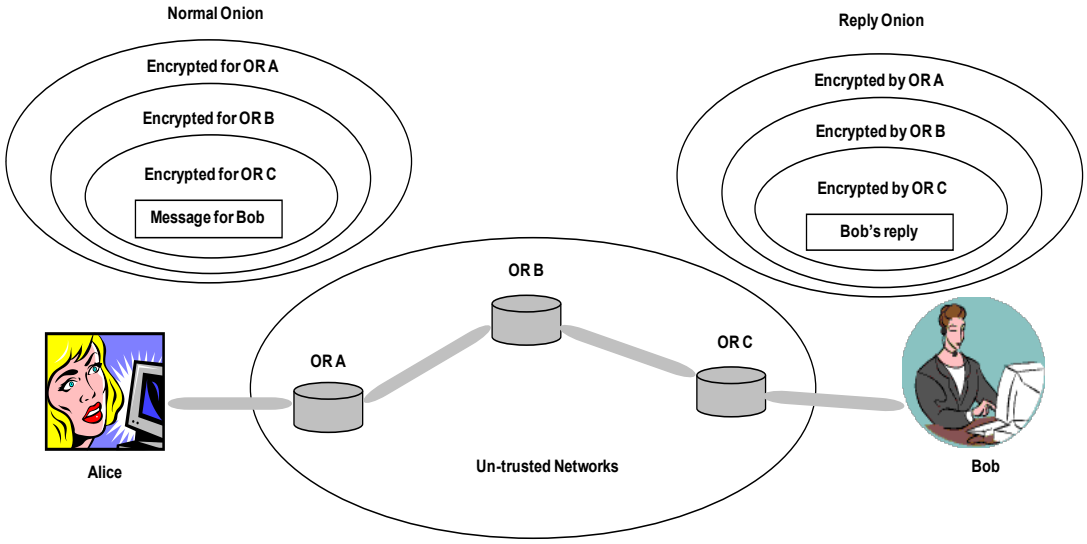


Figure 9. Onion Routing Protocol

Onion Routing is implemented based on public key cryptography, as shown in Figure 9. Suppose the sender Alice intends to send message to Bob. The message is encrypted repeatedly by the sender, to obtain what is called an onion (see "Normal Onion" in Figure 9), and transmitted via three ORs (e.g., OR A, OR B and OR C). Note that before sending the message, ORs have been selected and ordered by the sender Alice to form the communication chain or circuit. Consequently, Alice has encrypted the message using the public key of each chosen OR to build necessary layers of encryption. These layers have been successively constructed, in reverse order of ORs in the sending path. Once an OR receives the message, it peels away a layer of encryption by using its private key and obtains the routing information to forward the message to the next OR. The last OR in the chain (e.g., OR C) peels off the last layer of encryption and delivers the original message to Bob. For Bob's response, a reply onion is generated in a similar way.

The advantage of onion routing is that it is not necessary to trust each intermediary OR. If a given OR in the communication chain is compromised, the communication can be completed and remains anonymous since each OR in the chain is only aware of its neighboring ORs (i.e., the preceding and next ones) during transmission. However, it cannot provide ultimate

anonymity since attackers can still observe which individuals are actually sending and receiving messages. In practice, the anonymity network Tor [DMS04] is the predominant technology that employs onion routing. It provides a flexible, low overhead, easily accessible and traffic analysis resistant infrastructure [GRS99].

3 Authentication and Integrity

3.1 Introduction

Authentication and integrity techniques are used to detect tampering of (encrypted) messages or data. Techniques based on public key algorithms (e.g., digital signature) detect tampering with extra benefits (e.g., non repudiation). However, public key technology performs slowly, since it involves computationally expensive techniques as mentioned before, thus incurring expensive computation overheads. As a result, techniques based on symmetric key algorithms are more commonly used for data authentication and integrity. Typically, a message digest is computed from the original message with the secret key (the sender and receiver use the same key) and attached to the message. At reception time, the recipient computes the digest of the message (using the secret key). He compares it with the attached digest and deduces if the message has been tampered or not. When the sender and receiver are the same entity (e.g., a user protecting data at rest), the process is similar.

In this section, we introduce basic techniques used to authenticate data, like cryptographic hash function (e.g., MD5 and SHA-1), message authentication code, and a technique called *Authenticated Encryption* (AE) that provides confidentiality (i.e., encryption) and integrity (i.e., authentication) guarantees at same time.

3.2 Cryptographic Hash Functions

A cryptographic hash function is a deterministic procedure that takes in input a variable size data and returns a fixed size bit string called a cryptographic hash value (or digest). The hash function is one-way, meaning that given a cryptographic hash function *H* and a cryptographic hash value *h*, it is computationally infeasible to find a message M such that H(M) = h. This property is called *pre-image resistance*. In order to resist all known types of cryptanalytic attacks, it also has additional properties including:

- *Second pre-image resistance* (also called weak collision resistance). It is computationally infeasible to find any second input (e.g., message M') which has the same hash value as a given input (e.g., message M), i.e. H(M) = H(M'). This property is sometimes referred as weak collision resistance.

- *Collision resistance* (also called strong collision resistance). It is computationally infeasible to find a couple of distinct inputs (e.g. messages M and M') which lead to the same hash value, i.e. H(M) = H(M'). Such a couple is called a cryptographic hash collision.

The two most commonly used cryptographic hash functions include *Message Digest 5* (MD5) and *Secure Hash Algorithm 1* (SHA-1). Both hash functions process the data by blocks of 512 bits, the size of the hash function output being 160 bits for SHA-1 and 128 bits for MD5. Regarding their security, MD5 has been broken, successful attack on SHA-1 has been reported as well in 2005 [WYY05, Sch05]. Consequently, more advanced members in SHA family (e.g. SHA-2) are required to provide better security level. To ensure the long term robustness of applications using hash functions, there is a competition for SHA-3 as the replacement of SHA-2, and it will become FIPS standard in 2012 [SHA3].

3.3 Message Authentication Code

Cryptographic (un-keyed) hash functions used for authentication are called Modification Detection Codes (MDC). To achieve the authentication purpose, we have to encrypt the generated hash values. Otherwise, we face two problems: (1) identical messages would generate identical hash values; and (2) attackers could tamper the original message and generate a valid hash value for the forged message (cryptographic hash functions are publicly known).

Keyed hash functions used for authentication purpose are called Message Authentication Codes (MAC). They take a secret key and an arbitrary-length message as input, and produce a fixed-size output (called authentication tag or MAC value). The same output cannot be produced without knowledge of the key. MAC algorithms can be based on block ciphers like OMAC [IwK03], CBC-MAC [BKR94] and PMAC [BlR02], or based on cryptographic hash functions like HMAC [HMAC]. In current SPT platform, hash functions are implemented in software and extremely slow, and only AES algorithms are available, thus we have to resort to block cipher (e.g., AES algorithm) based MAC algorithms. Moreover, only ECB, CBC and CTR modes of operation are supported. As a result, we choose CBC-MAC algorithm as the authentication method. In CBC-MAC, the message is encrypted with some block cipher (e.g.,

AES) using the CBC mode. Such encryption produces a chain of blocks such that each block depends on the encryption of the previous blocks, and the last encrypted block is used as authentication tag. The dependency property of the CBC mode ensures that any change in the message will cause change in the tag. In our implementation, we use a variation called XCBC-MAC to resist to message concatenation attacks [FrH03, Anc04].

The method used to generate the MAC is an interesting topic. In practice, the data need to be encrypted and authenticated. There exist three ways to build the MAC of a data: (1) *Encrypt-and-MAC*: the plaintext is encrypted; the MAC is also computed using the plaintext (2) *MAC-then-Encrypt*: MAC is first built using the plaintext; then the MAC and the plaintext are encrypted together to produce the ciphertext. (3) *Encrypt-then-MAC*: the plaintext is first encrypted; then the MAC is built on the ciphertext. The first approach (Encrypt-and-MAC) exhibits several security weaknesses [BeN00]. The second one (MAC-then-Encrypt) requires decrypting the message before checking the MAC value which incurs extra cryptographic costs. The third approach (Encrypt-then-MAC) seems thus the most interesting one for our works. It is considered secure in our context and the integrity checking can be done in a straightforward way [Anc04].

3.4 Authenticated Encryption Algorithms

In order to provide confidentiality and integrity guarantees for protected data, a simple and straightforward way is to glue encryption and MAC, as stated above. However, this would require two independent keys, one for the encryption and one for building the MAC. Moreover, it would require two passes of processing on the data, which would increase the cost (the sum of the encryption cost and the MAC cost).

Authenticated Encryption (AE) algorithms provide confidentiality and integrity simultaneously, based on a secret-key. Some AE algorithms perform a single-pass on the message, but are protected by patents [Anc04]. AE algorithms requiring two passes of processing (one for encryption, another one authentication) have been developed [Dwo04, Dwo07]. Their advantage over a basic composition of encryption and MAC is that a single cryptographic key is sufficient for the entire scheme, instead of two independent keys (this enables saving key initialization costs). We refer the readers to [BeN00, RBB+01] for more details.

Chapter 3
Database Encryption and State-of-the-Art

Database security encompasses three main properties: confidentiality, integrity and availability. Roughly speaking, the confidentiality property enforces predefined restrictions while accessing the protected data, thus preventing disclosure to unauthorized persons. The integrity property guarantees that the data cannot be corrupted in an invisible way (not refer to complying with defined integrity constraints). Finally, the availability property ensures timely and reliable access to the database.

To preserve data confidentiality, enforcing access control policies defined on the database management system (DBMS) is a prevailing method. An access control policy mainly includes a set of authorizations which takes different forms depending on the underlying data model (e.g., relational, XML). Several models can be followed to manage the authorizations, like the well known Discretionary Access Control (DAC), Role-Based Access Control (RBAC) and Mandatory Access Control (MAC) [Les08]. Whatever the access control model, the authorizations enforced by the database server can be bypassed in a number of ways. For example, an intruder may infiltrate the information system and mine the database footprint on disk. Another source of threats comes from the fact that many databases today are outsourced to Database Service Providers (DSP). In that case, data owners have no other choice than trusting DSPs arguing that their systems are fully secured and their employees are beyond any suspicion, an assumption frequently denied by facts [HIL+02a]. Finally, a database administrator (DBA) has enough privileges to tamper the access control definition and to spy on the DBMS behavior.

With the spirit of an old and important principle called defense in depth (i.e., layering defenses such that attackers must get through layer after layer of defense), the resort to cryptographic techniques to complement and reinforce the access control has received much attention from the database community [HIL+02, AKS+02, IBM07, Hsu08, Ora09] during the last decade. The purpose of database encryption is to ensure the database opacity by keeping the information hidden to any unauthorized persons (e.g., intruders). Even if attackers get through the firewall and bypass access control policies, they still need the encryption keys to decrypt data.

Encryption can provide strong security for data at rest, but developing a database encryption strategy must take many factors into consideration. For example, where should the encryption be performed, in the storage layer, in the database, or in the application layer where the data has been produced? How much data should be encrypted to provide adequate security? What data could be kept in clear without threatening secrecy? What should be the encryption algorithm and mode of operation to satisfy security as well as performance requirements? Who should have access to the encryption keys? How to minimize the impact of database encryption on performance?

In this chapter, we do not discuss all potential solutions to answer the questions above. However, we provide some knowledge on encryption enforcement in traditional databases to help introducing the state-of-the-art techniques and the new contributions made in this manuscript. Section 1 provides preliminary knowledge. In this section, we first describe possible options for encryption enforcement, encryption algorithms and modes of operation. Next, we address the key management issue, which plays a critical role in securing database. Then, we list encryption support in well-known DBMSs and relevant products. Finally, we describe database encryption strategies based on *Hardware Security Modules* (HSM). In a second section, we give an overview of the previous works conducted in the database community about database encryption and integrity techniques.

1 Database Encryption

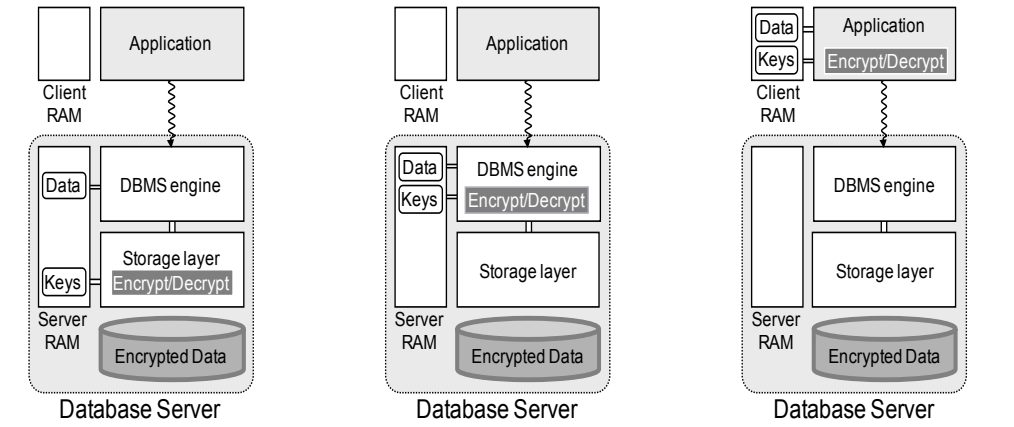
1.1 Encryption Levels

Storage-level encryption amounts to encrypt data in the storage subsystem and thus protects the data at rest (e.g., from storage media theft). It is well suited for encrypting files or entire directories in an operating system context. From a database perspective, storage-level encryption has the advantage to be transparent, thus avoiding any change to existing applications. On the other side, since the storage subsystem has no knowledge of database objects and structure, the encryption strategy cannot be related with user privileges (e.g., using distinct encryption keys for distinct users) nor to data sensitivity. Thus, selective encryption (i.e., encrypting only portions of the database in order to decrease the encryption overhead) is limited to the file granularity. Moreover, selectively encrypting files containing sensitive data is risky since it must be ensured that no replica of this sensitive data remains somewhere in the clear (e.g., in log files, temporary files, etc).

Database-level encryption allows securing the data as it is inserted to or retrieved from the database. The encryption strategy can thus be part of the database design and can be related with data sensitivity and/or user privileges. Selective encryption is possible and can be done at various granularities, such as tables, columns and rows. It can even be related with some logical conditions (e.g., encrypt salaries greater than 10K€/month). Depending on the level of integration of the encryption feature and the DBMS, the encryption process may incur some changes to applications. Moreover, it may cause DBMS performance degradation since encryption generally forbids the use of indexes on encrypted data. Indeed, unless using specific encryption algorithms or mode of operation (e.g., order preserving encryption, ECB mode of operation preserving equality), indexing encrypted data is useless.

For both strategies, data is decrypted on the database server at run time. Thus, the encryption keys must be transmitted or kept with the encrypted data on the server side, thereby providing a limited protection against the server administrator or any intruder usurping the administrator identity. Indeed, attackers could spy the memory and discover encryption keys or plaintext data.

Application-level encryption moves the encryption/decryption process to the applications that generate the data. Encryption being performed before introducing the data into the system, the data is then sent encrypted, stored and retrieved encrypted, and finally decrypted within the application [HIL+02a, DDJ+03, BoP02]. This approach has the benefit to separate encryption keys from the encrypted data stored in the database since the keys never have to leave the application side. However, applications need to be modified to adopt this solution. In addition, depending on the encryption granularity, the application may have to retrieve a larger set of data than the one granted to the actual user, thus opening a security breach. Indeed, the user (or any attacker gaining access to the machine where the application runs) may hack the application to access unauthorized data. Moreover, transmitting a superset of the result incurs an additional communication workload. Finally, such a strategy induces performance overheads (index on encrypted data is useless) and forbids the use of some advanced database functionalities on the encrypted data, like stored procedures (i.e., code stored in the DBMS which can be shared and invoked by several applications) and triggers (i.e., code fired when some data in the database are modified). In terms of granularity and key management, application-level encryption offers the highest flexibility since the encryption granularity and the encryption keys can be chosen depending on application logic. The three strategies described above are pictured in Figure 10.



a. Storage-level encryption **b. Database-level encryption** **c. Application-level encryption**

Figure 10. Three options for database encryption level

1.2 Encryption Algorithms and Modes of Operation

Independently of the encryption strategy, the security of the encrypted data depends on the encryption algorithm, the encryption key length, and its protection. In this section, we mainly focus on the option of encryption algorithms and modes to achieve security goals. Since the key length depends on the algorithm used and is usually recommended by authorities (e.g. NIST), we skip this discussion. In the next section, we address specifically key management issues.

Even having adopted strong algorithms, such as advanced encryption standard (AES), the cipher text could still disclose plaintext information if an inappropriate mode is chosen. For example, if encryption algorithm is implemented in *Electronic Codebook Mode* (ECB), identical plaintext blocks are encrypted into identical cipher text blocks, thus disclosing repetitive patterns (see Section 1.3 in Chapter 2). In the database context, repetitive patterns are common (as many records could have same attribute value), so much care should be taken when choosing the encryption mode. Moreover, simple solutions that may work in other contexts, like using counter mode with an initialization vector based on the data address, may fail in the database one because data can be updated. Indeed, with the preceding example, performing an exclusive OR between old and new versions of the encrypted data discloses the exclusive OR between old and new versions of plaintext data.

Consequently, the choice of an adequate encryption algorithm and mode of operation must be guided by (or at least should take into account) the specificities of the database context: repetitive patterns, updates, and huge volume of encrypted data. In addition, the protection should be strong enough since the data may be valid for very long time periods (several years).

Moreover, we also need to consider performance degradation caused by cryptographic workload and security requirements. As in some contexts (e.g. outsourced database), exotic encryption algorithms (see section 2.2) which exhibit certain security weakness (at point of view of cryptanalysis) could be used to trade off better performance, while satisfying the security requirements expected in that context.

1.3 Key Management

Key management refers to the way cryptographic keys are generated and managed throughout their life. Because cryptography is based on keys that encrypt and decrypt data, the database protection solution is only as good as the protection of the keys. The location of encryption keys and their access restrictions are thus particularly important. For illustration purposes, we assume, in the following, database-level encryption.

For database-level encryption, an easy solution is to store the keys in a database table or file with restricted access, potentially encrypted by a master key (itself stored somewhere on the database server). But any administrator with sufficient privileges could also access these keys and decrypt the data without ever being detected.

To overcome this problem, specialized tamper-resistant cryptographic chipsets, called hardware security module (HSM), can be used to provide secure storage for encryption keys [Hsu08, Ora09]. Generally, the encryption keys are stored on the server encrypted by a master key which is stored in the HSM. At encryption/decryption time, encrypted keys are dynamically decrypted by the HSM (using the master key) and removed from the server memory as soon as the cryptographic operations are performed, as shown in Figure 11a.

An alternative solution is to move security-related tasks to distinct software running on a (physically) distinct server, called security server, as shown in Figure 11b. The security server then manages users, roles, privileges, encryption policies and encryption keys (potentially relying on an HSM). Within the DBMS, a security module communicates with the security server in order to authenticate users, check privileges, and encrypt or decrypt data. Encryption keys can then be linked to user or to user's privileges. A clear distinction is also made between the role of the DBA, administering the database resources, and the role of the SA (Security Administrator), administering security parameters. The gain in confidence comes from the fact that an attack requires a conspiracy between DBA and SA.

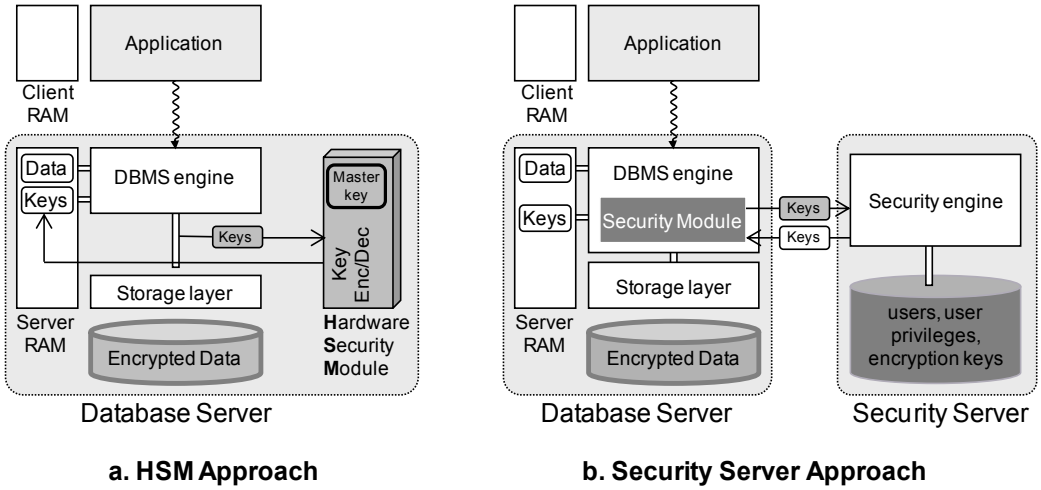


Figure 11. Key management approaches

While adding a security server and/or HSM minimizes the exposure of the encryption keys, it does not fully protect the database. Indeed, encryption keys as well as decrypted data still appear (briefly) in the database server memory and can be the target of attackers.

1.4 Crypto-Protection in DBMS products

Since several years, most DBMS manufacturers provide native encryption capabilities that enable application developers to include additional measures of data security through selective encryption of stored data. Such native capabilities take the form of encryption toolkits (e.g. Encryption-GENERAL for MySQL [Pac11]) or packages (Oracle8i/9i [Ora01]), functions that can be embedded in SQL statements (IBM DB2 [IBM07]), or extensions of SQL (Sybase [Syb08] and SQL Server 2005 [Hsu08]). To limit performance overhead, selective encryption can be generally done at the column level but may involve changing the database schema to accommodate binary data resulting from the encryption process [Hsu08].

SQL Server 2008 [Hsu08] introduces *Transparent Data Encryption* (TDE) which is actually very similar to storage level encryption. The whole database is protected by a single key called *Database Encryption Key* (DEK), itself is protected by more complex means, including the possibility to use HSM. TDE performs all of the cryptographic operations at the I/O level, but within the database system, and removes any need for application developers to create custom code to encrypt and decrypt data.

TDE (same name as in SQL server 2008 but with different functionalities) has been introduced in Oracle 10g/11g, greatly enlarging the possibilities of using cryptography within the DBMS [Ora11]. Encryption keys can now be managed by an HSM or be stored in an external file named *wallet* which is encrypted using an administratively defined password. Selective encryption can be done at column granularity or larger (table space, i.e., set of data files corresponding to one or several tables and indexes). To avoid the analysis of encrypted data, Oracle proposes to include in the encryption process a *Salt*, a random 16 bytes string stored with each encrypted attribute value. An interesting but rather dangerous feature is the possibility to use encryption mode that preserves equality (typically a CBC mode with a constant initialization vector), thus allowing, for instance, to use indexes for equality predicates encrypting the searched values.

The database-level encryption with the security server approach mentioned above is proposed by IBM DB2 with the *Data Encryption Expert* (DEE [IBM07]) and by third-party vendors like Protegrity [Mat04], Packet GENERAL [Pac11], RSA BSAFE [RSA02] and SafeNet [Saf11] (appliance-based solution). The third-party vendors' products can adapt to most DBMS engines (Oracle, IBM DB2, SQL Server, MySQL and Sybase).

1.5 Crypto-Protection Strategies using HSM

Currently, existing architectures endowed with database encryption are not fully satisfactory since, as mentioned above, encryption keys appear in plaintext in the RAM of the server or on the client machine where the application runs. HSM acts as a safe storage to minimize the risk, diminishing the keys' exposure during their lifetime. Research is being conducted to make a better use of HSM, avoiding exposing encryption keys during the whole process. Two architectures can be considered: server-HSM, when the HSM is shared by all users and is located on the server; client-HSM, when the HSM is dedicated to a single user and is located near the user, potentially on the client machine. These two architectures are pictured in Figure 12.

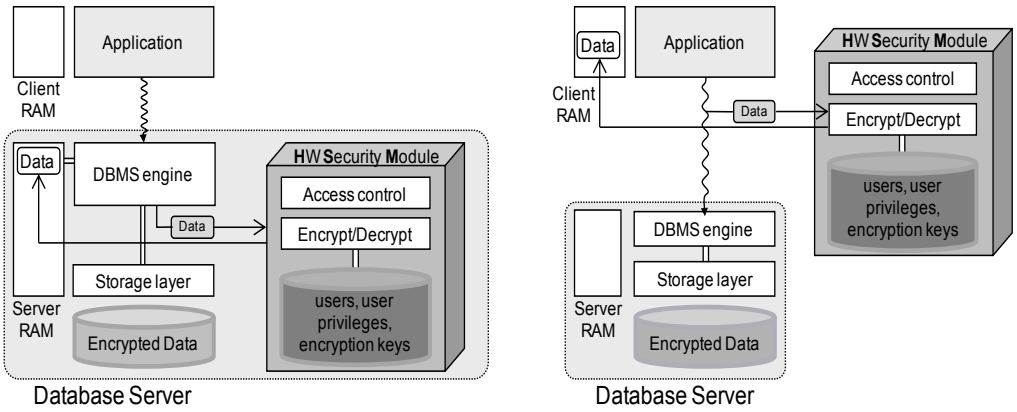


Figure 12. HSM-based new database encryption strategies

Logically, the server-HSM is nothing more than a database-level encryption with a security server embedded in the HSM. The HSM now manages users, privileges, encryption policies and keys. It has the same advantages as the database-level encryption with security server approach but does not expose encryption keys at any moment (since encryption/decryption is done within the HSM). Moreover, the security server cannot be tampered since it is fully embedded in the tamper-resistant HSM. With this approach, the only data that appears in plaintext is the query results that are delivered to the users. The main difficulty of this approach is its complexity, since a complex piece of software must be embedded in an HSM with restricted computation resources (due to security reasons). In Chapter 4, we give the design of embedded database coping with such constraints and show the feasibility of this approach. Regarding the research done in the database community, a simpler version of server-HSM solution has been prototyped in TrustedDB [BaS11], where tamper-proof trusted hardware is utilized only in critical query processing stages (involving sensitive operations). For non-sensitive operations, TrustedDB still dispatch them to the un-trusted host server (in Database-As-a-Service Model). The performance evaluation shows that the query cost are orders of magnitude lower than any (existing or) potential future software-only mechanisms.

While the client-HSM approach seems very similar to the server-HSM one and brings the same benefit in terms of security, it poses several new challenges. Indeed, the HSM is now dedicated to a single user and is potentially far from the server, thus making difficult any tight cooperation between the database server and the HSM. Thus, the database server must work on encrypted data and provide to the HSM a super-set of the query results, decrypted and filtered in

the HSM. Despite these difficulties, since the HSM is dedicated to a single user, the embedded code is simpler and less resource demanding, making this approach practical [BoP02].

2 State-of-the-Art

Database encryption has received more attention from the database community in the last decade. Encrypted databases incur many restrictions and challenges like operations and computations on encrypted data, view-based protection, performance degradation etc, many researchers in the database community have thus investigated relevant indexing and encryption techniques to enable efficient queries over encrypted database, without sacrificing database security. In this section, we will introduce the state-of-the-art works on these two aspects respectively. Finally, we also introduce works done in the community to guarantee database integrity properties, in particular to ensure the completeness, correctness and freshness of query results.

2.1 Indexing Encrypted Data

Index structures are critical components in DBMS, in particular to evaluate efficiently selection and join predicates. For an encrypted database, there are basically two approaches to build indexes: (i) building the index on the plaintext, and (ii) indexing the ciphertext. By nature, the second approach cannot tackle range searches, assuming classical encryption algorithms which are not order preserving (see next section). Therefore, most techniques proposed are adopting the first approach. In the following, we will present some representative works based on this approach, both in traditional DBMS model and in Database-As-a-Service (DAS) model [HIL+02a].

In traditional DBMS model, the server manages the data in-house and answers data requests from clients. Usually, the DBMS is partially trusted in this model, which means database server itself together with its memory and the DBMS software is trusted, while the secondary storage area is exposed to various attacks [SVE+09]. As a result, database encryption techniques are used to meet the requirements for data confidentiality and detection of unauthorized modifications.

In such model, common indexing techniques like B+ tree are often used, but are encrypted to enforce confidentiality. In [IMM+04], the authors mentioned an indexing scheme which construct B-tree index on the plaintext values and encrypt each page of the index separately.

When a specific page of the index is required during query processing, the whole page will be loaded into memory and decrypted. Consequently, this approach provides full index functionality (e.g., range queries and selection) while keeping the index itself secure. However, this scheme is implemented at operating system level, thus causing difficulties when it is impossible to modify the operating system implementation. In addition, such scheme is not flexible since encrypting different portions of database using different keys is not well supported. As the uniform encryption of all pages is likely to provide many cipher breaking clues, the indexing scheme provided in [BaM76] suggests encrypting each index page using a different key depending on the page number. Unfortunately, this scheme is implemented at the level of the operating system as well, and it is not satisfactory.

A new structure preserving indexing encryption scheme that does not reveal any information on the database plaintext values was proposed in [EWS+04]. Indeed, each index value (i.e., B+ tree node) is the result of encrypting the corresponding plaintext value in the database concatenated with its row-id, such that no correlation exist between the index values and the database ciphertext values. Furthermore, the index does not reveal the statistics or order of the database values. It supports equality and range queries. The encryption scheme allows separation of structural data (which is left unencrypted) from database content, hence preserve the structure of database. Thanks to this feature, DBA could manage the indexes (i.e., B+ tree) without the need of decrypting its values. However, this proposal assumes that the cell coordinates (including table-id, row-id, and column-id) are stable and do not change at the time of insert, update and delete operations, which puts high restrictions to the implementation of the DBMS. In their subsequent work [SWE+05], they give a more comprehensive discussion on the challenges for designing a secure index for an encrypted database, including prevention of information leakage; detection of unauthorized modifications; preservation of the index structure; support for discretionary access control; key storage and encryption granularity. Moreover, they suggest a secure database index encrypted at the granularity level of individual values, hence obtaining performance and structure perseverance (i.e., structure data are left clear). [Ulr06] cryptanalyses these schemes with respect to possible instantiations, gave counter-examples, and showed how to modify the schemes such that the original schemes lead to secure index encryption, i.e., by resorting to the authenticated encryption with associated data.

Regarding commercial database products, Oracle 8i allows encrypting values in any of the columns of a table. However, the encrypted column can no longer participate in indexing as the

encryption is not order-preserving [Ora00]. Other DBMS vendors also recommend not to encrypt indexed columns since it leads to full scan the encrypted column regardless of whether an index exist [Mat04].

In Database-As-a-Service (DAS) model [HIL+02a], data owners (e.g., organizations, companies) outsource their data in encrypted form to servers belonging to potentially un-trusted Application Service Providers (ASP). The main objective of this line of research is finding techniques for delegating data storage and the execution of queries to un-trusted servers while preserving efficiency [BoP02, HIL+02a]. To achieve such objective, indexes have to be built on server side to allow queries over encrypted database, without leaking any information about plaintext.

Hacigümüs et al. [HIL+02b, HIM04] proposed an approach based on information hiding-bucketization. Typically, tuples are encrypted using conventional methods, but an additional bucket id is built for each indexed attribute value. More precisely, the range of values in the specific domains of the attribute is divided into buckets, and each bucket is labeled explicitly using bucket id, which is then stored along with the encrypted tuples at the server. Since the bucket id represents the partition (i.e., bucket) to which the unencrypted value belongs, it replaces the constant (e.g., specific value) appearing in the query predicate during query processing. Bucketization could support range queries, join predicates, order by and group by clauses. The returned results will contain false positives, which should be filtered out by the data users (e.g., clients) in a post-processing step after decrypting the returned results. The number of false positives depends on the width of the partitions (e.g., bucket size) involved. Note that, each partition has the same length interval (i.e., same number of tuples are associated with each bucket), and there exists a trade-off between query efficiency and security levels. Indeed, to minimize the computation on client side, it is desirable to have more accurate results (few false positives) and finer bucketization. Unfortunately, such bucketization will disclose information about plaintext and suffer from estimation exposure, since the relative size of buckets reveals information about the distribution of the data, and relationships between fields in a tuple can be revealed as well [GeZ07a]. On the contrary, coarse bucketization improves security level but degrades query efficiency. In this line of research, Hore et al. [HMT04] analyze the bucketization technique and propose techniques to optimize the bucket sizes to generate privacy-preserving indices at the server side with a minimal information leakage. [WaD08] introduces a local overlapping bucket algorithm (LOB) achieving higher security by

trading off efficiency. More precisely, they propose a way to evaluate security and efficiency using probability distribution deviation and overlapping ratio, and a heuristic based greedy algorithm to determine optimal overlapping bucket. The efficiency of the proposed algorithm is shown under static scenario (no updates). However, the security of this algorithm is easily broken under dynamic scenarios.

Damiani et al. [DDJ+03, CDP05] propose hash based indexes which balance efficiency and security requirements, and offers quantitative measures to model potential inference attacks by exploiting indexing information. More precisely, the indexes are built based on direct encryption (for each indexed cell, the outcome of an invertible encryption function over the cell value is used as index) and hashing (the outcome of secure hash function over the cell value is used as index). Direct encryption approach preserves plaintext distinguishability and together with precision and efficiency in query execution. However, this approach opens the doors to frequency-based attacks. Hashing approach counters such attacks and provides more protection as different plaintext values are mapped onto the same index. Moreover, the authors also provide a measure of inference exposure of the encrypted/indexed data, by modeling the problem in terms of graph automorphism. Such index is suitable for exact match queries. To enable interval based queries in DAS context, the authors suggest building a B+ tree index over the plaintext values and encrypting the B+ tree at the node level. Therefore, the clients have to perform a sequence of queries that retrieve tree nodes (one query per level of the tree) until reaching the leaf level. Moreover, the references between the B+ tree nodes are encrypted together with the index values, thus the index structure is concealed.

In DAS model, except specially designed indexing techniques mentioned above, conventional index structures are also used to enable queries on encrypted database at server side, such as B+ tree [PaT04, LHK+06, LHK+10], bucket index [WDL+10] and R tree [YPD+08, MSP09, LHK+10] for multi-dimensional database etc. Moreover, to guarantee the correctness, completeness and freshness of returned results, authentication information is added to these indexes as well (see Section 2.3).

Other index techniques used for keyword searching [SWP00, Goh03] and XML databases are beyond the scope of this manuscript. Moreover, specific encryption algorithms (e.g., order-preserving) open another direction for building indexes for encrypted data, since they encrypt data in such a way that comparison or evaluation can be done directly on encrypted data, thus (i) indexes (e.g., B+ tree) could be built on ciphertext directly without need to encrypt them, (ii) or

they could be used to encrypt conventional indexes directly. Thanks to their specific features, index accesses will not incur cryptographic operation. We will introduce these algorithms in the next section.

2.2 Encryption Scheme

Many specific encryption algorithms have been investigated to enable computing queries directly on encrypted data, without leaking information about underlying plaintext.

Rivest et al. [RAD78] described a first approach for solving such problem called *Privacy Homomorphism* (PH), which allow mathematical operation on ciphertext values corresponding to operations on plaintext values. For instance, given ciphertext E(x) and E(y) (i.e., encryption form of plaintext value x and y), one can obtain E(x θ y) by performing E(x) θ E(y), thus computation (e.g. aggregation) can be done without involving cryptographic operations (e.g. decryption).

[HIM04] gives a first application of PH to aggregation queries in encrypted relational databases, by allowing basic arithmetic operation (+, −, ×) to be performed directly over encrypted tables. [OSC03] investigates a set of homomorphism encryption and decryption functions to encrypt integer-valued attributes while preserving their order/distance, and proposes a computing architecture to process simple queries (e.g., select) over an encrypted database. Their subsequent work [ChO06] addresses more complex SQL queries such as aggregate queries and nested queries. Unfortunately, since integer values preserve their order after encryption, thus information about the input distribution are revealed, which can be exploited by attackers. A simple additively homomorphic stream cipher was proposed in [CMT+05], it allows efficient aggregation of encrypted data in wireless sensor networks. In [EvG07], the authors present an encryption scheme based on PH to support an extensive set of relational operations (like select, projection and Cartesian product), and allows effective search on encrypted tables. [GeZ07a] gives a comprehensive solution for the SUM and AVG aggregate queries by using a secure homomorphic encryption scheme in a novel way (i.e., operates on a much larger (encryption) block size (e.g., 2K bits) instead of single numeric data values). [Cha09] introduces two symmetric-key homomorphic encryption schemes which allow the encrypted data to be added and multiplied by a constant. Moreover, they are secure to ciphertext-only attacks (an attack model where the attacker have access only to a set of cipher

texts) and have the nice property that the same data leads to different representations in the encrypted domain.

Some encryption algorithms are proposed to tackle comparison operations specifically. [AKS+04] makes use of *Order-Preserving Encryption Schemes* (OPES) to enable comparison operations to be directly applied over encrypted numeric data. Therefore, equality and range predicates, joins, group by and order by clause, as well as aggregates (MIN, MAX and COUNT operations except SUM and AVG) can be computed directly over encrypted data. Moreover, the results of query processing using OPES are exact. However, this scheme is only secure under the "ciphertext-only attack" model and breaks down when the adversary possesses background knowledge about domain. [BCL+09] relaxes the standard security notions (indistinguishability against chosen-plaintext attack) for encryption which is difficult to achieve for practical OPES, and propose an efficient OPES which is proven secure under relaxed security notions (based on pseudo-randomness of an underlying block cipher). [LiO05] introduce Prefix-Preserving Encryption (i.e., the prefix of values are preserved after encryption) to evaluate range queries on encrypted numeric data. This can be used to evaluate range queries and to build efficient encrypted indexes. Moreover, it examines proposed scheme under both ciphertext only attack and known plaintext attack (i.e., an adversary gain knowledge about certain number of <plaintext, ciphertext> pairs). However, both order-preserving and prefix-preserving schemes suffer from a common problem, i.e., they preserve statistics. Ge and Zdonik [GeZ07b] proposed an encryption scheme called FCE which allows fast comparison through partial decryption (i.e., the comparison procedure stops as soon as a difference is found), hence it can be used for range queries and indexing. Note, in this scheme, the server is assumed trusted and has the cryptographic key to decrypt data, even partially. However, it exhibits security weakness [GuJ09] and is thus not suitable to encrypt highly sensitive data.

In summary, the above introduced approaches based on specific encryption techniques (e.g., PH, order preserving or prefix preserving encryptions) aim at offering means to a query processor to compute results over the encrypted data. Obviously, the same arms are available to the attackers. Thus, by nature, those approaches can be considered as safe only under specific situations, when the adversary has (very) limited background knowledge. This has been demonstrated in [HHI+07].

2.3 Database Integrity

Encrypting the data only guarantees data confidentiality, but gives no assurance on data integrity, i.e., on the fact that the data has not been illegally forged or modified (authenticity), or replaced by older versions (freshness). In this section, we give an overview of authentication approaches proposed in database community.

In traditional DBMS model, [Den84] resorts to an isolated trusted filter, a security kernel responsible for enforcing the requirements for multilevel security in the database system (being a security kernel, it is nonbypassable, tamperproof, and verifiable). It computes an un-forgeable cryptographic checksum over the records to detect malicious modifications. The trusted database TDB [MVS00] uses standard cryptographic techniques (e.g., MHT), adapted to the underlying log-structured storage model, to prevent malicious corruption of the data. Moreover, it uses increment-only counters to prevent replay attacks. GnatDB [Vin02], which is a limited version of TDB designed for small cardinality databases, avoids building Merkle Hash Tree but relies on a trusted microcontroller. [SWE+05] allows protecting data against information leakage and unauthorized modifications by using MAC function, dummy values and pooling. More precisely, dummy values are inserted to the index at each insertion made by the user, thus reducing the adversary's level of confidence about the position of a value within the index. In addition, new inserted values in the pool are extracted in random order and inserted into the database table and index, to cope with dynamic leakage attacks (gaining information by analyzing the changes performed in the database over a period of time). [Ulr06] analyze the security weakness lying in this scheme, and propose to use authenticated encryption with associated data to guarantee confidentiality and integrity properties. Other methods such as MAC [ECG+09] and authenticated encryption [HGX+08] are also widely used.

Regarding the DAS model, all approaches investigated in database community could be divided into three classes: authenticated data structures, signature based approach and probabilistic approaches [WDL+10]. Authenticated data structures, such as DAGs [MND+04], skip list [BaP07] and Merkle Hash Tree (MHT) [Mer89] have been well investigated. In the following, we focus on MHT which is widely used to authenticate query results. Compared with signature based approach, MHT hashes are faster to compute and more concise than signatures, which leads to shorter construction time and smaller storage overhead [LHK+06].

In [DGM+00], Devanbu et al. exploited MHT to prove the completeness and authenticity of relational query results produced by un-trusted third party publishers. The scheme requires the data owner to construct MHT (i.e., binary tree constructed by using hash values) over each database table, and disseminate the signed root digest to users directly. [LHK+06] proposed an embedded Merkle B-tree (MB-tree) which combines the MHT with the B+ tree, and provides query authentication including freshness guarantees in dynamic environments where data is frequently updated. The concept of MHT also applied to R-tree for verifying the soundness and completeness of multi-dimensional database [YPP+09]. In [MSP09], the proposed *Partially Materialized Digest* (PMD) scheme approach is opposed to existing ones. Indeed, PMD does not incorporate the MHT into the data index for query processing. Instead, it uses a main index (MI) solely for storing and querying the data, and a separate digest index (DI) that contains the MHT-based verification information in a compressed form, to guarantee the query results are authentic and complete.

Regarding signature based approach, [PJR+05] introduces a signature scheme for users to verify that their query results are complete and authentic. The scheme supports range selection on key and non-key attributes, project as well as join queries on relational databases. Mykletun et al. [MNT04] proposed the use of two specialized signature schemes to allow aggregation of multiple individual signatures into one aggregated signature, thus reduce the signature size and verifying cost. [NaT05] developed an approach based on signature aggregation and chaining to provide evidence of completeness of query result set. In this research line, more works have been followed. [PZM09] introduces a protocol, built upon signature aggregation, for checking the authenticity, completeness and freshness of query answers. The propose protocol provides an important property, i.e., allowing new data to be disseminated immediately and detect outdated values (beyond a pre-set age). Moreover, the protocol caches a small number of strategically chosen aggregate signatures, to reduce proof construction time.

Except authenticated data structures and signature based approaches, there exist some probabilistic approaches for authenticating database: challenge token [Sio05] and fake tuple [XWY+07]. Compared with the others two approaches, the probabilistic ones can support richer types of queries (e.g., range, aggregation etc) at the cost of inexact authentication.

Unfortunately, all integrity approaches presented above are not adequate to ensure integrity properties for embedded PDS engine. On one side, we could take advantage of some specificities of PDS engine design and tamper resistance of SPT, and easy some integrity issues

(e.g., data freshness), instead of introducing heavyweight integrity mechanisms (e.g., signature based approach). On the other side, the special design and hardware limitations exhibited by SPT make the usage of traditional integrity methods inefficient. For instance, MHT does not adapt for the NAND Flash well, since the latter provides poor support for random writes. As a result, more efficient integrity methods have to be investigated to address integrity issues lying in PDS approach. We will develop them explicitly in Chapter 5.

Chapter 4
PDS Architecture and Embedded Data Management

In this chapter, we start by illustrating the Personal Data Server (PDS) vision through different scenarios motivating our approach. Next, we mainly focus on the PDS global architecture and PDS engine design, including hardware constraints, main elements (functionalities) of PDS, storage and indexing model, as well as query processing techniques dedicated for PDS engine. In the next chapter, we will address the crypto-protection for PDS data explicitly, satisfying the security requirements without incurring changes to existing design (presented in this chapter).

1 Motivating Examples

1.1 Healthcare Scenario

Alice carries her electronic healthcare folder (along with other information) on a PDS. She has an account on e-Store, a Supporting Server provider. She downloaded in her PDS, from the Ministry of Health, a predefined healthcare database schema, an application to exploit it, and an access control policy defining the privileges attached to each role (physician, nurse, etc). Alice may manage the role assignment by herself or activate specific user policies predefined by e.g., a patient association. When she visits Bob, a new physician, she is free to provide her SPT or not, depending on her willingness to let Bob physically access it (this is a rough but effective way to control the sharing of her data, as with a paper-based folder). In the positive case, Bob plugs Alice's PDS on his terminal, authenticates to the PDS server with his physician credentials, queries and updates Alice's folder through his local Web browser, according to the physician's privileges.

Bob prescribes a blood test to Alice. The test result is sent to Alice by the medical lab in an encrypted form, through e-Store acting here as a secure mailbox. The document is downloaded from e-Store and wrapped by Alice's PDS to feed the embedded database. If this document contains information Alice would like to keep secret, she simply masks this document so that it remains hidden from any user querying the database except her. The lab keeps track of this medical act for administrative purposes but does not need anymore to keep a copy of its medical content. If Alice loses her PDS, its tamper-resistance renders potential attacks harmless. She will then recover her folder from an encrypted archive stored by e-Store using, e.g., a pass-phrase.

Alice suffers from a long-term sickness and must receive care at home. Any practitioner can interact at home with Alice's PDS thanks to his netbook, tablet PC or PDA without need for an Internet connection. To improve care coordination, Bob convinces Alice to make part of her folder available 24/7, during a one-month period, to him and to Mary, a specialist physician. Alice uploads the requested part of her folder encrypted on e-Store. The secret key is exchanged with Bob's and Mary's PDSs in order for them to be able to download Alice's data on their own PDS and query it. While Alice's data is now replicated on Bob's and Mary's PDSs, Bob and Mary cannot perform actions on the replica exceeding their privileges and this replica will be destroyed after a one-month period because their PDS will enforce these controls. Bob and Mary's actions are recorded by their own PDSs and sent back to Alice through e-Store for audit purpose. To make this sharing scenario possible, patients and practitioners are all assumed to be equipped with PDSs and these PDSs are assumed to share a compliant database schema. Finally, if the Ministry of Health decides to compute statistics or to build an anonymized dataset from a cohort of patients, the targeted PDSs will perform the processing and deliver the final result while preventing any leakage of sensitive data or identifying information.

1.2 Vehicle Tracking Scenario

John, a traveling salesman, drives a car from his company during working hours and shares his personal car with Cathy, his daughter. Both have a PDS that they plug in the car to register all their personal trips. Several applications are interested in the registered GPS locations. John's insurance company adapts the insurance fee according to different criteria (e.g., the distance traveled, type of road used, and speed). Cathy will probably pay more than her father because she lacks enough driving experience. The Treasury is also interested by this information to compute John's carbon tax according to similar criteria, though the computation rules are different. Finally, John's company would also like to track John's moves to organize his rounds better. GPS raw data is obviously highly private. Fortunately, John's PDS externalizes only the relevant aggregated values to each application. In other words, each application is granted access to a particular view of the data registered in John's database.

1.3 BestLoan.com & BudgetOptim Scenarios

Alice needs a loan to buy an apartment. She would like to find the best rates for her loan and, thus, relies on the service of BestLoan.com (BL for short), a mortgage broker. To assess Alice's financial situation, BL needs to get access to sensitive information from Alice's PDS such as

salary, bank statements and tax information. Alice's data can be securely shared with Donald, a BL employee, as follows: (1) Alice opts in for the BL application and downloads the security policy associated to it in her PDS, (2) Donald authenticates to Alice's PDS with his credentials embedded in his own PDS and requests the required data, (3) Alice agrees to share this data with Donald for a specified duration (e.g., two weeks), (4) finally Donald downloads the data in his PDS, all this by exchanging messages and data through the e-Store Supporting Servers. Donald cannot perform actions on Alice's data exceeding their privileges or the retention period fixed by Alice because his PDS will preclude these actions. If Alice distrusts Donald, she can audit his activity and can at any moment opt out of the BL application (with the effect of deleting Alice's data in Donald's PDS), all this again by exchanging messages through the e-Store.

Alice now wants to optimize her budget and thus opts in for the BudgetOptim application (BO for short). BO runs locally on Alice's PDS with a GUI running on the terminal. BO accesses details of Alice's invoices, telecom bills, etc. in order to suggest more advantageous services according to her consuming profile. With BO application, Alice does not share data with anybody. This last scenario is typical of many private applications that can process personal data (e.g., diet advices, tax minimization, pension simulation, vaccine reminders, etc.).

2 The PDS approach

The idea lying in the PDS approach is to embed in Secure Portable Tokens (SPT for short) software components capable of acquiring, storing and managing personal data. However, it does not amount to a simple secure repository of personal documents. Instead, the objectives of this approach are:

- Allow the development of new, powerful, user-centric applications and serve data requests from existing server-based applications managing personal data. Consequently, it requires a well organized, structured, consistent and queryable representation of personal documents;
- Provide the user of the SPT with a friendly control over the sharing conditions related to her data and with tangible guarantees about the enforcement of these conditions

These two objectives lead to the definition of a real secure and portable Personal Data Server (PDS for short). Unfortunately, a SPT cannot provide on its own all the required database functionalities of a PDS (e.g., durability, if the SPT is lost or destroyed, availability when the

SPT is disconnected, global queries involving data from several SPTs), thus we have to resort to external servers, called hereafter supporting servers, to rebuild such functionalities.

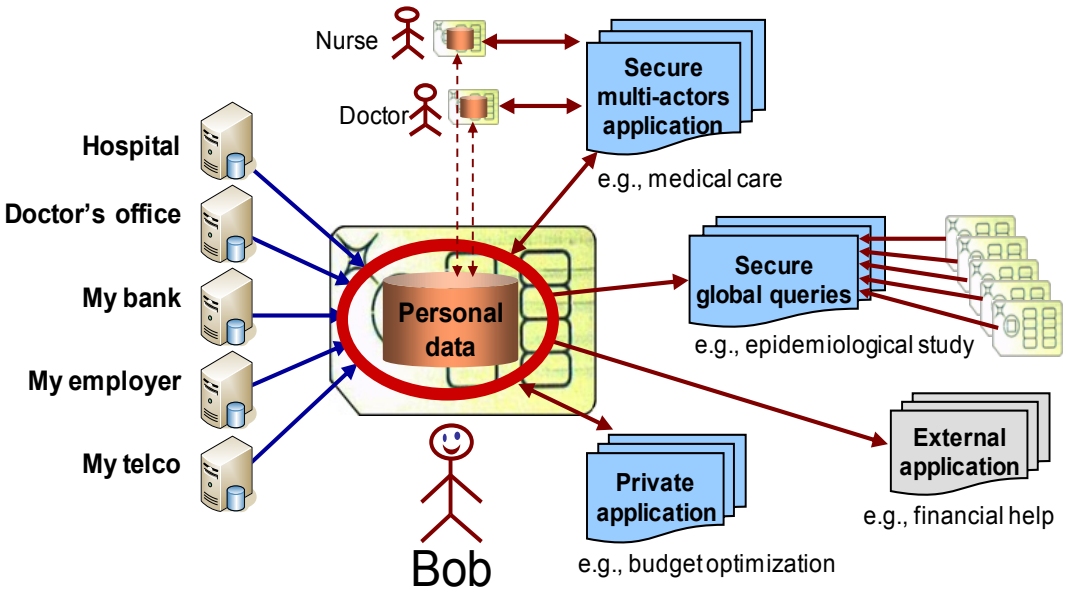


Figure 13. The Personal Data Server Approach

Figure 13 shows various applications that could be built based on this approach (supporting servers are not explicitly shown). Typically, Bob's personal data, delivered by different sources, is sent to his PDS which can then serve data requests from private applications (serving Bob's interest, e.g. Budget optimization), secure multi-actors applications (accessed through actors' PDS, e.g. medical care) and external applications (e.g. financial help). Bob's PDS can also take part in secure global processing (e.g. epidemiological study).

In order to support such powerful applications, a first requirement is to embed a full-fledged database engine (PDS engine) in the SPT (we assume the database model is relational in this manuscript but this choice has little impact on the global architecture). However, as SPT exhibits intrinsic limitations (e.g. limited RAM, NAND Flash constraints), thus the database engine design has cope with these constraints (refer to Section 4).

In addition, the SPT microcontroller only provides limited security perimeter, such that (i) data stored on NAND Flash in SPT is vulnerable to confidentiality and integrity attacks, (ii) some (shared) data or message have to be externalized to supporting servers for durability, availability, global processing purpose, thus leading to potential information exposure.

Consequently, corresponding protection schemes have to be devised by resorting to cryptography technology (see Chapter 5).

3 PDS Global Architecture

3.1 Problem Statement

SPTs appear today in a wide variety of form factors ranging from SIM cards to various forms of pluggable secure tokens (but this manuscript makes no assumption on the form factor), see Figure 1. As mentioned before, whatever the form factor, SPTs share several hardware commonalities. Their secure microcontroller is typically equipped with a 32-bit RISC processor (clocked at about 50 MHz today), a cryptographic coprocessor, a tiny static RAM (about 64 KB today), a small internal stable storage (about 1 MB today) and security modules providing the tamper-resistance. The internal stable storage provides Execute-In-Place (XiP) capability to the embedded code and can store sensitive metadata.

The microcontroller is connected by a bus to a large external mass storage (Gigabytes of NAND Flash) dedicated to the data. However, this mass storage cannot benefit from the microcontroller tamper resistance. SPTs can communicate with the outside world through various standards (e.g., USB2.0, Bluetooth, 802.11).

In the secure chip domain, hardware progresses are fairly slow, because the size of the market (billions of units), and because the requirement for high tamper-resistance leads to adopt cheap and proven technologies [Eur08]. Nonetheless, SPT manufacturers forecast a regular increase of the CPU power, stable storage capacity and the support of high communication throughputs (up to 480 Mb/s). RAM will unfortunately remain a scarce resource in the foreseeable future due to its poor density. Indeed, the smaller the silicon die, the more difficult it is to snoop or tamper with its processing, but RAM competes with CPU, ROM and NOR in the same silicon die.

In summary, a SPT can be seen as a low power but very cheap (a few dollars), highly portable, highly secure computer with reasonable storage capacity for personal use. Regarding the design of an embedded DBMS engine, the mentioned hardware characteristics can be translated into four main constraints [ABG+11]:

- *Constraint C1* “Poor RAM/stable storage ratio”: the RAM capacity is rather low and must be shared between the operating system, the DBMS and the applications. Conversely, fewer

constraints exist on the external stable storage and the ratio RAM/stable storage has decreased by two orders of magnitude in the last years. The trend that this ratio continues to decrease makes this constraint even severe.

- *Constraint C2* "Scarce secure stable storage": only the internal XiP stable memory benefits from the tamper-resistance of the microcontroller. Although its capacity is regularly growing in order to tackle the pressure of embedded applications, it still remains a scarce resource for other usages like storing sensitive data.
- *Constraint C3* "External stable storage vulnerability": external memory must be protected against confidentiality and integrity attacks since it is beyond the secure perimeter of tamper-resistance rendered by microcontroller. Indeed, a secure microcontroller cannot embed large stable storage due to the silicon die size constraint mentioned above.
- *Constraint C4* "NAND Flash update behavior": the NAND Flash is badly adapted to fine-grain data (re)writes. The memory is divided in blocks, containing (e.g., 64) pages themselves containing (e.g., 4) sectors. The write granularity is the page (or sector) and pages must be written sequentially within a block. A page cannot be rewritten without erasing the complete block containing it and a block wears out after about 10^4 repeated write/erase cycles. To tackle these constraints, updates are usually managed out of place with the following side effects: (1) a Translation Layer (TL) is introduced to ensure the address invariance at the price of traversing/updating indirection tables, (2) a Garbage Collector (GC) is required to reclaim stale data and may generate moves of valid pages before reclaiming a non empty block and (3) a Wear Leveling (WL) mechanism is required to guarantee that blocks are erased evenly. TL, GC and WL are black-box firmware. Their behavior is difficult to predict and optimize with the consequence that random writes can be up to order(s) of magnitude more costly than sequential writes [BJB+09]. In particular settings, these components can be deactivated. The technology trend is to increase the density of Flash (e.g., MLC vs. SLC), thereby ever worsening these constraints. Other stable storage technology could be envisioned in the future, like Phase-change memory (PCM), but the term of their appearance is still unclear, especially for such kind of devices. Hence, according to SPT manufacturers, all four constraints will remain effective in the foreseeable future.

Among above constraints, C1 and C4 mainly impact the PDS engine design, thus specific storage and indexing model need to be devised to cope with such constraints (see Section 4). C2

and C3 have greater influence on the crypto-protection designs, which is the core contribution made by this manuscript, and we address them explicitly in the next chapter.

3.2 Personal Database

The Personal Database is assumed to be composed of a small set of relational database schemas, typically one per application domain (e.g. e-health, e-administration etc). Database schemas are defined by *DB Schema Providers*. Depending on the domain, a *DB Schema Provider* can be a government agency (e.g., Ministry of Health) or a private consortium (e.g., a group of banks and insurances). *Content Providers* are external information systems (e.g. administrations or companies) that deliver personal data (e.g., blood test, salary form), encoded in XML.

We make the simplifying assumption that each XML document conforms to one XML schema defined by a standardization organization (e.g., HL7) or by a *DB Schema Provider* (e.g., the Ministry of Health). To allow building a consistent and structured view of a set of related documents, an XML document (e.g., a prescription) is enriched with all referential data required to fill the embedded database accurately (e.g., detailed data related to the doctor who wrote the prescription and to the drug prescribed). Hence, the data contained in different documents related to a given doctor or drug can be easily retrieved by SQL queries and cross documents processing becomes possible (e.g., get the list of current medical treatments or compute average blood pressure during the last month). Then the enriched document is pushed in an encrypted form to the recipient PDS through supporting servers. The recipient PDS downloads the XML document and wraps it into a set of tuples thanks to mapping rules. The benefit of declarative mapping rules is not only that it simplifies the work of the DB Schema Provider but primarily that the safety of these rules can be controlled.

Figure 14 illustrates the wrapping of a prescription, enriched with doctor and drug referentials sent from a hospital. The document conforms to an XML schema for healthcare, and is wrapped into four tables (two of them being referentials) from the healthcare database schema. However, not all documents are wrapped and integrated in the database. Some documents (e.g., an X-ray image) can stay encrypted in the supporting servers and simply be referenced by the embedded database.

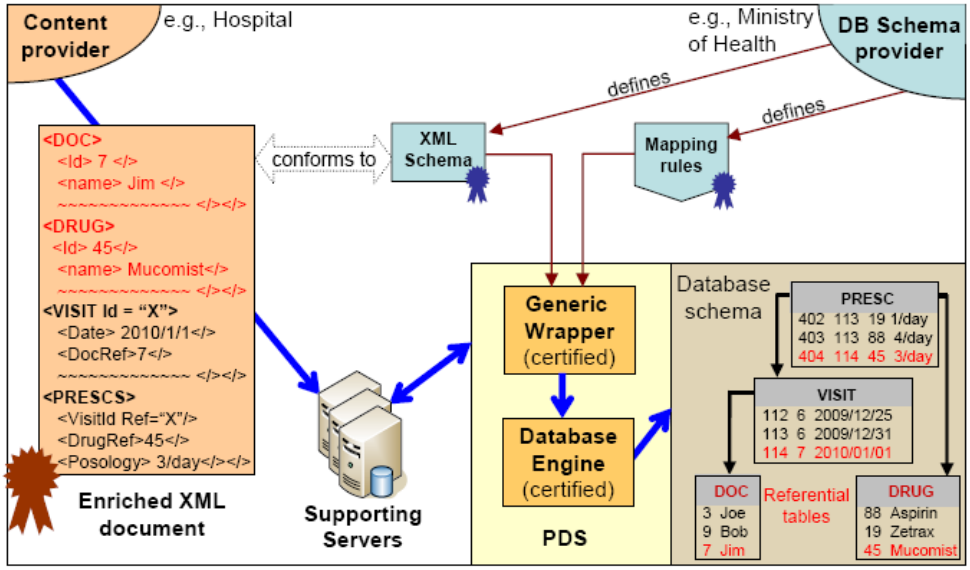


Figure 14. Wrapping a document into the PDS database

3.3 Applications

Applications are developed by Application Providers (e.g., BestLoan.com), they are defined on top of the published DB schema(s) of interest and can use all database functionalities provided by the embedded DBMS (i.e., DDL and DML statements). Each application defines a set of collection rules specifying the subset of documents required to accomplish its purpose (e.g., the five most recent salary forms are required by BestLoan.com). These rules are expressed at the document level to make them meaningful to the PDS holder (helping him to opt in or opt out of this application) and are mapped at the database level to be enforced similarly to access control rules.

Most applications are assumed to perform only selection queries, insertion of new documents is not precluded (e.g., a treatment prescribed at home by the doctor). An updating application will play the same role as a *Content Provider* and the insertion will follow the same process.

3.4 Embedded Software Architecture

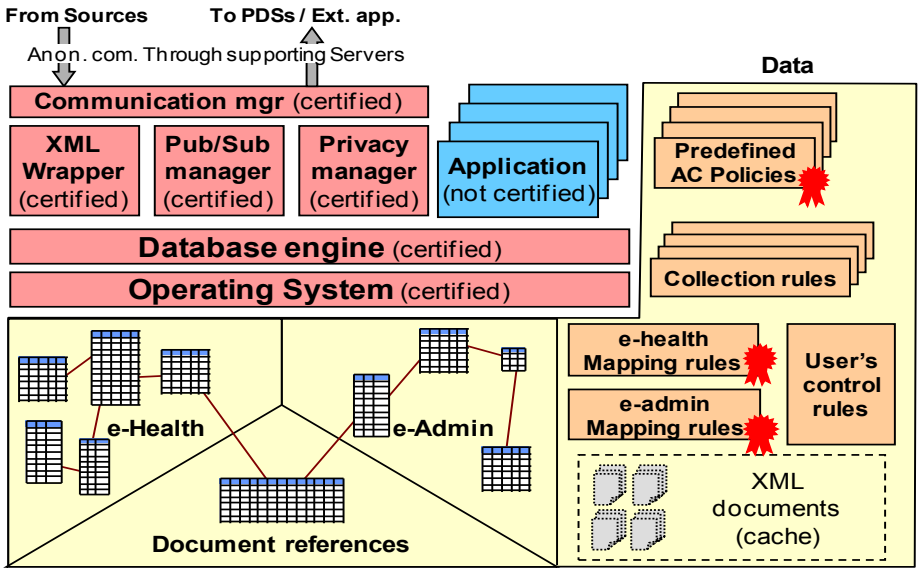


Figure 15. PDS generic software, application, and database

- *The Communication Manager* implements communications with the outside world through supporting servers. On one side, it receives various personal documents (e.g. salary forms, insurance forms, invoices, phone call sheets, banking statements, etc) from external sources. On the other side, it sends shared data or messages to other PDSs or external applications through supporting servers. The Communication Manager integrates a protocol such as Onion-routing [GRS99] to implement anonymous communications like in TOR [DMS04], hence the PDS can send/receive shared data/messages or participate to global query without revealing her identity.
- *The XML Wrapper* wraps the downloaded XML documents into a set of tuples based on mapping rules provided by *DB Schema Provider*, these mapping rules define the transcription of documents into a structured database.
- *The Pub/Sub Manager* implements secure Publish/Subscribe mechanism addressing minimal exposure issue (see Section 3.5) for users' control over the shared data. Indeed, PDS holders (i.e. publishers) publish shared data in encrypted form on the supporting servers, the recipient PDSs (i.e. subscribers) have to subscribe to this data first, then receive the decryption key if the publisher accepts this subscription.
- *The Privacy Manager* is used to enforce user's control rules, which can be fixed by the PDS holder herself to protect her privacy, namely masking rules, retention rules and audit rules.

User's control rules are enforced by all PDSs, both on the PDS holder's data and on the data downloaded after a subscription.

- *Cached Documents*: Considering that some applications may need to view original XML documents, thus PDS store them in a local cache. PDS could also retrieve them from supporting servers in the case of cache misses.

3.5 User Control

The prime way for the PDS holder to control the usage of her data is to opt-in/out of applications and to decide situations where she physically delivers her PDS to another individual (e.g., a doctor). Assuming that the PDS holder's consent has been given, the actions that any individual can perform are regulated by a predefined access control policy.

Predefined access control policies are usually far too complex to be understandable by the PDS holder (e.g., the RBAC matrix regulating the use of the French EHR contains more than 400 entries). It is therefore mandatory to provide the PDS holder with simple tools to protect her sensitive data following her wish. A first way consists in managing the privileges through a simple GUI, as illustrated in the healthcare scenario. A second way is to give the user the ability to mask documents in the database. The records corresponding to a masked document are no longer considered at query execution time, except if the query is issued by the PDS holder herself (through an application). To make this process intuitive, the DB Schema Provider can predefine masking rules (e.g., hide documents by doctor, pathology, time period, etc.) exploiting the expressive power of the DBMS language and easily selectable by the user through a GUI.

The PDS holder (called hereafter the donor) can also impose privacy preserving rules whenever data leaves her PDS to enter another PDS. This sharing is required when a donor's data must be made available while her PDS is disconnected (see the healthcare scenario). This sharing must be ruled by the following principles:

- *Minimal exposure*: in a nominal use, only the results of authorized queries are externalized by a PDS and raw data always remains confined in the PDS. When donor's raw data is made available to others, this must be done in such a way that minimal data (limited collection principle) is exchanged during a minimal duration (limited retention principle) and with the minimum number of recipient PDSs (need-to-know principle) to accomplish the purpose of this externalization.

- *Secure delete*: if the donor decides to delete a document before the retention period expires, all replicas of the corresponding raw data hosted by the recipient PDSs must be deleted.
- *Audit*: the donor must have the ability to audit the actions performed by all recipient PDSs on replicas.

3.6 Supporting Server

Supporting servers are essential part in PDS global architecture, they provide storage and timestamp services to implement the functions that PDSs cannot provide on their own, namely:

- *Asynchronous communication*: since PDSs are often disconnected, documents, shared data and messages must be exchanged asynchronously between Content Providers and PDSs and between PDSs themselves through a storage area, these communications are anonymous.
- *Durability*: the embedded database must be recovered in case of PDS loss or destruction. The PDS holder's personal data can be recovered from the documents sent by *Content Providers* through the supporting servers (assuming these documents are not destroyed). Data downloaded from other PDSs also can be recovered from the data published in the supporting servers (assuming their retention limit has not been reached). Other data (user's control rules definition, metadata built by applications, etc.) must be saved explicitly by the embedded DBMS on the supporting servers (e.g., by sending a message to itself).
- *Global processing*: a temporary storage area is required to implement processing combining data from multiple PDSs, such as statistical queries and data anonymization.
- *Timestamping*: the SPT hardware platform is not equipped with an internal clock since it takes electrical power from the terminal it is plugged in. Hence, a secure time server is required to implement auditing and limited retention.

4 Embedded Database Design

4.1 Design Guidelines

The hardware constraints discussed above induce two challenges for the PDS engine design. In this section, we address both challenges and deduce corresponding design rules.

Challenge 1: Computing complex queries on Gigabytes of data with tiny RAM

This challenge is a direct consequence of constraint C1. Select-Project-Join-Aggregate queries must be executed on Gigabytes of data with Kilobytes of RAM. It is well known that the

performance of “last resort” join algorithms (block nested loop, sort-merge, hybrid hash) quickly deteriorates when the smallest join argument exceeds RAM size [HCL+97]. Jive join and Slam join use join indices [LiR99] but both require that the RAM size is of the order of the square root of the size of the smaller table. In addition, swapping data in the terminal or in the external NAND Flash is precluded in the PDS context due (1) to the dramatic amount of swapping required considering the ratio between the RAM size and the potential size of the tables to be joined and (2) to the cost of encryption (only the microcontroller is trusted). This means that not only joins but all operators involved in the query must be evaluated in a pipeline fashion, in order to minimize RAM consumption and intermediate results’ materialization. Materialized views are not an option considering applications are not known in advance since they can be dynamically downloaded in the SPT. This leads to the first design rule:

Rule R1: Design a massive indexing scheme where all (key) joins are precomputed and allowing pipeline computation of any combination of selections and joins.

Multi-way join indexes called Sub-tree Key Table, and Climbing Indexes, allowing to speedup selections at the leaves of a join tree were proposed in [ABB+07]. Combined together, these indexes allow selecting tuples in any table, reaching any other table in the join path in a single step. Queries can then be executed in a pure pipeline fashion without consuming RAM or producing intermediate results. This work must be considered as a first step towards the definition of indexing models and query execution techniques dedicated to PDS engine.

Unfortunately, such massive indexing scheme adapts badly on NAND Flash due to its fine grain access pattern (e.g. read, write and rewrite), hence, implementing them effectively on Flash becomes another challenge:

Challenge 2: Implement a massive indexing scheme compatible with the NAND Flash constraint

Traditional indexing techniques (e.g., B+ Tree) are poorly adapted to NAND Flash because of the high number of random writes they incur [WCK03]. All improvements we are aware of (e.g., BFTL [WCK03], Lazy-Adaptive Tree [AGS+09], In Page Logging [LeM07]) rely on the idea to defer index updates using a log (or Flash-resident cascaded buffers) and batch them to decrease the number of writes. The side effect is a higher RAM consumption (to index the log or to

implement write-coalescing of buffers) and a waste of Flash memory space, which is incompatible with constraint C1. This leads to the second design rule:

Rule R2: Design a dedicated storage and indexing model matching natively the NAND Flash constraints, i.e., proscribing random writes.

The problem addressed here is to find a global design matching rules R1 and R2 simultaneously, tackling both rules together is mandatory to guarantee the design consistency. Unfortunately, these two rules are conflicting by nature. Indeed, rule R1 leads to define data structures having usually a fine grain read/write/rewrite pattern thereby hurting R2. Consequently, we introduce two main principles to tackle this challenge, namely database serialization and database stratification. These two principles impact all components of a DBMS engine: storage, indexing, buffering, transaction management, query processing, we will introduce them respectively in the following.

In the following, we first sketch in Section 4.2 the main lines of the global design and postpone to the next section a detailed discussion about the principal challenge that is, how to build efficient sequential indexes in Flash. Finally, Section 4.4 shows query processing (including integrating updates and deletes) using the designed storage and indexing model.

4.2 Database Serialization and Stratification

To reconcile rule R1 with R2, we have to break the implication between massive indexing and fine-grain (re)write pattern, this is precisely the objective pursued by *database serialization*.

4.2.1 Database serialization

The database serialization paradigm is based on the notion of *Sequentially Written Structures* defined as follows:

Definition A *Sequentially Written Structure* (SWS) is a data container satisfying three conditions: (1) its content is written sequentially within the (set of) flash block(s) allocated to it (i.e., pages already written are never updated nor moved); (2) blocks can be dynamically added to a SWS to expand it at insertion time; (3) allocated blocks are fully reclaimed when obsolete and no partial garbage collection ever occurs within a block.

If all database structures are organized as SWS, including all forms of indexes required by rule R1, base data, logs and buffers, rule R2 would be satisfied since the SWS definition proscribes random (re)writes, hence the dramatic overhead of random writes in Flash would be avoided. Moreover, the TL cost would be saved and the GC could be implemented for free on a block basis.

However, the database serialization objective is easy to express but difficult to achieve. It requires solving the following problems:

1. *Base data organization*: natural solutions can be devised to organize the base data as SWSs. Typically, a table can be stored as a sequence of rows in a Row Store scheme or as a set of sequences of attribute values in a Column Store one. In this thesis, we consider Row Store scheme.
2. *Join Indexes*: join and multiway join indexes [ABB+07] can be directly mapped into SWSs as well. Indeed, inserting new base data incurs simply inserting sequentially new entries in these indexes (sorted join indexes are not considered). Since they are managed is the same way, we call DATA the SWSs dedicated to both base data and join indexes.
3. *Selection indexes*: any form of clustered index (e.g., tree-based or hash-based) is proscribed since inserting new base data would generate random node/bucket updates. We say that an index is cumulative if the insertion of new base data incurs the strict adjunction of new index entries (e.g. Bitmap index). In Section 4.3, we propose smarter forms of cumulative indexes, by deriving new information improving the lookup process from the previous state of an index and the current data to be inserted. Although cumulative indexes are less efficient than their clustered counterpart, they can provide a significant gain compared to scanning DATA. We call IND the SWSs dedicated to cumulative indexes.
4. *Flash Buffers*: DATA and particularly IND being made of fine grain elements (tuples, attribute values or index entries), inserting without buffering would lead to waste a lot of space. Indeed, constraint C4 imposes adding a new page in a SWS for each elementary insertion. Moreover, the density of a SWS determines the efficiency of scanning it. The objective of buffering is to transform fine-grain to coarse-grain writes in Flash. Elements are gathered into a buffer until they can fill a complete SWS page, which is then flushed. But buffers cannot reside in RAM, partly because of constraint C1 and because the SPT has no electrical autonomy. Hence, buffers must be saved in NAND Flash and the density of buffer pages depends on the transactions activity. To increase buffer density (and therefore

save writes), elements from different SWSs are buffered together. Having one buffer per insertion rate is a simplifying factor (e.g., *n* IND indexing different attributes of a same DATA can be grouped together and be filled and flushed synchronously). When enough data items have been buffered to fill a complete page of each IND, buffers are flushed. Buffers must be organized themselves as SWS to comply with the serialization objective. A buffer is actually managed as a sliding window within its SWS, a Start and a End markers identifying its active part (i.e., the part not yet flushed). We call BUF the SWSs implementing buffers.

5. *Updates/deletes*: applying updates and deletes directly in a target SWS (DATA, IND or BUF) would violate the SWS definition. Instead, updates and deletes are logged in dedicated SWSs, respectively named UPD and DEL. To manage updates, the old and new attribute values of each updated tuple are logged in UPD. At query execution time, UPD is checked to see whether its content may modify the query result (i.e., if a logged value matches a query predicate). If so, the query is adjusted to eliminate false positives (i.e., tuples matching the query based on their old value but not on their new value) and to integrate false negatives (i.e., tuples matching the query based on their new value but not on their old value). UPD and DEL are also checked at projection time, to project up-to-date values and to remove deleted tuples from the query result. To avoid accessing UPD and DEL on Flash for each result tuple during query processing, dedicated structures are built in RAM at each session (see Section 4.4.2).
6. *Transaction atomicity*: rollbacking a transaction, whatever the reason, imposes undoing all dirty insertions to the SWSs. To avoid the presence of dirty data in DATA and IND, only committed elements of BUF are flushed in their target SWSs as soon as a full page can be built. Hence, transaction atomicity impacts only the BUF management. In addition to the Start and End markers of BUF, a Dirty marker is needed to distinguish between committed and dirty pages. Rollback insertions leads (1) to copyback the elements belonging to the window [*Start*, *Dirty*], that is the committed but unflushed elements, after *End* and (2) to reset the markers (*Dirty=Dirty-Start+End, Start=End, End=Dirty*) thereby discarding dirty data.

At first glance, database serialization is a powerful paradigm to build a robust and simple design for an embedded DBMS engine complying with rules R1 and R2. However, such a design scales badly. Indeed, a cumulative index cannot compete with its clustered counterpart and the accumulation over time of elements in UPD and DEL will unavoidably degrade query

performance. There is a scalability limit (in terms of IND, UPD and DEL size) after which the user's performance expectation will be violated. To tackle this issue, we propose database the stratification paradigm addressed below.

4.2.2 Database stratification

This section describes the stratification process which transforms a SWS database organization into a more efficient SWS database organization. Let us call DB_0 the initial serialized database, i.e., stratum 0. DB_0 is composed of (1) the SWSs materializing the buffer part and (2) the SWSs storing the base data, the indexes, the update and delete logs. For simplicity, we call this latter part SDB which stands for Sequential Database. Thus, $DB_0 = BUF_0 \cup SDB_0$ where $SDB_0 = DATA \cup IND \cup UPD \cup DEL$.

When the scalability limit is reached, a new stratum DB_1 is built such that $DB_1 = BUF_1 \cup SDB_1 \cup CDB_1$ where BUF_1 and SDB_1 are initially empty and CDB_1, which stands for Clustered Database, is the optimal restructuring of DB_0. The data part of CDB_1, denoted by CDB_1.DATA, results from the integration in SDB_0.DATA of all updates and deletes registered in SDB_0.UPD and SDB_0.DEL and of all data items currently buffered in BUF_0. The index part of CDB_1, denoted by CDB_1.IND, corresponds to a clustered reorganization of SDB_0.IND. For instance, cumulative indexes in SDB_0 can be replaced by B-Tree like indexes in CDB_1 (see Section 4.3). Hence, at the time DB_1 is built, it corresponds to the best database organization we may expect. As soon as reorganization starts, new insertions occur in BUF_1 and SDB_1, thus CDB_1 remains stable. The next time the scalability limit is reached, a new stratification step occurs. Stratification is then an iterative mechanism summarized as follows:

Stratify(DB_i) → DB_{i+1}
// CDB_0=∅;
CDB_{i+1}.DATA = Merge (CDB_i.DATA, BUF_i.DATA, SDB_i.DATA, SDB_i.UPD, SDB_i.DEL);
CDB_{i+1}.IND = ClusterIndex (CDB_i.IND, BUF_i.IND, SDB_i.IND, SDB_i.UPD, SDB_i.DEL);
Reclaim (BUF_i, SDB_i, CDB_i);
BUF_{i+1}= ∅;
SDB_{i+1}= ∅;

At each stratification step, CDB_{i+1}.DATA and CDB_{i+1}.IND are built, as new SWSs, and BUF_i, SDB_i and CDB_i are totally reclaimed. Hence, rule R1 is preserved since CDB_{i+1}.IND is assumed to provide an optimal indexing scheme; R2 is preserved since random writes are never produced despite the reorganization (new SWSs are written sequentially and old SWSs are fully reclaimed). Hence, stratification is very different in spirit from batch approaches deferring

updates thanks to a log since such deferred updates produce random rewrites. The price to pay however is a complete reconstruction of CDB_i at each step. This raises the following remarks:

- According to Yao's formula [Yao77], the cost of producing CDB_i is (almost) independent of the size of SDB_i since little locality can be expected when reporting SDB_i elements into CDB_i. Hence, there is a high benefit to maximize the size of SDB_i in order to reduce the number of stratification steps, thereby decreasing the global cost of stratification.

- According to the duration of each stratification step, there is a high benefit to perform it incrementally in order to run this process in background (not in parallel). The consequence is that the query processing must accommodate data which coexist in two strata at a particular point of time.

4.3 Indexing Techniques

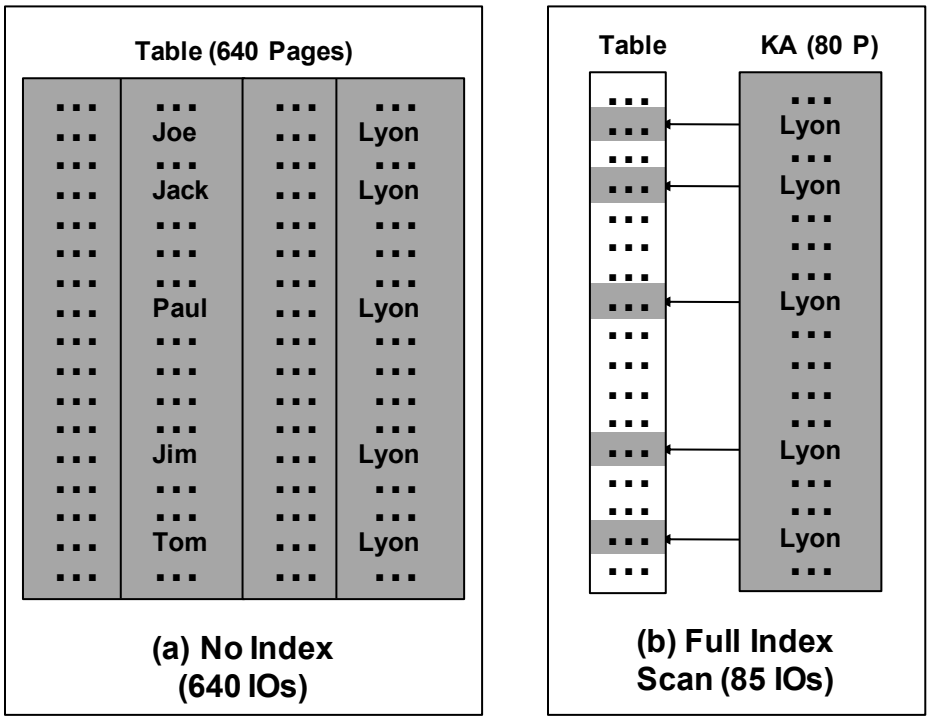
With no index, finding matching tuples leads to sequential scan of the whole table (see Figure 16.a). Simple cumulative indexes can be created by replicating the indexed attribute (called key) into a SWS called *Key Area* (KA) (see Figure 16.b). The row identifiers of matching tuples are found by a sequential scan of KA, thus the name *Full Index Scan*. *Full Index Scan* is simple and works for any exact match or range predicate. For variable or large size keys, a collision resistant hash [MOV97] of the key can be used in place of the key itself. However, this restricts the index use to exact match predicates. For keys varying on a small cardinality domain, the key can also be advantageously replaced by a bitmap encoding. Adequate bitmap encoding can be selected to support range predicates [ChI99].

Smarter forms of cumulative indexes can be devised to support exact-match predicates. In [YPM09], Yin et al. propose to summarize a sequential index using *Bloom Filters* (BF) [Blo70]. A BF represents a set of arbitrary length values in a compact way and allows probabilistic membership queries with no false negatives and a very low rate of false positives. For example, the false positive rate produced by a BF built using 3 hash functions and 12 bits per value is 0.1, it decreases to 0.05 with 16 bits per value and to only 0.006 with 4 hash functions. Hence, BF provides a very flexible way to trade space with performance.

In our context, BFs are used to summarize KA, by building one BF for each flash page of KA. Finding matching tuples can then be achieved by a full scan of the KA summary (denoted SKA in Figure 16.c), followed by a direct access to the KA pages containing a result (or a false

positive with a low probability). Only those KA pages are finally scanned, thereby saving many useless IOs in case of selective predicates. For low selectivity predicates, this index, called *Summary Scan*, becomes less efficient since it qualifies a large number of KA pages.

Given the sequential structure of a cumulative index, Summary Scan can still be optimized by chaining index entries sharing the same key value. The chains are stored in a SWS called PTR (see Figure 16.d). To cope with the SWS constraints, the chaining must be backward, i.e., the occurrence n+1 of a given key value points to occurrence n. SKA and KA are then used only for searching the last occurrence of the searched key (see Figure 16.d). If a key value appears n time in the index, in average, 1/n of SKA and 1 page of KA will be scanned. Then, the pointer chain is followed to find all matching tuples' row identifiers. While this index, called *Summary Skip* is efficient for lookups, however, maintaining such pointer chain can be very costly when keys have few occurrences. It leads, in the worst case to a full scan of SKA to find the previous occurrence of the inserted key.



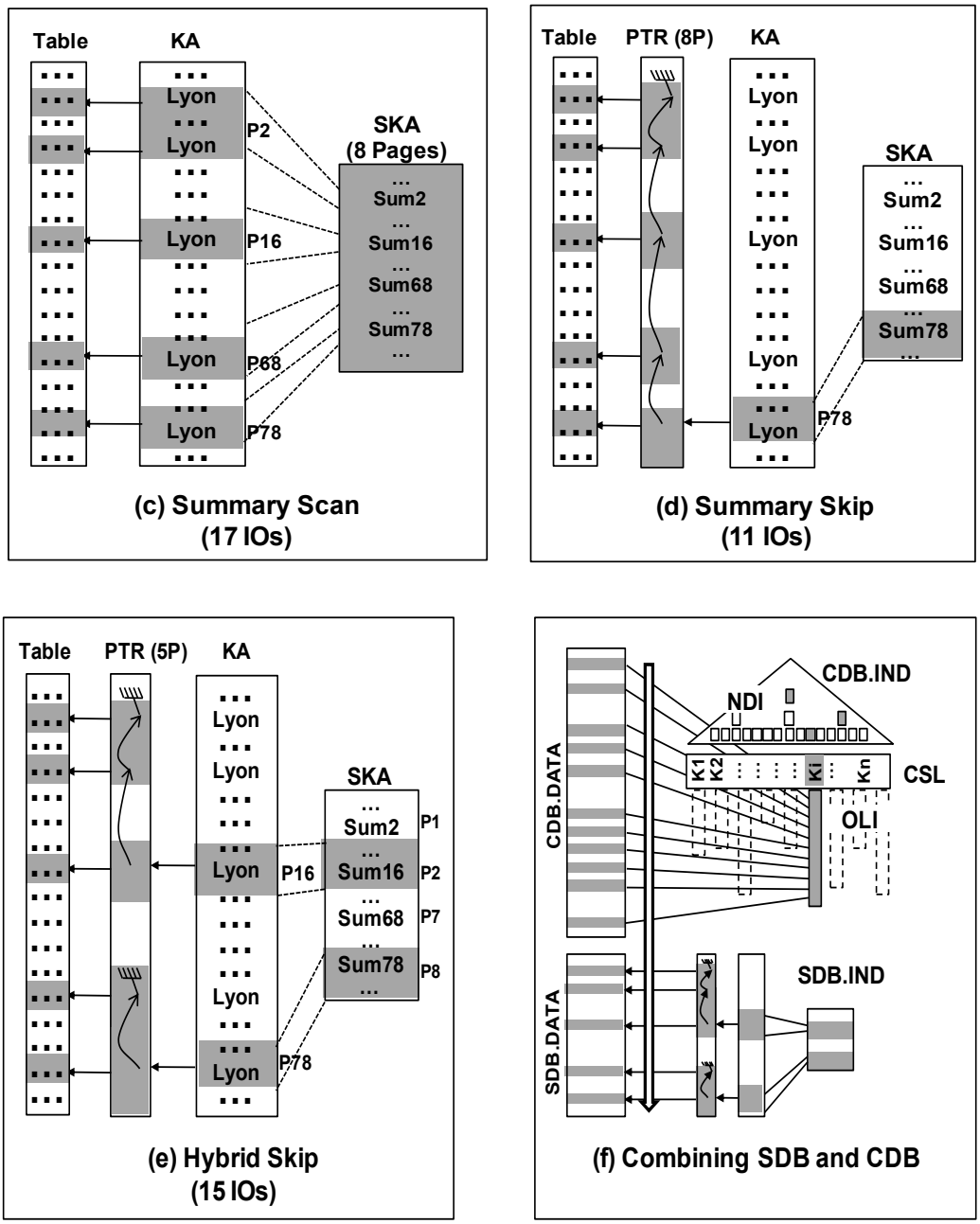


Figure 16. Cumulative and clustered index

(in all figures, grey parts are the ones that are effectively accessed)

In order to bound the insertion cost, a solution is to specify the maximal number of SKA pages (e.g. *n* pages), that should be scanned before stopping. If the previous occurrence of the key is not found once this limit is reached, the new key is inserted with a preceding pointer set to NULL, thereby breaking the pointer chain. At query time, when a NULL pointer is found, the

algorithm switches back to SKA, skipping the *n* next pages of SKA, and continues searching the preceding key occurrence according to the Summary Scan strategy. Indeed, by construction, the KA pages summarized by these n pages of SKA cannot contain the searched value (otherwise, they would have been chained). We call this strategy *Hybrid Skip* since it mixes *Summary Scan* and *Summary Skip* (See Figure 16.e). This strategy has two positive impacts: (1) at insertion time, contrary to *Summary Skip*, the search cost in SKA is bounded to n pages independently of the index size; (2) the pointer can be encoded on a smaller number of bits (linked to the number of keys summarized by *n* pages of SKA), thereby reducing the access cost of PTR.

Thanks to their sequential structure, all cumulative indexes have the property of producing an ordered list of matching tuples' row identifiers (corresponding to the insertion order of these tuples). This property is essential to combine partial results of several (low selectivity) selections because it allows efficient merges, with no RAM consumption (see Section 4.4).

The construction of clustered indexes at stratification time can take advantage of three properties: (1) clustered indexes are never updated (SWS) and can then rely on more space and time efficient data structures than traditional B-Tree, typically filled at 75% to support updates; (2) all the data items to be indexed are known in advance; (3) the complete RAM can be dedicated to the index construction. However, the constraint is to organize the clustered index in such a way that the lists of row identifiers associated to the index entries are kept sorted on the tuples insertion order. This requirement is mandatory to be able to merge efficiently lists of identifiers issued from multiple indexes or from a range search in one index. Moreover, this ordering allows combining query results in SDB_i (cumulative indexes) and in CDB_i (clustered indexes) by a simple concatenation of the lists of matching tuples (see Figure 16.f). Indeed, CDB_i tuples always precede SDB_i in their insertion orders.

The resulting clustered index structure is as follows. A compact ordered list of row identifiers (OLI in Figure 16.f), is built for each index entry K_i. The set of index entries is represented by a compact sorted list, named CSL. Finally, a compact non-dense index, named NDI, stores the highest key of each page of CSL in a first set of flash pages, itself indexed recursively up to a root page. The index is built from the leaves to the root so that no pointers are required in NDI. Clustered indexes are efficient and highly compact because they are built statically and are never updated.

4.4 Query Processing

This section instantiates the techniques presented above to implement a massive indexing scheme and its associated query execution model which has been proposed earlier [ABB+07], thus satisfying rule R1. Then we explain how to integrate updates and deletes in the query processing.

4.4.1 Query Execution Model

In the following, we consider a tree-based database schema which is also used in the performance section, as shown in Figure 17.a. It represents a medical database where table *Prescription* (*Pre*) is the root of the schema referencing tables *Visit* (*Vis*) and *Drug* (*Dru*), table *Vis* references *Doctor* (*Doc*) and table *Dru* references *Laboratory* (*Lab*) and *ClassOfDrug* (*Cla*).

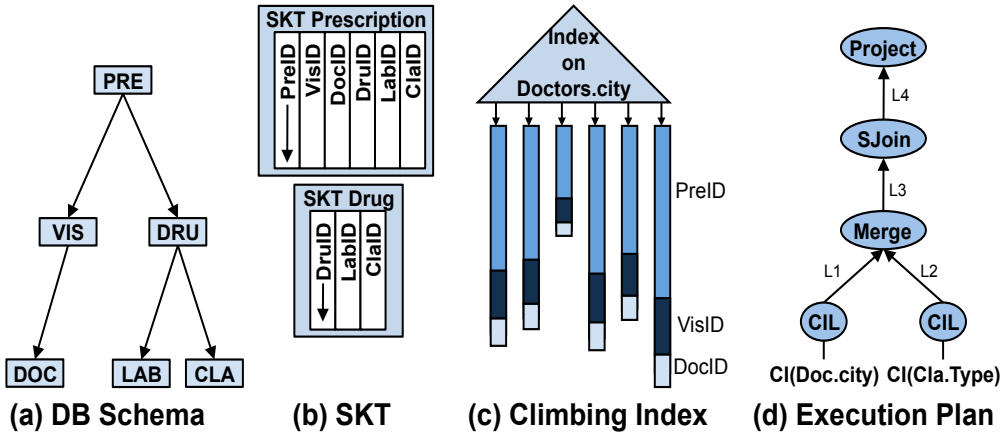


Figure 17. Query processing with SKT and climbing indexes

Generalized join indexes called *Subtree Key Tables* (SKTs) are created for each node tables of the tree-based schema. Each SKT joins all tables in a subtree to the subtree root and stores the result sorted on the identifiers of the root table (SKT entries contain the ids of the matching tuples). For example, the SKT_{Pre} rooted at *Pre* is composed of the joining sets of ids $\{id_{Pre}, id_{Vis}, id_{Dru}, id_{Doc}, id_{Lab}, id_{Cla}\}$ sorted on id_{Pre}. This enables a query to associate a prescription directly with, e.g., the doctor by whom it was prescribed. Selection indexes called *Climbing Indexes* (CI) are created on all attributes involved in selection predicates. A CI created on attribute A of table T maps A values to lists of identifiers of T, as well as lists of identifiers of each table T' ancestor of T in the tree-based schema (Figure 17.a). For example, in $CI_{Doc.City}$, the value *'Lyon'* is mapped to lists of id_{Doc}, id_{Vis}, and id_{Pre} identifiers. Combined together, SKTs and CIs allow

selecting tuples in any table, reaching any other table in the path from this table to the root table in a single step and projecting attributes from any other table of the tree.

Base tables and SKTs are implemented as simple SWSs. A CI is implemented by one cumulative/clustered index for each level of the CI up to the root. For example, $CI_{Doc.City}$ is made of three cumulative/clustered indexes delivering respectively lists of id_{Doc}, id_{Vis}, and id_{Pre} identifiers.

Figure 17.d shows the Query Execution Plan (QEP) of a query which joins all the tables, evaluates the selection predicates on two indexed attributes (*Doc.City='Lyon'* AND *Cla.Type='AZT'*), and projects some attributes (similar to *Multi1* query in performance section, but with meaningful attribute name to ease understanding). The operators required to execute this query are the following:

- $CIL(CI, P, \pi) \rightarrow \{id_T\}\downarrow$ looks up in the climbing index CI and delivers the list of sorted IDs referencing the table selected by π and satisfying a predicate P of the form (attribute θ value) or (attribute ∈{value});
- $Merge(\cap i\{\cup j\{id_T\}\downarrow\}) \rightarrow \{id_T\}\downarrow$ performs the unions and intersections of a collection of sorted lists of identifiers of the same table T translating a logical expression over T expressed in conjunctive normal form;
- $SJoin(\{id_T\}, SKT_T, \pi) \rightarrow \{< id_T, id_{Ti}, id_{Tj} \ldots >\}\downarrow$ performs a key semi-join between a list of identifiers of a table T and SKT_T, and projects the result on the subset of SKT_T attributes selected by π; this result is sorted on id_T; Conceptually, this operation implements a Join but its cost sums up to read the right SKT entries.
- $Project(\{< id_T, id_{Ti}, id_{Tj} \ldots >\}, \pi) \rightarrow \{< Att_i, Att_j, Att_k \ldots >\}$ follows tuples identifiers and retrieves attributes selected by π; the attribute values are buffered in RAM in a hierarchical cache keeping most frequent values to avoid Flash IOs.

The query can be executed as follows in a pipeline fashion:

```
Project (L4, <Doc.Name, Dru.Name, Pre.Qty> → Result
 SJoin (L3, SKTPre, <idPre, idDru, iddoc>) → L4
  Merge (L1∩L2}) → L3
   CIL (Cla.Type, ='AZT', Pre) → L2
   CIL (Doc.City, ='Lyon', Pre) → L1
```

The RAM consumption for this query is limited to one Flash page per CI, the rest of the RAM being used for projection (cache).

4.4.2 Updates/Deletes Integration

As mentioned before, updates and deletes are logged in dedicated SWSs, named UPD and DEL, rather than being executed in place. This virtualization implies compensating queries at execution time, by combining DATA, IND and BUF with UPD and DEL to compute the correct result. In the following, we present techniques used to integrate updates and deletes for single predicate query, but generalize these techniques to multi-predicate queries is straightforward.

Regarding global query processing presented in previous section, only operators *CIL* and *Project* are impacted by the integration of updates and deletes (see below).

- ***Deletes Integration***

Regarding deletes, each result tuple which has been recorded in DEL must be withdrawn from the query result. To perform this check efficiently, a dedicated structure DEL_RAM is built in RAM to avoid accessing DEL on Flash for each result tuple.

To limit RAM occupancy, only the identifiers of the deleted tuples, excluding cascading deletes, are stored in DEL_RAM. Indeed, in a tree based DB schema, deleting a tuple in a leaf table (e.g., one tuple *d* in *Doctor*) may incur cascading the deletes up to the root table (e.g., all *Visit* and *Prescription* tuples linked to *d*). Only d identifier is recorded in this case.

At query execution time, the *SJoin* operator accesses the SKT of the root table of the query, to get the identifiers of all (node table) tuples used to form the tuples of the query result. At that time, each of these identifiers is probed with DEL_RAM and the tuple is discarded in the positive case, without inducing any additional I/O. In above example, if a query applies to *Prescription* and selects a prescription *p* performed by the deleted doctor *d*, *SJoin* will returns *d* identifier from the *Prescription* SKT, *d* identifier will be positively probed with DEL_RAM and *p* will be discarded.

- ***Updates Integration***

Regarding updates, old and new attribute values are logged in UPD for each updated tuple. For the sake of conciseness, we ignore the case of successive updates of the same attribute of the same record in this thesis, but the proposed techniques can be easily adapted. To compensate the query, the query processor must (1) for each projected attribute, check its presence in UPD and get its up-to-date value in the positive case (2) compensate index accesses to eliminate false positives, i.e., tuples returned by the index based on their old value (in BUF, IND) but which should be discarded based on their new value (in UPD), and (3) compensate index accesses to integrate false negatives, i.e., tuples matching the query based on their new value but not returned by the indexes based on their old value. Those three steps are detailed below.

- *Projection*. Similarly to the delete case, a structure UPD_RAM is maintained in RAM to speedup the membership test in UPD. UPD_RAM stores the addresses of modified values in DATA and is rebuilt at each session by scanning UPD. Each attribute (address) to be projected is probed with UPD_RAM, and only when the probe is positive, UPD is accessed on Flash. In addition, UPD is indexed on the attribute addresses in a way similar to a DATA SWS (i.e. using BF), thereby drastically reducing the overhead caused by update processing at project time.
- *Selections – removing false positives*. False positives are elements of UPD which match the query predicate based on their old value and does not match it anymore based *on* their new value. The set of tuples (identifiers) matching this criteria is extracted from UPD and stored in a RAM structure called FP_RAM. FP_RAM is used to probe each output of the index scan. For example, to get the doctors living in '*Paris'*, excluding false positives, using the index *Doc.City*, we first retrieve from UPD the doctors who left '*Paris'* (since the last stratification), store their IDs in FP_RAM, then probe each result of the index with FP_RAM and discard positive ones.

  Tackling climbing index accesses is trickier. For example, for a climbing index access on *Doc.City* at level *Pre* (i.e, the prescriptions of doctors living in '*Paris'*) the result is a list of prescriptions, while UPD provides doctors identifiers. To address this issue, we propose such solution: For each element of FP_RAM (doctors) the corresponding prescriptions are found, e.g., by a sub-query into the climbing index on doctors' IDs, since the number of updated values is likely to be much lower than the number of results returned by the index.

- *Selections – Integrating false negatives.* False negatives are elements of UPD which does not match the query predicate based on their old value and match it based on their new

value. The set of tuples identifiers matching this criterion is extracted from UPD and must be merged with the index output. For example, to get the doctors living in '*Paris'*, including false negatives, using the index on *Doc.City*, we first retrieve from UPD the doctors who moved to '*Paris'* (since the last stratification) and merge them with the results.

For a climbing index access, we use the same technique presented above. For example, for a climbing index access on *Doc.City* at level *Pre* (i.e, the prescriptions of doctors living in '*Paris'*), each doctor tuple in UPD is linked to the (sorted) set of its prescriptions which have simply to be merged with the index output. Using this technique, the integration of false negatives into the index output generates a very small overhead, given that UPD is also indexed on the new attribute value (speeding up the retrieval of false negative identifiers).

To conclude about RAM consumption, three types of structures are needed: DEL_RAM stores the deleted tuples identifiers, UPD_RAM stores the addresses of updated attributes, and FP_RAM stores the identifiers of modified tuples for an index access (its size is negligible compared with DEL_RAM and UPD_RAM). In the experiments conducted in Chapter 6, the RAM consumption corresponding to the 3000 deletes and 3000 updates was about 18KB.

5 Conclusion

In this chapter, we have introduced PDS approach, the motivation examples, the PDS global architecture, and we have detailed its main elements and functionalities. We have also described the main challenges to implement the approach. One of them is implementing a PDS database engine coping with the intrinsic constraints of the hardware platform (the SPT). Two design rules derived from those constraints (namely RAM limitation and NAND Flash constraints) have driven the conception of the embedded PDS database engine.

In order to protect personal data stored within the PDS or externalized on supporting servers, we have to devise adequate protection schemes relying on cryptography techniques. This is a main contribution of this manuscript, presented in the next chapter. Note that the security expectations will be reached without modifying the IO patterns and execution model which are already optimized according to the constraints of the platform.

Chapter 5
Cryptography Protection in PDS

1 Introduction

As a radical different way to manage personal data, the PDS approach provides the functionalities rendered by a traditional DBMS server by relying on a secure hardware component called SPT. The level of trust which can be put in the PDS comes from the following factors:

1) The PDS software inherits the tamper resistance of the SPT making hardware attacks highly difficult.
2) The basic software including operating system, database engine and PDS generic tools, can be certified according to the Common Criteria (all declarative rules are ratified as well), making software attacks also highly difficult.
3) The PDS basic software can be made auto-administered thanks to its simplicity, in contrast to its traditional multi-user server counterpart. Hence, DBA attacks are also precluded.
4) Compared to a traditional server, the ratio Cost/Benefit of an attack is increased by observations 1) and 2) and by the fact that a successful attack compromises only the data of a single individual.
5) Even the PDS holder cannot directly access the data stored locally. After authentication (e.g., by a pin code), she only gets the data according to her privileges thanks to enforced user's control rules.

Unfortunately, the security perimeter provided by the SPT is limited. As stated in Section 3.1 of Chapter 4, the external massive stable storage (i.e., NAND Flash) is beyond the protection scope of the SPT microcontroller, hence Flash resident data suffer from confidentiality and integrity attacks. Integrity attacks make sense because the PDS holder herself can try to tamper the database (e.g., she could perform a replay attack to be refunded several times for the same drug, or try to change an access control rule or the content of an administrative document, e.g., a diploma). More formally, we consider four kinds of threats in our context:

- *Data Snooping*. An attacker examines (encrypted) data to deduce some unauthorized information
- *Data Altering*. An attacker deletes or modifies (even randomly) some data

- *Data Substituting.* An attacker replaces valid data with another valid data
- *Data Replaying*. An attacker replaces valid data with its older version

In addition, personal data also have to be externalized on supporting servers for availability, durability and global processing purpose. Regarding supporting servers, we assume they are Honest but Curious, a common assumption regarding Storage Service Providers. This means that they correctly provide the services that are expected from them (typically serve store, retrieve, and delete data requests) but they may try to breach confidentiality of any data that is stored locally. Therefore, care must be taken to ensure that the traditional functions of a central server (durability, availability, global queries) can be implemented using supporting servers in a secure manner, that is without violating confidentiality and integrity properties of externalized data and without revealing the participant PDSs' identities (e.g., publishers or subscribers participating into a data exchange).

To address the above mentioned security issues, we investigate some protection schemes, by resorting to cryptography techniques. The main objectives are: (a) enforcing data confidentiality (for Flash resident data and data externalized on supporting servers), i.e., preventing any information disclosure, (b) detecting any tampering (e.g., substituting, altering, replaying) on the data, (c) enabling supporting servers to serve users requests (e.g., store, retrieve and delete data) without any information leakage about externalized data and PDSs' identities. To remain efficient, the proposed crypto-protection schemes should not impact data organization (storage and index structure) nor query execution strategy, which have been designed to minimize the number of IOs in NAND Flash (and thus the query response time, since IO cost is dominant).

The PDS context poses many challenges to the crypto-protection design and enforcement. To cite a few: (a) scarce secure non-volatile storage is available on chip (e.g., hundreds of Kilobytes for code and metadata) which challenges the management of versions numbers (required when addressing replay attacks); (b) the PDS engine relies on a massive indexing scheme, leading to many fine granularity data access, which incurs a high cryptography overhead with traditional integrity methods (which are adapted to protect coarse grain text messages); (c) the limited computing capabilities offered by the SPT requires efficient design and enforcement of cryptographic techniques; (d) supporting servers are not considered as trusted, but as Honest-but-Curious, and may threaten the anonymity of the participating PDSs; and (e) the cryptography module of existing SPTs have limited cryptographic capabilities which constrains the choice of cryptography techniques to be used in practice.

In section 2, we focus on the crypto-protection of the data stored within the SPT (i.e., in the external NAND Flash). To make the explanation illustrative and concise, we first select some representative data structures used in the PDS engine and introduce their characteristics. Next, we propose a set of crypto-protection building blocks to counter the potential attacks mentioned above, adapted to the hardware constraints of the SPT. Finally, we illustrate explicitly how to enforce proposed building blocks on the selected data structures. In Section 3, we give a preliminary design of the main communication protocols between the PDSs and the supporting servers, and show the feasibility of PDS approach in terms of global architecture.

2 Crypto-Protection for Embedded Data

2.1 Data Structures and Access Patterns

We present here the main data structures designed for the PDS engine, which are considered as indispensable components for efficient on board query processing. Storage structures are made of base tables, join indices (called *Sub-tree Key Tables* or SKTs) and selections indexes (see Chapter 4). Selections indexes are of two kinds: *Cumulative Indexes* and *Clustered Indexes*. We present here representatives of each data structure and corresponding access patterns. The study of their characteristics is important to derive the adequate security countermeasures (i.e., crypto-protection building blocks).

- **Cumulative Index - Hybrid Skip**

We choose Hybrid Skip as the representative of cumulative indexes, used for indexing Sequential Database (SDB for short). This is reasonable because (i) it is a combination of all the data structures that we have proposed as forms of cumulative indexes (see Chapter 4, Section 4.3) and (ii) it performs well on both insertion and query aspects.

As shown in Figure 16, Hybrid Skip index includes three structures: KA, SKA and PTR. KA stores the replicated indexed attribute values or their hash values (i.e., key). SKA is a summary of KA using Bloom Filters [Blo70]: one Bloom Filter (BF) is built for each KA page. SKA enables membership test before searching KA, which leads to access only the KA pages which, with very high probability, do contain expected results. PTR stores the chains which link index entries sharing the same key value together. In Hybrid Skip, the pointer in the chain is encoded in a small number of bits depending on the number of keys summarized by n pages of SKA (n is the maximal number of SKA pages that should be scanned before stopping in Hybrid Skip). For

example, if n = 2, the pointer size is 11 bits. Moreover, the chains is backward due to SWS definition, and is broken if the n next pages of SKA (in reverse order, as chaining is backward) do not contain the expected value. The end of chain value is noted NULL.

During query execution, SKA is accessed in reversed order, and each BF is tested (sequentially) until finding a positive one. At that time, the corresponding KA page is accessed. It is scanned until the last occurrence of the searched key in the page is reached (the keys are placed sequentially at insertion inside the page). If no occurrence is found (i.e., false positive cases), the execution process goes back to SKA and continues searching (by testing BFs sequentially) in the reverse order. On the other hand, if the last occurrence is found in KA page, according to its location, corresponding PTR page is loaded to get the tail of pointer chain. Furthermore, more PTR pages are loaded to obtain all row identifiers of tuples sharing the searched value by traversing the chain (row identifiers are deduced from the position of the pointers in the chain). Once a NULL pointer is reached, the execution process switches to SKA, skips next n pages and continues searching (in reverse order as well) as described in previous steps. Finally, we could obtain a complete ordered list of matching tuples' row identifiers in Sequential Database (i.e., SDB).

- **Clustered Index**

For clustered indexes, we have proposed a sort-based structure made of three components: OLI, CSL and NDI (see Section 4.3, Chapter 4). All row identifiers of tuples sharing the same value are clustered together based on their insertion order and constitute a list named OLI (*Ordered List of Identifiers*). The key value (i.e., index entry) for each OLI is stored in a sorted list of key values. Since this sorted list of keys is compact (no free spaces are required on the contrary of B-Tree leaves) it is called CSL (*Compact Sorted List*). In order to speedup lookups in this list, we index it with a Tree-like structure forming a *Non-Dense Index* (at current level, we only store the highest key for each page at inferior level) named NDI. Different from B-Tree, (i) CSL and NDI pages are fully filled since there is no updates/inserts/deletes in-place, thus no need to allocate space to alleviate the overhead of tree adjustment incurred by such changes; (ii) since NDI is static and built from the leaves to the root in recursive way, there is no need to store pointers in NDI to locate its children.

During the lookup process of equality query, we use binary search algorithm into NDI pages (keys are sorted) and deduce the child NDI page given the position of the key resulting from

that binary search. The process is repeated until reaching the CSL level, like traversing the nodes of a B-Tree. Binary search is used as well in the target CSL page, and once reaching the searched key, OLI is accessed to obtain the ordered list of matching tuples' row identifiers in Clustered Database (CDB for short). For the inequality query (e.g., range), the lookup process is similar but all CSL pages containing key values that qualifies the inequality predicates are accessed, and the corresponding (maybe several) OLI are retrieved as well to obtain the row identifiers of resulting tuples (for equality query, a single OLI is accessed only).

- **Join Index - SKT**

A SKT acts as a join index. It joins all the tables in a subtree to the subtree root, each index entry containing the row identifiers for the matching tuples in subtree tables (see chapter 4). The entries in a SKT are naturally sorted on the row identifiers of the subtree root table.

If a query requires projecting the data values from tables in the subtree, corresponding SKT entries will be accessed according to the row identifiers of root table obtained from traversing cumulative and clustered indexes as mentioned above. Such access is often considered as random since the row identifiers obtained from indexes, although sorted, are not clustered in same SKT pages (note: full scan SKT also makes sense, but it is not a common case). In addition, the query processor only retrieves useful identifiers, required for projecting the tuples from a SKT entry instead of the whole part of the entry.

- **Base Table**

Tables could be stored using the Row Storage scheme or Column Storage one. In our context, we used the Row Storage one but this has little impact on cryptographic protection. Indeed, (1) only tuples from the query result are accessed, based on the row identifiers obtained from SKT entries, or partially sequentially using the row identifiers obtained from a selection index (indexing the root table); (2) only the subset of columns (required for projection) are accessed. Thus, similar to the SKT, the access is partial, i.e., only retrieve the data of interest.

2.2 Crypto-Protection Building Blocks

In this section, we introduce the designs of crypto-protection building blocks, considering the limited resource and the hardware constraints (e.g., NAND Flash particularities). The building blocks include smarter version management strategy by taking advantage of database

serialization and stratification paradigms, authenticated encryption for fine granularity data generated by massive indexing scheme, smart selection scheme which enables searching directly on encrypted data, data placement strategy which allows decrypting only the data-of-interest during processing.

Note that the proposed building blocks do not necessarily rely on cryptography techniques (e.g., data placement), but all of them aim at constructing an efficient and secure crypto-protection scheme for embedded PDS engine and can be potentially combined depending on the data structure and its access patterns (see Section 2.3).

2.2.1 Version Management

Fighting against replay attacks imposes to include a version number when computing the cryptographic hash of the protected plaintext, or when encrypting the plaintext. We have to securely store this version number, and maintain the association between each data item and its version number. At the time of checking integrity (e.g., check version), we obtain the good version from the secure store and compare it with the one stored along with the data.

The main difficulties related to version management are the choice of the granularity at which a structure must be protected (i.e., the number of data items protected using a same version number) and poor update support on NAND Flash (in the cases of version update). Indeed, if a version number is shared by few data items (e.g., each data item has its own version), then a huge amount of version numbers have to be maintained. Storing these versions within the secure memory (e.g., internal memory like NOR) becomes impossible due to its limited size. Maintaining a hierarchy of versions in the external storage and only storing its root in secure memory, in the spirit of Merkle Hash Trees [Mer89] and TEC Tree [ECL+07], generates fine-grain random writes for each level of hierarchy, thus disturbs NAND Flash optimal write patterns (i.e., sequential write) [BJB+09]. Moreover, such tree structures bring side effects on query processing. As we have to traverse all tree levels from the leaf up to the root when checking the version for each data item, access latency and extra crypto overhead (e.g., decrypt Flash pages) are generated.

In our context, we can take advantage of the serialization and stratification principles adopted by PDS engine and design a smarter version management strategy. More precisely, the following specificities exhibited by the PDS engine can be exploited: (i) in all SWSs, obsolete

blocks can only be reclaimed during stratification, and they can only be reallocated to later strata. Hence when stratification starts, all SWSs of the current stratum are frozen; and (ii) no updates in-place exist, but updates are logged in a dedicated SWS called UPD and processed using specific algorithms during query execution.

The above features enable a simple version management strategy where: (i) the whole stratum remaining valid or becoming obsolete altogether, the stratum number (allocated incrementally and kept unique) stands for the version number, which means maintaining versions at stratum granularity (only that number is stored in secure memory) thus avoiding the storage problem; (ii) data are never updated in-place within one stratum thanks to UPD, but are updated only after stratification (data with different versions locate in different strata) thus avoiding the problem of version update on NAND Flash.

Based on such a version management strategy, we have to store the current stratum number in the secure memory. At the time of query processing, in order to check if the data items are used in the correct stratum (i.e. check version) or not, we have to obtain the stratum number from secure memory. Note that accessing secure memory is direct and efficient (typically, secure memory is NOR with access times comparable to RAM), making the proposed version management strategy highly efficient.

2.2.2 Granularity of encryption and integrity checking

In our context, personal data stored on NAND Flash suffer from snooping, substituting, altering and replaying attacks, thus adequate cryptographic techniques have to be used to fight against such attacks. Typically, snooping attacks could be precluded thanks to encryption, while illicit integrity violations (e.g., substituting, altering, replaying attacks) are traditionally detected by computing a Message Authentication Code (MAC), or using hash value [MOV97], or use authenticated encryption algorithms to encrypt and authenticate data simultaneously. At the time of checking integrity, we usually regenerate the MAC (or hash value) and compare it with the one stored with the data. If they match, the integrity of data item is not violated. Otherwise, the data item has been tampered.

However, SPT only provides limited cryptography capabilities and disfavors some traditional methods. Indeed, only AES algorithm is available on SPT, hash function is implemented in software and is extremely slow. According to their performance (both in hardware and in

software) reported in [NES03], such limitation will remain valid in the foreseeable future. As a result, some efficient methods such as authenticated encryption algorithms (e.g., OCB, EAX, CCM etc) are precluded and the options of cryptographic techniques are constrained as well.

Moreover, the granularity of traditional encryption and hashing algorithms (e.g., 128 bits for AES and 512 bits for SHA family) is not compliant with the database granularity (e.g., in the order of pointer size, such as 3 bytes) since the engine relies on a massive indexing scheme, hence using traditional methods will cause prohibitive cryptographic cost. For instance, the potential way to encrypt and authenticate data in our platform is using MAC method based on a block cipher (AES encryption can be done efficiently in hardware while software-based cryptographic hash computation is extremely slow) with the granularity of MAC depending on the block size of underlying block cipher (e.g., 16 bytes in our case). For very small data item (e.g. a pointer encoded in 3 bytes within the PTR structure), if we use MAC method and encrypt-then-MAC composition way (any composition methods have similar cost since they all have encryption and MAC two steps), first we have to pad the data to reach one block size before encrypting, then perform MAC algorithm on encrypted data. Therefore, it requires two passes of cryptographic processing and incurs significant crypto overhead.

In addition, MAC size is not negligible even after truncation (e.g., 10 bytes). According to [MOV97], smaller MAC size will lead to security weakness (i.e., collision problems). In order to reduce MAC storage space, several data items could share the same MAC (e.g., all the pointers inside one page share one MAC). However, the drawback of this method is that it incurs significant crypto cost at the time of integrity checking. For instance, when we only retrieve one pointer randomly (e.g., following inverted list) and check its integrity, we have to perform MAC computation on the whole page accessed, which is clearly sub-optimal.

Based on above statement, the problem of building MAC mainly comes from two aspects: (i) Storage expansion. For small granularity data, the storage expands significantly after encryption due to padding. Moreover, extra MAC storage is required. (ii) Two passes of cryptographic operation. MAC based approach needs two passes of processing regardless of data size, one for encryption and one for authentication. Hence the cryptographic overhead increases approximately by a factor of 2, which is significant in an embedded environment. Due to these reasons, we investigate a more efficient method that could solve or avoid such problems.

In literature [ECL+07], Elbaz et al. propose an authenticated encryption method based on block cipher, which provides confidentiality and integrity guarantees simultaneously and named *Block-Level Added Redundancy Explicit Authentication* (AREA). This method only requires one pass of processing and provides integrity guarantees at low cost, operates at the granularity of traditional block cipher and does not involve complex algorithms. It is, thus quite suitable to encrypt and authenticate data in our context. This method works as follows: a Nonce (Number used only once) is concatenated to small granularity plaintext data (e.g., 4 bytes), and then is padded potentially in order to obtain a complete block (e.g. 16 bytes) before encryption. After decryption, the Nonce obtained should match its initial value. Otherwise, at least one bit in the data item (including Nonce or padding bits) has been modified or tampered.

The security of this construction is based on (i) the diffusion property of the block cipher, (ii) the uniqueness of the Nonce, (iii) the Nonce size. Property (i) means that one bit-flip in the input block will cause change for each bit of the output blocks with reasonable probability (e.g., 50%). Property (ii) makes sure that data items with same value will be padded with different Nonce values and be encrypted into different ciphertext to resist to statistical attacks. Property (iii) concerns the probability of a successful attack. Indeed, for a block with n bits, the Nonce size being m bits, the chance of an attacker to modify the block without being detected (i.e., the Nonce remaining valid) is $1/2^m$. Therefore, the larger the Nonce size, the more difficult it is to breach the security. The side effect is that the size of the payload becomes smaller (as stated below).

To fight against *Data Substituting* and *Data Replaying* attacks, the Nonce value should contain the absolute address and the version number of the data items. This information does not require being kept secret to guarantee integrity (with reasonable Nonce size). Consequently, AREA can be used to authenticate large quantity of data (e.g., several GBs) without generating secure storage overheads. This point is very important in our environment where the amount of secure memory is limited.

Compared to traditional Encrypt-then-MAC methods, the advantages of using AREA are: (i) it operates at fine granularity (e.g., several bytes), which is suitable to our massive indexed PDS engine; and (ii) it offers integrity checking at low cost. Indeed, we just have to compare the decrypted Nonce value with the good one to detect tampering, without performing any extra cryptographic operation. Compared with traditional MAC method which requires two passes

over the data, AREA is still thus a preferable option (in terms of cryptographic cost) for any size data and whatever be the access pattern (sequential or random).

However, AREA has an intrinsic drawback: compared with traditional MAC methods, it has larger storage expansion. For instance, for *x* bytes data (assuming *x* is multiple of the block size, i.e., the block size of the cipher, like 16 bytes), using AREA would require *x*(m/n)* bytes (block size being *n* bits, and the Nonce size being *m* bits) for storing the Nonce. The storage expansion is thus proportional to the size of the plaintext. With MAC method, only a fixed size space (e.g., 16 bytes) is required for storing MAC regardless of the plaintext size. In practice, we have to take the storage expansion factor into consideration as well (including other factors such as introduced cryptographic overhead, implementation feasibility and complexity etc), and choose adequate solutions (to ensure confidentiality and integrity guarantees) for the data to be protected.

2.2.3 Searching on Encrypted Data

According to the storage and indexing model of PDS engine (presented in Chapter 4), at query time the execution engine often requires searching within a set of values those values matching certain conditions (e.g., when accessing the KA built on city attribute to find out row identifiers of tuples with value 'Lyon'). If we encrypt these values in traditional way (e.g., assuming using the CBC mode), identical values are encrypted into different ciphertexts to resist to statistical attacks. Each value requires then to be decrypted to identify the qualified values, inducing many (costly) decryption operations. To conduct the search more efficiently, we have to enable searching over encrypted data without decryption.

The state-of-the-art works done in the database community have opened some avenues in this direction (see section 2, Chapter 3). Hacigümüs et al. [HIL+02b] propose exploiting bucketization indices to compute queries over encrypted database in DAS model, and many other following studies have been made to design indices that have a good trade-off between privacy and efficiency [DDJ+03, HMT04]. Another famous avenue to enable searching on encrypted data using exotic encryption algorithms, such as Order-Preserving Encryption [AKS+04], Prefix-Preserving Encryption [LiO05], Privacy Homomorphism (PH) [OSC03, HIM04, EvG07, GeZ07a].

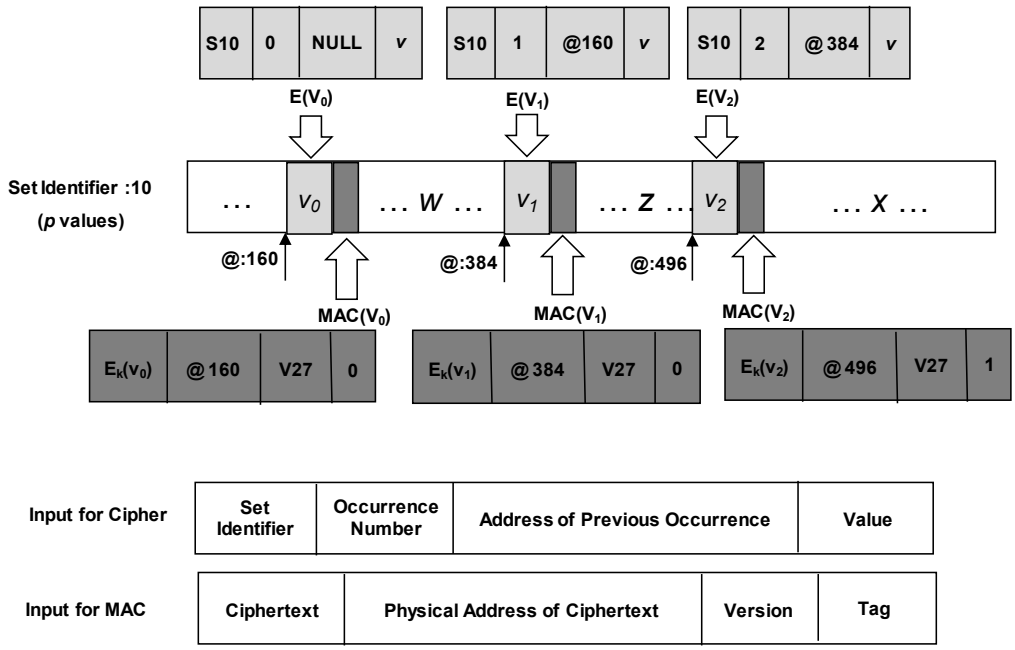


Figure 18. Smart Selection Scheme

Unfortunately, the state-of-the-art techniques are not able to solve our problem due to the following reasons: (i) the computing environment in our context is trusted, thus we only need to do some evaluation (i.e., selection) on ciphertext instead of computing whole query on them, hence the complexity of the problem is eased. (ii) The results returned by the searching process should be exact and accurate (i.e., no false positives). As the searched results have to be joined and projected in the following processing, thus accurate selection improves query efficiency (without wasting time on processing non-resulting tuples). (iii) Limited computing capabilities do not provide support for using exotic encryption algorithms (i.e., only AES algorithm is available on SPT).

In this manuscript, we propose a new method called *Smart Selection* to evaluate equality predicates without decryption (and without security breach). The idea is to encrypt each key according to its value and the number of occurrences of that key that are already present (in the set). At query time, the next encrypted text to search for can be deduced from the key value of the search and the current number of matching results. In the following, we will present in details how encryption and integrity are performed while enabling *Smart Selection*.

- **Encryption**

As shown in Figure 18, to speedup the retrieval of *n* occurrences of a given value *v* within a set of *p* values, the successive occurrences of the same value are concatenated with additional information before encryption (note, to avoid the same values producing several identical cipher blocks, the values are concatenated after additional information). This information makes each occurrence unique thus precluding that the same values generate the same ciphertext, thus resisting to statistical attacks. Indeed, such information includes: (i) *Set Identifier*. This field intends to preclude statistical attacks among different sets because other fields are local and are within the scope of a set (e.g., assuming one KA page is a set, without set identifier, the first occurrence of value '*Lyon*' in different KA pages would lead to the same ciphertext). (ii) *Occurrence Number*. This field indicates the occurrence order of value in a set. It starts from 0 and is incremented. The objective of this field is to make each occurrence distinct and to prevent integrity attacks such as substitution. (iii) *Address of Previous Occurrence*. This field indicates the address of the previous occurrence in the set. Through this field, it chains all occurrences of value *v* in reverse order (called address chain hereafter), and provides benefits for integrity checking (see the following). For the first occurrence, since it has no previous occurrence with values *v,* this field is set to NULL.

- **Building MAC**

In the previous section, we have introduced AREA, an efficient authenticated encryption method that provides integrity guarantee at low cost. Unfortunately, we can not combine AREA with smart selection since they are conflicting by nature. Indeed, using AREA requires embedding the absolute address (e.g. physical address) in Nonce to resist to substituting and altering attacks. However, if we embed this information in the encryption process of the smart selection scheme, we cannot anymore search on encrypted data (see next part). Consequently, we have no better solution than building MAC for encrypted values.

In order to reduce the cost for (re)building and checking the MAC, we build the MAC as follows: (i) we use the Encrypt-then-MAC composition way. According to [BeN00], this composition method is secure and efficient. Indeed, we can check MAC for data items and verify their integrity without decrypting the data. (ii) In order to fight against altering, substituting and replaying attacks, we integrate the absolute address (e.g., physical address) and the version number in the MAC building process. This also brings extra benefits. Indeed, if

version or address information have been updated (e.g., moving one data item to elsewhere), we only need to rebuild the MAC without decrypting data values, since they (e.g., absolute address, version number) are not encrypted with data values.

Based on the above statement, the input for MAC algorithm includes (i) *ciphertext to authenticate*, (ii) *physical address of ciphertext*, (iii) *version information*. As shown in Figure 18, except the above items, it also includes (iv) *Tag* field, which intends to identify the last occurrence of value *v* in the set. For example, only *Tag* field of the chain tail (e.g., v_n) has been set, while other occurrences have unset *Tag*. This field is used to check the completeness of results. For instance, if we have found the chain tail and prove its correctness by checking MAC (see next part), we can infer that there is no other occurrence of value *v* after this tail in the set. Obviously, this objective could also be achieved by adding *Tag* as input for encryption instead of MAC process. Since both ways lead to the same cryptographic cost, thus we only consider the former approach in this manuscript.

- **Searching and Integrity Checking**

In this part, we present the search process and explain how to detect tampering by taking advantage of the encryption and MAC techniques mentioned above.

To retrieve the *p* occurrences of a given value *v* within a set of *n* values, the searched value *v* is concatenated before encryption with the *Set Identifier* (e.g., S10), the *Occurrence Number* (initially set to 0) and the *Address of Previous Occurrence* (initially set to NULL) as defined above. The item: *Set Identifier* || *Occurrence Number* || *Address of Previous Occurrence* || *v* is then encrypted. The result of this encryption is then compared with the ciphertext of the *p* values in the set. If a match occurs, it means that the first occurrence has been found. Then, the *Set Identifier*[1] (e.g., S10), the incremented *Occurrence Number* (e.g., set to 1) and the *Address of Previous Occurrence* (e.g., @160 in Figure 18), concatenated with the value *v*, is encrypted again. The result of this second encryption is then compared with the elements of the set after the first occurrence. The same process is repeated until the last element of the list is reached.

At the end of the search, the MAC (with *Tag* set) of the last occurrence is checked to verify its integrity. Checking the MAC of the last occurrence is sufficient to guarantee the completeness and correctness properties (i.e., integrity) of the whole result. For example,

[1] Note that the value of *Set Identifier* is unchanged inside a same set

altering attack is detected, because if any occurrence in the chain has been tampered, the comparison fails at the time of the search and subsequent matching values are omitted; the last value reached by the search is thus not the last one in the chain, and its MAC is generated with the *Tag* unset; the tampering can thus be detected. Besides, substitution or swapping attacks are detected as well since all occurrences are chained and their *Occurrence Numbers* are allocated incrementally. For example, assuming this field does not exist, the completeness properties of the results can be violated without being detected. In the above example, if we swap v_0 and v_{n-1} (the 1st and n-1th occurrence of value *v,* note, the *Occurrence Numbers* are not included), we can find the v_0 (at the position of v_{n-1}) and v_n, while the integrity checking for v_n succeed as well, thus we missed $v_1 \ldots v_{n-1}$ and obtain incomplete results. Therefore, the *Occurrence Number* field is mandatory to detect missing occurrences.

However, there exist some exceptional cases: (1) the first occurrence has been tampered; (2) no occurrence in that set is qualified. In both cases, the search returns an empty result. To detect any potential tampering, the integrity for the whole set must be checked. Exceptional case (1) is scarce by nature, but case (2) may incur a significant overhead depending on the frequency of empty result. When adapting *Smart Selection* to storage and indexation structures, this case must be carefully taken into account (See Section 2.3.2).

In summary, *Smart Selection* scheme has the important advantages of only generating one encryption per qualified result (to locate the last occurrence, an extra encryption is required for searching v_{n+1}) and one MAC checking (in the general case) independently of the size *p* of the set. However, this scheme only supports equality query.

2.2.4 Data Placement

As block cipher (e.g., AES algorithm) operates at a block granularity, it makes sense to cluster data-of-interest (according to the query processor) within same blocks to reduce encryption/decryption overheads.

Previous works on databases encryption take data placement into consideration. The *Partition Attributes Across* (PAX) storage model [ADH02] groups all values of each attribute together inside the page to minimize the number of decryption operations when accessing to a given set of attributes. In the same spirit, [IMM+04] proposes the *Partition Plaintext and Ciphertext* (PPC) storage model to separate sensitive and non-sensitive data within disk pages

(disk page are split into two mini-pages), thus reducing the cryptographic operations (only sensitive data is encrypted). Note that in most contexts, IO is the dominant factor (over cryptographic costs). This leads to propose adaptations of the existing storage and indexing models (optimized for IOs) which enable cryptographic protections at a lower cost, but without impacting the IO patterns and causing storage penalties.

In the PDS context, this will also be the case; the data placement strategy will aim at reducing the amount of cryptographic operations and improve the performance of the query processing, without violating the design rules of the PDS engine (thus without incurring any changes to the IO access patterns). Details are given in the next section.

2.3 Instantiating the building blocks in the PDS case

In this section, we give a description of how the crypto-protection building blocks that we proposed can be transposed to the case of the PDS engine. We first instantiate the building blocks in the specific context of the PDS engine (for both *AREA* and *Smart Selection*). Next, we show how those instances (of *AREA* and *Smart Selection*) can be used on the data structures of PDS (as introduced in Section 2.1).

2.3.1 Instantiating the Crypto-Protection Building Blocks for the PDS

- **AREA for the PDS**

For the implementation of AREA method, the major problem is designing the Nonce value. In the above sections, we have pointed out that Nonce value should contain absolute address (e.g., physical address) and version information (using stratum number in our context) for the data item to be authenticated, as shown in Figure 19. The stratum number takes 2 bytes, hence the number of strata could reach up to 65536, while the physical address takes 4 bytes, thus support 4 GB sized database (to support larger database, we can increase the Nonce size). Such design ensures the uniqueness of Nonce values, indeed, same values will produce different ciphertext after encryption (prevent snooping attack), hence statistical attacks are precluded. Thanks to the Nonce, integrity attacks such as substituting, replaying, altering can be detected as well (see following). Therefore, AREA achieves expected security levels in our context.

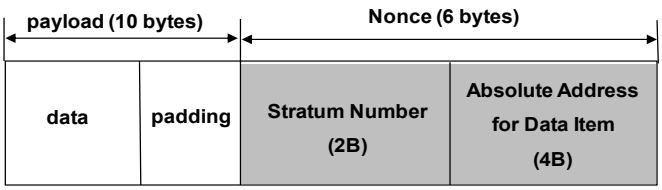


Figure 19. Nonce Design in AREA

At the time of checking integrity, more precisely, (i) we compare retrieved stratum number in Nonce with the current stratum number stored in secure memory (i.e., internal NOR Flash), to check if they are coherent and to detect replay attack, (ii) we check the absolute address (since the physical address is known when accessing the data item) for the targeted data item to detect substituting and altering attacks.

The total size for the Nonce value is 6 bytes. The probability of conducting a successful attack without being detected is $1/2^{48}$, which is considered secure in our context. Using AES, it leads to a payload of 10 bytes only, since the block size for the AES cipher is 16 bytes.

- **Smart Selection for the PDS**

In the PDS context, the choice for the granularity of the set of items protected with *Smart Selection* is the page. Indeed, for any SDB_i part of the database (i.e., the sequential part of the database storing tables, indexes, update and delete information), choosing a multi-page granularity would lead to break the SWS precept. Actually, SDB_i must be filled page by page sequentially; considering a multi-page granularity would require updating previous pages (to modify tag values from 1 to 0 for all items sharing an existing value).

When instantiating the *Smart Selection* scheme at the page granularity (see Figure 20), hence the *Set Identifier* is represented by the absolute address of the page. In Figure 20.a, the counter stands for the occurrence number of the targeted data item, and in Figure 20.b the stratum number acts as a version number.

To build the MAC, we have chosen the CBC-MAC based on the AES algorithm (which is available on current chips). Note that, since CBC-MAC suffers from message concatenation attack, we use its variants called XCBC-MAC [FrH03] in practice. At the time of checking integrity, for the targeted data item, we get the stratum number from the secure memory,

concatenated to the absolute address and tag (set), to regenerate MAC. Then we compare the generated MAC with the one stored with data item retrieved, and verify if its integrity has been violated or not.

As mentioned above, during the lookup process in the *Smart Selection* scheme, we do not decrypt data and only do equality evaluation on ciphertext. When the size of the encrypted value (See Figure 20.a) is small, we can easily use the ciphertext when searching and building MAC. However, if the data is large (e.g., 100 bytes), the process becomes more costly for two reasons: (i) the storage footprint of the ciphertext is larger, i.e., more than 100 bytes after encryption, and (ii) the overhead of MAC algorithm for large data thus leads to double the overall cryptographic cost. In order to solve this, we propose using a cryptographic hash representation of the data as ciphered data (e.g., 16 bytes only) for Smart Selection. Selection becomes then more efficient, but at the cost of proscribing projections. Given the considered data structures in PDS, projection is often only required when running the stratification process. However, the projections steps involved in the stratification process can be performed differently, as exposed later, and without any strong impact on the performance (see Chapter 6), making this optimization acceptable is most cases.

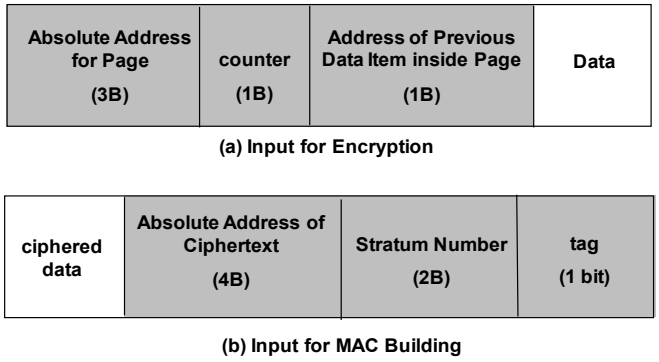


Figure 20. Smart Selection scheme for PDS (page granularity)

2.3.2 Protecting the PDS structures

For conciseness, we only consider the most representative data structures of the PDS, and consider that techniques for the other data structures can be deduced easily. Typically, we only describe the protection of the Hybrid Skip index, since the other indexes exhibit a similar but simpler format.

- **Cumulative Index - Hybrid Skip**

SKA is built based on the BF technology, which allows membership test with low false positive rate, as introduced above. Before discussing its protection, we investigate a better data placement (inside the page) for BFs, considering the following facts. (i) Each BF is a bit array (e.g., *m* bits). We only check few bits (e.g., 3 bits) in the array during the testing. The position and number of bits we need to check are determined by the hash functions used to build BF (e.g., 3 hash functions). (ii) BFs inside the page are retrieved sequentially without skipping. As a result, relevant bits (i.e., bits at the same position) from all BFs inside the page can be grouped, and encrypted together (called cluster, see Figure 21), in order to save decryption cost. During scanning, we only decrypt clusters of interest, hence the number of cryptographic operations is reduced largely (depends on the number of bits to check in the BF array).

It is preferable to use AREA (instead of Smart Selection Scheme) to ensure the security for SKA because (i) AREA provides integrity guarantees at low cost, (ii) the size of a cluster is small (e.g. *k* bits), because the number of BFs that can be stored inside one SKA page is small (e.g., $k = 8$), considering each BF summarizes one KA page and reaches rather large size. (iii) AREA is adequate for fine granularity data. In addition, in order to save crypto overhead further (and space as well), we can group adjacent clusters and encrypt them within one AES block (since each cluster is small). However, we do not allow the case that one cluster crosses two different AES blocks, because accessing such cluster leads to two cryptographic operations.

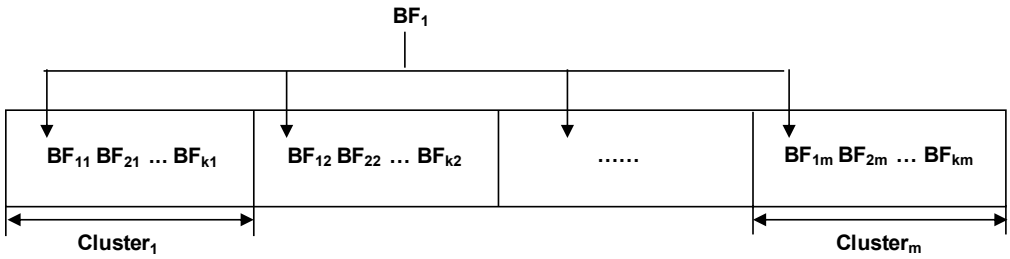


Figure 21. Data Placement for SKA

Thanks to SKA, we only access KA pages potentially containing results. Inside the KA page, we search for all keys sequentially (they are stored based on their insertion orders). Each key is mapped to its corresponding SKT entry, as well as the tuple inside the table. Hence, the data

layout inside the KA page should be kept unchanged, otherwise, the functionalities of cumulative indexes becomes invalid.

To protect each KA page, we use *Smart Selection* scheme, because all keys inside the KA page are accessed sequentially. As mentioned before, in the case of no results inside the page, we have to check the integrity for the whole page, leading to significant cryptographic overhead. In the PDS context, thanks to the BF technology used in the SKA (membership test is done before searching), the probabilities of such cases are reduced to extremely low level (amounts to false positives rate of BF).

Regarding the PTR structure, it is randomly accessed and the size of each pointer is small (e.g., 11 bits). Since AREA suits for fine granularity data very well, while traditional Encrypt-then-MAC method leads to significant storage expansion and checking cost. Hence it is preferable to use AREA to ensure its confidentiality and integrity,

- **Clustered Index**

Since we need to do inequality evaluation on NDI (presented in Section 2.1), it precludes the application of Smart Selection, which only supports equality predicates. In addition, AREA adapts for fine granularity accessing and random accessing, thus NDI is protected using AREA. To speedup searching, we use dichotomy algorithm during searching (since keys are sorted). For the same reason, OLI are protected using AREA as well.

For CSL, each value is distinct and we only need to find the expected one through equality testing, this feature fits well the application scope of Smart Selection scheme. If using AREA, we have to decrypt full CSL page before searching, while using Smart Selection scheme only requires two cryptographic operations (one for encryption and one for MAC computation). As a result, protecting CSL using Smart Selection scheme makes more sense.

- **SKT**

Each SKT serves as a join indexes. It is organized as a table of identifiers. Each column stores the rows identifiers of a given table. We refer to Section 2.1 for more details. Row identifiers are usually small (e.g., 4 bytes). When accessing a SKT, only a subset of the columns is required (i.e., corresponding to projected tables). Usually, the query processor accesses not contiguous rows of a SKT (since rows identifiers are obtained by querying selection indexes).

AREA is suitable for protecting fine granularity data accessed randomly making it a natural choice for SKT protection.

- **Table**

Table in row store scheme has similar access patterns to SKT. Indeed, only subset of the tuple can be retrieved separately and randomly (e.g., for projection purpose). However, different from SKT entry, the tuple may have very large attributes (e.g., 100 bytes). If using AREA, the storage expansion incurred by crypto-protection becomes too important (for both Flash consumption and IO cost at query time). For example, considering the payload is 10 bytes in AREA, for an attribute or a tuple with 100 bytes, it requires 10 AREA blocks to store it, and the total size of encrypted data could reach 160 bytes.

AREA is however the best choice to protect tables because: (i) AREA provides integrity at low cost and only requires one pass of processing on projected data item, while normal methods require two passes of processing (see Section 2.2.2) (ii) Our context is more sensitive to read performance (e.g., projection tables). As each stratum is written only once, never updated and read intensively, the write overhead incurred by AREA becomes acceptable. (iii) Regarding read IOs, the IO cost largely depends on IO itself, instead of data size per IO, thus read performance is not impacted greatly either by the larger storage expansion factor of AREA (except the cases when data item size is rather large, refer to Chapter 6 for more details). (iv) The external NAND storage has no strict limit according to SPT manufacturers (e.g., Gemalto), thus the storage will not become a problem in our context. As a result, it makes sense to trade better performance (i.e., crypto gain) with more storage space. Performance measurements (see Chapter 6) will confirm these qualitative arguments.

In summary, this section introduced the protection schemes for main data structures used in PDS engine (see Table 1), to meet security expectations. In next chapter, we will quantify cryptography impact and show their effectiveness.

Table 1. Data Structures and Cryptography Protection Schemes

Data Structures		Protection Scheme	Reasons
Cumulative Index	SKA	(Data Placement) AREA	AREA suits for fine granularity data (less crypto overhead)
	KA	Smart Selection	sequential equality testing inside KA page
	PTR	AREA	AREA suits for fine granularity and random accessed data
Clustered Index	NDI	AREA	provide support for inequality evaluation
	OLI	AREA	AREA suits for fine granularity data (less crypto overhead)
	CSL	Smart Selection	smart selection incurs less crypto overhead
SKT		AREA	AREA suits for fine granularity and random accessed data
Table		AREA	AREA provides integrity at low cost and incurs less crypto overhead

3 Secure Communication Protocols

3.1 Introduction

In order to implement traditional functionalities of a central server such as durability, availability and global processing, PDS approach has to resort to external supporting servers, which are supposed to serve data store, retrieve and delete requests on an unbounded storage area. However, supporting servers are Honest-but-Curious, i.e., they function correctly, but they may try to breach the confidentiality of externalized data or anonymity of participant PDSs.

Consequently, to achieve expected security goals while satisfying PDS application requirements, any externalized data is required to be encrypted, both on supporting servers and communication channels. Moreover, communication channels between supporting servers and PDSs and between PDSs themselves should use anonymizing network like Tor [DMS04], based on the Onion-routing protocol [GRS99], to make the communication anonymous. In addition, secure communication protocols between supporting servers and PDSs should be devised to ensure supporting servers function correctly, without any knowledge about PDSs' identities.

In this manuscript, we focus on designing adequate secure communication protocols, to show the feasibility of PDS approach. Other issues concerning security are considered as future works (see Chapter 7).

In the following, we first introduce the message format and describe each field inside the message. Next, we give the design of message exchanging protocols. This is not straightforward.

For instance, the primary concern for the feasibility of PDS approach is that recipient PDSs must be able to retrieve messages or data sent to them. Although communications are anonymous, the difficulty lies in selecting the relevant messages or data without disclosing any information that could help the supporting servers to infer PDS identity. To address this issue, we propose a protocol tagging messages with anonymous markers (see Section 3.3).

Finally, we provide protocols for handling data delete requests, as deleting obsolete/invalid data is necessary in PDS functionalities. Such requests are difficult for two reasons: (i) the requesting PDS must exhibit a proof of legitimacy to destroy the data, and (ii) the deletion must be effective, either in the case that an attacker spies and replicates all messages sent to the supporting servers, or in the case that the physical image of the targeted data has been stolen or cannot be physically destroyed.

3.2 Message Format

Table 2 gives detailed explanation about the notations used in the following protocols.

Table 2. List of Symbols used in Protocols

$E_X^{pub}[M]$	Encryption of message M with the public key of entity X.
$E_k[M]$	Encryption of message M with secret key k
$M_1 \| M_2$	Concatenation of messages M_1 and M_2
$H[M]$	Cryptographic hash of message M
$rand_k()$	A pseudorandom number generator using a secret key k, specific to each actor (PDSs or Content Providers)
$ID(X)$	Publicly known identifier of entity X
TS	Timestamp generated by a time server
$\boldsymbol{N}$	Null value

The structure of the messages that are sent and stored on supporting servers is shown in Figure 22. Typically, a message contains following fields:

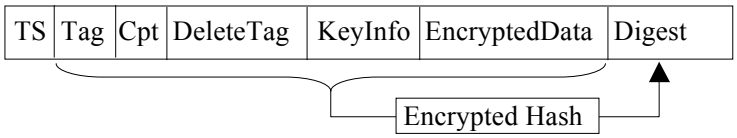


Figure 22. Message Structure

- *TS* is acquired by the supporting servers thanks to a secure time server. It is added to the message to allow the recipient PDS to filter out the messages it has already received.
- *Tag* is used to mark the messages intending for a given recipient PDS, thus allows the recipient PDS to retrieve its messages on the supporting servers.
- *Cpt* is a counter associated to each sender/receiver pair (or to each marker). It is incremented by the sender and used by the receiver to check the correctness of the messages ordering. In addition, it also could be used to detect missing messages and to guarantee the completeness of messages. In the following protocols, this field is ignored to simplify description.
- *DeleteTag* is a proof of legitimacy for the delete operation. It is computed by cryptographic hash functions and used by supporting servers to verify if the delete request comes from an authorized PDS or not. We will explain its usage explicitly later (see Section 3.4).
- *KeyInfo* is a session key used to encrypt the *EncryptedData* field. As the encryption uses symmetric encryption algorithms, the *KeyInfo* has to be transmitted along with the messages, and then used by the receiver to decrypt externalized data. For the *KeyInfo* itself, it is encrypted with the public key of the receiver.
- *EncryptedData* is the actual content of the message, encrypted with the session key *KeyInfo*.
- *Digest* is a hash of the previous fields, encrypted with the session key *KeyInfo* and used to check the integrity of the message, since an attacker could tamper the messages transmitted on communication channels.

Note that some fields do not play critical role for each protocol, and are deliberately not shown in the exchanged messages to make the description more concise.

3.3 Protocol for Exchanging Messages

For a given recipient PDS, the difficulty for retrieving the messages lies in the possibility to anonymously select messages on supporting servers, without revealing any information that

could be exploited by the server to infer the identity of the PDS. To address this issue, we propose to resort to anonymous markers, leading to the protocol described below.

The protocol to establish anonymous markers works in two phases (see Figure 23). In the first phase, the sender computes a tag T (which will be used to tag the next messages) thanks to the pseudorandom number generator. The computed tag T is transmitted encrypted with the session key Ks, itself encrypted with the public key of the receiver. This first message between a sender and a receiver is itself tagged with the public identifier of the receiver ID(R). Note that, while the receiver identifier is transmitted in clear-text in this first message, it does not disclose sensitive information because (1) the sender is anonymous and (2) for a sender/receiver pair there is only one message of that kind. Hence, an attacker could only count the number of entities who established a communication with a given PDS.

In the second phase, data is exchanged using the defined marker T, the timestamp TS, and the session key Ks encrypted with the public key of the receiver. Note that the reuse of markers with timestamps allows a passive observer to determine that new data items are shared possibly between the same sender and the receiver. Since all communications are anonymous this information cannot be exploited further to link a particular data item to one specific sender or receiver. However, this information could be hidden by changing the marker periodically, transmitting the new marker in the last message using the current marker.

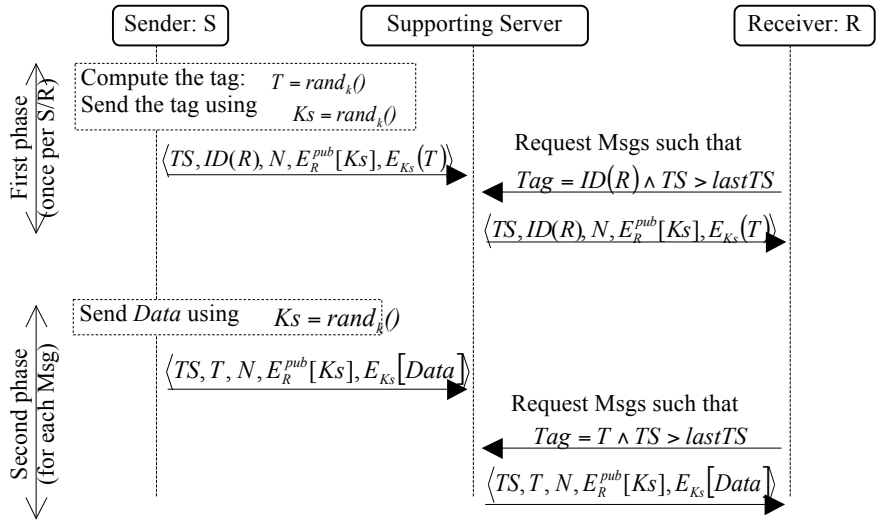


Figure 23. Communication using Markers

3.4 Protocols for Deletion

In this section, we tackle with two major issues for handling delete request, as mentioned in the introduction. First, we present how to exhibit deletion proof of legitimacy to supporting servers in a secure manner, while delete request could be issued by the sender or receiver (e.g., delete audit data). Next, we provide solutions to ensure the effectiveness of deletion, even in the case where physical image of the targeted data can not be destroyed or has been replicated or stolen.

3.4.1 Deletion with proof of legitimacy

A proof of legitimacy is required to guarantee that only the PDS which produces a data can delete it. Audit data is a special case where the PDS which is granted permission to delete some audit data (i.e., the publisher) is actually not the PDS which produces it (i.e., the subscriber). We illustrate below the protocol used when the delete right is delegated to the receiver. The protocol when the sender keeps the delete right can be deduced easily. The idea is based on cryptographic hash functions pre-image resistance property. The sender computes a random value called Delete Proof or DP and applies a cryptographic hash, thus obtaining DT, the Delete Tag. To transmit the delete right to the receiver, the sender simply adds DP to the data before encrypting it. When the receiver receives the message, it extracts DP and stores it. At delete time, the receiver sends a delete request, sending DP to the Supporting Server. Since given the hash value DT, it is computationally infeasible to find DP, such that DT = H(DP) (pre-image resistance property), the supporting server knows that the delete request was sent by an authorized PDS (see Figure 24). Moreover, the protocol for the case where the sender keeps the delete right can be deduced easily (see Figure 25).

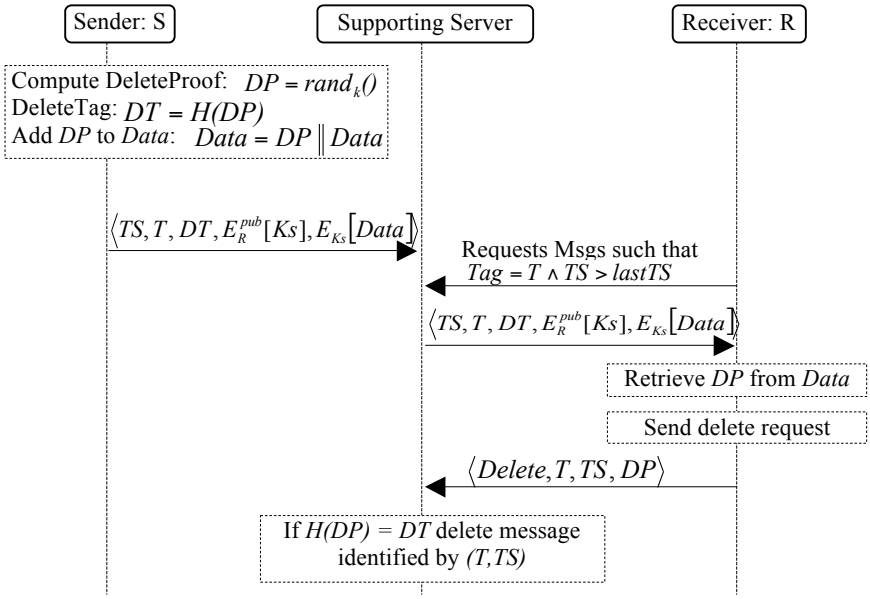


Figure 24. delete request issued by receiver

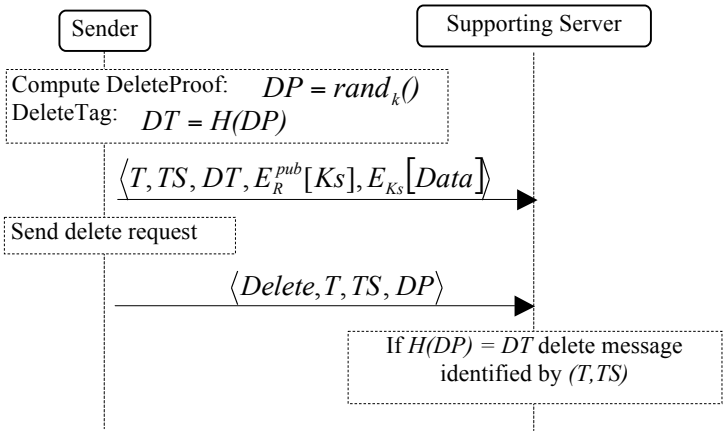


Figure 25. delete request issued by sender

3.4.2 Secure deletion

All data stored in the supporting servers have been carried by messages. Hence deleting a data item on the supporting servers amounts to deleting the corresponding message. Since the communications may be spied by an attacker and the messages copied, there is no other solution for enforcing the deletion than removing permanently the access to this message. This can be implemented as follows. The sender and the receiver establish a secret key using the Diffie-Hellman key agreement protocol and use it to encrypt the message (thus do not fill the KeyInfo field). When, e.g., the sender decides to delete the message, he destroys his partial secret and sends a message to the receiver requiring deletion of his partial secret. Even if an attacker

tampers one of the SPT after the deletion occurs, he cannot recover the message. This idea is simple but the protocol to implement it is more complex due to the fact that each party must be able to recover this message (assuming it has not been yet deleted) in case of a SPT failure (i.e., to ensure the durability property).

4 Conclusion

In this chapter, we design a set of crypto-protection building blocks. We instantiate the building blocks in the PDS context, and show how they can be used to protect some representative data structures used in PDS engine. We also propose secure communication protocols to ensure external supporting servers can serve data store, retrieve and delete requests, without knowing the identities of PDSs participating to the communication.

In the next chapter, we will use experiments to show the efficiency of the proposed solutions and their impact to the performance of PDS engine.

Chapter 6
Performance Evaluation

In this chapter, we show experimentally the effectiveness of the proposed crypto-protection schemes, designed for the embedded PDS engine. In the first section, we describe the experimental environment including hardware parameters, database schema and queries used in the experiments. Then we measure the performance of the crypto-protection schemes in isolation, compare them with other traditional methods and show their advantages. Finally, we measure an implementation of our techniques on the PDS engine and discuss the impact of the cryptographic protection.

1 Experimental Environment

1.1 Hardware Platform

Developing embedded software (i.e., PDS engine) for secure microcontrollers is a complex task, which is normally done in two phases: (i) development and validation on simulators, (ii) adaptation and flashing on secure microcontrollers. Our current PDS engine prototype is in the phase one and runs on a software simulator of the SPT platform. This simulator is IO and crypto-accurate, i.e., it computes exactly the number of IOs (by pattern) and cryptographic operations (by type), by far the most costly operations in our context. In order to provide performance results in seconds, we use SPT's basic timing information provided by our industrial partner Gemalto. Table 3 shows these basic timings, other hardware related parameters, as well as some parameters used in the prototype (for Bloom Filters). Note that, reading or writing on Flash can be done only at the granularity of a sector (512 B) or at the granularity of a page (2 KB). Indeed, to eliminate hardware errors, error correction codes (ECC) must be computed for each read/written sector. We read or write data by sector to minimize RAM usage (typically for buffering), a scarce resource on the SPT.

To check the validity of our time estimations, we calibrate the output of this simulation with performance measurements done on a previous prototype named PlugDB. PlugDB has already reached phase two (i.e., runs on a real hardware platform), has been demonstrated [ABG+10] and is being experimented in the field in an experimental project of secure and portable medical-social folder [AAB+09]. While PlugDB is simpler (no serialization, nor stratification, basic crypto-protection which is done at sector granularity), and it shares some commonalities

with the current design. Based on our preliminary measurements (e.g., through some queries where the code of PlugDB has similarities with the ones we designed for PDS engine), such calibration has shown its rationality.

Table 3. SPT simulator Performance Parameter

Category	Parameter Description	Performance
Flash	size of a Flash sector	512 bytes
	size of Flash page	2048 bytes (i.e. 4 sectors)
	read one Flash sector (including transfer time)	37.8 μs / IO
	write one Flash page (including transfer time)	262.8 μs / IO
	transfer one byte data from NAND register to RAM	0.025 μs / byte
CPU	CPU frequency (in MHZ)	50 MHZ
Cipher	block size for the AES cipher	16 bytes
	CPU cycles required for encrypting block sized data	120 cycles / block
	CPU cycles required for decrypting block sized data	120 cycles / block
Bloom Filter	number of hash functions used	3 hash functions
	encoding size for each key	2 bytes
	false positive rate of bloom filter	0.005 (percentage)

1.2 DBMS Settings

1.2.1 Database

In our experiments, we consider a synthetic medical database (see Figure 17) with cardinalities similar to TPCH with a scale factor SF = 0.5, leading to 3M tuples for the largest table: *Prescription*. Each table has 5 indexed attributes including ID, Dup10, Dup100, MS1, MS10 (climbing indexes use Hybrid Skip for their cumulative part with n = 2, *n* being the maximal number of SKA pages that should be scanned before stopping). ID is the row identifier of a tuple (amounts to the tuple's physical address in our settings), Dup10, DUP100, MS1 and MS10 are all CHAR(10) type, populated such that exact match selection retrieves respectively 10 tuples, 100 tuples, 1% and 10% of the table. Including the required foreign keys and other non-indexed attributes, the tuple size reaches 160 bytes.

The database schema cardinalities are given in Table 4 (see Figure 17 for the database schema). For each attribute, we indicate its name, its type, the table it belongs to, and the number of distinct values (considering a scale factor SF = 0.5). Primary key attributes are underlined and foreign keys are in italic. Attributes with star label (*) are indexed using a climbing index. This schema leads to build 30 climbing indexes, translated into 64 cumulative

indexes, from which 29 for the Prescription table (5 from each sub-table + 4 defined at Prescription level). In addition, we also built Subtree Key Tables on tables *Prescription*, *Visit* and *Drug*.

Table 4. The medical database schema cardinalities

Attribute \ Table		Pre	Vis	Doc	Dru	Lab	Cla
ID*	INTEGER	3M	75K	7500	400K	5K	10K
DUP10*	CHAR(10)	300K	7500	750	40K	500	1K
DUP100*	CHAR(10)	30K	750	75	4K	50	100
MS1*	CHAR(10)	100	100	100	100	100	100
MS10*	CHAR(10)	10	10	10	10	10	10
A1	CHAR(10)	3M	75K	7500	400K	5K	10K
A2	INTEGER	3M	75K	7500	400K	5K	10K
A3	DATE	3M	75K	7500	400K	5K	10K
COMMENT	CHAR(98)	3M			400K		
COMMENT	CHAR(94)		75K				
COMMENT	CHAR(90)			7500		5K	10K
IDVIS	INTEGER	75K					
IDDOC	INTEGER		7500				
IDDRU	INTEGER	400K					
IDLAB	INTEGER				5K		
IDCLA	INTEGER				10K		

1.2.2 Query

To measure the query performance, we define a set of 18 queries (see Table 5): 12 queries, termed $Mono_i$, involve an exact match selection predicate on a single table on attribute ID, DUP10 or DUP100, joins this table up to *Prescription* and projects one attribute per table. 3 queries, termed $Multi_i$, involve 2 or 3 exact match predicates on MS1 or MS10. Finally, 3 queries, termed $Range_i$, involve a range predicate on ClassOfDrug.DUP100.

Table 5 presents for each query its name, its number of results, the SQL expression of the query, and its execution plan. The operators CIL, Merge, SJoin and Project are those defined in Chapter 4.

Table 5. Query Set

Mono1 (1)	select Pre.A1 from Pre where ID = 1500000; Project (Pre@1500000, <Pre.A1>)
Mono2 (8)	select Pre.A1, Dru.A1 from Pre, Dru where Pre.idDru=Dru.ID and Dru.ID = 20000; Project (L_2, <Pre.A1, Dru.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru >) → L_2 CIL ($CI_{Dru.ID}$, Dru.ID=20000, Pre) → L_1
Mono3 (10)	select Pre.A1 from Pre where Pre.DUP10 = 'VAL_150000'; Project (L_1, <Pre.A1>) CIL ($CI_{Pre.DUP10}$, Pre.DUP10='VAL_150000', Pre) → L_1
Mono4 (30)	select Pre.A1, Dru.A1, Cla.A1 from Pre, Dru, Cla where Pre.idDru = Dru.ID and Dru.idCla = Cla.ID and Cla.ID = 5000; Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CIL ($CI_{Cla.ID}$, Cla.ID=5000, Pre) → L_1
Mono5 (80)	select Pre.A1, Dru.A1 from Pre, Dru where Pre.idDru = Dru.ID and Dru.DUP10 = 'VAL_2000'; Project (L_2, <Pre.A1, Dru.A1 >) SJoin (L_1, SKT_{Pre}, <idPre, idDru >) → L_2 CIL ($CI_{Dru.DUP10}$, Dru.DUP10='VAL_2000', Pre) → L_1
Mono6 (100)	select Pre.A1 from Prescription Pre where Pre.Dup100 = 'VAL_15000'; Project (L_1, <Pre.A1 >) CIL ($CI_{Pre.DUP100}$, Pre.DUP100='VAL_15000', Pre) → L_1
Mono7 (300)	select Pre.A1, Dru.A1, Cla.A1 from Pre, Dru, Cla where Pre.idDru = Dru.ID and Dru.idCla = Cla.ID and Cla.DUP10='VAL_500'; Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CIL ($CI_{Cla.DUP10}$, Cla.DUP10='VAL_500', Pre) → L_1
Multi1 (300)	select Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1 from Pre, Vis, Doc, Dru, Lab, Cla where Pre.idVis = Vis.ID and Vis.idDoc = Doc.ID and Pre.idDru = Dru.ID and Dru.idLab = Lab.ID and Dru.idClaID = Cla.ID and Doc.MS1 = 'VAL_50' and Cla.MS1 = 'VAL_50' Project (L_4, <Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1>) SJoin (L_3, SKT_{Pre}, <idPre, idDoc, idVis, idDru, idLab, idCla>) → L_4 Merge ($L_1 \cap L_2$}) → L_3 CIL ($CI_{Doc.MS1}$, Doc.MS1='VAL_50', Pre) → L_2 CIL ($CI_{Cla.MS1}$, Cla.MS1='VAL_50', Pre) → L_1
Mono8 (600)	select Pre.A1, Dru.A1, Lab.A1 from Pre, Dru, Lab where Pre.idDru = Dru.ID and Dru.idLab = Lab.ID and Lab.ID = 2500; Project (L_2, <Pre.A1, Dru.A1, Lab.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idLab>) → L_2 CIL ($CI_{Lab.ID}$, Lab.ID=2500, Pre) → L_1
Mono9 (800)	select Pre.A1, Dru.A1 from Pre, Dru where Pre.idDru = Dru.ID and Dru.DUP100 = 'VAL_200'; Project (L_2, <Pre.A1, Dru.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru>) → L_2 CIL ($CI_{Dru.DUP100}$, Dru.DUP100='VAL_200', Pre) → L_1
Mono10 (3K)	select Pre.A1, Dru.A1, Cla.A1 from Pre, Dru, Cla where Pre.idDru = Dru.ID and Dru.idCla = Cla.ID and Cla.DUP100='VAL_50'; Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CIL ($CI_{Cla.DUP100}$, Cla.DUP100='VAL_50', Pre) → L_1
Range1 (3K)	select Pre.A1, Dru.A1, Cla.A1 from Pre, Dru, Cla where Pre.idDru = Dru.ID and Dru.idCla = Cla.ID and Cla.DUP100 > 'VAL_99';

	Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CIL ($CI_{Cla.DUP100}$, Cla.DUP100>'VAL_99', Pre) → L_1
Multi2 (3K)	select Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1 from Pre, Vis, Doc, Dru, Lab, Cla where Pre.idVis=Vis.ID and Vis.idDoc = Doc.ID and Pre.idDru = Dru.ID and Dru.idLab = Lab.ID and Dru.idCla = Cla.ID and Doc.MS10 = 'VAL_5' and Cla.MS10 = 'VAL_5' and Lab.MS10 = 'VAL_5'; Project (L_5, <Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1>) SJoin (L_4, SKT_{Pre}, <idPre, idVis, idDoc, idDru, idLab, idCla>) → L_5 Merge ($L_1 \cap L_2 \cap L_3$}) → L_4 CIL ($CI_{Doc.MS10}$, Doc.MS10='VAL_5', Pre) → L_3 CIL ($CI_{Cla.MS10}$, Cla.MS10='VAL_5', Pre) → L_2 CIL ($CI_{Lab.MS10}$, Lab.MS10='VAL_5', Pre) → L_1
Mono11 (6K)	select Pre.A1, Dru.A1, Lab.A1 from Pre, Dru, Lab where Pre.idDru = Dru.ID and Dru.idLab = Lab.ID and Lab.DUP10 = 'VAL_250'; Project (L_2, <Pre.A1, Dru.A1, Lab.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idLab>) → L_2 CIL ($CI_{Lab.DUP10}$, Lab.DUP10='VAL_250', Pre) → L_1
Range2 (15K)	select Pre.A1, Dru.A1, Cla.A1 from Pre, Dru, Cla where Pre.idDru = Dru.ID and Dru.idCla = Cla.ID and Cla.DUP100 > 'VAL_95'; Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CIL ($CI_{Cla.DUP100}$, Cla.DUP100>'VAL_95', Pre) → L_1
Range3 (30K)	select Pre.A1, Dru.A1, Cla.A1 from Pre, Dru, Cla where Pre.idDru = Dru.ID and Dru.idCla = Cla.ID and Cla.DUP100 > 'VAL_90'; Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CIL ($CI_{Cla.DUP100}$, Cla.DUP100>'VAL_90', Pre) → L_1
Multi3 (30K)	select Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1 from Pre, Vis, Doc, Dru, Lab, Cla where Pre.idVis=Vis.ID and Vis.idDoc = Doc.ID and Pre.idDru = Dru.ID and Dru.idLab = Lab.ID and Dru.idCla = Cla.ID and Cla.MS10='VAL_5' and Lab.MS10 = 'VAL_5'; Project (L_4, <Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1>) SJoin (L_3, SKT_{Pre}, <idPre, idVis, idDoc, idDru, idLab, idCla>) → L_4 Merge ($L_1 \cap L_2$}) → L_3 CIL ($CI_{Cla.MS10}$, Cla.MS10='VAL_5', Pre) → L_2 CIL ($CI_{Lab.MS10}$, Lab.MS10='VAL_5', Pre) → L_1
Mono12 (60K)	select Pre.A1, Dru.A1 Lab.A1 from Pre, Dru, Lab where Pre.idDru = Dru.ID and Dru.idLab = Lab.ID and Lab.DUP100 = 'VAL_25'; Project (L_2, <Pre.A1, Dru.A1, Lab.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idLab>) → L_2 CIL ($CI_{Lab.DUP100}$, Lab.DUP100='VAL_25', Pre) → L_1

2 Performance Evaluation

In this section, we first evaluate the performance of the proposed crypto-protection techniques, measured in isolation (i.e., without embedding in the PDS engine), and compare their performance with that of traditional methods (e.g., Encrypt-then-MAC). Next, we give the selection performance using cumulative indexes (only CIL operator involved) and the overall performance of the system (including queries, stratification and insertion cost), to analyze the impact of cryptography protection.

2.1 Performance of Crypto-Protection Techniques

2.1.1 AREA

AREA provides confidentiality and integrity guarantees simultaneously while integrity guarantee comes at low cost. Due to Nonce storage, it causes larger storage expansion compared to the traditional method. Figure 26 compares AREA with the traditional Encrypt-then-MAC method, in terms of cryptography overhead, storage expansion and overall read/write cost (including IO and crypto). The numbers in the figure are obtained by dividing the performance of AREA with the performance of traditional method. We vary the (plaintext) data item size up to 512 bytes (i.e., the size of a NAND Flash sector).

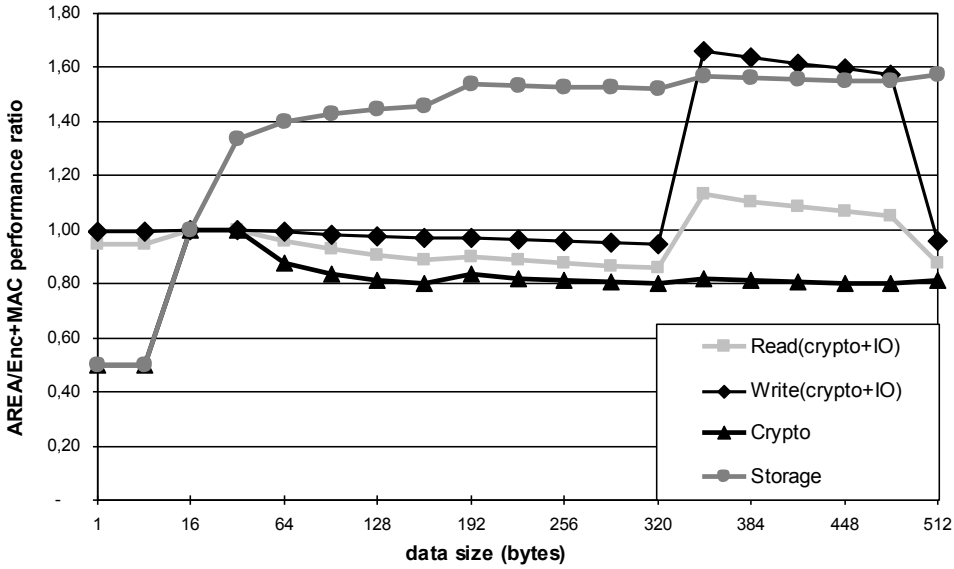


Figure 26. AREA vs. traditional method

When the data item is small (e.g., no more than 16 bytes), AREA outperforms traditional method on each aspect (e.g., crypto cost, storage and read/write performance), i.e., ratio is less than 1. Regarding crypto, the ratio is 0.5 because Encrypt-then-MAC requires encrypting the plaintext data, then computing the MAC on the resulted encrypted text. We assumed that auxiliary information added for MAC computation (e.g., version, address) does not change the size of data to be processed, e.g., by using exclusive-or operation. AREA only requires one encryption step, although on a larger data. Regarding storage expansion, for traditional method, the MAC itself takes 16 bytes storage, thus it consumes more storage than AREA (with Nonce) for up to 16 B plaintext data. For read/write performance (including IO and crypto cost), since

the data item is small, it can be stored/accessed in one IO, leading to the same IO cost. Relatively to IO cost, the crypto cost difference is negligible (for small data). Therefore, AREA and traditional method have similar read and write cost, i.e., ratio ≈ 1.

For large data item (e.g., larger than 64 bytes), AREA reduces the crypto overhead by about 20% (instead of 50%), compared with traditional method. This is because AREA only processes 10 bytes plaintext in one operation, and the remaining 6 bytes are reserved for Nonce storage. The storage expansion ratio of AREA is near 1.6, which can be easily deduced. Regarding read/write performance, both methods are similar since they largely depend on the number of IOs. When data item size is between 352 bytes and 480 bytes, AREA leads to 2 IOs due to its larger storage expansion, while traditional method only requires 1 IO, thus the performance of AREA (for write) is relatively bad.

As a conclusion, AREA is adequate to encrypt and authenticate small to medium granularity data (up to 320 B, the overall read and write performance are better with AREA). In our context, we apply AREA on almost all data structures of embedded PDS engine (e.g., bases tables) because (i) Most PDS engine data are fine granularity (e.g., indexing data, attribute values); (ii) Data is written once and read intensively emphasizing the read performance; (iii) The storage expansion is not problematic, because the non-secured Flash storage has no strict limit in our context.

2.1.2 Smart Selection

We compare smart selection with the following methods: (i) a brute force method that encrypts and authenticates the whole set (i.e., one KA page); (ii) an oracle solution that only decrypts and checks the integrity for the results, an optimal but obviously unreachable case. The former is thus an upper bound while the latter is a lower bound. Both methods use AREA (remember that smart selection is not compatible with AREA), which has shown its significant advantages over traditional method in the previous experiment. For simplicity, we consider that the size of each key value is small and could be processed with one cryptographic operation.

Figure 27 shows the resulting performance. We consider that the number of values (e.g., p) in the set is 64, but varying this number will not change the conclusions. In the experiment, we vary the number of results (e.g., n) up to 64, such that all values in the set belong to the results.

According to the experimental results, smart selection outperforms brute force solution, since the former only pay crypto cost for potential results, while the latter requires decrypting all values in the set. Indeed, smart selection searches values on the ciphertext directly without decryption, paying one encryption for each result (except the last one) during the search, and one for checking the MAC of the last occurrence of the matching values. We did not show on the figure the case for no result. In that case, which should be very rare, thanks to bloom filters (only with false positive cases), smart selection will have to check the integrity for the whole set, thus leading to the performance similar to the Brute Force algorithm (although a little better because of storage expansion of AREA).

Compared with oracle solution (with AREA, one operation/result), smart selection is slightly more expensive. Indeed, smart selection requires two extra operations: (i) to identify the last result (i.e., searching for the $n+1^{th}$ occurrence, n being the number of results), (ii) check the MAC of the last result.

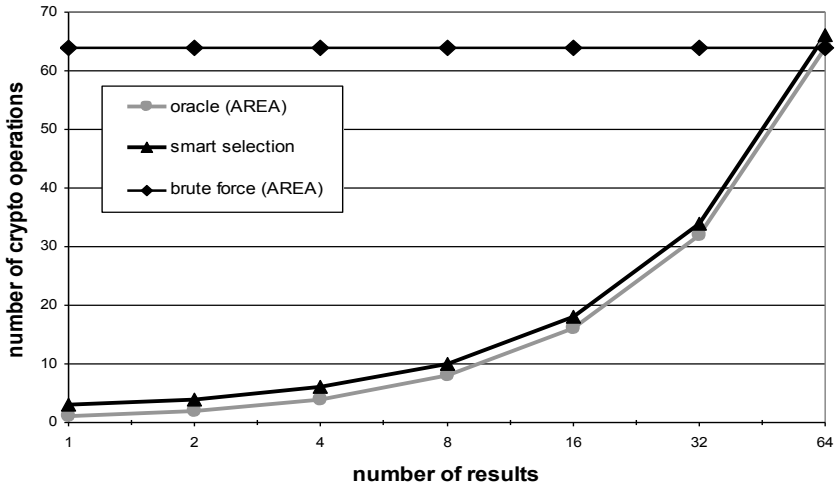


Figure 27. Smart Selection vs. traditional methods (p = 64)

In summary, the crypto overhead for brute force solutions depends on the p value (i.e., number of values in the set), while oracle solution and smart selection only depend on the n value (i.e., number of results). Furthermore, oracle solution needs n operations, thus performs slightly better than smart selection scheme (requires n + 2 operations). Therefore, smart selection is a highly efficient way for equi-selection on encrypted values.

2.2 Performance of Cumulative Indexes

This section describes the performance of cumulative indexes first without cryptography, and then focuses on the overhead incurred by crypto-protection.

2.2.1 Performance without Cryptography

We evaluate the cost of exact match lookups and insertions for the serialized indexes proposed in Section 4.3 of Chapter 4. Differently from conventional databases, indexes can be beneficial even with very low selectivity; indeed, random or sequential reads on NAND flash (with no FTL) have the same performance. Therefore, it makes sense to vary selectivity up to 100%. We consider a single table of 300K records, stored in SDB, with 11 attributes, populated with a varying number of distinct values (3, 15, 30, …, up to 300K), which are distributed uniformly. For insertions, we consider the worst case (i.e., inserting a single tuple, then committing). Figure 28 and Figure 29 show respectively the results for exact match index lookups and insertions (in one single index).

As expected, Full Index Scan has a very good insertion cost and a very bad lookup performance (full scan cost whatever the selectivity). Summary Scan reaches pretty good insertion costs but lookups do not scale well with low selectivity predicates (the whole SKA and KA are scanned). In low selectivity cases, its lookup performance is even worse than Full Index Scan, since the latter only full scan KA.

Compared to Summary Scan, Summary Skip performs better in terms of lookup since we take advantage of more compact structure PTR and use inverted pointer chain to retrieve all results, thus the IO cost is largely reduced. However, Summary Skip needs to maintain PTR structure, thus its insertion performance is always worse than Summary Scan. Such performance is extremely expensive when the inserted value is not frequent (more expensive than any other type of indexes), because we have to find its previous occurrence to maintain the pointer chain (in the worst case, the whole SKA need to be accessed in reverse order).

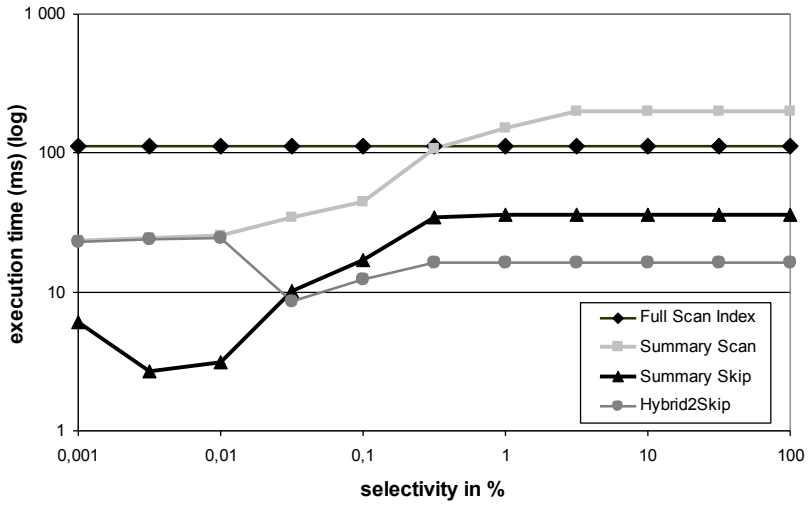


Figure 28. Query Performance of Cumulative Indexes (without crypto)

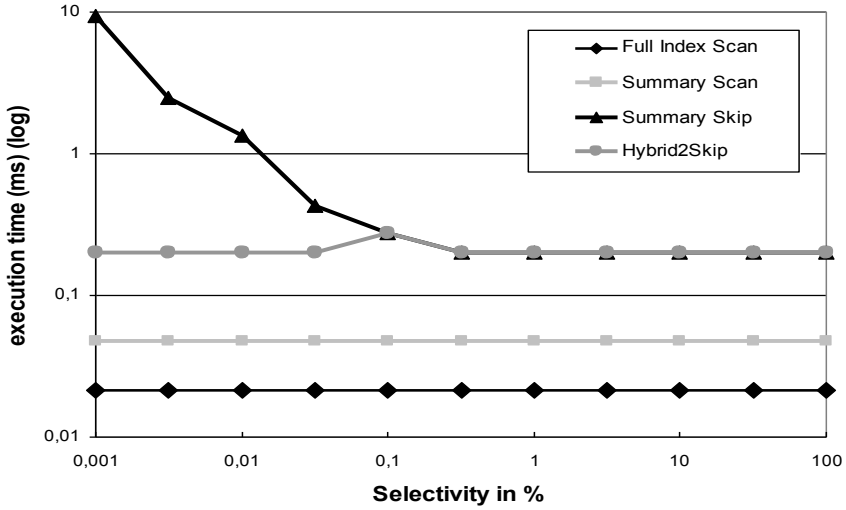


Figure 29. Insertion Performance of Cumulative Indexes (without crypto)

Hybrid Skip appears as the best compromise, with a bounded insertion cost. Indeed, when the selectivity is higher than 0.1% (the pointer chain is broken), it stops searching the previous occurrence as the number of scanned SKA pages has reached *n* limit. When the selectivity is low and the pointer chain is not broken, it has the same insertion cost as the Summary Skip. On the other side, Hybrid Skip has very good lookup costs. With low selectivity predicates, it even outperforms any other indexes, including Summary Skip, because pointers accessed in PTR are smaller in size (e.g., several bits). As the pointer size only depends on the number of keys

summarized by *n* SKA pages, instead of depending on the cardinality of the whole SKA. Therefore, the PTR structure is more compact leading to less IOs.

2.2.2 Analysis for Cryptography Impact

This section describes the crypto impact to the cumulative indexes. To simplify analysis, we measured the indexes performance with and without enforcing cryptography respectively, and use the performance ratio to quantify such impact (the subsequent analysis in the following sections are done in the same way). The obtained results are shown in Figure 30. Note that, the ratio with 1 means the cryptography does not incur any extra overhead.

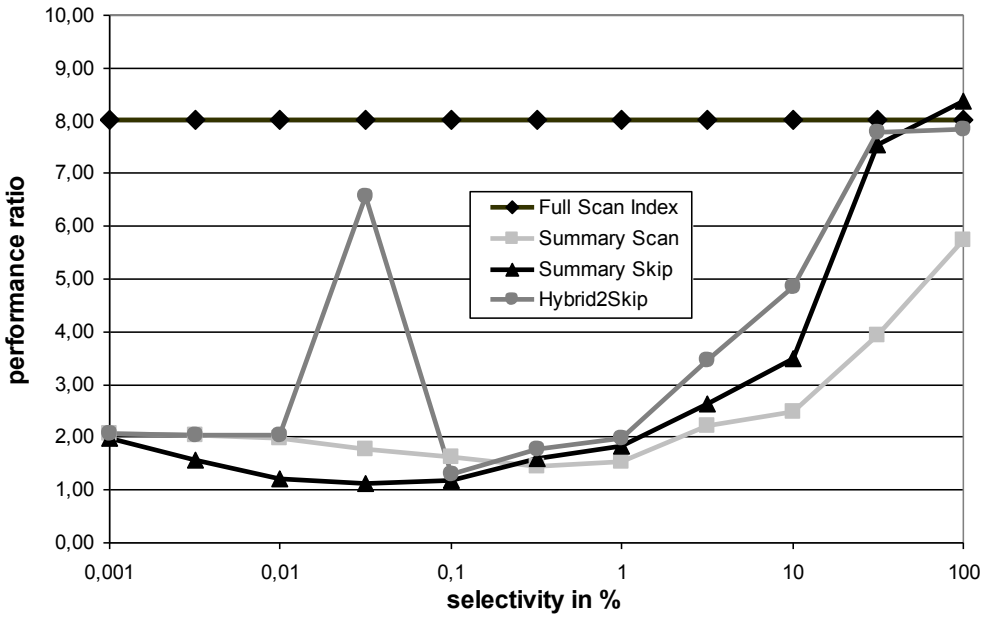


Figure 30. Crypto Impact on Cumulative Indexes for Selection

For *Full Scan Index*, the lookup cost is increased significantly after enforcing cryptography, i.e., ratio amounts to 8. This is caused by three reasons. (i) KA is protected by AREA, thus consumes more space due to Nonce storage, thus requires more IOs to do full scan KA. More precisely, the number of IOs is increased by 60% (since the storage expansion factor of AREA is 1.6). (ii) All values in the KA page need to be decrypted during scan, thus incurs prohibitive cost, i.e., 307.2 μs / IO (vs. 37.8 μs / IO for Flash sector read).

The lookup performance for other indexes including *Summary Scan*, *Summary Skip* and *Hybrid Skip* are less impacted than *Full Scan Index*, since they speedup lookup process by using BF, pointer chain etc, thus decrypt much less data.

More precisely, for *Summary Scan*, when the selectivity is high (e.g., less than 1%), its lookup cost is doubled at most. The extra cost mainly comes from IOs, this is because (i) few results are retrieved (and decrypted), (ii) SKA adopts data placement strategy and leads to largely reduced crypto overhead, i.e., only interesting data are decrypted.

Hybrid Skip has similar performance as *Summary Scan*, since both of them have similar IOs to load SKA, PTR (or KA for *Summary Scan*). The exceptional point in the curve for *Hybrid Skip* is due to the different selectivity limit (with/without crypto), when the pointer chain (in PTR) is broken (with fixed *n* value): when selectivity is equal to 0.05%, the *Hybrid Skip* (without cryptography) performs as the *Summary Skip* (the pointer chain is not broken, see Figure 28). After encryption, it performs as *Summary Scan* (the chain is broken), hence leads to complete different performance. As a result, the ratio between both is quite large. Note that this does not mean that *Hybrid Skip* is a bad strategy with crypto protection (the graph presents ration and not absolute values or comparable values between techniques). In contrast, *Summary Skip* has less crypto overhead compared to others, because it only scans partial SKA to find the last occurrence of searched value.

When the selectivity becomes lower (e.g., larger than 1%), the crypto overhead increases significantly for all three indexes, because a large number of results need to be decrypted and there is less and less benefit in using smart strategies, like SKA, smart selection or reverse pointer chains. Note that, on absolute values, *Hybrid Skip* remains with or without crypto the best strategy.

Regarding crypto impact on insertions for cumulative indexes (only considering insertion on KA, SKA and PTR), the crypto overhead incurred becomes less significant, as shown in Figure 31. Indeed, for each insertion (insert only one tuple then committing), the cost is dominated by expensive write (e.g., 272.8 μs / IO). The overhead increases by about 40% on average for *Full Index Scan* and *Summary Scan*, because only 1 IO is needed per insertion (for KA, since SKA is only computed and stored at the moment one KA sector is full). The incurred overhead is smaller for *Summary Skip* and *Hybrid Skip,* about 20% because 2 IOs are needed per insertion (one for inserting key in KA and one for inserting pointer in PTR), thus mitigating the crypto

impact. Note that, when the selectivity is low, the crypto overhead increases due to smart selection behavior in KA. Indeed, we have to perform several encryptions to find the previous occurrence of the inserted value, starting from the first one of the KA page (for maintaining the reverse chaining).

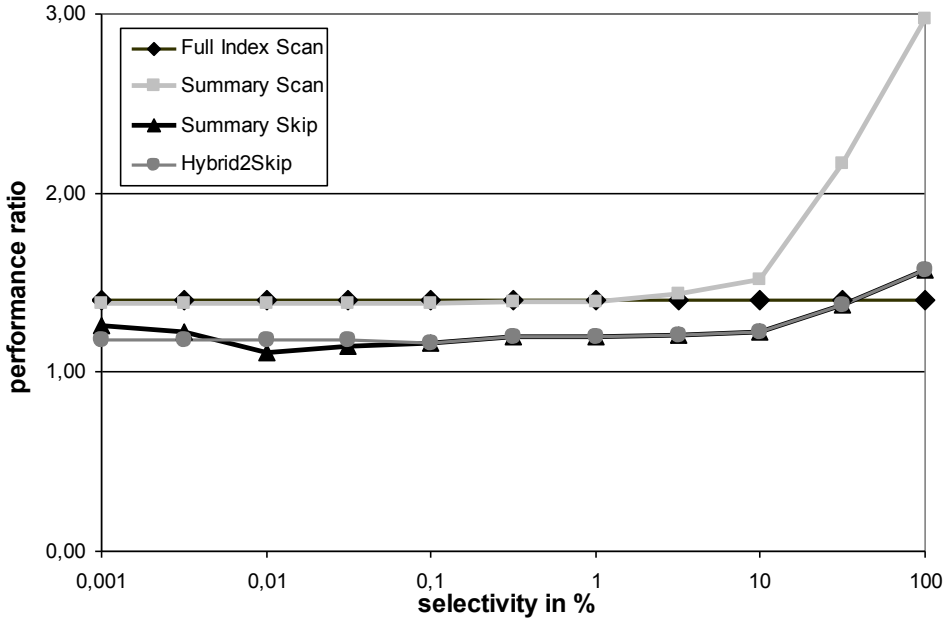


Figure 31. Crypto Impact on Insertions for Cumulative Indexes

In summary, the crypto overhead for selection is low for high selectivity and can reach a factor of 8 in very low selective cases. Such impact seems rather large, but, as we will see in the next section, the selection cost is not dominant in the whole query (which includes accessing SKT, projection etc) in these cases. In addition, the crypto overhead for insertions in cumulative indexes is acceptable.

2.3 Performance of Overall System

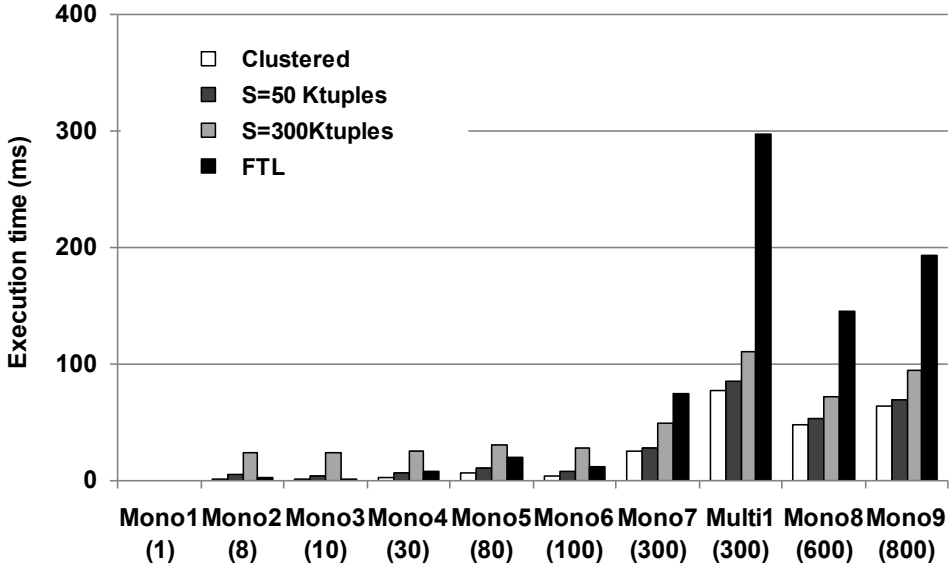
For the performance of overall system, we describe it on three aspects: queries, stratification and insertions (including insertions to SKT and base tables). As for cumulative indexes, we first address their performance without crypto-protection and then analyze the crypto-protection impact.

2.3.1 Query Performance and Cryptography

We measure the query response time with 3 settings: (1) the database has just been reorganized and thus SDB = ∅; (2) SDB = 50K; (3) SDB = 300K; and we compare with the same queries executed over a database traditionally accessed through a Flash Translation Layer (FTL). The index model is the same in all settings. For the FTL case, we do the optimistic assumption that the database is fully clustered, as in CDB, while traditional B-Tree is only filled at about 69% [Yao78]. Figure 32 presents the measurements for the 18 queries, ordered by the number of resulting tuples (X axis). We split the graph in two in order to have different scales for response time (0-400ms, 0-10s).

For selective queries (1-800 results), selection cost is relatively important with large SDB (300K) while with SDB = 50K the response time is very near the clustered one. Considering several predicates (Multi1) increases this cost, as expected, because it needs to traverse several indexes to obtain the same number of results as the others. The FTL approach is less efficient as soon as the number of results exceeds 300 for an SDB of 300K (or 30 for 50K). Note, the cost of Mono1 is almost zero because it retrieves a prescription having a specific ID, which is the tuple's physical address in our setting.

For less selective queries (3K-60K results), the cost is dominated by the projection step. Consequently, the SDB size has little influence. The FTL approach performs very badly under this setting because of the high number of IOs generated at projection time (then the indirection overhead incurred by FTL).



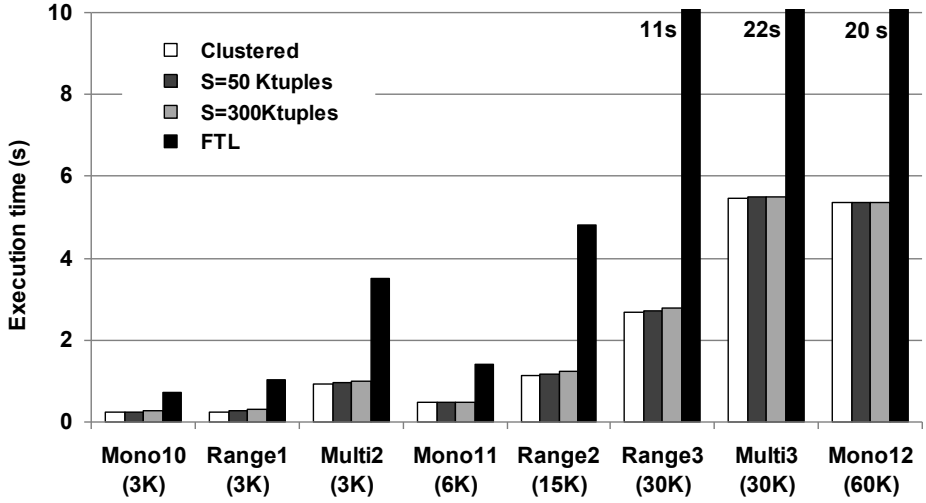


Figure 32. Performance of 18 queries with different settings

Regarding updates and deletes, measurements (SDB=50K) and (SDB=300K) have been done after having deleted randomly 3000 tuples (with cascade delete option) and having updated 3000 tuples (uniformly distributed on DUP10, DUP100, MS1, MS10 and A1 attributes of each table). We observed a small influence of updates and deletes on performance, because they are evenly distributed. Considering more focused updates (on the queried attribute value) may lead to bigger degradation, which stay rather limited thanks to massive indexation of UPD.

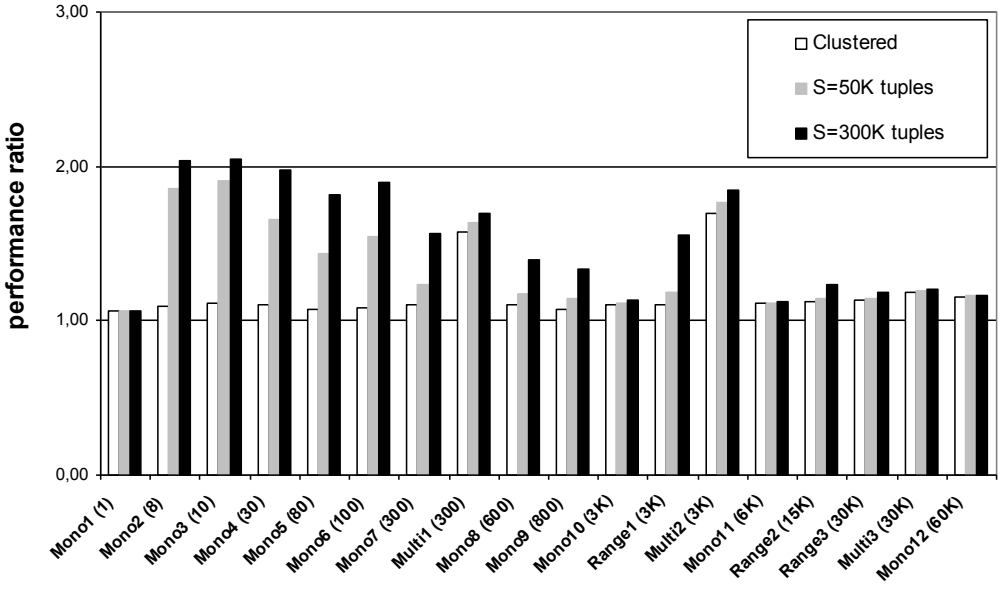


Figure 33. Crypto Impact on Queries

Figure 33 shows the crypto impact on all queries. For selective queries (1-800 results) with cumulative indexes, their total cost is doubled at maximum. Because (i) the selection cost is relatively important with larger SDB (e.g., 300K), and less for smaller SDB (e.g., 50K), as discussed before. (ii) The selection cost is roughly doubled after enforcing cryptography (refer to Figure 30). (iii) SKT and tables are protected by AREA, thus the incurred crypto workload for projection purpose is minimal.

When the selectivity becomes lower (e.g., more than 800 results), the crypto impact becomes less significant, i.e., increases the total cost by 10% ~ 20% only (except the query with several predicates, i.e., Multi2) because projection cost is dominant due to large number of IOs.

On the other side, the query cost on purely clustered database increases by 6% ~ 18% due to cryptography, independently of selectivity (excluding the queries with several predicates). Because (i) the Clustered Indexes are highly efficient structures and naturally sorted after stratification, which support efficient retrieving (e.g., dichotomy algorithm) even after encryption (using AREA). (ii) For the data protected by smart selection scheme in CDB (i.e., CSL), they do not have redundant duplicates (i.e., each key in CSL is unique). Therefore, it allows efficient selection requiring only 2 cryptographic operations (the algorithm is slightly adapted to avoid searching the next element).

Based on above observations, we can conclude that the overhead incurred by cryptography is acceptable and does not impact query performance greatly.

2.3.2 Stratification Performance and Cryptography

For stratification, we describe first its performance without cryptography. We consider a fixed size for CDB and vary the size of SDB (varying the *Prescription* table from 50K to 300K tuples, other tables grow accordingly). The reorganization cost varies linearly from 7 min (for 50K SDB) to 8.9 min (for 300K SDB). The stratification cost mainly results from (i) reorganizing IND, DEL and UPD and (ii) reading CDB_i and rewriting CDB_{i+1}. Consequently, different SDB settings do not change the cost greatly.

During the stratification, IND, DEL and UPD are reorganized, thus generate intermediate results to allow efficient merge. Since these intermediated results are processed in batch and do not support queries, we protect them using AREA. After encryption, the stratification cost grows to 18 min (for 50K) and 22.1 min (for 300K). The incurred overhead is rather large,

because (i) all data need to be decrypted and re-encrypted during processing, and (ii) storage expansion after encryption induces more IOs.

Even with large overhead increase after enforcing cryptography, the stratification performance will not become the problem in our context. Indeed stratification runs in background, and does not block queries. In addition, the time spent on stratification is still reasonable.

2.3.3 Insertion Performance and Cryptography

Let us finally consider the crypto impact on insertion cost for one tuple (we compute average cost by inserting different numbers of tuples up to flushing), including insertions to table, SKT and all its indexes (assumed using Hybrid Skip). Since the number of cumulative indexes depends on the table, the insertion cost also depends on the table (Hybrid Skip insertion cost is rather stable w.r.t., the selectivity – see Figure 29).

Without cryptography, the insertion cost varies between 1.8 ms for *Doctor* table to about 6.8 ms for *Prescription* table. With cryptography, the cost increases to 2 ms for *Doctor* table and to 8 ms for *Prescription* table, i.e., it increases the cost by about 20%. The explanation is similar to the ones for insertions of cumulative indexes. Indeed, the write IO cost is very expensive compared to the extra crypto cost, thus making the cryptography impact less significant. Consequently, we can infer that cryptography does not appear a bottleneck regarding insertion performance.

3 Conclusion

In this chapter, we compare our proposed crypto-protection solutions with other traditional methods, and the experimental results show the effectiveness of our solutions. Furthermore, we implement our crypto-protection solutions on PDS engine, measure the system performance with and without cryptography respectively, in terms of queries, insertion and stratification, and analyze the crypto impact. The experiments show the effectiveness of our solutions, the incurred overhead is acceptable and does not violate user patience expectations. Note that, a large portion of overhead comes from increased IOs due to storage expansion (unavoidable) after encryption, while IO cost is dominant in our context.

Chapter 7
Conclusion and Future Works

Our ultimate goal is to provide personal and secure data management facilities, resisting to privacy violations and security vulnerabilities, resulting, e.g., from abusive usage or attacks (even those conducted by internals or administrators). To progress in this direction, we proposed the Personal Data Server (PDS) vision, where personal data is securely managed by a self-administrated DBMS embedded into a secure device remaining under the control of its owner. In this vision, we did resort to a new hardware called Secure Portable Token, combining the tamper resistance of a smart card microcontroller with the mass storage capacity of NAND Flash.

To enforce advanced access control over the embedded personal data, and as well to enable powerful applications to be developed on top of the user's personal database, a full-fledged DBMS engine (the PDS engine) had to be embedded in the SPT. One of the challenges was to cope with the hardware constraints (e.g., limited RAM and NAND Flash particularities) exhibited by the SPT. Initial contributions were the design of a query engine able to deal with Gigabytes of data without resorting to external resources. A massive indexing scheme inspired by the data warehouse context was devised to circumvent this issue. In addition, to match the NAND Flash constraints, the notion of serialization of all the database structures (including indexes, buffers, logs, etc.) was introduced. For scalability reasons, the idea of database stratification, where database structures are periodically rebuild (and clustered) in background, was proposed.

The first goal of this thesis was, given those existing contributions, to provide security solutions to protect the personal database footprint, stored in the external NAND Flash of the SPT (and thus being outside of the tamper resistant sphere).

To restore common features of traditional DBMS systems, like data durability, availability, data sharing and other global processing functionalities, *supporting servers* were introduced. Those servers were assumed not to be trusted, but Honest-but-Curious, i.e., they may breach the confidentiality of externalized data or violate the anonymity of the participant PDSs' identities. The second objective of this thesis was to propose security protocols to protect the

communications between personal data servers (PDSs), and the interactions with the supporting servers.

We summarize below the main contributions of this manuscript, and then present some research perspectives.

1 Summary

The main contribution of the thesis is linked to the cryptographic protection of the embedded database footprint (objective 1). A second important contribution is to propose the basis for a set of secure protocols to validate the PDS vision (objective 2). A third aspect of the PhD was linked to the prototyping effort. Those three aspects are summarized below.

1.1 Cryptographic building blocks

To protect Flash resident data from confidentiality and integrity attacks, we proposed a set of crypto-protection building blocks. To ensure the efficiency of the proposed building blocks (and maintain good system performance), the cryptographic overhead had to be minimized. Therefore, our design has taken into account the specificities of the PDS engine (e.g., fine granularity data accesses, specific access patterns), the resource limitations of the platform (e.g., constrained NOR Flash size) and the particularity of the database storage media (i.e., NAND Flash memory). The proposed crypto-protection building blocks include *version management* techniques, adaptation of the *AREA encryption* method, a specific encryption model called the *smart selection* enabling equality search in a dataset avoiding decryption and *data placement* strategy. We summarize below those different aspects of the proposal.

(1) *Version management*. We have proposed to only store the stratum number (as the version number) in the secure memory (i.e., internal NOR Flash). This is an important advantage inherited from the serialization and stratification paradigms. At the time of version checking (integrity), we only check if the accessed data item belongs to the good stratum. As this strategy only keeps a minimal number of versions (i.e., stratum numbers) in the secure memory, it avoids the traditional problems linked with the storage of multitude of version numbers (usually, one version number per data item). In addition, our technique does not cause any extra overhead at query time (since NOR Flash is in direct access, like RAM), and makes versions updating problem on NAND Flash (since the NAND Flash provides poor support for random writes) vanish.

(2) *Adaptation of the AREA encryption scheme*. AREA is an authenticated encryption method based on traditional block cipher (e.g., AES algorithm). It provides integrity guarantees at low cost, and requires only one pass of processing on the targeted data items. On the contrary, traditional integrity methods either introduce complex algorithms (e.g., Authenticated Encryption algorithms like OCB), or require two passes of processing (e.g., Encrypt-then-MAC), which is much more expensive in terms of cryptographic operations. The only drawback of AREA is that it incurs a storage expansion due to the embedded Nonce. The original AREA, as proposed by Elbaz et al. in [ECL+07], was requiring to keep the Nonce secret. This was problematic in our context (secure NOR Flash is not large enough, and storing Nonce on NAND Flash would highly increase the query cost). However, we found that AREA is in fact independent of the secrecy of Nonce (see Section 2.2.2, in Chapter 5)

(3) *Smart selection scheme*. Smart selection allows selection to be done directly on ciphertext (without data decryption). The selection process thus incurs a minimal crypto workload, being proportional to the number of results. More precisely, it requires one crypto operation (i.e., encryption) for each result, and two additional crypto operations for locating the last occurrence and checking its integrity. This solution is near the optimal (oracle) case (regarding crypto overhead), which consists in decrypting only the qualified results. Note that, this scheme only supports equality queries.

(4) *Specific data placement strategies.* Using adequate data placement strategy (by changing the data layout), the query process only decrypts data-of-interest and skips the uninteresting ones. Therefore, the cryptographic overhead is largely reduced. Its benefits come for free as soon as it does not hurt the IO patterns at query time. In our context, due to SWS definition, this requires applying data placement at the NAND Flash page level.

1.2 Secure protocols for the PDS

Regarding communications with external supporting servers, we proposed a preliminary design for the main communication protocols required in the PDS architecture. In particular, we have proposed secure protocols to enable the supporting servers to serve *retrieve*, *store* and *delete* requests. Using our protocols, the confidentiality of the externalized data is preserved and the anonymity of each participant PDS is guaranteed. Note that we have focused on showing the feasibility of the PDS approach in terms of global architecture, and have let concrete implementations considerations for future works.

1.3 Prototyping and experience in the field

A part of the PhD thesis has been dedicated to prototyping. First, a demonstrator has been implemented on a hardware token furnished by Gemalto (our industrial partner) to demonstrate the query processing engine endowed with a preliminary version of our cryptographic techniques. This prototype has been demonstrated at SIGMOD 2010 [ABG+10].

Second, a PDS like system prototype, dedicated to a social-medical application, has been developed by the SMIS team. A field experiment has started in September 2011. It is supposed to involve up to 120 users including doctors, patients and social workers. Cryptographic techniques have been used in that prototype. It was an important prerequisite to obtain the agreement of the French CNIL for conducting this experience. More details on this prototype, on the related projects and funding can be obtained at: http://www-smis.inria.fr/~DMSP/dmsp_presentation.php.

2 Research Perspectives

Our first perspective, for the very short term, concerns the secure communication protocols. In this manuscript, we just proposed a preliminary design to attest the feasibility of the PDS approach, without going into implementation details.

Besides, we plan to extend our results to a more general setting. The following directions can be investigated:

- The crypto-protection building blocks that are proposed in this manuscript are dedicated to the structures designed for the PDS engine. We plan to identify other contexts where the underlying data structures share similarities with the ones considered in PDS. For example, we will study techniques analogous with ours in the contexts of log-structured file systems and DBMSs.
- We also plan to characterize the database structures employed in traditional DBMS, and devise some cryptographic protection using the same approach as the one we used for the PDS structures. We are pretty convinced that we can obtain interesting results, at least for structures having a sequential behavior like bitmap indexes or join indices, even though adaptation of our techniques are needed to support update-in-place (our version management proposal must be revised).

- A third axis can be to improve the AREA scheme, by "chaining" the consecutive AREA blocks when encrypting large data items (where several consecutive AREA blocks are required). This could enable reducing the storage overhead by making the Nonce size smaller in each intermediate element of the chain (considering consecutive elements are keeping a link with each other, like in the CBC mode, more concise Nonce information are used). For example, we could embed address/version information in the Nonce of the first AREA block (with normal Nonce size), and in the subsequent blocks embed their offset (to the first block) and order (among all blocks) instead of a "full" Nonce.
- Another possible extension can be to try using of the *smart selection scheme* in the context of outsourced database (or in the Database as a Service model). Indeed, the whole selection process using smart selection does not involve any decryption operations. It can thus be used on the un-trusted party. Moreover, compared with existing DaS execution techniques, it would not generate any false positive (that have to be subsequently filtered out at the client side). We can thus expect some gains in the case of equi-selection queries.

Bibliography

[AAB+09] Allard, T., Anciaux, N., Bouganim, L., Pucheral, P., Thion, R., Trustworthiness of Pervasive Healthcare Folders, Pervasive and Smart Technologies for Healthcare Information Science Reference, 2009.

[AAB+10] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. Le Folgoc, B. Nguyen, P. Pucheral, Ij. Ray, Ik. Ray, S. Yin, Secure Personal Data Servers: a Vision Paper, VLDB, 2010.

[ABB+07] Anciaux, N., Benzine, M., Bouganim, L., Pucheral, P. and Shasha, D., GhostDB: Querying Visible and Hidden data without leaks, SIGMOD, 2007.

[ABG+10] Anciaux, N., Bouganim, L., Guo, Y., Pucheral, P., Vandewalle J-J. Yin, S., Pluggable Personal Data Servers, SIGMOD, 2010.

[ABG+11] Anciaux, N., Bouganim, L., Guo, Y., Le Folgoc, L., Pucheral, P., and Yin, S. Database Serialization and Stratification: a New Paradigm to Manage Embedded Data. Submitted, 2011.

[ADH02] Ailamaki, Anastassia and DeWitt, David J. and Hill, Mark D, Data page layouts for relational databases on deep memory hierarchies, VLDB Journal, 2002.

[AGS+09] Agrawal, D., Ganesan, D., Sitaraman R., Diao Y. and Singh S., Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices, VLDB, 2009.

[AKS+02] R. Agrawal, J. Kiernan, R. Srikant, Xu. Yirong. Hippocratic databases, VLDB, 2002.

[AKS+04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu, Order preserving encryption for numeric data, SIGMOD, 2004.

[Anc04] Nicolas Anciaux, Database systems on chip, PHD thesis, 2004.

[BaM76] Bayer, R. and Metzger, J. K., On the Encipherment of Search Trees and Random Access Files, ACM Trans Database Systems, 1976.

[BaP07] Giuseppe Di Battistal, Bernardo Palazzi, Authenticated Relational Tables and Authenticated Skip Lists, Data and Applications Security, 2007.

[BaS11] Sumeet Bajaj, Radu Sion, TrustedDB: a trusted hardware based database with privacy and data confidentiality, SIGMOD, 2011.

[BCL+09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill, Order-Preserving Symmetric Encryption, EUROCRYPT, 2009.

[BCO+04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, Public Key Encryption with Keyword Search, EUROCRYPT, 2004.

[BeN00] Mihir Bellare, Chanathip Namprempre, Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm, Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology, 2000.

[BJB+09] Bouganim, L., Jónsson, B. T. and Bonnet P. uFLIP: Understanding Flash IO Patterns, CIDR, 2009.

[BKR94] M. Bellare, J. Kilian, and P. Rogaway, The security of the cipher block chaining message authentication code, journal of Computer and System Sciences, 1994.

[Blo70] Bloom, B. H., Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, 1970.

[BlR02] J. Black, P. Rogaway, A block-cipher mode of operation for parallelizable message authentication, in Advances in Cryptology: EUROCRYPT, 2002.

[BoG09] Luc Bouganim, Yanli Guo, Data Encryption, Encyclopaedia of Cryptography and Security, Springer, 2009.

[BoP02] Bouganim L, Pucheral P, Chip-secured data access: confidential data on untrusted servers, VLDB, 2002.

[CDP05] Ceselli Alberto, Damiani Ernesto, Paraboschi Stefano, Modeling and Assessing Inference Exposure in Encrypted Databases, ACM Transactions on Information and System Security, 2005.

[Cha81] David L. Chaum, Untraceable electronic mail, return addresses, and digital pseudonyms, Communications of the ACM, 1981.

[Cha09] Aldar C-F. Chan, Symmetric-Key Homomorphic Encryption for Encrypted Data Processing, 2009 IEEE International Conference on Communications, 2009.

[ChI99] Chan, C. Y., Ioannidis, Y. E., An Efficient Bitmap Encoding Scheme for Selection Queries, SIGMOD, 1999.

[ChM05] Yan-Cheng Chang, Michael Mitzenmacher, Privacy Preserving Keyword Searches on Remote Encrypted Data, ACNS, 2005.

[ChO06] Sun S. Chung, Gultekin Ozsoyoglu, Anti-Tamper Databases: Processing Aggregate Queries over Encrypted Databases, ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops, 2006.

[CMT+05] Claude Castelluccia, Einar Mykletun, Gene Tsudik, Efficient Aggregation of encrypted data in Wireless Sensor Networks, The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2005.

[CSI11] Computer Security Institute, 15th Annual 2010/2011 Computer Crime and Security

Survey, 2011.

[DBLoss] DataLossDB, url = http://datalossdb.org/

[DDJ+03] Damiani E, De Capitani Vimercati S, Jajodia S, Paraboschi S, Samarati P. Balancing confidentiality and efficiency in untrusted relational DBMS, Proceedings of the 10th ACM conference on computer and communications security, 2003.

[Den00] D. E. Denning, Information Warfare and Security, Addison-Wesley, 2000.

[Den84] Denning, D., Cryptographic Checksums for Multilevel Database Security, 1984.

[DES] DES, Data Encryption Standard. FIPS PUB 46, 1977.

[DGM+00] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine, Authentic third-party data publication, In IFIP Workshop on Database Security, 2000.

[DHV03] J. Deepakumara, H.M. Heys, and R. Venkatesan, Performance Comparison of Message Authentication Code (MAC) Algorithms for Internet Protocol Security (IPSEC), Proc. Newfoundland Electrical and Computer Eng. Conf., 2003.

[DMS04] Dingledine, R., N. Mathewson, and Syverson P., Tor: The Second-Generation Onion Router, USENIX, 2004.

[DRD08] Changyu Dong, Giovanni Russello, Naranker Dulay, Shared and Searchable Encrypted Data for Untrusted Servers, DBSec, 2008.

[Dwo04] Dworkin, M., Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, NIST Special Publication 800-38B, 2004.

[Dwo07] Dworkin, M., Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, NIST Special Publication 800-38D, 2007.

[ECC] The Case for Elliptic Curve Cryptography. National Security Agency, url = http://www.nsa.gov/business/programs/elliptic_curve.shtml.

[ECG+09] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres, Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines, transaction on computer science, 2009.

[ECL+07] Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin, Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks, in CHES, 2007.

[eStream] eStream website, url = http://www.ecrypt.eu.org/stream/

[Eur08] Eurosmart, Smart USB token, White paper, Eurosmart, 2008.

[EvG07] Sergei Evdokimov and Oliver Guenther, Encryption techniques for secure database outsourcing, Cryptology ePrint Archive, Report 2007/335, 2007, url = http://eprint.iacr.org/.

[EWS+04] Yuval Elovici, Ronen Waisenberg, Erez Shmueli, and Ehud Gudes, A structure preserving database encryption scheme, in Secure Data Management, 2004.

[FrH03] Frankel, H.Herbert, The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec, RFC, 2003.

[Gem] Gemalto. url = http://www.gemalto.com/france/

[GeZ07a] Tingjian Ge and Stan Zdonik, Answering aggregation queries in a secure system model, VLDB, 2007.

[GeZ07b] T. Ge, S. Zdonik, Fast, Secure Encryption for Indexing in a Column-Oriented DBMS, In: IEEE 23rd International Conference on Data Engineering, 2007.

[Goh03] Eu-Jin. Goh., Secure indexes, Cryptology ePrint Archive, Report 2003/216, 2003. url=http://eprint.iacr.org/2003/216/.

[GRS99] Goldschlag, D., M. Reed, and Syverson P., Onion Routing for Anonymous and Private Internet Connections, Communications of the ACM, 1999.

[GSW04] Philippe Golle, Jessica Staddon, Brent R. Waters, Secure Conjunctive Keyword Search over Encrypted Data, ACNS, 2004.

[GuJ09] Y. Guo et S. Jacob, Confidentialité et intégrité des grandes bases de données, Journées "Codage et Cryptographie", 2009.

[HCL+97] Haas, L. M., Carey, M. J., Livny, M. and Shukla, A. Seeking the truth about ad hoc join costs, VLDB Journal, 1997.

[HGX+08] Hou Fangyong, Gu Dawu, Xiao Nong, Tang Yuhua, Secure Remote Storage through Authenticated Encryption, International Conference on Networking, Architecture, and Storage, 2008.

[HHI+07] Hakan Hacigümüs, Bijit Hore, Balakrishna R. Iyer, and Sharad Mehrotra, Search on encrypted data, in Secure Data Management in Decentralized Systems, 2007.

[HIL+02a] H. Hacigümüs, B. Iyer, C. Li, S. Mehrotra., Providing database as a service, International conference on data engineering (ICDE), 2002.

[HIL+02b] Hakan Hacigümüs, Bala Iyer, Chen Li, and Sharad Mehrotra, Executing sql over encrypted data in the database-service-provider model, SIGMOD, 2002.

[HIM04] Hakan Hacigümüs, Balakrishna R. Iyer, and Sharad Mehrotra, Efficient execution of aggregation queries over encrypted relational databases, DASFAA, 2004.

[HMAC] National Institute of Standards and Technology (NIST), FIPS Publication 198: HMAC - Keyed-Hash Message Authentication Code, 2002.

[HMT04] Bijit Hore, Sharad Mehrotra, and Gene Tsudik, A privacy-preserving index for range queries, VLDB, 2004.

[Hsu08] Hsueh S, Database encryption in SQL server 2008 enterprise edition, SQL server technical article, 2008. url = http://msdn.microsoft.com/en-us/library/cc278098.aspx.

[IBM07] IBM corporation, IBM database encryption expert: securing data in DB2, 2007.

[IMM+04] Balakrishna R. Iyer, Sharad Mehrotra, Einar Mykletun, Gene Tsudik, and Yonghua Wu, A framework for efficient storage security in rdbms, EDBT, 2004.

[IwK03] Iwata, T., Kurosawa, K., OMAC: One-key CBC MAC, In Fast Software Encryption, 2003.

[LeM07] Lee, S. and Moon, B., Design of flash-based DBMS: an in-page logging approach, SIGMOD, 2007.

[Les08] Paul Lesov, Database Security: A Historical Perspective, University of Minnesota CS 8701, 2008.

[LHK+06] Li F., Hadjieleftheriou M, Kollios G, Reyzin L, Dynamic authenticated index structures for outsourced databases, SIGMOD, 2006.

[LHK+10] Feifei Li, Marios Hadjieleftheriou, George Kollios, Leonid Reyzin, Authenticated Index Structures for Aggregation Queries, ACM Transactions on Information and System Security, 2010.

[LiO05] Jun Li and Edward Omiecinski, Efficiency and security trade-off in supporting range queries on encrypted databases, DBSec, 2005.

[LiR99] Li, Z. and Ross, K. A. Fast joins using join indices. VLDB Journal, 1999.

[Mat04] Mattsson U, Transparent encryption and separation of duties for enterprise databases: a practical implementation for field level privacy in databases, Protegrity Technical Paper, 2004. url = http://www.protegrity.com/whitepapers

[Mer89] Ralph C. Merkle, A certied digital signature, in Proceedings on Advances in cryptology, CRYPTO, 1989.

[MND+04] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong and Stuart G. Stubblebine, A General Model for Authenticated Data Structures, Algorithmica, 2004.

[MNT04] Mykletun, E., Narasimha, M., Tsudik, G., Authentication and Integrity in Outsourced Databases, NDSS, 2004.

[MOV97] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997. url=www.cacr.math.uwaterloo.ca/hac.

[MSP09] Kyriakos Mouratidis, Dimitris Sacharidis, HweeHwa Pang, Partially materialized digest scheme: an efficient verification method for outsourced databases, VLDB Journal, 2009.

[MVS00] Umesh Maheshwari , Radek Vingralek , William Shapiro, How to build a trusted database system on un-trusted storage, Proceedings of the 4th conference on Symposium on Operating System Design & Implementation, 2000.

[NaT05] Maithili Narasimha, Gene Tsudik, DSAC: Integrity for Outsourced Databases with Signature Aggregation and Chaining, CIKM, 2005.

[NES03] NESSIE, Performance of Optimized Implementations of the NESSIE Primitives, Public Report, 2003.

[NIS01] NIST, Advanced Encryption Standard (AES), FIPS Publication 197, 2001.

[Ora00] Oracle Corporation, Database Encryption in Oracle 8i, 2000.

[Ora01] Oracle Corporation, Database encryption in Oracle 9i, technique white paper, 2001.

[Ora11] Oracle Corporation, Oracle advanced security transparent data encryption best practices, White Paper, 2011.

[OSC03] Gultekin Ozsoyoglu, David A. Singer, and Sun S. Chung, Anti-tamper databases: Querying encrypted databases, in Proc. of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security, 2003.

[Pac11] Packet General Networks, PCI-GENERAL™: A Secure MySQL Database Appliance, 2011, url = http://www.packetgeneral.com/.

[PaT04] H. Pang and K.-L. Tan, Authenticating query results in edge computing, ICDE, 2004.

[PJR+05] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, Kian-Lee Tan, Verifying Completeness of Relational Query Results in Data Publishing, SIGMOD, 2005.

[PZM09] Pang H, Zhang J, Mouratidis K, Scalable verification for outsourced dynamic databases, VLDB, 2009.

[RAD78] Rivest, R.L., Adleman, L., Dertouzos, M.L.: On Data Banks and Privacy Homomorphisms, In: DeMillo, R., Dobkin, D., Jones, A., Lipton, R. (eds.) Foundations of Secure Computation, Academic Press, 1978.

[RBB+01] P. Rogaway, M. Bellare, J. Black, T. Krovetz, OCB: A block-cipher mode of operation for efficient authenticated encryption, ACM Conference on Computer and Communications Security (CCS), 2001.

[Rij] Rijndael site. url=http://www.esat.kuleuven.ac.be/~rijmen/rijndael/.

[RSA02] RSA Security company, Securing data at rest: developing a database encryption strategy, Whiter paper, 2002.

[RSA78] R. L. Rivest, A. Shamir, L. M. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, 1978.

[SAB+05] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, C-Store: A column-oriented DBMS, VLDB, 2005.

[Saf11] Safenet, Database Protection, 2011, url = http://www.safenet-inc.com/products/data-protection/database-protection/.

[Sch] F. Schutz, Cryptographie et Sécurité, Cours. url=http://cui.unige.ch/tcs/cours/crypto

[Sch05] Bruce Schneier, Cryptanalysis of SHA-1, 2005. url = http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html.

[SHA3] NIST Computer Security Resource Center, cryptographic hash Algorithm Competition, 2010. url= http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[Sio05] Radu Sion, Query Execution Assurance for Outsourced Databases, VLDB, 2005.

[SVE+09] Erez Shmueli, Ronen Vaisenberg, Yuval Elovici, Chanan Glezer, Database encryption: an overview of contemporary challenges and design considerations, ACM SIGMOD Record, 2009.

[SWE+05] Erez Shmueli, Ronen Waisenberg, Yuval Elovici, Ehud Gudes, Designing Secure Indexes for Encrypted Databases, DBSec, 2005.

[SWP00] Dawn Xiaodong Song, David Wagner, Adrian Perrig, Practical Techniques for Searches on Encrypted Data, SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy, 2000.

[Syb08] Sybase Inc, Sybase adaptive server enterprise encryption option: protecting sensitive data, 2008. url=http://www.sybase.com.

[TDES] N. I. of Standards and Technology, Triple-des algorithm, FIPS 46-3, 1998.

[Ulr06] Ulrich Kühn, Analysis of a Database and Index Encryption Scheme - Problems and Fixes, SDM, 2006.

[Vin02] Radek Vingralek, GnatDb: a small-footprint, secure database system, VLDB, 2002.

[WaD08] Jieping Wang, Xiaoyong Du, LOB: Bucket Based Index for Range Queries, The Ninth International Conference on Web-Age Information Management, 2008.

[WBD+04] Brent Waters, Dirk Balfanz, Glenn Durfee, D. K. Smetters, Building an Encrypted and Searchable Audit Log, In The 11th Annual Network and Distributed System Security Symposium, 2004.

[WCK03] Wu, C., Chang, L., and Kuo, T., An Efficient B-Tree Layer for Flash-Memory Storage Systems. RTCSA, 2003.

[WDL+10] Jieping Wang, Xiaoyong Du, Jiaheng Lu, Wei Lu, Bucket-based authentication for outsourced database, Journal: Concurrency and Computation: Practice & Experience, 2010.

[Wes06] Dr. Alan F. Westin, Privacy and HER Systems: Can we avoid a looming conflict, At the Markle Conference on Connecting Americans to Their Health Care, 2006.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, Finding Collisions in the Full SHA-1, Crypto, 2005.

[XWY+07] Xie M, Wang H, Yin J, Meng X, Integrity auditing of outsourced data, VLDB, 2007.

[Yao77] Yao, S., Approximating block accesses in database organizations, Communication of the ACM, 1977.

[Yao78] Yao, A., On Random 2-3 Trees. Acta Informatica, 1978.

[YPD+08] Yang, Y., Papadopoulos, S., Papadias, D., Kollios, G., Spatial Outsourcing for Location-based Services, ICDE, 2008.

[YPM09] Yin, S., Pucheral, P. and Meng, X., A Sequential Indexing Scheme for flash-based embedded systems, EDBT, 2009.

[YPP+09] Yin Yang, Stavros Papadopoulos, Dimitris Papadias, George Kollios, Authenticated indexing for outsourced spatial databases, VLDB journal, 2009.