Beyond Just Data Privacy

Bob Mungamuru
Stanford University
bobji@stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

ABSTRACT

We argue that designing a system that "guarantees" the privacy of its information may not be enough. One must also consider the price for providing that protection: For example, is the information preserved adequately? Does the system perform well? We illustrate this point by presenting the concept of a *configuration* that can capture the security, longevity and performance aspects of managing information. Configurations can be useful for describing the policies used to safeguard information, as well as in selecting the right mix of security, longevity and performance levels.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Security, Integrity, Protection*
C.4 [**Performance of Systems**]: Reliability, Availability and Serviceability

General Terms

Design, Reliability, Security, Performance

Keywords

configurations, implementability, secret sharing, replication, encryption

1. INTRODUCTION

Information privacy and security are critical issues in today's high risk world. The news is full of stories of credit card numbers being stolen, patient records being misplaced, a photo agency being sued because they lost an artist's digital images, and governments collecting more data than they say they are. At the same time, there has been a lot of progress on fundamental techniques for sharing information while still protecting the privacy of individuals (e.g., [1, 2, 8, 10, 11]). For example, with k-anonymity techniques [10], we can release some data (aggregated in some way) so that the data of one individual is hard to identify. Several multi-party computation techniques have also been studied, where for instance two sites can join two relations without making the non-joining tuples known (e.g., [1]).

Nevertheless, we feel that many current research and development efforts in the area of information privacy and security are not fully considering other important factors such as:

- *Longevity.* Is the information also safe from hardware and software failures?
- *Performance.* Does the system that safeguards our information perform adequately?
- *Usability.* Are the privacy and longevity models intuitive and easy to manipulate?

To illustrate the tradeoffs between these factors, consider the following two extreme information systems. System A simply deletes all the data it receives. From the point of view of privacy, System A is perfect: there is no danger that information will be leaked or stolen. However, since it does not preserve data, it is not a very useful system. (One could also argue that System A's performance is excellent, as it answers all its queries extremely fast, always returning the null set for an answer.)

At the other end of the spectrum, consider System B that replicates the data it receives at many Internet storage sites. Making many copies is clearly good for longevity, since it would take many site failures to destroy our data. However, System B is weak in terms of privacy and security, since the more copies there are, the higher the probability of break-in or information leakage to an attacker.

Thus, we believe that in designing a secure information management system, one must consider all important factors, and ensure adequate levels along all dimensions. Developing an algorithm or system that makes very strong privacy or security guarantees, but that does not provide adequate performance, data longevity or usability is not enough.

A second weakness of many current privacy and security approaches, in our opinion, is that they view privacy and security as all-or-nothing. For example, a multi-party join algorithm (described briefly earlier)

3^{rd} Biennial Conference on Innovative Data Systems Research (CIDR 2007), January 7-10, 2007, Asilomar, California, USA.

will "guarantee" that no information that is not in the join result will be leaked to participants. No distinction is made between leaking one bit and leaking all of the database: both cases are considered leakage and are not permitted by the algorithm. Of course, the "guarantee" made by the algorithm is conditioned on certain strong assumptions, e.g., that the participants are "honest but curious" [1]. Of course, in practice, the assumptions may or may not hold, so there is an inherent probability that the privacy will be preserved. And one would expect that in the real world, there is a different probability associated with the loss of a few records than with the loss of the entire database.

We believe that privacy and security can be modeled as continuous variables, capturing the fact that data losses or leakages can be small or large, or can be likely or unlikely. Such models may allow us to better capture the tradeoffs between privacy and security and the other factors we discussed earlier. For example, we may be willing to "weaken" our security guarantees "a bit" in order to achieve a system that performs better or that provides higher data longevity. Of course, the challenge is how to capture notions such as "weaken" or "a bit" more precisely.

In summary, our position is that it is time to explore new information models and new metrics that make it possible to strike a good balance between the competing factors that arise when we try to safeguard data. Current research in the area of privacy and security has brought us excellent fundamental algorithms, but we can learn more about how these can be used in practice if we look at privacy and security as continuous variables that can affect other continuous variables such as performance, longevity and usability.

In the rest of this paper, we briefly summarize some initial work we have been doing at Stanford (as part of the PORTIA Project [3]), in order to capture the tradeoffs between security, longevity and performance in information systems. At the end of the paper, we mention some open problems that we believe can help explore these and other tradeoffs.

2. CONFIGURATIONS

Since we are interested in tradeoffs between longevity and security, we begin by defining a pair of simple data operators – *Copy* and *Split* – that directly impact these factors. A Copy operator makes n copies of its input, thus improving the longevity of the input data. A Split operator divides its input data into n "shares" so that all n shares are needed to reconstruct the input. Thus, a Split makes data leakage less likely, since n different shares must be obtained in order to get at the input data.

The Copy and Split operators defined here capture an extremely broad set of options for safeguarding data. Copy operators could represent anything from a simple tape backup to a complex peer-to-peer data trading scheme [4]. Split operators capture encryption with any number of keys, as well as XOR'ing with random bit-streams and other more exotic schemes. The generalization we describe in Section 2.1 is able to describe an even broader set of real-world techniques.

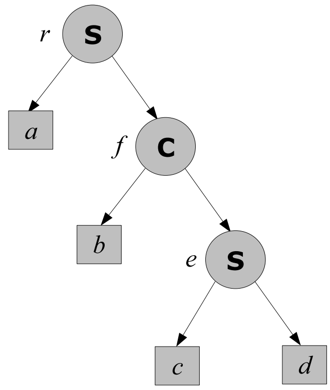


Figure 1: Example configuration.

A *configuration* is a composition of Split and Copy operators, used to achieve different levels of longevity and security. Figure 1 illustrates one possible configuration that might be used to safeguard a database. Our database is represented by the root r, which is initially split (say, using encryption) into shares a and f. After splitting r, two copies of f are made, labelled b and e. One copy, b, is materialized and stored. The other copy, e, is split again, this time say, by XOR'ing with a random sequence of bits. The random bit-sequence is stored at c, and the XOR'ed result is stored at d. The terminal vertices a, b, c and d at the bottom of the tree each represent a materialized data object, say, owned by users Alice, Bob, Carol and Dave. The non-terminals e, f and r represent transient data elements that are not materialized. In particular, the root r is not stored anywhere. Therefore, there is no single materialized data object that can be leaked that will cause the entire database r to be leaked.

A configuration defines how a database and its associated copies and shares (e.g., ciphertext, keys, bit-streams) are managed: a) the downward arrows tell us how the terminal data elements are derived from the root, and b) if we reverse all the arrows so that they point upward, we see which terminal data elements are needed to reconstruct the original database.

Also note that configurations are not restricted to be trees, but can be rooted directed acyclic graphs (DAGs). For example, consider Figure 2, where d and e are copies of the root data r. Here, the vertex b is *shared* by both d and e – it might represent a single encryption key that is used to encrypt both d and e. Thus, if Alice, Bob and Carol own data elements a, b and c respectively, then Bob has to collaborate with either Alice or Carol in order to access the decrypted database r. The terminal representing Bob's key has arrows from both Split operators, since his key is needed to reconstruct either copy of r.

Since a configuration describes how the database is preserved and secured, it is the key to understanding how the two factors of interest, longevity and security, interact (if we wished to study additional factors, we would need to develop a model that also captured the

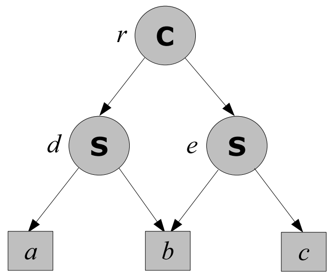


Figure 2: A configuration with sharing.

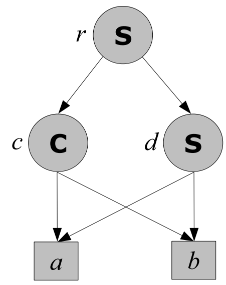


Figure 3: An unimplementable configuration.

other factors). Also important in understanding the tradeoffs is the observation that not all configurations "make sense", as we discuss in Section 3. For example, it is possible to compose Split and Copy operators in such a way that we violate the semantics of splitting (i.e., that all n shares are required to reconstruct). Or, we might unintentionally introduce "circularity" e.g., the encryption key used by a Split operator somehow depends on the value of the ciphertext it is supposed to generate. We refer to configurations that don't "make sense" as *unimplementable* and will typically want to avoid them in our designs.

In [7], we formalize the notions of operators, configurations and implementability. Here we briefly state some of the concepts that are useful for the rest of our discussion. A configuration Θ is comprised of a set of terminal vertices $\mathcal{T}$, and non-terminal vertices $\mathcal{N}$. We assume that the elements of $\mathcal{T}$ are labelled $a, b, c, \ldots$ and so on, and that the root is labelled r. Corresponding to any configuration Θ is a Boolean expression F_Θ, referred to as its *access formula*. F_Θ is constructed by recursively representing Copy operators as disjunctions and Split operators as conjunctions. F_Θ may include parentheses (i.e., it is a particular factorization) and is always monotone (i.e., no negation). For example, in Figure 1, we have $F_\Theta = a(b + cd)$. The *satisfying assignments* of F_Θ, denoted $\mathcal{S}(\Theta) \subseteq 2^{\mathcal{T}}$, tell us which terminals an attacker has to break into in order to reconstruct the sensitive data at the root, r. In Figure 1, $\{a, b\}$ and $\{a, c, d\}$ are satisfying assignments. Conversely, the *falsifying assignments* of F_Θ, denoted $\mathcal{F}(\Theta) \subseteq 2^{\mathcal{T}}$, are those subsets of terminals that, if destroyed, would make our sensitive data unrecoverable. In Figure 1, $\{a\}$ and $\{b, d\}$ are examples of falsifying assignments. It can be shown that the correspondence between a configuration and its access formula is one-to-one. As such, we will often represent a configuration Θ directly by its access formula F_Θ.

2.1 Secret Sharing

Copy and Split are special cases of a more general *secret sharing* operator. A k-of-n secret sharing operator, denoted $T^{k,n}$, decomposes data into n shares such that any $k \leq n$ are sufficient to reconstruct the data. The classic example of a $T^{k,n}$ operator would be Shamir's scheme [9]. A less obvious example would be RAID, where error-correction codes are used to distribute data across an array of disks, and failures of one or more of these disks can be tolerated without causing the data to be lost. A Split operator is simply $T^{n,n}$, and a Copy operator is $T^{1,n}$. Although we focus on Split and Copy operators in this paper, all of the ideas that we discuss can be readily extended to include the more general $T^{k,n}$ operator.

The behaviour of a $T^{k,n}$ operator is characterized by the m-*invertibility* property, which is formally defined in [7]. In short, the m-invertibility property implies that a $T^{k,n}$ operator has only $k-1$ degrees of freedom amongst the n shares it generates. For example, an encryption operator with two outputs (key and ciphertext) has just one degree of freedom, not two, since fixing the input data and choosing a key determines the value of the ciphertext. The reduced degrees of freedom, in turn, causes certain configurations to not "make sense", as we discuss next.

3. A TAXONOMY

As suggested in Section 2, there exist configurations that do not "make sense". As we will describe shortly, a configuration "not making sense" is a consequence of the m-invertibility property.

For example, consider $F_\Theta = ab(a+b)$, illustrated in Figure 3. The data we are safeguarding is represented by the root r, and split between the children c and d, say, using encryption. Now, suppose c is the encryption key and d is ciphertext. Then, c's children a and b (i.e., copies of c) will each be materialized copies of the key used to encrypt r. However, the vertex d is an encrypted version of r, which is re-split into a secondary key and ciphertext. Thus, one of d's children (either a or b) must be an encrypted version of d. But both a and b have already been designated as copies of c! We cannot, therefore, make a consistent assignment of keys and ciphertext to the vertices in the configuration.

The same inconsistency arises irrespective of whether c or d is the ciphertext. We cannot even resolve this issue using a different implementation of the Split operator (e.g., XOR). The problem with this configuration is that, due to the m-invertibility of the Split operators, only one child of r and one child of d can be freely chosen, and the other must be computed. There exists no consistent assignment of computed and free values to the operators in the configuration. It is in this sense that the configuration does not "make sense". It is not physically realizable using Split and Copy operators. We say that this configuration is *unimplementable*.

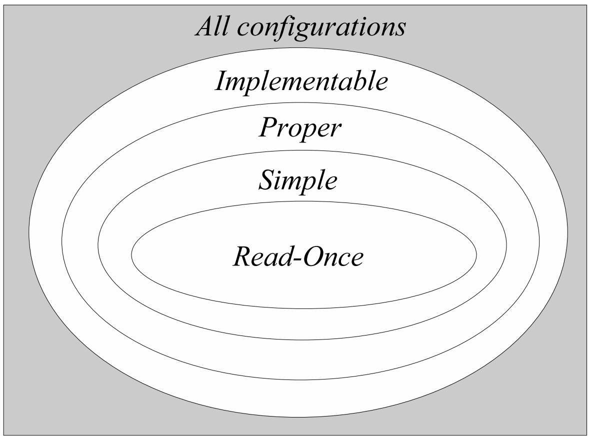


Figure 4: The space of possible configurations.

The configuration illustrated in Figure 1, on the other hand, does not present a similar problem. For example, we can choose a and c as the computed children of r and d, respectively. Alternatively, we can choose b and c, or b and d. Any of these assignments is consistent with the m-invertibility property. We say that the configuration in Figure 1 is *implementable*.

These examples suggest that the space of all possible configurations might be partitioned according to some notion of physical realizability. We can then impose further restrictions on the semantics and structure of our configurations, to arrive at a finer-grained classification scheme. In [7], one such taxonomy is presented, comprised of four nested subsets: *implementable*, *proper*, *simple* and *read-once*. Proper configurations are a subset of the implementable ones, wherein no two shares generated by an operator are constrained to be equal (since such a constraint would violate the intended semantics of the operator). Simple and read-once configurations have further structural properties that allow the size of the generated shares to managed effectively. Algorithms for verifying membership of a configuration in a given class are also provided in [7]. The resulting taxonomy is summarized in Figure 4.

Our taxonomy of configurations is analogous to the classification of transaction schedules, where the space of all schedules (configurations) is divided into desirable, serializable schedules (implementable configurations) and non-serializable ones (unimplementable configurations). Once the schedule space is understood, sub-classes can be determined (e.g., two-phase locking) that guarantee serializability and are easier to enforce in practice. In our case, we analogously identify sub-classes of implementable configurations (e.g., simple and read-once) that have more efficient membership tests. Having such efficient tests then makes it feasible for a design tool to search for good configurations that provide desired protection from data loss and/or break-ins.

When choosing a configuration to safeguard our data, we will always want to use atleast an implementable configuration. It can be shown experimentally, however, that implementability is a highly selective property, in the sense that only a small proportion of the space of possible configurations is implementable. Thus, it is important to check that any configuration that we design (say, using the techniques of Section 5) is actually implementable.

4. METRICS

Now that we can describe different privacy-longevity configurations, the next question is: how does one design systems that not only provide good privacy, but also good longevity, performance and usability? However, it is presently unclear how we might evaluate the effectiveness of a given configuration along such dimensions. How might we measure privacy, longevity and performance? How might we specify our system requirements under such measures? Ideally, we would see a "continuum" of good systems as we tradeoff continuously between these measures, as opposed to just the two extremes that were described in Section 1. For example, we might slightly weaken privacy guarantees in exchange for substantially improved longevity, or perhaps spend more on resources in exchange for both improved privacy and longevity. We now briefly describe some metrics over the space of configurations, which capture these dimensions (see [6] for more detail). In Section 5 we discuss how we might tradeoff between these metrics.

4.1 Probabilities of Failure

One possible continuous measure of the privacy and longevity provided by a configuration is *failure probabilities*, namely the probability of break-in and the probability of data loss. The *probability of break-in*, $P(\Theta)$, is the probability that an attacker breaks into enough terminal vertices to be able to reconstruct the root, r. Similarly, the *probability of data loss*, $Q(\Theta)$, is the probability that enough terminals are lost that we can no longer recover the data at r.

Consider the configuration $F_\Theta = ab + bc$ for example, illustrated in Figure 2. Let us assume that each of a, b and c is broken-into independently with probability $\frac{1}{4}$. An attacker wishing to reconstruct r must do one of three things. He must either break into terminals a and b only, or terminals b and c only, or all three of a, b and c. Thus, the probability of data loss will be the sum of probabilities of these three mutually exclusively outcomes i.e., $P(\Theta) = 2\left(\frac{1}{4}\right)^2 \frac{3}{4} + \left(\frac{1}{4}\right)^3 = \frac{7}{64}$. Similarly, an attacker must destroy any of the following sets of terminals to cause r to be lost: $\{b\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$ or $\{a, b, c\}$. Assuming the attacker destroys each terminal independently with probability $\frac{1}{4}$, we sum over the probabilities of these five outcomes to find $Q(\Theta) = \frac{1}{4}\left(\frac{3}{4}\right)^2 + 3\left(\frac{1}{4}\right)^2 \frac{3}{4} + \left(\frac{1}{4}\right)^3 = \frac{19}{64}$.

Formally, we define a pair of independent probability spaces $(\Omega_P, \mathbb{P})$ and $(\Omega_Q, \mathbb{Q})$, which represent an attacker's attempts to reconstruct and destroy our data, respectively. Ω_P and Ω_Q are referred to as *sample spaces*. The outcomes $\omega \in \Omega_P$ are subsets of terminals that the attacker manages to break into. Elementary outcomes $\omega \in \Omega_Q$ are subsets of terminals that are destroyed by the attacker. Thus, $\Omega_P = \Omega_Q = 2^{\mathcal{T}}$. $\mathbb{P}$ and

$\mathbb{Q}$ are discrete probability measures over events in Ω_P and Ω_Q, respectively, so that $\sum_{\omega \in \Omega_P} \mathbb{P}(\omega) = 1$ and $\sum_{\omega \in \Omega_Q} \mathbb{Q}(\omega) = 1$. Finally, we have $P(\Theta) \equiv \mathbb{P}(\{\omega \in \mathcal{S}(\Theta)\})$ and $Q(\Theta) \equiv \mathbb{Q}(\{\omega \in \mathcal{F}(\Theta)\})$.

The physical meaning of $\mathbb{P}$ and $\mathbb{Q}$ is as follows. $\mathbb{P}$ and $\mathbb{Q}$ describe an experiment that lasts a fixed period of time, say, ten years. We wish to answer questions such as: what is the probability that our data will still be available ten years from now? Or, how likely is it that no break-ins occur over the next ten years? The answers to these questions (i.e., $P(\Theta)$ and $Q(\Theta)$) depend on the ten-year security and reliability characteristics (i.e., $\mathbb{P}$ and $\mathbb{Q}$) of the terminals across which our data is distributed.

We will choose "good" configurations by solving the following problem: Given $\mathcal{T}$, $\mathbb{P}$ and $\mathbb{Q}$, find the "best" Θ. That is, given a set of physical resources, and knowledge of their failure characteristics, what is the configuration that best utilizes these resources?

4.2 Depth

A configuration's *depth*, $D(\Theta)$ is the maximum number of vertices between the root and any of the terminals. The depth is related to performance, since it is the number of processing steps needed to compute the terminal data elements from the original data. For example, the configuration in Figure 1, has depth three. We discuss performance in more detail in Section 6.

4.3 Non-Terminals

The *number of non-terminal vertices*, $N(\Theta)$, is a measure of the computational resources required in computing the terminal data elements. It is a performance measure similar in spirit to measuring depth, although not exactly the same. A Split operator with, say, six children all of whom are Split or Copy operators would have a small depth (i.e., $D(\Theta) = 2$), but would still require seven operators total. Measuring depth alone would not capture this.

4.4 Class

As discussed in Section 1, within the space of all possible configurations, we can identify *classes* that have desirable semantic and structural properties. We will always require a configuration to be at least implementable, but sometimes we may wish to impose a stronger restriction (i.e., proper, simple or read-once). We denote by $\mathcal{C}(\Theta)$ the class of a given configuration.

4.5 Terminals

The *number of terminal vertices*, $M(\Theta)$, is a measure of the physical storage required to deploy the configuration. When we search for good configurations, we will always impose an upper bound on $M(\Theta)$. Recall that in a configuration, only the data at the terminal vertices is stored physically. Thus, a bound on $M(\Theta)$ can be thought of as a resource constraint.

4.6 Groups

Finally, we may stipulate that certain *groups* must be allowed to reconstruct the data. We refer to these as *allow groups*. For example, we may require terminals a and b to be together sufficient to reconstruct the data. Such a statement is equivalent to requiring that $\{a, b\} \in \mathcal{S}(\Theta)$. We may also stipulate that certain groups, referred to as *deny groups*, be denied the ability to reconstruct the data. For example, breaking into c and d should not be sufficient to reconstruct the root. Such a statement is equivalent to specifying that $\mathcal{T} \setminus \{c, d\} \in \mathcal{F}(\Theta)$ (the '\' denotes set difference). As an illustration, one possible configuration that meets these requirements is shown in Figure 1.

5. OPTIMIZATION

We now return to the task of searching for good configurations, and exploring the tradeoff between security and data longevity. As suggested in Section 1, it does not make sense to simply search for the "best" configuration. The best possible (non-trivial) configuration for privacy is simply a Split, but it is the worst for longevity. Similarly, the best configuration for longevity is a Copy, but it is worst for privacy. Moreover, we can do arbitrarily well along either of these dimensions by simply using unbounded numbers of terminals! A better question to ask would be: subject to some minimum level of privacy, and an upper bound on the number of terminals, which is the configuration that provides us the most longevity? Using the metrics introduced in Section 4, we can write down the following optimization problem:

$$\begin{array}{ll} \min\limits_{\Theta} & Q(\Theta) \\ \text{s.t.} & P(\Theta) \le P_0 \\ & \{\omega_0^s, \omega_1^s, \dots\} \subseteq \mathcal{S}(\Theta) \\ & \{\omega_0^f, \omega_1^f, \dots\} \subseteq \mathcal{F}(\Theta) \\ & M(\Theta) \le M_0 \\ & N(\Theta) \le N_0 \\ & D(\Theta) \le D_0 \\ & \mathcal{C}(\Theta) \in \mathcal{C}_0 \end{array} \tag{1}$$

Here, P_0 is an upper bound on $P(\Theta)$ that indicates the highest probability of break-in we are willing to tolerate. M_0, N_0 and D_0 are our constraints on the various metrics introduced in Section 4. The sets $\{\omega_i^s\}$ and $\{\omega_i^f\}$ are the allow and deny groups, respectively, as described in Section 4.6. $\mathcal{C}_0$ is the class that we require our configuration to fall into, as discussed in Section 3. The set of physical terminals $\mathcal{T}$ and their failure characteristics $\mathbb{P}$ and $\mathbb{Q}$ (which are needed to compute $P(\Theta)$ and $Q(\Theta)$) are known beforehand. In (1), we are maximizing longevity by minimizing $Q(\Theta)$, the probability of data loss. Note that we could have, instead, maximized privacy (i.e., by minimizing $P(\Theta)$) subject to some minimum longevity requirement.

Thus, we can explore the tradeoff space between security and longevity by varying the constraints in (1), re-solving the problem, and seeing which systems we get. For example, Figure 5 is a plot of the $Q(\Theta)$ (i.e., longevity) we can achieve at various P_0 values (i.e., lower bounds on privacy), for a particular scenario with four terminals. This type of graph illustrates how we might sacrifice "a bit" of privacy for a relatively large gain in data longevity.

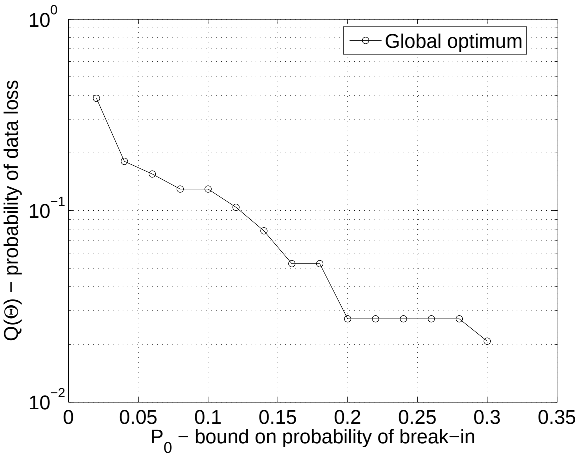


Figure 5: A privacy-longevity tradeoff curve.

In principle, this completes the task of designing good configurations. If we could solve the optimization problem in (1) exactly, then we would be done. Of course, the exact solution of (1) is extremely difficult due to the enormity of the space of possible configurations. Therefore, we must resort to approximate techniques for good solutions to (1). In [6], we formulate the problem in (1) more rigorously, and provide efficient techniques for its solution.

6. PERFORMANCE

In Sections 2 through 5, we illustrated how two important aspects of information systems, namely data privacy and data longevity, can be managed jointly. We now propose how our framework might be extended to include performance considerations. The basic idea is that, while our choice of configuration (i.e., composition of Split and Copy operators) is a tradeoff between longevity and privacy, our choice of how we implement Split operators involves a tradeoff between performance and privacy, and our choice of Copy operator implementations is a tradeoff between performance and longevity.

The tradeoff between performance and privacy is best illustrated through examples. Consider the Split operator at vertex r in Figure 1. Here, the data at r (of size n, say) is being split into two shares, a and f. Suppose the Split is implemented by XOR'ing r with a randomly generated n-bit sequence, stored at a, sending the XOR'ed result to f. XOR'ing in this manner provides perfect secrecy in an information-theoretic sense. However, in spite of XOR being a very fast operation, we now have $n + n = 2n$ bits of data to manage, which impacts the performance of subsequent operations. Suppose instead that we implement the Split at r using a stream cipher such as 3DES. If a is the ciphertext and f is a 128-bit key, we now have just $n + 128$ bits to manage, which is roughly half of $2n$ – a large savings. The price we pay is that while some stream ciphers are fast, it is not as fast as XOR'ing, and moreover we get only 128 bits of security (instead of n bits).

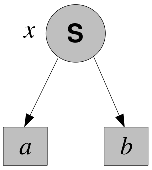


Figure 6: A Split operator.

In both the XOR and stream cipher cases, if the encoding is done on an entire relation, query performance is greatly degraded since we must reconstruct the entire relation before running any queries. One way to alleviate this problem is to instead use a block cipher to encrypt the fields (or rows) of a relation individually, as opposed to the entire relation as a blob. Execution of queries over encrypted data has been studied in the past (e.g., [5]). The disadvantage is that some information is leaked when encryption is done at a finer granularity. Moreover, block ciphers are typically slower than both stream ciphers and XOR'ing.

Thus, the decision of how to implement each Split operator is a delicate balance between performance and privacy considerations. Similar examples can be given to show that choosing between Copy operator implementations involves a tradeoff between performance and longevity. For brevity, we will focus only on Split operators here. In general, to get better performance, we pay a privacy penalty.

In the remainder of this Section, we present three small examples that raise some interesting questions about how we might quantify the tradeoff between performance and privacy.

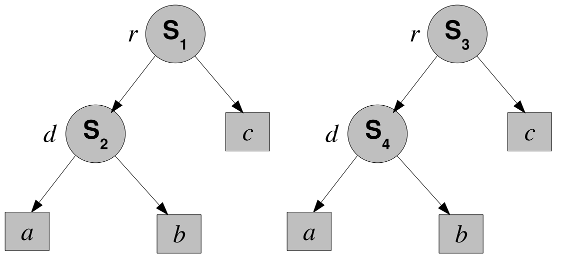
6.1 Possible Metrics

Our first example is the Split operator illustrated in Figure 6, where the data x is decomposed into shares a and b. We denote by p_a and p_b the probabilities of break-in of a and b, and assume that break-ins occur independently. In Section 4 we computed p_x, the probability of break-in for x, as $p_x = p_a p_b$. Implicit in this computation is what we might call the *perfect split* assumption. That is, if an attacker does not break-into both terminals a and b, the probability that he infers x is assumed to be zero.

Suppose that the Split at x is implemented using an encryption algorithm that is known to be breakable (e.g., DES with 56-bit keys), with the ciphertext stored at a and the key at b. Then, even if an attacker obtains only a, there is actually some non-zero probability that he will infer x via a brute-force search over the domain of b. Thus, we would really like to compute the probability of break-in at x as:

$$\begin{aligned} p_x &= p_a p_b \\ &+ p_a(1-p_b)\tilde{P}_b \\ &+ p_b(1-p_a)\tilde{P}_a \\ &+ (1-p_a)(1-p_b)\tilde{P}_{ab} \end{aligned} \quad (2)$$

Here, $\tilde{P}_a$, $\tilde{P}_b$ and $\tilde{P}_{ab}$ are respectively the probabilities



(a) S_1 and S_2 (b) S_3 and S_4

Figure 7: Different Split implementations.

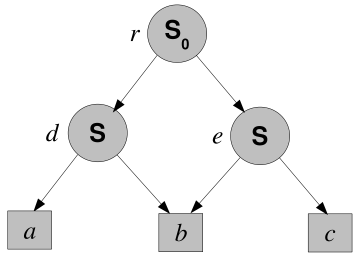


Figure 8: An insecure Split, S_0.

that an attacker will be able to guess a, b and $\{a, b\}$ in spite of not being able to break into to those terminals. In the DES example, we expect $\tilde{P}_a \approx 0$ and $\tilde{P}_{ab} \approx 0$, but not $\tilde{P}_b$. That is what makes DES "weak".

The probabilities $\tilde{P}_a$, $\tilde{P}_b$ and $\tilde{P}_{ab}$ are dependent on the strength of the Split operator implementation selected for x, the compute power available to the attacker, the time frame we are looking at, and so on. With the added terms in (2), we have effectively relaxed the assumption that splits are perfect. In particular, operator strength has a large impact – if we had used 3DES with 256-bit keys, $\tilde{P}_b$ would be negligible for any attacker and time frame of practical interest.

An interesting problem in this context is to find a good metric for operator strength. One possible measure of strength might be the *joint entropy* of the free children of x (i.e., b), as this captures the remaining uncertainty after an attacker steals a. It is thus a measure of the value of a "partial break-in". Another interesting question is on the functional dependency of $\tilde{P}_a$, $\tilde{P}_b$ and $\tilde{P}_{ab}$ on operator strength, attacker strength and time frame. What is the form of this dependency? What other the other factors that impact operator strength?

Consider, as a second example, the configurations shown in Figures 7(a) and 7(b). The configurations are the same composition of Split operators, but Figure 7(a) uses implementations S_1 and S_2, whereas Figure 7(b) uses implementations S_3 and S_4. We would like to know the impact on performance and privacy of the each choice. In particular, we focus on the performance of two basic data operations – a reconstruction and a decompostion of r.

Suppose we are trying to reconstruct r using shares a, b and c. To do so, we must first compute d (from a and b), and subsequently r (using d and c). Assuming that the computation at r cannot start until d is finished, these computations must be performed in sequence. Thus, the total time T_R spent reconstructing r is $T_R = t_R(d) + t_R(r)$, where $t_R(x)$ is the execution time of a reconstruction at vertex x. Similarly, the time T_D needed for a decomposition is $T_D = t_D(d) + t_D(r)$, where $t_D(x)$ is the time spent decomposing data at vertex x.

Clearly, $t_R(x)$ and $t_D(x)$ will depend on the operator implementation at x, as well as the sizes of the inputs and outputs to x. What is the functional form of this dependence? If the share d computed by S_1 in Figure 7(a) (or S_3 in Figure 7(b)) is large in size, then the operator S_2 (resp., S_4) will have a larger input to operate on, which has an impact on performance. How do we account for this performance impact when choosing between the configurations in Figures 7(a) and 7(a). What are the other factors that impact operator performance? Other than reconstructions and decompositions, what other data operations should we be considering?

Moreover, there are combinations of operator implementations that should not be used together, for practical reasons. For example, we may want to avoid "double encryptions" i.e., if S_1 is 3DES, we may not want to use 3DES again at S_2. This is conceptually similar to the implementability issue discussed in Section 3, where certain compositions of Split and Copy operators did not "make sense". How do we capture this constraint in our optimization problem?

Our third example illustrates a privacy-performance tradeoff that often arises in practice. As mentioned earlier, we can encode an object either in its entirety, or piecemeal. For instance, a relation can be encrypted in a single operation, or one tuple at a time. While the latter approach can lead to better performance, the data may be less secure, especially if the same key is used to encrypt all the tuples.

Consider the configuration shown in Figure 8. The data at r is Split into shares d and e, which are each then Split again into $\{a, b\}$ and $\{b, c\}$ using b as a shared free child. The interesting twist is that the Split S_0 at r is *insecure* e.g., it is a vertical partitioning of the attributes of a relation, or it is a horizontal partitioning of the rows. There is no encoding of data done in an insecure Split. While obeying the semantics of splitting (i.e., both c and d are needed to fully reconstruct r), there is now significant value to knowing just one of c or d, even without knowing the other share. Until now, we have assumed that individual children of a Split have zero value.

Insecure splits have clear performance benefits. Suppose we want to update the value of some data in c. We don't need to touch a to do so. Thus, our performance measure for an update of c does not need to include $t_R(e)$ or $t_D(e)$. We therefore need a system-level performance measure that account for the fact that S_0 is an insecure Split. Perhaps, in this case, we should use a weighted sum of $t_R(d)$ and $t_r(e)$.

At the same time, we have weaker privacy as a result of using the insecure Split. While before there was no value associated with having just c or d alone, there is now significant damage done if an attacker obtains either one of c or d. We may want to combine the probabilities of an attacker inferring c, d and r, perhaps using another weighted sum. What is the correct combination to use? In general, how do we combine performance and privacy measures in a hierarchy involving multiple insecure splits? What is the correct value to attach to a "partial leak" of data (i.e., only one of c or d)?. How do we compare the configuration in Figure 8 to another configuration where there is no insecure Split e.g., if the entire relation is encrypted as a blob? As we discussed, the performance is likely to be worse whereas the privacy will be better.

To balance performance and privacy, we can take an approach that is similar in spirit to Sections 4 and 5. We would first use our metrics to measure the impact of different operator implementation choices on the performance and privacy of a system. Then, since performance and privacy are competing objectives, we can fix a minimum privacy requirement and find the best performing system that meets this requirement (or vice versa). Whereas before we fixed the set of physical resources and searched for the best configuration, now we fix a configuration and a set of implementation options, and search for the best choice of implementation for each vertex.

7. DISCUSSION

We have illustrated how two important aspects of information, its security and its longevity, can be *jointly* modeled and evaluated. We have suggested how performance aspects can also be captured.

We envision that configurations could be used in a system design tool that lets users select their strategy for safeguarding information. For example, the tool could provide a GUI where users could build and annotate configurations, describing where their data is stored (terminals), how it is processed (non-terminals), and what systems and people are responsible for the different components. The tool could check for implementability, warning the user if the configuration has flaws, and perhaps suggesting alternatives that provide similar features but do not have problems. The tool could also compute the performance and strength of various proposed implementation choices for the non-terminal vertices.

Another approach would be for the user to define constraints, e.g., how many terminals are desired, what groups of users require access to which information (see Section 4), how much execution time we are allowed. The design tool could then run some of the optimization procedures described in Sections 5 and 6, and suggest one or more configurations to the user. The users could be presented with tradeoff curves like the one in Figure 5 that quantify the "price" that must be paid to obtain desired levels of privacy, longevity and performance.

Of course, the next step is to flesh out the details of our framework to account for the performance-privacy tradeoff, as well as usability and other issues. For example, how do we formulate and efficiently solve the optimization describing the privacy-performance tradeoff? What is the equivalent formulation for the longevity-performance tradeoff? As for modeling usability, there is also a lot of work ahead. Configurations, as we have presented them here, describe how *one* object (file, database) is protected. They need to be extended or re-designed to handle multiple objects, of different granularities, and possibly forming hierarchies or related in other ways. Other concepts such as roles and permissions need to be incorporated. The framework described in Sections 4 and 5 may provide a partial answer. For example, through the failure distributions $\mathbb{P}$ and $\mathbb{Q}$ described in Section 4, we can express a many-to-many relationship between data objects and users. Allow and deny groups represent an access control list. However, our model needs to be extended to fully and more naturally encode a richer set of security constructs.

8. ACKNOWLEDGEMENTS

This work was partially supported by NSERC and by the NSF through the PORTIA project.

9. REFERENCES

[1] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. SIGMOD*, 2003.

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

[3] D. Boneh, J. Feigenbaum, A. Silberschatz, and R. N. Wright. Portia: Privacy, obligations, and rights in technologies of information assessment. *IEEE Data Engineering Bulletin*, 27, 2004.

[4] B. Cooper and H. Garcia-Molina. Peer-to-peer data trading to preserve information. *ACM Transactions on Information Systems*, 20(2):133–170, Apr. 2002.

[5] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proc. SIGMOD*, 2002.

[6] B. Mungamuru, H. Garcia-Molina, and S. Mitra. How to safeguard your sensitive data. In *Proc. IEEE SRDS*, 2006.

[7] B. Mungamuru, H. Garcia-Molina, and C. Olston. Security configuration management. *Stanford InfoLab Technical Report*, 2005.

[8] P. Samarati. Protecting respondent's privacy in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[9] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[10] L. Sweeney. k-anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[11] V. S. Verykios, E. Bertino, I. N. Fovino, L. P. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in privacy preserving data mining. *SIGMOD Record*, 33(1):50–57, 2004.