Configurations: Understanding Alternatives For Safeguarding Data

Bob Mungamuru
Stanford University
bobji@stanford.edu

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

Christopher Olston
Carnegie-Mellon University
olston@cmu.edu

Abstract

Configurations are introduced as a new model for the description and analysis of secure data systems. Both the longevity and privacy of sensitive data are considered. The model uses two basic operators: Copy, which replicates data for longevity, and Split, which decomposes data into “shares” (e.g., ciphertext and a key) for privacy. The operators can be recursively composed to describe how data and their associated copies and shares are managed. Various classes of configurations are defined that have desirable properties with respect to physical realizability and semantic correctness. Formal techniques are provided to verify these properties for a given configuration.

1 Introduction

There are many ways to safeguard data from loss and unauthorized access, but there are two fundamental operations that cover many of the options: Copy and what we call Split. Making copies safeguards against the loss of data, while splitting data safeguards against unauthorized access. With a Split, data is “decomposed” into \( n \) shares (e.g., an encrypted version and keys), in such a way that all \( n \) shares are needed to reconstruct the original data, and access by an adversary to any proper subset of the shares is not a security breach. The Split operator can be implemented in many ways. For instance, in a 3-way Split (i.e., \( n = 3 \)), two shares can be randomly generated sequences of bits and the third share can be a tuple XOR-ed with the random sequences. From our point of view, a Split might even be a partition of the attributes of a relational database into subsets (see [1]), as long as each subset of attributes is not sensitive on its own, and only the combination of all attributes allows us to reconstruct the original database.

Copy and Split operators can be composed in interesting ways to describe how a database and its copies and shares are managed. To illustrate, consider Figure 1, which shows what we call a configuration.
The terminal vertices at the bottom of the tree represent materialized data, while the non-terminals represent data that is not materialized. The non-terminals are annotated with either an $S$ for a Split operation, or a $C$ for a Copy. In this configuration, a database containing sensitive data (represented by the root $g$) has been split into a materialized share $a$ and a non-materialized share $f$. For our illustration, let us say that $a$ is a publicly accessible encrypted version of database $g$ and that $f$ is the encryption key. We make two copies of key $f$: one copy $b$ is stored on a disk, whereas the other copy $e$ is split again into shares $c$ and $d$. We have labeled the terminals $b$, $c$ and $d$ with names (Bob, Carol and Dave) to indicate the person who “owns” the stored object. For instance, Bob owns one copy $b$ of the key with which $g$ was encrypted, and hence can reconstruct $g$ on his own. However, Carol and Dave cannot access $g$ on their own: they need to combine their keys to reconstruct the key $f$ required to decrypt $g$. This configuration can thus be useful if, say, Bob is the CEO of a company and should have access to the database on his own. However, the company wants to have another way to retrieve the data in case something happens to Bob (e.g., he forgets his key $b$, or the disk where the key is stored fails), but does not fully trust assistants Carol or Dave. Thus, shares of $f$ are given to Carol and Dave so that only together can they decrypt the database.

![Diagram of example configuration]

Clearly, many applications require both longevity and privacy for sensitive data. For example, most major financial institutions are required by law to retain records of financial transactions for a set number of years. At the same time, these financial records contain much personal information about customers, and therefore must be kept strictly confidential. A configuration would then describe how an institution replicates (Copy) and encrypts (Split) their databases, and distributes the resulting keys (shares) between their New York, London, Tokyo and Hong Kong data centers (terminals). Similarly, consider the US National Archives, which works to preserve government documents permanently, for the public record. At the same time, many of these government documents are classified, and must be held in secret until
it is time to declassify them. Thus, a configuration would describe how the Archives could ensure both longevity (via Copy operators) and privacy (via Split operators) for their enormous (and growing) digital document libraries. These are just two of the many real-world situations where both data privacy and longevity are needed, and where it would be useful to have a configuration that precisely describes how the data is safeguarded.

Note that the labels on terminals in Figure 1 are purely for illustration. In particular, observe that from a logical point of view, which share is “ciphertext” and which is an “encryption key” is not important. Similarly, a share is by no means restricted to be an encryption key, since a Split isn’t necessarily implemented using encryption – we use the word “share” generically here to mean the transient and materialized data objects that are derived from the original database, through Split and Copy operations. What is important is that the configuration defines how sensitive data and its associated copies and shares are managed: a) the downward arrows tell us how the terminal data elements are derived, 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 data.

Also note that configurations are not restricted to be trees, but can be rooted directed acyclic graphs (DAGs). For example, suppose we divide our employees into (possibly nondisjoint) groups, and deem that the database can only be accessed if all members of a group cooperate. Each employee is to be assigned a single share to keep track of for this purpose. Figure 2 shows a configuration that achieves the desired property, if the groups are: $G_1 = \{\text{Bob, Carol}\}$; $G_2 = \{\text{Carol, Dave, Elton}\}$. In this example, the original database $i$ is, say, encrypted with key $h$, of which two copies are made: $f$ and $g$. Each of $f$ and $g$ is in turn re-encrypted. For illustrative purposes, think of $b$ as the result of encrypting $f$ with key $c$, and $e$ as the result of encrypting $g$ with a combination of keys $c$ and $d$. We call $c$ a shared vertex because it has more than one parent in the DAG. In this example, the same data $c$ is used as input (during reconstruction) to both $S$ operators $f$ and $g$.

1.1 The Space of Configurations

We envision a GUI tool that would be used to create and manipulate configurations. The aim of this paper is to lay the groundwork for such a tool by understanding the space of options for configurations. In [6], we address the problem of efficiently searching for “good” configurations.

In particular, some configurations that arise naturally turn out not to “make sense”, whereas others have very desirable semantic and structural properties. For example, suppose we stipulate that cross-group cooperation (i.e., one member of $G_1$ and one member of $G_2$) is required to gain access to a particular piece of data. As before, each employee is to be assigned a single share for this purpose.
Figure 2: Configuration with sharing of keys.

Figure 3 shows a configuration adapted from the one shown in Figure 2 with the intent of enforcing the new rule. This configuration does not make sense, however, because sharing $c$ between $f$ and $g$ implies that $f$ and $g$ must be identical. There could be some Split operator that actually generates identical shares $f$ and $g$ (e.g., dividing $h$ by 2), or it could happen that for some specific value of $h$, the encryption key and the resulting ciphertext are identical. But we do not want to consider a configuration that only works in specific cases, or that forces us to use Split operators of questionable value (i.e., splitting $h$ into identical components does not provide much security, since just one of the shares $f$ and $g$ is sufficient to reconstruct $h$). Thus, we wish to rule out configurations like the one of Figure 3, which we classify as unimplementable.

Figure 3: Configuration that is not implementable.

A configuration design tool should automatically check for implementability. Given a configuration that turns out to be unimplementable, the tool could suggest alternative configurations. For example, the configuration of Figure 4 is logically equivalent to the one shown in Figure 3, and is implementable (logical equivalence of configurations is defined formally later – for now think of logically equivalent
configurations as ones that imply identical semantics for data reconstruction).

It is not difficult to show (see Section 4) that all tree-shaped configurations like the one in Figure 4 are implementable. This observation suggests a simple solution for dealing with unimplementable configurations: convert all configurations to trees. Unfortunately this naive approach turns out to be inadequate, for three reasons:

- In cases such as Figure 2, doing so would needlessly alter the user’s original configuration.
- Conversion to a tree may add extra en/decryption overhead, in some cases. For example, converting Figure 2 into a tree (not shown) necessitates introduction of an additional S-vertex layer.
- There exist implementable DAG configurations for which there is no logically equivalent tree configuration. An example can be constructed by adding a third group $G_3 = \{\text{Elton, Fred}\}$ to the configuration of Figure 2.

Hence, a more sophisticated understanding of the space of configurations is required.

In this paper we lay the foundations for a proper configuration design tool. The principal task is to study the space of configurations, and to give characterizations for them. This work is analogous to that on 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 (which we call 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.

1.2 Contributions

In summary, in this paper we study how a sensitive database and its derived copies and shares can be managed, through replication and further splitting, in order to safeguard against data loss and break-ins. In particular, we make the following contributions:

- We present a formal model for configurations, which describe alternatives for managing a data object and its associated copies and shares. The model allows recursive composition of the basic Copy and Split operators, with sharing of components.

- We define the notions of implementable, proper, simple and read-once configurations, and introduce properties that characterize the various subsets of configurations.

- We provide formal techniques to verify these properties for a given configuration.

- We describe how our framework can be used to design “good” configurations that meet a user’s privacy and longevity requirements, and best utilize available physical resources.

2 Related Work

Traditional research on safeguarding data generally considers privacy and longevity as separate issues. Furthermore, the work on data privacy (e.g., privacy-preserving data mining [11], k-anonymity [10], privacy policies [2]) often assumes that data within a database system (or file system) itself is safe, and that the only danger is when data is explicitly given to users or other systems. However, data within a database system can still be compromised (e.g., the physical disks can be stolen, the system can be hacked, the operator can be bribed, the disks can fail, the bits can rot). Moreover, some data is not even stored inside the database system (in particular, encryption keys are often stored outside database systems). Thus, configurations provide a “second line of defense”, safeguarding the underlying data from system-level compromises.

Threshold security schemes [12, 3, 9] (including secret sharing and ramp schemes) can be thought of as a generalization of our Split and Copy operators. A threshold operator $T^{k,n}$ splits data into $n$ shares – any $k$ shares can reconstruct the data ($k \leq n$), and fewer than $k$ shares do not leak any information. Our Split operator is simply $T^{n,n}$, while our Copy operator is $T^{1,n}$. Current work on threshold operators does not consider composition of operators, as we do here. Note that composition
is essential to describe asymmetrical scenarios like the one in Figure 2 (a single threshold operator cannot describe that situation), which is important in data management contexts. We discuss threshold operators in Appendix A, although the focus in this paper is on Split and Copy operators.

Our model uses Copy operators to provide longevity for sensitive data. There has been extensive work on this topic [7], and in particular on using replication as a means of ensuring longevity (e.g., [5, 8]). Most work in this area does not consider privacy issues. Conversely, most work on database privacy is not concerned with longevity issues. Our work considers both privacy and longevity, and studies ways of composing existing privacy and longevity primitives.

3 Basic Model

In this section, we formally introduce our model. Consider the case of a single logical data object, which is accessed as a unit (it is straightforward to extend our model to accommodate multiple independent data objects). The data object of interest has an associated configuration, \( \Theta \), which specifies how it is safeguarded. A configuration consists of a rooted directed acyclic graph (DAG), in which each vertex is labeled with the name of a data object; a vertex is said to carry a particular data object. Terminal vertices represent materialized data objects. The root vertex carries the original data object of interest, which is not materialized directly (unless the root comprises the entire tree).

Each non-terminal vertex (i.e., vertices having at least one child) is one of two operators: Copy (\( C \)) and Split (\( S \)). Copy operators (or, “\( C \)-vertices”) represent \( n \)-way replication of data, in which each of the \( n \) children carries an identical copy of the data object carried by the parent Copy operator. Split operators (or, “\( S \)-vertices”) denote splitting of data into shares, in which the shares carried by the \( n \) children taken together hold enough information to reconstruct the data object carried by the parent Split operator, and all \( n \) shares are required to reconstruct the parent data object successfully.

As discussed in Section 1, Split operators describe an extremely broad set of practical techniques for decomposing data into shares. Encryption is arguably the most popular implementation of a Split. XOR-ing with random bit-sequences is of additional theoretical interest since it guarantees information-theoretically secure splits – knowing any less than all of the children yields zero information about the parent. A Split operator’s behaviour is characterized by the 1-invertibility property:

**Definition 1.** An operator \( F \), over \( n \) inputs, is said to be 1-invertible if it has the following two properties:

1. Fixing the output and any \( n - 1 \) inputs allows us to uniquely determine the remaining one input.

2. Fixing all \( n \) inputs allows us to uniquely determine the output.
In other words, a 1-invertible operator has only \( n \) degrees of freedom (i.e., not \( n+1 \)) between its \( n \) inputs and one output.

As we see in Section 4, the 1-invertibility property is the root cause of certain configurations not making sense.

Consider a Split operator implemented by XOR-ing the parent data with \( n-1 \) randomly generated bit-sequences. The “inputs” in Definition 1 are the \( n \) shares (i.e., the \( n-1 \) random bit sequences and the XOR’ed sequence) whereas the “output” is the reconstructed parent data. Property 1 of Definition 1 says that, by fixing \( n-1 \) bit-sequences (i.e., \( n-1 \) inputs) and the parent data object (i.e., the output), we can uniquely determine what the XOR’ed sequence should be (i.e., the \( n^{th} \) input). Property 2 says that given the \( n-1 \) randomly generated bit-sequences and the XOR’ed sequence, we can uniquely reconstruct the original data at the parent. A similar description shows that encryption with \( n-1 \) keys, along with other more exotic Split implementations (e.g., [1]) all have this 1-invertibility property.

In Appendix A, we describe a generalization of Copy and Split operators (i.e., threshold or secret sharing operators) that captures an even broader set of practical techniques for safeguarding data. The behaviour of threshold operators is described by the \( q \)-invertibility property, which is a generalization of 1-invertibility. Although we will focus primarily on Split and Copy operators in this paper, we believe that all of the ideas that we discuss (e.g., configurations, composability, implementability) can be readily extended to include the more general threshold operator.

As suggested in the Section 1, a configuration represents two processes. If we take a top-down view (downward arrows), a configuration tells us how to break down the original data at the root to generate the (materialized) data at the terminals. If we take a bottom-up view, and reverse all the arrows (upward), the configuration tells us how to reconstruct the original data starting at the terminals. Although our diagrams show downward arrows, the reader should remember that during reconstruction, the data flows “up” the DAG.

The reconstruction aspect of a configuration can be represented by an access formula that shows how the terminal vertices are combined to yield the root. For example, the access formula for the configuration of Figure 1 is \( \Theta = a \land (b \lor (c \land d)) \). Thus, an access formula is a prepositional logic formula with terminal vertex labels as literals, Copy operators represented by logical disjunction, and Split operators represented by conjunction (no negation occurs, so all access formulae are monotonic). Note that disjunction and conjunction capture our intended safeguarding strategy – any one of the children of a \( C \)-vertex is sufficient to read the parent’s data, whereas all of the children of an \( S \)-vertex are required to read the parent.

A satisfying assignment of the access formula gives a set of materialized data objects that must
be accessed in order to reconstruct the root data object successfully. In our example, the satisfying assignment \( a = b = T, c = d = F \) tells us that \( a \) and \( b \) (the true values) can be used used to reconstruct root \( g \). Similarly, a falsifying assignment gives a set of materialized data objects whose elimination would render successful reconstruction of the root data object impossible. In our example, the falsifying assignment \( a = F, b = c = d = T \) tells us that loss of \( a \) makes it impossible to reconstruct \( g \). Two configurations are said to be logically equivalent if their access formulae are equivalent according to the rules of predicate logic. Surprisingly, as we will see in Section 5, configurations that are logically equivalent may actually have different properties regarding implementability and the protection they offer.

4 Classifying Configurations

We saw, through the examples in Section 1, that some configurations “make sense” and some do not. Those examples suggest that configurations might be classified according to some notion of semantic correctness. We now present one such classification scheme, and describe the properties that characterize each subset. The diagram in Figure 5 illustrates that the space of all possible configurations contains a subset that we call implementable. Further subsets of implementable are called proper, simple and read-once. Roughly speaking, we are interested in implementable configurations because they can be realized, regardless of the particular value of the data object to be encoded. Proper configurations are interesting because they accurately reflect the intended semantics of Split and Copy operators. The distinguishing characteristic of simple configurations is such that it can be very easily verified. Finally, the read-once configurations represent, in a structural sense, the simplest class of configurations. Simple and read-once configurations also allow the decomposition and reconstruction processes, respectively, to be executed efficiently.

In the subsections that follow, we provide a precise characterization of these classes of configurations. The experiments in Section 5.2 suggest that our taxonomy is, indeed, highly discriminating.

4.1 Implementable Configurations

We illustrate the notion of an implementable configuration through counterexamples. Consider the configuration shown in Figure 6. Its access formula is given as \( \Theta = (a \land b) \land (a \lor b) \). Our original data is carried at the root \( e \), and split between its children \( c \) and \( d \), say, using encryption. Now, suppose the encryption key is carried by \( c \); then, its children \( a \) and \( b \) (i.e., copies of \( c \)) will each be materialized copies of this same key. However, the vertex \( d \) carries \( e \)'s ciphertext, which is re-split into a secondary key and ciphertext. Thus, one of \( d \)'s children (either \( a \) or \( b \)) must carry \( d \)'s ciphertext. But both \( a \) and
b have already been designated as carrying encryption keys! We cannot, therefore, make a consistent assignment of keys and ciphertext to the vertices in the configuration, and it is in this sense that the configuration does not “make sense”. The configuration is said to be unimplementable.

![Diagram of configurations](image)

Figure 6: An unimplementable configuration.

The same problem arises in this example no matter which assignment of key and ciphertext we begin with, and irrespective of which implementation of Split operators we use. It is the configuration itself that is problematic. We can demonstrate this by studying the constraints implied by the configuration.

Begin by interpreting each labeled vertex \(a, b, c, d\) and \(e\) as a variable that holds the (stored or derived) data carried by that vertex. The Copy operator in Figure 6 implies that \(a = c\) and \(b = c\), i.e., the data objects carried by \(a, b\) and \(c\) must all be equal. Similarly, at each \(S\)-vertex, the 1-invertibility property implies that exactly one child must be designated as computed and all others as free. For example, at vertex \(d\), we must select a value for either share \(a\) or \(b\) (the free child) and use it (together with \(d\)) to compute the other share (the computed child). Moreover, the free child must not be constrained to have the same value as the computed child, since (as discussed in the Section 1) this would force us to use a restricted class of Split operators. We capture both of these constraints by the expression:
Let \( a = -1 \) and \( b = -1 \). Here we use “−1” as a placeholder for the value computed by the Split operator; it is not really the integer −1. Other Split operators will compute values −2, −3 and so on. The XOR symbol “⊕” (not to be confused with the bitwise XOR described earlier) tells us that only one of \( a \) or \( b \) can take on the computed value −1.

To summarize, the configuration of Figure 6 implies the following constraints:

\[
\begin{align*}
    a &= b = c \\
    (a = -1) &\oplus (b = -1) \\
    (c = -2) &\oplus (d = -2)
\end{align*}
\]

It is easy to see that these constraints do not have any feasible solution. If we chose \( a = -1 \), the equality constraint forces us to select \( b = -1 \), which violates the second constraint. If we chose \( b = -1 \) initially, we have the same problem. When there is no feasible solution to the constraints implied by a configuration, we say that the configuration is not implementable.

There is a second form of inconsistency that gives rise to unimplementable configurations. To illustrate, consider the configuration shown in Figure 7 (with access formula \( \Theta = a \land (a \land b) \land (b \land c) \land c \)).

This configuration implies the following constraints:

\[
\begin{align*}
    (a = -1) &\oplus (d = -1) &\oplus (e = -1) &\oplus (c = -1) \\
    (a = -2) &\oplus (b = -2) \\
    (b = -3) &\oplus (c = -3)
\end{align*}
\]

These constraints do have a feasible solution, for example \( d = -1, a = -2, b = -3, c = 1, e = 2, f = 3 \). As mentioned earlier, the negative numbers represent the vertices computed by the Split operators. The positive numbers, on the other hand, represent the free vertices whose value can be freely chosen by the
user. For instance, \( f = 3 \) represents the value of the root data object this configuration is protecting. The value \( c = 1 \) represents the share that is used by both the \( f \) and the \( e \) Split operators. Again, the actual numbers shown here (e.g., 1, 3) are not important; what is important is that we can choose values without violating the constraints.

Unfortunately, this solution still has a problem! The difficulty lies in the sequence in which the Split operators are invoked: Since \( d \) is computed by the Split at \( f \), the \( f \) Split must occur before the \( d \) Split. However, the \( f \) Split uses the value \( a \) computed by the \( d \) Split as a input, so the \( f \) Split must come after the \( d \) Split! This circularity makes the assignment \( d = -1, a = -2, b = -3, c = 1, e = 2, f = 3 \) problematic.

We can check for Split circularity by constructing a vertex dependency graph. Figure 8 shows the dependency graph for our sample solution. For each \( S \)-vertex we add arcs from the free variables to the variable computed by the Split operator. For example, for the Split operator at \( f \), \( d \) is the computed child i.e., \( d \) is the child that was used to satisfy the constraint for the Split \( (d = -1) \). Thus, we have arcs from \( f, a, e, c \) to \( d \). The cycle in this graph tells us that the solution is not sequentially consistent.

![Figure 8: Vertex dependency graph with \( d = -1, a = -2 \) and \( b = -3 \).](image)

Before we give up on the configuration of Figure 7 and declare it unimplementable, we need to check if there is some other feasible solution that is sequentially consistent. For instance, we may consider the solution \( a = -1, b = -2, c = -3, d = 1, e = 2, f = 3 \). The resulting vertex dependency graph is shown in Figure 9.

Once again, we notice that a cycle is formed, this time between the vertices \( a, b \) and \( c \). As such, this assignment is also sequentially inconsistent. Continuing in this manner, we find that every feasible solution is sequentially inconsistent. Therefore, we conclude that the configuration of Figure 7, too, is unimplementable.

As a final example in this section, consider the configuration of Figure 10 (with access formula
\[ \Theta = (a \land b) \lor (c \land d \land e) \]. In this case the constraints are as follows:

\[
\begin{align*}
    h &= f = g \\
    (a = -1) &\oplus (b = -1) \\
    (c = -2) &\oplus (d = -2) \oplus (e = -2)
\end{align*}
\]

For these constraints there are feasible solutions, for instance \( a = -1, c = -2, b = 1, d = 2, e = 3, f = g = h = 4 \). The corresponding vertex dependency graph is shown in Figure 11. Observe that we have merged \( h, f \) and \( g \) into a single vertex in the dependency graph, since the \( h = f = g \) constraint implies that \( \{h, f, g\} \) form an equivalence class. There are no cycles in this graph, and hence no sequential inconsistencies. As such, we conclude that this configuration is implementable.

The notion of implementability illustrated by our examples can be formalized as follows. We denote by \( V \) the set of all vertices in \( \Theta \). Set \( V \) is partitioned into \( N \), the set of non-terminal vertices, and \( T \), the set of terminal vertices. The non-terminals \( N \) are further partitioned into subsets \( S \) and \( C \), the sets of \( S \)- and \( C \)- vertices, respectively. We thus have \( V = N \cup T \) and \( N = S \cup C \). Observe that, for notational convenience, we are suppressing the dependence of these sets on the particular configuration \( \Theta \). Let \( m_S \), \( m_C \), \( m_T \), \( m_N \) and \( m \) denote the number of elements in \( S, C, T, N \) and \( V \), respectively. The function
Figure 11: Vertex dependency graph for $a = -1, c = -2$.

$p: V \rightarrow 2^N$ returns the parents of a given vertex $x$, and similarly, $c: N \rightarrow 2^V$ returns its children.

**Definition 2.** Given a configuration $\Theta$, the equality constraints implied by each $x \in C$ are:

$$\forall y \in c(x), \ y = x.$$  

The set $\{x \cup c(x)\}$ forms an equivalence class. The resulting set of constraints is denoted by $\delta_C(\Theta)$.

**Definition 3.** Given a configuration $\Theta$, the split constraint implied by each $x \in S$ is:

$$(y_1 = z(x)) \oplus (y_2 = z(x)) \oplus \cdots \oplus (y_n = z(x))$$

where $\{y_1, \ldots, y_n\} = c(x)$, and $z(x)$ is the value computed by the split operator. In our examples we used negative integers to represent computed values. The resulting set of constraints is denoted by $\delta_S(\Theta)$.

**Definition 4.** A solution $X: V \rightarrow Z$ assigns either computed values (negative integers in our examples) or free values (resp., positive integers) to each vertex in $V$.

**Definition 5.** Given a configuration $\Theta$, a solution $X$ is feasible if it satisfies $\delta_C(\Theta)$ and $\delta_S(\Theta)$. If $X$ is feasible, exactly one child of any $S$-vertex $x$ holds the computed value $z(x)$. For $x \in S$, we denote by $\alpha_X(x)$ the child that holds $z(x)$ and by $\beta_X(x)$ the set of all other children of $x$.

**Definition 6.** Given a configuration $\Theta$ and a feasible solution $X$, the vertex dependency graph, $G(X)$, is constructed as follows. Based on the equality constraints of $\Theta$, the vertices are partitioned into equivalence classes. Each equivalence class forms a node in $G(X)$. An arc exists from class $E_1$ to class $E_2$ iff there is an $x \in S$ such that $\alpha_X(x)$ is in $E_2$, and one of $\beta_X(x)$ or $x$ itself is in $E_1$.

**Definition 7.** A solution $X$ is sequentially consistent if $G(X)$ is acyclic.

**Definition 8.** A configuration is implementable if there exists a feasible solution that is also sequentially consistent. Otherwise, it is unimplementable.
Note that it is expensive to check whether a configuration is implementable. The first step is to assemble equivalence classes of vertices by visiting each \( C \)-vertex in the configuration. Then, for each \( S \)-vertex, we must designate one of its children as computed. If the \( i^{th} \) \( S \)-vertex has \( n_i \) children, then there are \( \prod_{i=1}^{m_s} n_i \) possible assignments (\( m_s \) was defined earlier as the number of \( S \)-vertices in \( \Theta \)). Finally, a solution, if feasible, must be checked for sequential consistency. This can be done using a depth-first search over the vertex dependency graph. In the worst case, each equivalence class will contain exactly one vertex from \( \Theta \), and so the dependency graph will have \( m \) vertices (\( m \) was defined earlier as the total number of vertices). Thus, the check for sequential consistency can be done in \( O(m + \sum_{i=1}^{m_s} n_i) \) time. In the worst case, each possible solution will be feasible but not sequentially consistent. Therefore, assuming it is relatively inexpensive to compute equivalence classes, checking implementability will require \( O((\prod_{i=1}^{m_s} n_i)(m + \sum_{i=1}^{m_s} n_i)) \) time.

### 4.2 Proper Configurations

As with the implementable configurations, we illustrate the distinguishing characteristic of proper configurations with a counterexample. Consider the configuration illustrated in Figure 12 (with access formula \( \Theta = (a \lor b) \land (b \lor c) \land d \)).

![Figure 12: An implementable configuration that is not proper.](image)

We can easily check that this configuration is implementable. Its equality and split constraints are:

\[
e = f = a = b = c
\]
\[
(e = -1) \oplus (f = -1) \oplus (d = -1)
\]

A feasible solution is \( d = -1, e = f = a = b = c = 1, g = 2 \), and one can see that this solution is also sequentially consistent. Thus, the configuration is implementable.

However, there is still something undesirable about this solution (or any other feasible solution). In particular, the equality constraint forces the two free children of the \( S \)-vertex, \( e \) and \( f \), to be equal, which is \textit{semantically} undesirable. That is, although the intent behind using the Split operator \( g \) was to
require all three shares \((e, f\) and \(d)\) in order to reconstruct the parent, we really only need either \(e\) or \(f\) along with \(d\), since \(e\) and \(f\) are known to be equal. This is akin to having an encryption scheme that uses the same key twice. An attacker that obtains \(d\) would only need to guess or steal one encryption key (either \(a, b\) or \(c\)) instead of two, in order to decrypt \(g\). More dramatically, if we were splitting \(g\) using XOR, and we knew \(e = f\), this would imply \(d = g\). That is, \(d\) would be an exact copy of the original database \(g\), offering zero privacy! As a result, we end up with less privacy in this configuration than we had intended. We call such configurations *improper*. 

Returning to the implementable configuration of Figure 10, we do not encounter such a problem. Recall that the constraints were \(h = f = g\), \((a = -1) \oplus (b = -1)\) and \((c = -2) \oplus (d = -2) \oplus (e = -2)\). Here there is a sequentially consistent feasible solution \((a = -1, c = -2, b = 1, d = 2, e = 3, f = g = h = 4)\) where the children of each \(S\)-vertex get different values. That is, we get to use different keys for each Split operator. We call such a configuration *proper*. One may prefer to use proper configurations because they capture the intended semantics of Split operators.

The following definitions formalize the notion of proper configurations.

**Definition 9.** Given a configuration \(\Theta\), the inequality constraints implied by each \(x \in S\) are:

\[
y_1 \neq y_2 \neq \cdots \neq y_n
\]

where \(\{y_1, \ldots, y_n\} = c(x)\). The resulting set of constraints is denoted by \(\delta_S(\Theta)\). Note that here the symbol \(!=\) is transitive e.g., when we write \(u \neq v \neq w\) we mean also that \(u \neq w\).

**Definition 10.** Given a configuration \(\Theta\), a solution \(X\) is strictly feasible if it satisfies \(\delta_S(\Theta) \cup \delta_S(\Theta) \cup \delta_S(\Theta)\).

**Definition 11.** A configuration is said to be proper if there exists a solution that is both strictly feasible and sequentially consistent. An implementable configuration that is not proper is called improper. Note that any proper configuration is implementable, by definition.

Returning to the example of Figure 12, the constraints used to check properness are now:

\[
e = f = a = b = c
\]

\[
(e = -1) \oplus (f = -1) \oplus (d = -1)
\]

\[
e \neq f \neq d
\]

We immediately see that the constraints require both \(e = f\) and \(e \neq f\), and hence will not permit any strictly feasible solution. We conclude that this configuration is *improper*. 


4.3 Simple and Read-Once Configurations

We define two further subsets of configurations here, which are of practical interest since their distinguishing properties can be established by a relatively easy test. Recall that, in general, a configuration is a DAG. A vertex is said to be shared if it has more than one parent; otherwise it is unshared. The class of simple configurations is defined as follows:

**Definition 12.** A configuration $\Theta$ is said to be simple if every $S$-vertex has at least one unshared child, and every $C$-vertex has no shared children. That is,

1. $\forall x \in S$, $\exists y \in c(x)$ such that $p(y) = \{x\}$;
2. $\forall x \in C$, $\forall y \in c(x)$, $p(y) = \{x\}$.

**Theorem 1.** Any simple configuration is proper.

The reader is referred to Appendix B for a proof of Theorem 1.

**Theorem 2.** Proper configurations are a strict superset of simple configurations. That is, there exists a configuration that is proper but not simple.

**Proof.** The configuration $\Theta = (a \land b) \lor (a \land b \land c)$, in Figure 13, is proper but not simple. It is not simple because all the children of the $S$-vertex $d$ are shared. Yet the solution $a = -1$, $c = -2$, $b = 1$, $d = e = f = 2$ is strictly feasible and sequentially consistent.

![Figure 13: A proper configuration that is not simple.](image)

The check for whether a configuration is simple can be done using a modified depth-first search. Each vertex we visit, we color red all the (initially blue) children, to indicate that at least one of its parents has been visited. Then, if at any point during the depth-first traversal, we encounter a non-terminal vertex whose children were all already red before we reached there, then the configuration is not simple. Thus, we can check if a configuration is simple in $O(m + \Sigma_{i=1}^{m} n_i)$ time. In particular, it is much easier to verify that a configuration is simple, compared to checking implementability or properness. In this
sense, simple configurations are analogous to transaction schedules that use two-phase locking – the easy procedure to verify simplicity (two-phase locking) can be used to establish the more general properties of properness and implementability (serializability).

Aside from offering lower evaluation complexity, simple configurations can also be useful when using implementations of Split operators such as encryption, where one child (the ciphertext) often requires more storage space than the other children (the encryption keys). In such cases, we could store the larger data object at the unshared child, and only share the smaller objects. Interpreting edges in the DAG as communication links, this strategy would minimize communication costs during the reconstruction process, since only small objects are sent redundantly to multiple parents.

The class of read-once configurations are defined next:

**Definition 13.** A configuration $\Theta$ is said to be read-once if it is a tree. That is, all vertices in $\Theta$ are unshared: $\forall x \in V, \forall y \in c(x), p(y) = \{x\}$.

An example of a read-once configuration is shown in Figure 10. The name “read-once” is a reference to the fact that each terminal vertex appears exactly once in the access formula. The test for a configuration being read-once is even easier than that for the simple configurations – just check that all vertices are unshared. It is trivial to show that the read-once configurations are a strict subset of the simple ones.

Once again interpreting the DAG edges as links, read-once configurations completely eliminate redundant communication in the reconstruction process, since every materialized and non-materialized data object is transmitted exactly once. This enables an efficient stack-based implementation of the reconstruction process. Moreover, read-once configurations are also of interest for theoretical reasons. For example, there are results proving tight bounds on the size of the access formulae of a read-once configuration (i.e., monotone read-once Boolean expressions) required to achieve certain privacy and longevity guarantees [4].

5 Discussion

We have argued that configurations can be the foundation of a design tool that allows users to define strategies for safeguarding data, as well as evaluate alternative strategies. 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 decomposed (non-terminals), and which systems and users are responsible for which data objects. The tool would check for implementability of the selected configuration, warning the user if the configuration has flaws, and perhaps suggesting viable alternatives that provide similar features.

Alternatively, a user could simply provide requirements and constraints, e.g., how many terminals
Recall from Section 3 that two configurations are logically equivalent if their access formulae are equivalent according the rules of logic. For example, the configurations $\Theta = a \land (b \lor (c \lor d))$ and $\Theta = (a \land b) \lor (a \lor c)$ are logically equivalent. In general, if $\Theta_1$ and $\Theta_2$ are logically equivalent, then a satisfying (falsifying) assignment of $\Theta_1$ will also satisfy (falsify) $\Theta_2$. However, other properties are not invariant across logically equivalent configurations, such as the depth of the DAG, and the number of non-terminal vertices. In particular, the classification of the configuration might change as a result of a logical transformation. As an extreme example, consider $\Theta = d \land (b \lor (a \land (c \lor e)))$. Table 1 shows five configurations that are logically equivalent to it, each falling into a different class of configurations.

This example clearly demonstrates that logical equivalence only refers to the reconstruction of data, not to the full configuration. That is, if two configurations $\Theta_1$ and $\Theta_2$ are logically equivalent, then data can be reconstructed in the same way, e.g., by combining data from the same stored nodes. However,
the way the stored data is generated by $\Theta_1$ and $\Theta_2$ may be different. In particular, the 1-invertibility property (which impacts how data is generated) may impact different items in $\Theta_1$ than in $\Theta_2$, causing $\Theta_1$ and $\Theta_2$ to fall into different classes. The notion of logical equivalence is crucial to our approach in [6], as we shall see in Section 5.3.

5.2 Relative Sizes of Classes

We have argued that our classification of configurations is analogous to the classification of schedules. With schedules, the size of a class is important. For instance, if a class like two-phase locking were very small (i.e., very few instances), then it would impact concurrency (there would be very few ways to safely interleave actions safely).

Similarly, one can also ask questions about the size of the classes for security configurations. For instance, say a particular security design tool only allowed simple configurations. Would we be ignoring a significant fraction of the design space? Would we be disallowing many configurations that could potentially provide better privacy-longevity-performance tradeoffs than the simple configurations? Also, say it were the case that the vast majority of the configurations are, for example, proper. That is, only a few pathological examples were unimplementable and not proper. Then the importance of our characterizations would be diminished, as the design tool we advocate would seldom rule out invalid configurations.

To address questions like these we developed an evaluation framework, with the goal of determining the relative sizes of the configuration classes. In particular, we focus on configurations that have $m$ total vertices and $m_T$ terminal vertices, and estimate the relative sizes of the implementable, proper, simple and read-once classes. Since the space of possible configurations can be very large for even moderate values of $(m, m_T)$, one must necessarily use sampling: Generate say 10,000 configurations, and test them to see in what class they fall.

The challenge, of course, is how to sample configurations. Ideally, one would like to pick 10,000 configurations proposed by users (or proposed by a design tool that is searching for a configuration with given properties), but this approach is clearly not feasible since there are no “users” when one is exploring a new direction. Instead, we settle for generating and testing “random” configurations, as detailed below. Since we are sampling over random configurations, it is important to keep in mind that our results will only be suggestive, giving us a sense of how large the classes are, but without making specific predictions about what a user or a design tool might encounter.

To generate random configurations, a uniform distribution was used over the set of rooted DAGs with $m_T$ terminal vertices, $m - m_T$ non-terminals, such that each non-terminal vertex had a minimum of two
children and was an $S$-vertex with probability $\frac{1}{2}$. (There are other options for generating configurations, but they are not discussed here.) Each randomly generated configuration was tested for implementability, properness, etc. In a first experiment, $m_T = 5$ was fixed, and $m$ varied between 6 and 10. In a second experiment, $m = 10$ was fixed, and $m_T$ was varied between 5 and 9. For each $(m, m_T)$ pair, 10,000 random configurations were tested.

The results of the experiments are shown in Figures 14 and 15. In each figure, the horizontal axis give the free parameter, and the vertical axis reports the fraction of configurations of a given type. For example, in Figure 14, with 5 terminal and 8 total vertices (i.e., 3 non-terminals), roughly 70% of the configurations were implementable, 32% were proper, 6% were simple, and just 0.3% were read-once. Similarly, in Figure 15, with 10 total vertices and 6 terminals (i.e., 4 non-terminals), 46% were implementable, 13% were proper, just 0.6% were simple and none were read-once.

![Figure 14: Classification of randomly drawn configurations with 5 terminal vertices (i.e., $m_T = 5$).](image1)

![Figure 15: Classification of randomly drawn configurations with 10 total vertices (i.e., $m = 10$).](image2)

The important conclusion here is that the implementability property is highly selective; in fact, as
shown by the data, with increasing numbers of non-terminal vertices, it becomes increasingly unlikely that a randomly drawn configuration is even implementable. It is even less likely that it will be proper, simple or read-once. This observation does not mean there are few simple of read-once configurations in absolute terms, only that there are many more of the other types. These results suggest that it is very important to have a configuration design tool that can avoid considering undesirable configurations. The results also suggest that allowing more than read-once (tree) configurations is good, since it greatly increases the number of options available, and thus hopefully increases the chances of finding a configuration that meets the user needs.

5.3 Designing and Evaluating Configurations

A design tool could help users search for desirable configurations. Given our understanding of configurations, we can now briefly discuss what this design process may look like. We present here a summary of the results in [6].

The first step is to select metrics that can quantify the longevity, privacy and performance offered by a configuration. One approach is to use failure probabilities i.e., the probability $P(\Theta)$ that our data gets lost or destroyed (longevity failure) and the probability $Q(\Theta)$ that our data is read by an attacker (privacy failure). Now, since only the terminals are materialized, they are the “units of attack”. That is, an attacker either destroys or breaks into individual terminals. Therefore, based on the access formula and the set of terminals that have “failed”, we can see if our original database has been read or destroyed by an attacker. Thus, given a joint failure distribution over the terminals, we can calculate the failure probabilities for the entire configuration.

The second step is to define an optimization problem. It can be shown that longevity and privacy are competing objectives – making copies helps longevity but creates more targets for break-ins, whereas splitting helps privacy but creates more objects that could potentially get lost or destroyed. Thus, our goal will be to find the configuration that minimizes the probability of data loss (or break-ins), subject to an upper bound on the probability of break-ins (resp., data loss), and a constraint on physical resources (number of terminals).

There may be other metrics of importance to us, which impose further constraints on our optimization problem. For example, the class of the configuration – we are typically interested only in implementable configurations, but in some cases a stronger restriction is desired. We may require that specific groups of users (i.e., terminals) be either allowed or denied access. For example, in Section 1, we described an example (Figure 1) where the groups \{Bob\} and \{Carol,Dave\} were allowed, but \{Carol\} and \{Dave\} were denied. We may also wish to constrain the depth of the DAG, since it impacts the amount of
computation required to decompose and reconstruct the original database.

The final step is to actually solve our optimization problem. Solving it exactly is intractable (except for small instances), since an exhaustive search through the space of configurations corresponds to searching through all factored monotone boolean formulae. Instead, we take a two-stage approach to finding a good approximate solution. In the first stage, we find a configuration $\Theta'$ that is optimum with respect to the metrics that are invariant across logical transformations. We show that these metrics include the probabilities of failure and the allow/deny groups. In the second stage, we try to find a configuration $\Theta^*$ that is logically equivalent to $\Theta'$ (thus implying that $\Theta^*$ remains optimal for the first stage), such that $\Theta^*$ satisfies the remaining constraints. This two-stage approach, although sub-optimum, can be shown to be much more efficient than a brute-force search. Intuitively, the efficiency arises because we are decoupling the search for a good truth table from the search for a good factorization. An exhaustive search would explore this space all at once. In [6], we formulate the optimization problem rigorously, and provide efficient techniques for its solution.

We can explore the tradeoff space between security and longevity by varying our constraints on various metrics, re-solving the optimization problem, and seeing which configurations we get. For example, Figure 16 is a plot of the probability of data-loss (i.e., longevity) we can achieve at various upper-bounds on the probability of break-in (i.e., lower bounds on privacy), in a scenario with four terminals, where break-ins and data loss each occur independently, with probability 0.2. The plot tells us, for instance, that if we can only tolerate a 10% probability of (system-level) break-in, we can achieve a 13% probability of data-loss. Alternatively, if we can tolerate a 20% probability of break-in, then there is a configuration that achieves a probability of data loss of just 2.7%. This type of graph illustrates how we might sacrifice “a bit” of privacy for a relatively large gain in data longevity.

![Figure 16: A privacy-longevity tradeoff curve.](image-url)
To summarize, our approach allows a designer to specify a set of requirements for her application. We then carry these requirements all the way through to a specific recommendation on how she should optimally safeguard her data. We achieve these results by first introducing failure probabilities as a measure of quality, and then using efficient approximation techniques to yield good solutions to an intractable optimization problem.

6 Conclusion

Privacy and longevity are both critical for sensitive, valuable data – it is important to consider both in designing a system to safeguard the data. In this paper we have introduced configurations as a “language” for describing how data is safeguarded, by composing simple Copy and Split primitives. We showed that the semantics of splitting implies the existence of a hierarchy of classes of configurations, with some classes containing configurations that were not practical, and others containing configurations that are easy to check for correctness. As we have discussed, we envision two immediate applications of this work. First, this framework allows an administrator to use a graphical interface to construct complex configurations by composing Copy and Split operators. The techniques in Section 4 are needed to ensure the correctness of his/her designs. Secondly, as described in Section 5.3 and [6], we may define metrics by which we measure the privacy and longevity of a configuration, and then formulate optimization problems against these metrics, which would yield configurations with desired properties. Our framework would be required, of course, in order to verify the feasibility of the computed configurations.

References


A Threshold Operators

Copy and Split are special cases of a more general threshold or secret sharing operator. A $k$-of-$n$ threshold operator, denoted $T^{k,n}$, decomposes data into $n$ shares such that any $k$ are sufficient to reconstruct the data, with $k \leq n$. 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.

The behaviour of a $T^{k,n}$ operator is characterized by the $q$-invertibility property:

**Definition 14.** An operator $F$, over $n$ inputs, is said to be $q$-invertible if it has the following two properties:

1. Fixing the output and any $n - q$ inputs allows us to uniquely determine all of the remaining $q$ inputs.

2. Fixing any $n - q + 1$ inputs allows us to uniquely determine the output and all of the remaining $q - 1$ inputs.


In other words, a q-invertible operator has only $n - q + 1$ degrees of freedom (i.e., not $n + 1$) between its $n$ inputs and one output. A $T^{k,n}$ operator is $(n-k+1)$-invertible.

In a configuration, when breaking down data (top-down), a $T^{k,n}$ operator fixes the parent (output) and “freely” chooses values for $k - 1$ shares (inputs). This allows the remaining $n - k + 1$ shares to be computed. We say that each $T^{k,n}$ operator in a configuration has $k - 1$ free children and $n - k + 1$ computed children. During reconstruction (bottom-up), knowing the value of any $k$ children allows us to compute the data at the parent.

The $q$-invertibility property characterizes the behaviour of a very broad class of operations. For example, in Shamir’s scheme [9], the secret to be shared is encoded as the $y$-intercept ($y_0$) of a private $(k-1)$th-order polynomial, $y = f(x)$. We generate $n$ shares of the secret by evaluating the polynomial at $n$ publicly known $x$-values. Since the $x$-values are public, knowing $f(x)$ at just $k$ $x$-values allows us to uniquely determine the secret $(k-1)$th-order polynomial, from which we determine the $y$-intercept, $y_0 = f(0)$. Conversely, knowing the $y$-intercept along with the value of the polynomial at $k-1$ other points uniquely determines the secret $(k-1)$th-order polynomial, from which the other $n - k + 1$ shares (i.e., $y$-values) can be computed.

An $n$th-order Split operator is simply $T^{n,n}$ (i.e., it is 1-invertible and has 1 computed child and $n - 1$ free children). Consider the encryption case, for example. The ciphertext and all $n - 1$ keys are needed for decryption, whereas knowing the original data and choosing $n - 1$ keys allows us to uniquely generate ciphertext. Similarly, an $n$th-order Copy operator is $T^{1,n}$ (i.e., it is $n$-invertible, with $n$ computed children and no free children). Reading any one copy is equivalent to read all other $n - 1$ copies as well as the original, whereas holding the original allows to uniquely generate $n$ copies.

## B Proof of Theorem 1

We will prove Theorem 1 by construction. The theorem states that any simple configuration is also proper. Consider a configuration $\Theta$ that is known to be simple. Let $T(V) = \{v_1, \ldots, v_m\}$ be a topological sort on the $m$ vertices of $\Theta$, which exists since $\Theta$ is a DAG. We will construct a strictly feasible, sequentially consistent solution, $X$, by visiting each vertex of $\Theta$ in the order specified by $T(V)$. All vertices are initially colored blue (i.e., have not been assigned a value). At the root $v_1$, set $X(v_1) = 1$ and color $v_1$ red (i.e., assigned). Visit each vertex $v_i, i = 1, \ldots, m$ in the order specified by $T(V)$, and do the following:

- If $v_i$ is an $S$-vertex, select one of the unshared children of $v_i$ and label it $x_i^\ast$. Set $X(x_i^\ast) = z(v_i)$ (i.e., $\alpha_X(v_i) = x_i^\ast$) and color $x_i^\ast$ red. For each remaining blue vertex $x \in c(v_i)$, color it red and set
The construction given above generates a solution \( v \) (i.e., \( \beta_X(x) = c(v_i) \setminus x^*_i \)) where \( \xi_x \) is some globally unique value (See Remark 1).

- If \( v_i \) is a \( C \)-vertex, \( \forall x \in c(v_i) \) set \( X(x) = X(v_i) \). Color red each \( x \in c(v_i) \).
- If \( v_i \) is a terminal vertex, do nothing.

**Remark 1.** Technically, we must ensure that \( \forall i, x \in c(v_i) \) implies \( \xi_x \neq z(v_i) \). One way to ensure this is to select \( \xi_x \) values outside the range of \( z(\cdot) \).

**Remark 2.** In the above construction, when we visit a vertex \( v_i \), all unshared children in \( c(v_i) \) are still blue. This is because a (non-root) vertex is colored red only after we visit one of its parents. Furthermore, \( v_i \) itself will be red since \( T(V) \) is a topological sort.

**Lemma 1.** The construction given above generates a solution \( X \) for configuration \( \Theta \). \( X \) is strictly feasible.

**Proof.** Let \( \delta_i(\Theta) \) denote the constraints implied by the \( i \)-th vertex in the ordering \( T(V) \). We use induction to show that after visiting the \( i \)-th vertex, the constraints \( \cup_{j=1}^{i} \delta_j(\Theta) \) are satisfied. Let \( i = 1 \). If \( v_1 \) is an \( S \)-vertex, then \( X(x_1^*) = z(v_1) \) and all other children of \( v_1 \) (initially blue) are given distinct values and colored red. Thus, the constraint \( \delta_1(\Theta) \) is satisfied. If \( v_1 \) is a \( C \)-vertex, then \( \forall x \in c(v_1) \), \( X(x) = 1 \), so again \( \delta_1(\Theta) \) is satisfied. This proves the base case.

Assume that the constraints \( \cup_{j=1}^{n} \delta_j(\Theta) \) are satisfied after visiting vertex \( i = n \). Let \( i = n + 1 \). If \( v_{n+1} \) is an \( S \)-vertex, then \( \delta_{n+1}(\Theta) \) is one split constraint and one inequality constraint. From Remark 2, \( x^*_{n+1} \) will certainly be blue immediately before we visit \( v_{n+1} \). By construction, we set \( X(x^*_{n+1}) = z(v_{n+1}) \), so the split constraint is satisfied. By the induction hypothesis, if \( x \in c(v_{n+1}) \) is red before visiting \( v_{n+1} \), then \( x \) is shared (see Remark 2) and \( X(x) = \xi_x \), globally unique (see Remark 1). Note that since the children of \( C \)-vertices are unshared, \( \exists y \in c(v_{n+1}) \) such that \( y \neq x \) and \( X(y) = \xi_x \). So, after visiting \( v_{n+1} \), the inequality constraint will also not be violated. Thus, if \( v_{n+1} \) is an \( S \)-vertex, \( \delta_{n+1}(\Theta) \) will be satisfied.

If \( v_{n+1} \) is a \( C \)-vertex, \( \delta_{n+1}(\Theta) \) is an equality constraint. All vertices in \( c(v_{n+1}) \) will be blue before visiting \( v_{n+1} \), since they are unshared (see Remark 2). Thus, after visiting \( v_{n+1} \), the equality constraint will be satisfied. Therefore, irrespective of the type of \( v_{n+1} \), \( \delta_{n+1}(\Theta) \) (and consequently \( \cup_{j=1}^{n+1} \delta_j(\Theta) \)) will not be violated, and by induction the resulting solution \( X \) will be strictly feasible. \( \square \)

**Lemma 2.** The construction given above generates a solution \( X \) for configuration \( \Theta \). \( X \) is sequentially consistent.

**Proof.** Consider the case where all equivalence classes in \( G(X) \) are singletons (i.e., individual vertices of \( \Theta \)). The argument can be extended to non-singleton equivalence classes. If a vertex \( x \in V \) is such that
it is not the computed child for any of its parents (i.e., $\forall y \in p(x), \ X(x) \neq z(y)$), then the corresponding vertex in $G(X)$ will have in-degree zero. So, if $G(X)$ has a cycle, then each vertex in the cycle is the computed child for some $S$-vertex in $\Theta$. By construction, each computed vertex in $X$ is also unshared in $\Theta$. Therefore, cycles in $G(X)$ are composed of unshared, computed vertices.

Suppose there exists a cycle in $G(X)$, and there is an arc from $v$ to $w$ as part of this cycle. Since $v$ points to $w$ in $G(X)$, $v$ must either be a parent or a sibling of $w$ in $\Theta$. If $v$ is a sibling of $w$, then they must share a common parent $x$ such that $w$ is the computed child of $x$. But $v$ is itself a computed child for some $S$-vertex $y \neq x$, which implies that $v$ is shared. This is a contradiction, since cycles in $G(X)$ are composed of unshared vertices. Therefore, if there is an arc from $v$ to $w$ as part of a cycle in $G(X)$, then $v$ must be a parent of $w$ in $\Theta$.

Thus, the existence of a cycle in $G(X)$ implies the existence of a cycle in $\Theta$. This is impossible, since $\Theta$ is a DAG. Therefore, $G(X)$ must be acyclic, which implies $X$ is sequentially consistent.

Lemmas 1 and 2 imply that there exists a strictly feasible, sequentially consistent solution $X$ to the constraints implied by the simple configuration $\Theta$. Therefore $\Theta$ is also proper.