Compounds: a next-generation hierarchical data model (I)

Markus G. Kuhn, Steven J. Murdoch, Piotr Zieliński

Computer Laboratory
Security Group

Abstract

Compounds provide a simple, flexible, hierarchical data model that unifies the advantages of XML and file systems. We originally designed it for *Project Dendros*, our distributed, revision-controlled storage system that aims to fully separate the control over data from its storage location. Compounds also provide an excellent extensible and general-purpose data format. A processing framework based on stackable *filters* allowed us to add rich functionality in a highly modular manner, including access control, compression, encryption, serialization, querying, transformation, remote access, and revision control.

XML started out as a plain-text file format; later a whole range of different APIs and abstract data models emerged (SAX, DOM, XML Infoset, XPath, etc.). The confusion between content and "syntactic sugar" in XML documents makes XML difficult to use with alternative representations, digital signatures and in revision-control systems. Compounds started out as a far simpler, yet more flexible abstract data model, for which several fully equivalent plain-text, binary and graphical representations are defined. Compounds can naturally represent XML in any of its data models.

Like XML, compounds are trees of strings. Unlike in XML, compounds clarify for each child whether its position relative to its siblings matters – as in structured text documents – or not – as in relational databases and file systems. This distinction not only simplifies the automatic merging of concurrent updates; it also supports encodings optimized for fast access.

Brief definition of compounds

A compound is a recursively defined structure. Each compound consists of five elements:

STRING — An arbitrary-length string of bytes.

TAG — A single byte, that distinguishes several kinds of STRING, for example:

- *control strings* are reserved for use by the compound processing framework (e.g., to activate *filters*)
- *meta strings* allow an application to distinguish textual meta information from normal text (e.g., for markup tags, such as XML element names)
- *text strings* are the standard type for any normal character data (UTF-8)
- *binary strings* are used for arbitrary non-text data (typically displayed in hexadecimal)

DIRECTORY — a mapping from *key* compounds to *value* compounds

KEY SET — a subset ("visible keys") of the set of key compounds in the DIRECTORY

LIST — a sequence of compounds

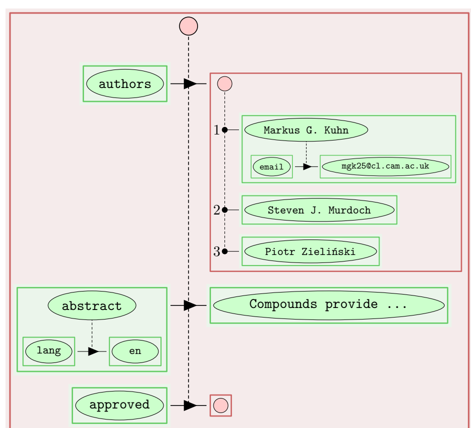
DIRECTORY access: A value compound can only be accessed by providing the corresponding key compound. If a provided key is not contained in the DIRECTORY, the reply will be a special **nil** value that is distinct from all compounds, to indicate that the provided key was invalid. The set of all valid keys may be infinite where a compound is not a stored data structure but is generated on-the-fly algorithmically. The set of valid keys cannot be enumerated, but a compound carries a KEY SET that enumerates a subset of all valid keys. These are called *visible keys*, because a user can discover their existence from the KEY SET.

Graphical representation

We draw a box ( ) around each *compound*. Inside, the STRING is shown as an ellipse (string), and the TAG determines its colour.

Compounds contained inside another compound are shown below its (string), connected to a vertical line. LIST elements appear to the right of this line, connected via a small circle; the first list element appearing highest. The key/value pairs of the DIRECTORY are connected via a small right-pointing arrowhead, with the key on the left and the value on the right of the vertical line. Invisible keys are indicated by white arrowheads.

A simple example compound:



The top-level compound has an empty string, an empty list, and three visible key/value entries in its directory. The value compound associated with the `authors` key has an empty string, an empty directory and a list with three elements. (As a convention, where the STRING is not used, it is set to be an empty control string, to distinguish unused strings from the empty text string.)

Text representation

For debugging, documentation, and direct manual access to compounds by developers or system administrators, we need a plain-text representation. Here, we outline a simple subset of the *Compound Representation Form – Text (CRF-T)* that we designed for such purposes. It is equally suited to discuss compounds informally via email and to handle them in a round-trip compatible way with plain-text editors (*emacs*, *notepad*, etc.).

In CRF-T, each compound starts with a representation of the tag and string, optionally followed by directory and list elements enclosed in parentheses, separated by commas. For all KEY SET elements, the corresponding key/value pairs from the DICTIONARY appear in arbitrary order, separated by `=`. All list elements appear on their own, in the order in which they appear in the list.

Each STRING is enclosed in `"..."` if it contains characters other than letters and digits. *Binary strings* are shown as hexadecimal digits enclosed in `<...>`. Some tag values are indicated by a punctuation prefix: `.` = *control string* and `*` = *meta string*. Examples:

```
size *bold <ff7f> .reset "Dr. Smith"
```

Directory elements (KEY = VALUE) and list elements (LIST ITEM) may be mixed arbitrarily. In the data model, only the relative positions of list elements will be preserved, not those of directory elements.

The previous example compound in CRF-T:

```
(
  authors=
  (
    "Markus G. Kuhn"
      (email="mgk25@cl.cam.ac.uk"),
    "Steven J. Murdoch",
    "Piotr Zieli\u0144ski"
  ),
  abstract(lang=en)=
    "Compounds provide a ...",
  approved=
)
```

Other representations

The full CRF-T encoding adds a TEX-like notation optimized for text-markup applications.

We have defined a sorting order for compounds and a unique, canonical binary encoding, CRF-S, that preserves this order in the lexicographic sorting order of the resulting byte strings. It is of particular use in connection with B-trees and cryptographic hash functions.

Our CRF-B encoding is a binary encoding optimized for compactness and efficient memory access.

Future variants will be particularly optimized for efficient on-disk access (good page locality, concurrent access, crash recovery).

2005-06-30

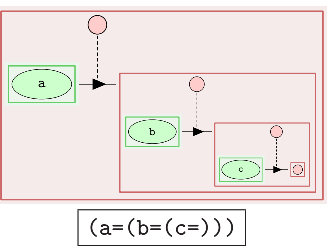
Compounds: a next-generation hierarchical data model (II)

Markus G. Kuhn, Steven J. Murdoch, Piotr Zieliński

Computer Laboratory
Security Group

Paths

Compounds can not only represent user data, but also locations in, operations on, and differences between other compounds. A compound of the form



`(a=(b=(c=)))`

is called a *path* when it represents a location within another compound, such as

`m(a=n(b=o(c=golf(1 2)), d=q(e=6)))`

The above path identifies the subcompound reached by descending through the keys `a`, `b`, and `c`. The abbreviated notation

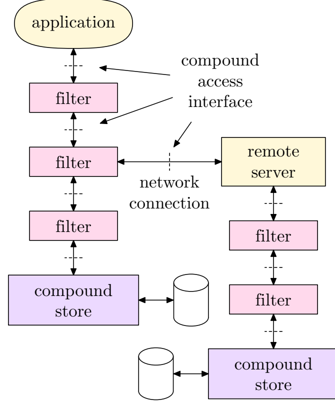
`a/b/c`

represents exactly the same path. Our binary compound representation handles paths just as efficiently as lists of the same length. Representing paths as a degenerated tree of nested compounds allows us to address several locations in a compound within a single path, as in

`a/b/{c, d/e}` ≡ `(a=(b=(c=, d=(e=))))`

Filters

The compound architecture not only defines a data representation concept, but also a modular processing model. In the simplest case, an *application* interacts with a *compound store* directly, via a standardized interface that provides support for navigating, reading and modifying a compound. A *filter* is an intermediate layer that can be inserted between an application and a compound store.

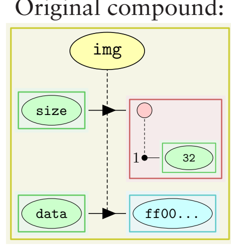


Filters use the same interface at both ends. Downwards, they act like an application to access the *original compound* below. Upwards they act like a compound store that offers a *filtered compound* to an application.

A filter provides some value-added service for an application. There are two types of filters.

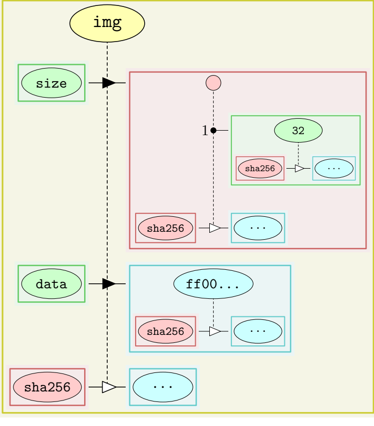
Augmentation filters do not change any visible aspect of the original compound. They merely add invisible keys, namely control strings reserved for the respective type of filter. A typical example would be a checksum filter that adds a SHA-256 hash of each compound:

Original compound:



⇓

Filtered compound:

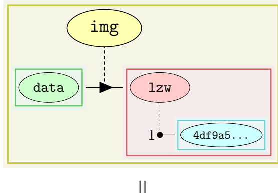


The added keys are not visible in the KEY SET and their values are usually calculated on-the-fly as the user queries their value. The *base filter* is an augmentation filter that adds all keys necessary to ensure that paths can address any part of a compound, including substrings and list elements, as in

`a/b/c/.s(1,2)` = `go`

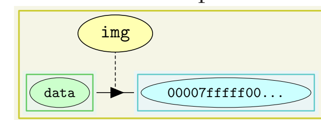
Substitution filters are triggered by certain control strings in the original compound. They transform the entire compound from the control string downwards. A typical example would be a filter that provides on-the-fly decompression of compressed strings or compounds:

Original compound:



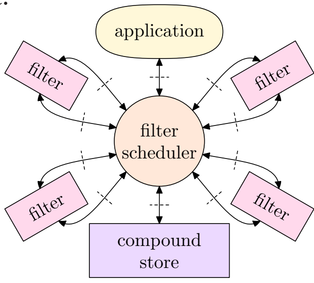
⇓

Filtered compound:



Other substitution filters implement symbolic links, remote access to other compound systems, or even entire revision control systems. Most substitution filters also work as augmentation filters and add an invisible key that grants direct access to the original compound below.

Several filters can be stacked, since they use the same interface at both ends. A filter scheduler automatically invokes filters as necessary, depending on the path accessed by the application and the control strings encountered along it.



Most of the convenience and functionality of a compound system is implemented in a highly modular way via filters. In a distributed implementation, the ability to reorder and bypass filters allows users to control, which filter is executed on which side of each network connection.

Compounds versus XML

XML documents can easily be mapped to compounds:

- XML element names, entity names, text, and attribute names/values all become compound strings
- child elements and text inside an element form the list of the compound representing the XML element
- element names become *meta strings*, while text maps to *text strings*, to distinguish them in lists
- XML attribute names and values form the dictionary of the compound representing the XML element in which they occur.

Compounds offer all the benefits of XML, but have fewer restrictions. They may be viewed as an extended and enhanced version of XML where:

- both attribute names and values can recursively be fully-fledged structured XML documents, not just flat strings
- new attributes can be attached to any string, not just to element names
- a clearer and simpler document model simplifies automated edits and the application of secure hash functions
- any part of a compound can hold arbitrary binary data, without any need for extra transparency encodings such as *base64*

More information:
`www.cl.cam.ac.uk/Research/Security/dendros`

2005-06-30