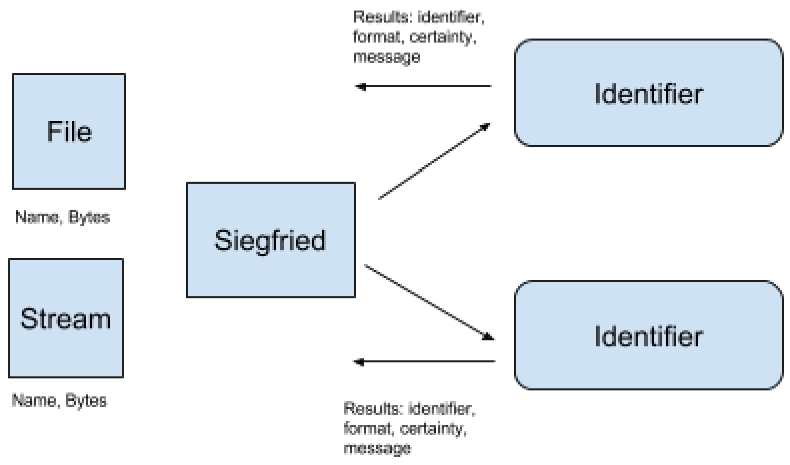
Identifiers

One or more identifiers are registered with a Siegfried service. The Siegfried service handles IO, but leaves all the actual work the to the identfiers.

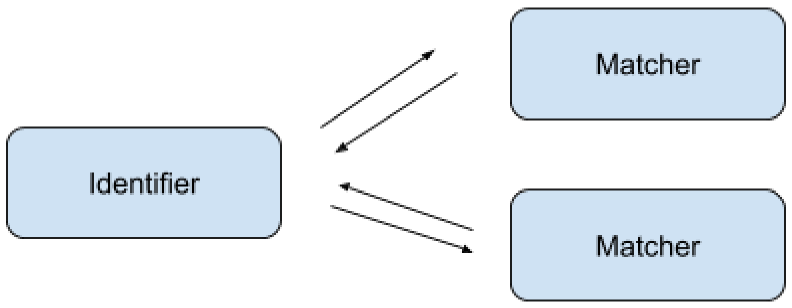
Identifiers are supplied the name and byte seqence of the file or stream. Names can be empty strings for pure byte stream identification. The identifiers return one or more results (*identifications*) on the supplied channel.

Identifiers control the matching process by determining which matchers to use, when to invoke new actions (such as invoke a container matcher), and when to finalise the matching process.


Results: identifier, format, certainty, message
File
Name, Bytes
Stream
Name, Bytes
Siegfried
Identifier
Identifier
Results: identifier, format, certainty, message

Matchers

Identifiers invoke and communicate with matchers. These do the actual work of identifying internal and external signatures.



Matcher interface:

ByteMatcher - takes a byte stream
NameMatcher - takes a file name
TextMatcher - takes a byte stream and a text format

Workflow for a byte signature match:
1. NameMatcher invoked on File, suggests a list associated with .pdf extension
2. ByteMatcher invoked on File, Hits PDF 1.2

Workflow for a container match:
1. NameMatcher invoked on File, suggests list associated with .doc extension
2. ByteMatcher invoked on File, Hits OLE2
3. NameMatcher invoked on OLE2, hits ComObject
4. ByteMatcher invoked on the ComObject

Workflow for a text match:
1. NameMatcher invoked on File, suggests list associated with .PY extension
2. ByteMatcher invoked on File, Hits Text UTF8
3. TextMatcher invoked on UTF8, Hits Python language file.

Results
Positive identification - Stop;
Positive identification - Stop this process and Try further matchers;
Suggestive identification - Continue and Try further matchers;

Processing Signatures

Signatures are processed for efficient matching.

The steps in processing are:
- flattening (adjoining frames are combined into plain sequence tests)

A segment function: that takes a signature and a max offset, then splits it into a set of bof-f, bof-w, vry, eof-f, and eof-w segments.

KeyFrame
Some kind of tree structure to represent partial matches. This is persistent and does not change. A set of pointers into this tree traverse it, their positions represent current state.

Identification

Step. 1 BOF and EOF checking
    BOF and EOF are lists of sequences that match an indexed list of anchors.
Step. 2 Based on above, add any conditional variable signatures
Step. 3 Variable offset checking

Segments

Segments are sets of Frames (within a signature) that share a single anchor.

```
type Segment []Frame
```

Result

```
type Result struct {
offset int
match Match
}

type Match int

enum {
FAIL Match = iota
```

PASS
CONTINUE
}

Anchors

All frames must be anchored, either individually or as groups of linked frames (segments). An anchor is essentially a fixed offset into a file. Frames can flow to the left or to the right of an anchor point.

Anchors can be:

- directly to BOF (i.e. the anchor is the BOF),
- directly to EOF,
- to a BOF or EOF sequence,
- or to a variable sequence.

Anchors are amalgamated into trees of frame choices. These look like:

```
[frame]<-[frame]<- ANCHOR <-[frame]
        [frame]<          <-[frame]
```

So an anchor is this type:

```
type Anchor struct {
lframe frame[s]
rframe frame[s]
}
```

Context

A context is
endianness; text encoding.
container: may need file pos, file name.

Actions

siegfried.ACTION:
enum: set context, set offset (for the subsequent pattern), partial match (could trigger adding variable pattern sequence), positive match, negative match, continue.
Results: Continue, Jump Left, Jump Right, Jump BOF, Jump EOF, fail, pass.

Only TestByte and TestByteL

```
TestByte(b byte, c *context) (result, offset)
TestByteL(b byte, c *context) (result, offset)
```

Range: NewRange func sets buf slice to right size.

Byte test - Container test - Container unizp - Text test - result.

New Sig
Siegfried provides:
Sig.AddNode(Offset, Pattern)
Sig.BofSeq []byte
Sig.VarSeqs []byte
Sig.EofSeq []byte

Sigs.Optimise []Sig

Algo

- Prefer a BOF/EOF sequence rather than direct anchor to maximise concurrency of searching
- For the BOF and EOF, start with left most pattern, take it as a sequence if possible,

Format Matching

Principle

Keep going until a positive match for a signature that has no preferences for any non-negatively matched signature.

Consider: idea of negative as well as positive match. Three **states**: POSMATCH, UNMATCH, NEGMATCH. Or three **sets**? Or both?

API

Siegfried will offer APIs for integration within other applications. Two APIs will be provided, a native Go API and an HTTP API. In addition, of course, there will be a CLI.

Golang

Server

Consider a REDIS style text protocol e.g. using http://golang.org/pkg/net/textproto/

CLI

IO

A single buffer that just keeps expanding to a MAX setting? MAX can be fairly large by default, care more about speed than memory.

Or a single large array that has BOF & EOF. 3* page size. With offset markers to control reading. With additional arrays for variable length in between. Can reserve those arrays to prevent them being overwritten.

Negative Identification

After BOF and EOF testing, we can say that a certain set of signatures are *negatively* matched. This may allow some avoidance of full file scans.