Signing PDF files

Hans Hagen

Here I discuss a feature of PDF documents that TEX users can safely ignore most of the time but that is part of the package. But before we arrive at describing what it is, first a few words about PDF and the way it is used.

When PDF showed up as a format, DVI was what TEX users had to deal with. It was possible to preview the result with a DVI viewer or convert it to some format suitable for a printer. The DVI format is rather minimal and has no resources like fonts and images embedded. Basically the file only positions glyphs and rules on a canvas and the glyphs are references to external fonts. Color and similar effects have to be implemented using the `\special` primitive that puts directives for the backend driver in the file. Although technically one could embed fonts and images using specials and thereby create a self-contained file it never happened, partly because it demands a dedicated viewer and (definitely at that time) increased runtime.

At that time PostScript was one of the popular output formats. For a long time I used DVIPSONE for (robust) printer output (with outline fonts) and DVIWINDO for previewing (because it was fast, supported color, graphics and hyperlinks). The initial road to PDF was to add another step to the conversion: using Acrobat to convert a PostScript (enhanced with so-called pdfmarks) into a PDF file. Later direct DVI to PDF converters showed up alongside pdfTEX that integrated an alternative backend. We can realize that without these more direct methods TEX would not be as popular as it is now. The fact that in ConTEXt we had a rather generic (abstract) backend, where specific drivers plugged in, indicates that we didn't foresee that PDF would eventually take over.

One of the reasons PDF showed up alongside PostScript is that it removes the interpretation part from the end result. Where PostScript is a programming language and the result needs to be interpreted in the printer or in a viewer (like GhostView), PDF is a collection of related objects that express what gets rendered in what way. Often the PDF files are smaller, also due to compression (so they transfer faster), and the lack of additional processing removes a bottleneck in high speed and high resolution printing. If you look at it this way, PDF is primarily a *printer format*.

Some tools in the Adobe suite of editing programs use(d) a mix of binary and PostScript to store the state. At some point PDF became the container format, although one could find curious mixes of PostScript, PDF, even configurations stored as typeset streams, but that might have been normalized by now. So, from this perspective PDF is a *storage format*, kind of an object-oriented one.

When PDF showed up it had some rudimentary support for annotations, like hyperlinks, and also for embedding of audio and video. I'm pretty sure that TEX was among the first programs to support this. Over time more annotation types showed up, like widgets (forms), complex media and 3D, JavaScript, comments and attachments. Widgets evolved a bit and one has to play rather safe (and not use all features) because some bugs and side effects became features and there is limited support in viewers, if only because often JavaScript is assumed. Media have always been sort of a mess, especially when the straightforward annotations were dropped. We have always supported them but I consider them unreliable in the long run. Most hyperlinks are working as expected, comments (a form of PDF annotations) when used wisely also work ok, although that depends on the viewer (interfaces change). No matter what one thinks of all this, here PDF is definitely a *viewing format*. Because comments can be added, PDF files can also be used in redacting workflows.

It is also worth remembering that in the early days only Acrobat was available for viewing; there was even a version for MS-DOS. The Reader only offered basic functionality and for the real deal one needed Exchange, which didn't come free. There was a complicated scheme for shipping a special version with documents: basically that was the business model for using PDF for an on-screen reading experience. This was not a success (cumbersome as well as expensive), and when Internet access became more common, using CD-ROM for distributing documents was soon obsolete, let alone installing a related closed source and operating system dependent viewer. We'll see that with document signing, another attempt at a business model is made.

Following up on that we now arrive at two additional aspects. One is accessibility and I could spend a whole article on that. Let's stick to the observations that the idea is that rendered content can be reinterpreted for reading aloud, for reflow (sic), maybe for cut-and-paste. Personally I wonder why one would use PDF to provide adaptive accessibility, because HTML is meant for that. Distributing a structured source might be more efficient when interpretation is needed. Anyway, it's not that hard to support but we end up with a bloated PDF file, while the PDF format started out with attempts to minimize size. I understand that publishers don't

doi.org/10.47397/tb/45-1/tb139hagen-pdfsign

like to distribute sources but if this is the solution one can wonder. The second addition is to render and present text in a way that cannot be tampered with and this is where signing comes in: can we somehow mark a document such that the receiver can trust its content. More about that later, but what we can say here is that PDF has become a *distribution format*. Conforming to standards, tagging, encryption and signing all play a role in this.

No matter what usage we consider, all of them depend on reliability. Can we show and process a document in the future? Can PDF be relied upon as a long-term archival format? From that perspective, standardization has to be mentioned. Already early in the history of PDF, plugins (and additional workflows) provided the printing industry ways to check if a file was okay. One should think of color spaces being used, fonts being present, etc. Some tools manipulated the PDF, not always with the best outcome, but we leave that aside. Other tools were a bit more tolerant than might be considered healthy, for instance by ignoring a bad xref table (basically the registry of objects in a PDF file) and either fixing it or just generating one from scratch. Although Acrobat can complain or fail to open a document, on the average commercial and open source tools are tolerant enough and the lack of a proper error log means that the PDF generators don't get fixed when that still can be done. It is also good to keep in mind that there is a whole industry around PDF generation, validation, manipulation, etc. and huge money making machines are not always on the retina of, for instance, TEX users who produce PDF. Of course by now there are so many documents in PDF format around that being tolerant kind of comes with the package. Validators like VeraPDF evolve and a document that is ok today (2023) might fail the test tomorrow, and the verdict even depends on the PDF framework being used (there are options). Where TEX users can often regenerate a document from source this is not true for the majority of documents produced elsewhere.

It is also important to notice that rather soon in the history of PDF, Ghostscript became an option for viewing and at some point commercial and open source viewers showed up. Not all were perfect and even today there are differences in quality and functionality. A good test is how well cut-and-paste deals with spaces and how well a test area gets selected. The open source viewers are slow in catching up, but because the evolution of media PDF annotations isn't that stable either for most purposes viewers like SumatraPDF (Windows) or Okular (Linux) is what I use today, especially now that Acrobat has moved to the cloud. There is also some competition from browsers that show PDF. For purposes of signing (which we'll get to next) one probably has to rely on Acrobat for a while, but we'll see.

So, what does signing bring to this? Digital signatures have been around for a while. You can for instance sign a document with a certificate (similar to what secure webservers do with sites). In that case the distributed blob has security-related information as well as the content. A validating application can take the content and check if it has been tampered with. It can do so off line (with limited security) but also go online and check the embedded certificate. With signed PDF files the same is true apart from the fact that here the signature is a partial one, not embedding the data. Instead the signature is embedded in the PDF file.

Before we move on we have to stress that signing is not the same as (password protected) encryption. A signed PDF file is by default just readable, unless one explicitly encrypts the file. These processes are independent. Here we ignore encryption; suffice it to say that ConTEXt can do it, but apart from users asking for it I don't know if it ever gets applied. We discuss the process of signing in the perspective of ConTEXt, although in itself it is not bound to that macro package.

Let's first look at how text ends up in a PDF file. Take this source file, in ConTEXt-speak:

```
\startTEXpage[offset=1dk]
   some text
\stopTEXpage
```

This leads to a so-called page stream that contains this (except normally you are likely to see garbage because compression is applied, so decompress first):

```
BT
/F1 10 Tf
1.195517 0 0 1.195517 7.485099 7.616534 Tm
[<0001000200030004000500060004000700 06>] TJ
ET
```

We switch to a font (`Tf`) with id `F1`, set up a text transform matrix (`Tm`) and render the four plus four characters (`TJ`) indicated by their index into a (in our case subsetted) font. The `0005` is not really a character: it refers to a space. It looks unreadable but one can figure out the text by consulting the `ToUnicode` resource associated with the font as it has the mapping from the index numbers to Unicode (with comments added):

```
<0001> <0073> % s
<0002> <006F> % o
<0003> <006D> % m
<0004> <0065> % e
```

```
<0005> <0020> %
<0006> <0074> % t
<0007> <0078> % x
```

There is nothing hidden here and one can actually even change the text by changing an index in the page stream, although of course you can only use indices that are available and you also have to accept weird rendering due to the change in progression when the referenced glyph has different dimensions. More extensive tampering with the document has more severe consequences. For instance, the page object looks like this:

```
3 0 obj
<< /Length 118 >>
stream
...
endstream endobj
```

So changing the content also demands changing the `Length`. Even worse, there is an entry in the object cross reference table:

```
0000000075 00000 n
0000000244 00000 n
```

that needs to be adapted, including all following entries. So, tampering is possible but not something that is likely to happen. Nevertheless we continue as if some guard against this is needed. We now assume the following document:

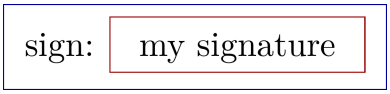
```
\nopdfcompression
\setupinteraction[state=start]

\definefield[signature][signed]

\defineoverlay[signature][my signature]

\starttext
\startTEXpage[offset=1ts,frame=on,
              framecolor=darkblue]
    sign: \inframed
      [background=signature,framecolor=darkred]
      {\fieldbody[signature][width=3cm,
                             option=hidden]}
\stopTEXpage
\stoptext
```

We get a small, one page, document:



This document has a widget that looks like this (some less relevant entries are omitted):

```
2 0 obj
<<
  /Type    /Annot
  /Subtype /Widget
  /FT      /Sig
  /T       <feff007300690067....074007500720065>
  /V       1 0 R
  /Rect    [ 38.445125 11.522571
             123.484489 23.471172 ]
>>
endobj
```

In principle we could add an appearance stream and decorate the widget but when adding signature support to ConTEXt I found that using a parent-kid approach, for instance, was not appreciated by some programs (I used mutool (mupdf), pdfsig (poppler), Okular and Acrobat Reader for some basic testing), so in the end the `V` key ended up in the root widget. It probably relates to fuzzy specifications, experiments with specific tool chains, non-public validation processes, etc. Round trip signing and verification seems not entirely trivial, so best to play safe.

When the value of a `Sig` widget is a string, signing is up to the viewer but when we have a dictionary the signature can be in the file. The `V` value of `1 0 R` is a reference to a dictionary with object number `1`. Here is what that value looks like when we generate this document:

```
1 0 obj
<<
  /ByteRange [ 2000000000 2000000000
               2000000000 2000000000 ]
  /Contents  <0000000000000....00000000000000>
  /Filter    /Adobe.PPKLite
  /SubFilter /adbe.pkcs7.detached
  /Type /Sig
>>
endobj
```

The `Filter` and `SubFilter` entries are sort of default, though alternatives are possible, which then requires additional information to be added and also a viewer (or validator) able to deal with it. We leave that aside. The `Contents` hex-encoded string is a placeholder for the signature and in our case is 4096 bytes long. We could compute a bogus signature and check the size instead. Here the `ByteRange` and `Contents` are actually invalid but viewers are (supposed to be) tolerant so it triggers no error. After all this widget is only consulted when signatures are checked. The general structure of the file is like this:

```
%PDF-1.7
....
1 0 obj
<< /ByteRange [ .... ] /Contents <....> .... >>
endobj
....
xref
```

The signature ends up between `<` and `>` and has to be calculated over the bytes specified by the `ByteRange` entries. Although one might think that

these can be arbitrary, in practice it looks like it is best sticking to the recommendation:

```
start of file
length up to position < of contents
position after > of contents
length up to end of file
```

So basically all except the `Contents` value is taken into account. Because the ranges are part of that, they need to be filled in properly, something that has to be done after the PDF file is finished because only then is the size known. In ConTEXt that could be done as part of the main run, but it makes little sense because we need to adapt the file anyway. If the file is called `sign-001.tex`, we get this:

```
mtxrun --script pdf --sign \
       --certificate=sign-001.pem \
       --password=test sign-001
```

The script will set the byte ranges and fill in the content. It does that by making a data file and running `openssl` with the appropriate parameters, although with `--library` one can avoid the temporary file and gain a bit. Just for the record: we don't depend on that library but have only a minimal delayed binding to a few functions, with Lua wrappers so it has no impact on (compiling) the LuaMetaTEX binary. Eventually one ends up with something like this (values abridged):

```
1 0 obj
<<
  /ByteRange [ 0000000000 0000006276
               0000010375 0000000380 ]
  /Contents  <3082061a06092a....a082060b308206>
  ....
endobj
```

Verifying can be done as follows:

```
mtxrun --script pdf --verify \
       --certificate=sign-001.pem \
       --password=test sign-001
```

which reports:

```
sign pdf | signature in file 'sign-001.pdf'
           matches the content
```

while changing a byte in the trailer id results in:

```
sign pdf | signature in file 'sign-001.pdf'
           doesn't match the content
```

For verifying we can load the PDF file and use the `ByteRange` specification but for signing this is less trivial: when we load a PDF file we load a structure that is ignorant of the position in the file. We could use the cross reference table to find the position in the file of the object but that assumes that this table is available. So here we have two alternatives. We can write an auxiliary file (`sign-001.sig`) at the end of the TEX run that has the relevant information. This approach permits us to keep the PDF file simple: we reserve enough characters for the ranges and content so we can overwrite them. If the file is lacking, the sign routine tries to locate the object in the PDF from the list of widgets and once we know its number we also know where the object is in the file. This alternative adds a little overhead because at least the cross reference table has to be loaded. Whatever route we take, it is still prettier than appending additional objects to the file and basically creating a new version, which not only makes the file larger but also keeps unused objects around. Applications like mutool and Acrobat prefer that route, though, in part because they add their own appearance streams.

We now need to discuss these certificates and that is where it becomes less convenient. For testing, I use a Let's Encrypt certificate but these officially cannot be used as they are flagged as web certificates. There is (what's new here) a whole industry behind this signing. You need to get a certificate someplace and for that often have to sign up for a yearly subscription. In the worst case you get a token instead of a file and then have to set up some delegated workflow. Feeding a document into a USB token is not the most efficient of all processes, so you will find alternative solutions where you end up with a dedicated machine in a server rack. This all makes it a no-go for a low or zero budget situation. It also means that for just *printing*, *viewing* or *storing* purposes signing doesn't make that much sense: it only adds overhead.

One can argue that signing is not that robust anyway. Just like we can add a signature to a file, so can anybody. It's all about trust. When a byte in a PDF file is changed validation fails anyway so that is already a signal; we don't need to verify the certificate for that. And it's not that hard to let a user upload a file to the origin and let it validate there where the private key is known. But wait, isn't it more convenient to do that without uploading? Sure, but here are some pitfalls. First of all, who knows if a certificate is still valid? An organization has to spend quite some money on it yearly. And (even root) certificates expire so in the end the document refers to something invalid anyway, which effectively makes the document expire after some time. Saving documents and providing them again might be cheaper and also has the advantage of archiving. For long term archiving signing makes little sense anyway (expiration, cracking).

So why do we bother to add signing to ConTEXt? The answer is simple: user demand. Just like being forced to use some PDF standard, users

can be forced to comply with what the organization prescribes with respect to signing, even when in the end it's just a demand, and nothing is actually done with it. So the main question is: after showing that it can be done, what eventually happens; how does the workflow look? It's comparable to tagging: it is sometimes demanded, but after that the lack of useful tools make it just a box to be ticked.

There are other alternatives to making users feel good about a document: provide a printed copy, keep the original someplace for downloading, maybe make it possible to regenerate a document from source, maybe even provide the resource. Generate a string hash and keep that available alongside the original. In the end it is all about trust, indeed.

Let's end on a positive note. Getting to know what has to end up in the file is not that trivial and as with much on the Internet, looking for solutions quickly brings you to a subset of partial and sometimes confusing answers and solutions. This is why in the end I decided to just look in the code base of `openssl` that comes with examples and eventually one can sort out something not too complex. One of the interesting observations was that the binary blob is a structured key/value sequence using technology from decades ago (1984), when data had to be transferred reliably between architectures and programming languages: Abstract Syntax Notation One (ASN.1). It makes old TEXies feel young when old tools survive beyond the modern short lifespan of fancy web technologies. I might eventually spend some more time on this, just for the fun of it.

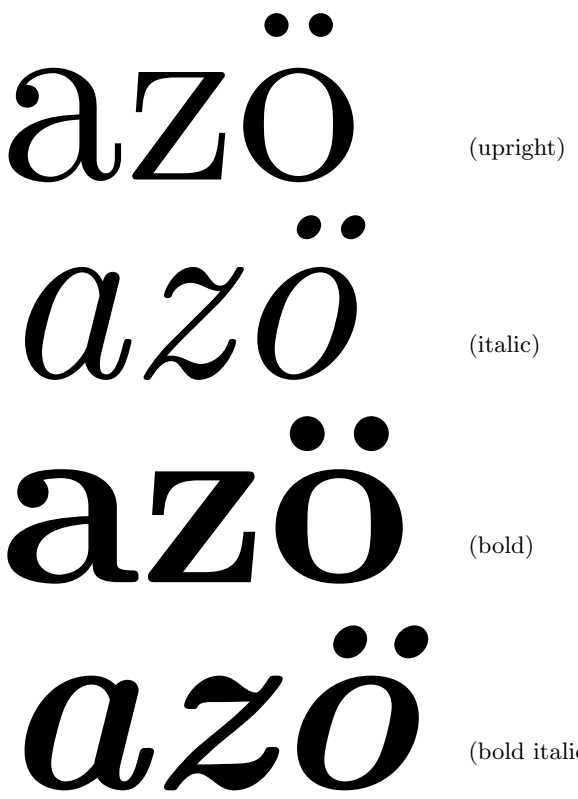
If you want to know more details: the official ISO standard on PDF has some sections on the matter; a more comprehensive summary can be found in "Digital Signatures in a PDF, Adobe Systems Incorporated, May 2012". There is also "CDS Certificate Policy, Adobe Systems Incorporated, October 2005" but I suggest to ignore that one unless you're forced to implement the more expensive route.

Some final words on the mentioned formats. For printing and storage this feature is not needed. Nor for regular viewing, because users probably don't care that much if a manual or book is signed and it's unlikely that certificates last that long (or stay secure for that matter). But it might make sense for distributing documents with some legal meaning in the short term. In that perspective having this feature in ConTEXt makes most sense in specific workflows. But it doesn't hurt to know that TEX is still able to adapt itself to these situations.

◇ Hans Hagen
Pragma ADE

Computer Modern shape curiosities

Hans Hagen



When playing with some (upcoming) new font features in LuaMetaTEX, I overlaid regular and bold versions of Latin Modern characters. I took an 'a' with diaeresis as a test.

While staring at the overlays I noticed that the little hook of regular was not present in the bold variant. After displaying the whole upright alphabet, that was the only difference in shapes. In the italic shapes, the 'z' was a bit different. And when blown up the dots are somewhat larger in the bold. (Computer Modern is the same, naturally.)

So, the question is: how many users who can immediately recognize Computer Modern have noticed this difference in 'a'? Another question is: did personal taste win over consistency?

We can also wonder if Latin Modern should have a few stylistic alternates, but maybe no one is willing to pay the prices in additional overhead. Of course most such details get hidden at a small 10 point size. When blown up enough, a few other interesting design details can be seen, but I leave noticing that to the reader. After all, these shapes were never meant to be seen that large.

◇ Hans Hagen
Pragma ADE

doi.org/10.47397/tb/45-1/tb139hagen-cmshapes