Gambling with Crypto

It isn't just typesetting

Pablo Rodríguez

Crypto... does not stand here for the fancy word of the moment (cryptocurrency), but only for cryptography. In these pages, I want to introduce some cryptographic features included in ConTEXt, which may help to secure some aspects of your documents. These cryptographic aids are hashes (for documents and texts) and digital signatures.

1. Why cryptography may help

First of all, I'm only an average computer user. I have been using ConTEXt for some years now, but I have no technical background. I cannot properly code, I can only copy and paste snippets (barely modifying them in that process). Just in case you wonder, my background is in humanities. This also means my mathematical knowledge is practically non-existing.

These pages intend to introduce how ConTEXt offers two cryptographic features. This section aims to be an explanation about their usefulness. Or at least, an introduction about how the rest of us may benefit from these features. These features are hashes (or digital fingerprints) and digital signatures.

Hashes are different computation methods to obtain digital fingerprints from any given content (files or simply data). They are digital fingerprints, since their values are strings of digits as characters (such as `c28e8fcc3`, but usually longer).

For *exactly the same data*, a given hash outputs the same value. If anything changes, the fingerprint is totally different. As you may already think, such feature is extremely useful to check whether any file has been modified or not. It might be a small image or a hundred page PDF document. Hashes are used to confirm that the data sent and received are exactly the same (to secure software downloads).

Digital signatures[1] may have a societal effect similar to handwritten ones, because they guarantee content integrity and non-repudiation of the signed document. This means that signature breaks when contents aren't *exactly the same* as the signed ones (using hash values to check their integrity). It also involves that the signature

[1] According to their specification documents, digital and electronic signatures are not the same. As for these pages, you may assume that when digital signatures are mentioned, electronic ones are meant.

comes from the person identified by the certificate (since a digital certificate should only be issued to the person certified by it).[2]

These signatures generate an external file to the signed data. For PDF documents, signatures are included in the documents. Embedding PDF signatures in documents has the main advantage that the conforming viewer checks both signature and certificates, displaying their status.

A signature–handling PDF viewer checks all signatures included in the document when opening it. After that, it reports whether each signature is valid (contents haven't been modified after signing). It also reports whether each included certificate is legitimate (it has been issued by a trusted certificate authority and it hasn't been revoked).

I hope these pages don't sound too technical. If this were the case, you might want to skip the two last subsections (3.1 "Technical remarks" and 3.2 "Some *OpenSSL* commands").

As for how hashes and signatures may help to the rest of us with our PDF documents, I have found three main cases:

1. As a proof of integrity for attached files (PDF documents or not).
2. As digital fingerprint for the output PDF document.
3. As a seal of integrity for the generated PDF document.

The second and third cases need some explanation.

Since ConTEXt may get digital fingerprints from any file, nothing prevents you from hashing any file and automatically renaming the file to include the hash value in its file name. It will be obvious that a pure Lua script would be faster than any ConTEXt source (besides the file being totally unrelated to mentioned source).

Digital signatures are a newly introduced (and not intensively tested) feature in ConTEXt. As you may experience later, there are still some glitches. Since personal certificates are sensitive stuff, I'd rather delay signing with any personal certificate until some extensive testing has been performed. For example, having to type a certificate password in plain text is something that could (and I guess, should) be improved.

2. Fingerprinting contents

ConTEXt includes the SHA2 algorithm family (with options from SHA256 to SHA512) as Lua commands.[3]

[2] In a similar way to official documents (such as driving licenses, passports or national identification cards) only being issued to their legitimate holders.

[3] SHA1 has been abandoned as insecure and other functions might be imported from *github.com/Egor-Skriptunoff/pure_lua_SHA* (SHA3, BLAKE2 and BLAKE3, optimized for speed).

Hashing may be applied to texts, such as in the following examples (only their last character differs).[4]

Look, no hands (and all teeth ;-)):

```
This is only an example
21721c837d8a4934170286c048fd1e213c8ea6ef18d263f727d1fb6786f7752b

This is only an example
cfb638e6105c8125c4680dcb572b74dabf6577d08d9e6aa076a0bf56219af6eb

This is only an example.
469590cc7924bf9c0ae5b0a3c8f7c4594343475993bef80f5f83cb162c2c14d4

This is only an example,
34e8082fea5754c4ac0ff1cf5e2e8205a21430b4a711ff4f3c18ed36fc4dee3f

This is only an example;
f6c666ace83d06b44658676ce3098af9405ea43e89e9a1d77ce00c4f4f8e262d

This is only an example:
a7ef6ccfc1dae7725a38a4373719b52d03230efee797139fbf2a3af1b7a6477f
```

As you may have experienced before, hashes are considered safe because only highly–dissimilar contents might get the same result.

If similar contents generate the same hash, you have a collision. A repeated hash poses the risk of functional indistiction of not totally different files.

Unique hashes may be theoretically impossible. In case of SHA512, hash strings contain 10 digits and 6 letters, which make 16^{128} possibilities (actually, 2^{512} values).

2.1 Useful code

At least for me, hashing files has been much more useful than hashing text contents. But here you have both definitions for `\texthash` and `\filehash`.

First, it comes the Lua code:

```
require("util-sha")

function document.sha_text(str)
  context(sha2.hash512(str))
end

function document.sha_file(str)
  if io.exists(str) then
    context(sha2.hash512(io.loaddata(str)))
  end
end
```

[4] The first sentence includes no character after the last word. But the second sentence includes a blank space after its last word. This is the reason why both hashes differ. In order to fit page width, these hashes are SHA256 (with 64 characters) and not SHA512 (with 128 characters).

In both functions, you may replace `hash512` in `sha2.hash512` with:

- Any of the hashes `hash256` or `hash384`. Their string is a quarter of their bit length (64 or 96 characters, instead of 128).
- The same name in uppercase (such as `HASH256` or `HASH512`) includes uppercase letters in the resulting hash. Consider that lowercase letters may be more readable (at least in some contexts).

The ConTEXt commands for these Lua functions may be defined as:

```
\startluacode
-- replace this line with previous Lua code
\stopluacode

\def\texthash#1{\cldcontext{document.sha_text("#1")}}

\def\filehash#1{\cldcontext{document.sha_file("#1")}}
```

2.2 External documents (or attachments)

Hashing files may be extremely useful with attached PDF documents. For the sake of accuracy, these files are not hashed as attachments, but they are attached in the PDF document and hashed as external files.

Considering the code from the previous subsection, a definition for `\attachmenthash` may read:[5]

```
\protected\def\attachmenthash#1{%
  \attachment[method=hidden, file={#1},
    title={SHA512: \filehash{#1}}]}
```

For example, typing `\attachmenthash{\jobname.tex}` (with the relevant previous code), the TEX source would be embedded in the resulting PDF document (and its SHA512 value would be displayed with the attachment).

In this case, it is important to consider that PDF[6] has two ways of attaching external data: as embedded files and as file annotations. ConTEXt enables both methods. `\attachment[method=hidden]` embeds the attachments without adding any content on the page, nor any annotation to the document. The default behavior (`\attachment[method=normal]`) adds an icon where file is attached and a file annotation to the page of the document.

As per (more or less) conformant viewers, extra information for attachments is handled by them in the following way (sorry, no *macOS* around here):

[5] This command may be improved with the option `\attachment[type={application/pdf}]` for PDF documents (or correponding MIME type for other kinds of attachments). Note that `\setupinteraction[state=start]` is required for attachments.

[6] I mean the format itself. Since it stands for *portable document format*, PDF is a format (not a document).

a. *Acrobat Reader*,[7] *Okular*, *Evince* and *PDF.js* (at least, within *Firefox*) handle both embedded and attached files, also displaying their extra info.

b. *MuPDF-GL* shows attachment annotations and the extra info. It cannot display PDF attachments by clicking on their icon, and it blatantly ignores any embedded file the PDF document might contain.

c. *SumatraPDF* behaves the same way as *MuPDF-GL* with attachment annotations. It shows embedded files as bookmarks and it only allows to save them. Even with PDF documents, they have to be saved and reopened as external documents. No extra info is displayed with embedded files.

d. As of July 2024, neither *Chrome* or *Edge* handle attachments in PDF documents (embedded or as annotations).

2.3 Fingerprint the output document

There may be cases in which you may benefit from having the hash value of the PDF document you are generating with ConTEXt in its file name.

ConTEXt generates PDF documents using more than a single run. Each run modifies an auxiliary file (`.tuc` or `.tua`), generated in the first run.

Lua code to rename the file to its SHA512 value may read (consider this needs to be inside **`\startluacode … \stopluacode`**):

```
luatex.wrapup(
  function()
    local function fs512(str)
      return sha2.hash512(io.loaddata(str))
    end

    os.rename(tex.jobname..".pdf", fs512(tex.jobname)..".pdf"))
  end
)
```

The previous code renames the output document from each run. Depending on what you are generating, you may end having a couple (or three) renamed files. This is basically useless, since you have to remove the PDF renamed documents from runs previous to the last one.

ConTEXt identifies the last run when the auxiliary file from previous run has not been modified in the current run. This also implies that the hash value for that file is the same in previous and current runs.

A possible approach to rename the PDF document in its final run could be as described below. It assumes ConTEXt is invoked with the `--purgeall` argument. I needed to rename previous local function from `fs512` to `f512` to fit the text width.

[7] In these pages, even if *Acrobat* is mentioned, this stands for *Acrobat Reader* (I don't have other version).

```
luatex.wrapup(
function()
 if environment.currentrun >= 2 then
  for f in lfs.dir(dir.current()) do
   if f:match("^[%d%a]+%.tx$") ~= nil and f:len() > 130 then
    if f512(tex.jobname..".tuc") == f:match("^([%d%a]+)%.tx$")
      then
     os.rename(tex.jobname..".pdf", f512(tex.jobname..".pdf")..".pdf")
     os.remove(f)
    else
     os.rename(tex.jobname..".pdf", f512(tex.jobname..".tuc")..".tx")
    end
   end
  end
 else
  os.rename(tex.jobname..".pdf", f512(tex.jobname..".tuc")..".tx")
 end
end
)
```

As before, this code needs to be enclosed in **\startluacode** … **\stopluacode**. These lines proceed in the following way:

1. A main conditional checks whether current run is the first one or not.
2. At the end of the first run, PDF output is renamed to SHA512 value from `.tuc` file.
3. On the second run, all files of current directory are looped.
4. Use a conditional to restrict the loop to files with names matching both a reserved extension and the length of the given hash value (at least, 128 characters for SHA512).[8]
5. Check whether SHA512 from the previous `.tuc` file matches the SHA512 value contained in current file name. In this sample, it would be its name without extension.
6. If there is a match, this is the last run in the compilation. Rename PDF output to contain its SHA512 value and remove current file.
7. If there is no match, rename PDF output to contain the SHA512 value from the `.tuc` file.

The code above is rightfully limited by text width. In order to ease code readability, special extension has been set to `.tx` and the file name (without extension) only contains the hash value. Since it is important to reduce the possibilities of unintended matches, restricting the process to fewer files to avoid undesired file deletions.

[8] The file name includes the extension. Since it contains two characters, total length is over 130 characters.

In fact, nothing prevents the extension from being `totally_temporary` (or something similar).[9] The file names could also start with `tex.jobname.."-"`. With both restrictions in file name and extension, the main loop will match the right files only.

2.4 What about buffers?

In some cases, you might want to include scripts, both as attachments and in the text body. In an explanation on how to record a screencast, I needed the invocation command for `ffmpeg` in the explanation. The video files were recorded to catch some issues with a given program and I needed to show that the batch file only contained a harmless `ffmpeg` invocation.

Since it is possible to type and attach a buffer directly by its name, hashing needs to load the buffer contents. The Lua code (enclosed in **\startluacode** … **\stopluacode**) and its matching ConTEXt command would read:

```
function document.sha_buffer(buffer)
  return sha2.hash512(buffers.raw(buffer))
end
```

```
\def\bufferhash#1{\cldcontext{sha_buffer("#1")}}
```

The previous code applied to an extremely basic script, which is only a way of invoking `ffmpeg` recording the whole screen in *Windows* with mouse and without audio, degrading the pixel format (for compatibility reasons), at a couple of frames per second (to minimize final video size):

```
\startbuffer[ffmpeg]
set PATH=%PATH%;%~dp0\ffmpeg\bin; &&
ffmpeg -framerate 2 -f gdigrab ^
  -i desktop ^
  -pix_fmt yuv420p ^
  -draw_mouse 1 ^
  error-video.mp4
\stopbuffer

\typebuffer[ffmpeg]

\attachment
  [name={record-video.bat},
   buffer=ffmpeg,
   method=hidden,
   title={SHA512: \bufferhash{ffmpeg}}]
```

Just as a comment, *Acrobat Reader* warns about security issues when saving batch files and even JPEG images. I might understand the first case, not so much the second one.

[9] In my documents, I use a string with 12 (extended) Greek characters as the extension, which avoids any possible unintended match.

This kind of general security warnings are similar to the description: “young adult male with knife in hand found in your dining room”. Well, he may be a dinner guest or he might be a serial killer.

My point is that such descriptions are incomplete, to say the least. They don’t help to decide and may generate panic among unexperienced users.

3. Signing documents

At least since July 2024, ConTEXt is capable of digitally sign PDF documents through *OpenSSL*.[10] It is a single command, such as:

```
mtxrun --script pdf --sign --certificate=certificate.pfx
--password=password document.pdf
```

Previous to that processing, you need to generate a PDF document with a signature field, such as in:

```
\setupinteraction[state=start]

\definefield[Signature][signed]
\defineoverlay[signature-text][my signature text]

\starttext
  \framed[width=1tw, height=1th, background={signature-text}]
    {\fieldbody[Signature][height=1fh, width=1fw, option=protected
      ]}
\stoptext
```

There are some questions to be taken in consideration:

a. The first argument of **\definefield** is the field name of the signature displayed by the PDF viewer.

b. The second argument is the field type, and `signed` is mandatory here.

c. The previous example has a background text (`my signature text`). It may be useful in some cases, but it is not required to include an overlay.

d. In that sense, nothing prevents a field of width and height of **\zeropoint**. The PDF specification mandates handling these signatures as not visible.[11] They are displayed in the signature pane of *Acrobat*.

Lua code to sign the completely generated PDF document would read:

[10] In *Windows*, it is also possible to install it within *MSYS2* and invoke ConTEXt from there.
[11] In section “12.7.4.5 Signature Fields”, on page 446 (*opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf#nameddest=G11.1697972*).

```
luatex.wrapup(
 function()
  local function f512(str)
  return sha2.hash512(io.loaddata(str))
 end
 if environment.currentrun >= 2 then
  for f in lfs.dir(dir.current()) do
   if f:match("^[%d%a]+%.tx$") ~= nil and f:len() > 130 then
    if f512(tex.jobname..".tua") == f:match("^([%d%a]+)%.tx$")
      then
     local Sign = "mtxrun --script pdf --sign %s %s %s.pdf"
     io.write(" Certificate file name? ")
     local Cert = "--certificate="..io.read()
     io.write(" Plain-text password? ")
     local Pass = "--password="..io.read()
     os.execute(Sign:format(Cert, Pass, tex.jobname))
     os.remove(f)
    else
     os.rename(tex.jobname..".pdf", f512(tex.jobname..".tua").."
       .tx")
    end
   end
  end
 else
  os.rename(tex.jobname..".pdf", f512(tex.jobname..".tua")..".tx")
 end
end
)
```

Variable definition was guided by fitting in text width. Password prompt assigned to a local variable is only to avoid permanent storage (either in the source file, or in the terminal buffer). Of course, this displays the password completely and it is as unsafe and insecure at it seems.

3.1 Technical remarks

The following comments are some considerations on what actually is the first implementation of digital signing in ConTEXt. Keep the following in mind when reading this.

- First and foremost, I'm just an average computer user. My background in math is practically non-existing.[12] Anything written in this article may be proven wrong.

[12] All my math knowledge is almost limited to "digital computing": I count using my fingers ;-P.

- ConTEXt is mainly a typesetting program (as Hans reminds us from time to time). It is great to have interactive features, but this one completely relies on external software to generate the signature.
- During the composition of this article, I found out the that Hans described the new signing feature in an article for *TUGboat*.[13] Since this journal is available only to subscribers during the first year, I hope these remarks still make sense after reading Hans' article.

As common in other EU countries, the Spanish national identification cards contain digital certificates. In fact, these actually contain two for mutually exclusive purposes: authentication and signing. The use of digital signatures is pretty standard to submit documentation to public services. Regulation EU/910/2014[14] grants the same legal effect to qualified electronic and handwritten signatures (article 25.2).[15]

Due to the high legal value of digital signatures, personal certificates should be stored in the most secure way possible. After experiencing that installed certificates don't require password for authentication or signature (also with *Acrobat*), devices may help to avoid having what may be seen as a permanent frying pane (an insecure practice, to say the least).

Using other software (such as *OpenSC*), *OpenSSL* may perform signatures using certificates from smart cards and other hardware devices that store digital certificates (known as PKCS#11). I haven't tested that possibility right now, because it would require a different *OpenSSL* invocation (not yet allowed by ConTEXt).

In its current form (as of July 2024), the signature itself includes signing time, but for some reason *Acrobat* ignores it and complains about missing time.[16]

At least in theory, it is possible to include a time stamp in the signature as an attribute. This is not another time stamping signature added to the document. In practice, first we neeed to know how to achieve that with *OpenSSL*. At least to me, it isn't very clear whether this is possible (in a way that ConTEXt could make use of it).

Right now, electronic signatures requested by ConTEXt are only basic ones, as far as I know.[17] This means, there is no certificate chain embedded in the signature (for long term validation). Again, this would require the invocation of a new argument (`-certfile`), and the user should also know which files contain the required certificates. I don't know whether this will be considered much ado about nothing.

---

[13] "Signing PDF files", *TUGboat* 45(1):145–149, 2024, *tug.org/TUGboat/tb45-1/tb139hagen-pdfsign.pdf* (available to the general public in 2025).

[14] *data.europa.eu/eli/reg/2014/910/oj*.

[15] *eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32014R0910#d1e2383-73-1*.

[16] The PDF specification requires to read the signing time from the signature itself (the one generated by *OpenSSL* in this case) in the first place. For some reason, *Acrobat* seems not to read it. ConTEXt adds no other signing time specified outside the signature (in the PDF document). Please before complaining, consider that this makes complete sense in ConTEXt, since a PDF document with a signed field may be actually signed months after its generation.

[17] Just in case you wonder, I analyzed the signature contents reading the output from command explained in next subsection (3.2 "Some *OpenSSL* commands").

3.2 Some *OpenSSL* commands

Since signing certificates are delicate files (in my personal case, enclosed in smart cards to avoid leaks), I have tested the signing feature added to ConTEXt with sample certificates created by myself. These were generated using *OpenSSL*.[18]

Self-signed certificates may be generated with:

```
openssl req -x509 -newkey rsa -keyout key.pem -out cert.pem -days
365
```

In order to have a single file (with certificate and private key) from the previous two generated files, this is the required command:

```
openssl pkcs12 -export -out certificate.pfx -inkey key.pem -in
cert.pem
```

But as Hans rightfully warned me,[19] *Acrobat* doesn't allow self-signed certificates to perform PDF signatures.[20] It will show an icon for invalid signature, citing invalid certificate as its cause. Internet is full of references to create a full certificate chain with OpenSSL.[21] Since I needed the sample certificates only to test that signatures were right, even self-signed certificates did the job in this case.

For the sake of experimentation, another algorithm would be ECDSA[22] (instead of previously generated RSA). Again, as long as self-signed certificates perform the PDF signature, *Acrobat* will complain.

A way of generating a private key and certificate in a single file may read:

```
openssl req -x509 -nodes  -days 365 -newkey ec -pkeyopt
ec_paramgen_curve:prime256v1 -keyout key.pem -out cert.pem &&
openssl pkcs12 -export -out certificate.pfx -inkey key.pem -in
cert.pem
```

Information about a given certificate may be extracted with:[23]

```
openssl x509 -noout -text -in certificate.pfx
```

Finally signature information is displayed with:

```
openssl cms -cmsout -print -in document.bin -inform der
```

---

[18] *www.openssl.org*. *Linux* distributions offer this software packaged with dependencies. *MSYS2* (*www.msys2.org*) also offers it for *Windows*.

[19] Since that comment is on the mailing list, see *mailman.ntg.nl/archives/list/ntg-context@ntg.nl/message/GYVO3PJTOICMB4PUXN2IKWR7YNPAFZQM/*. I hadn't experienced this issue in *Acrobat* before, because I had been using a sample certificate generated by *Reader* itself (that wasn't self-signed).

[20] Rationale in *www.adobe.com/devnet-docs/acrobatetk/tools/DigSig/changes.html* (reference provided thanks to *stackoverflow.com/a/77967318*).

[21] *openssl-ca.readthedocs.io* may be a very useful one.

[22] According to *csrc.nist.gov/Projects/Digital-Signatures*, FIPS 186-5 deprecated DSA as algorithm for signature generation, but not for verification of signatures generated prior to its implementation (*nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf#page=25*).

[23] `openssl pkcs12 -info -in certificate.pfx` provides less information.

This may be useful to see what contains the signature that ConTEXt requests from *OpenSSL* and embeds in the PDF document. All signatures get their name from the source PDF document (that is going to be signed) and the `.bin` extension.[24]

[24] The relevant code actually reads: `local binfile = file.replacesuffix(pdffile,"bin").`