

~~TIMECRYPTION~~



FRIENDLY TODAY.

EVIL TOMORROW.

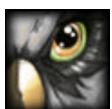
TIMECRYPTION

CLEAN NOW, MALICIOUS LATER

A talk by



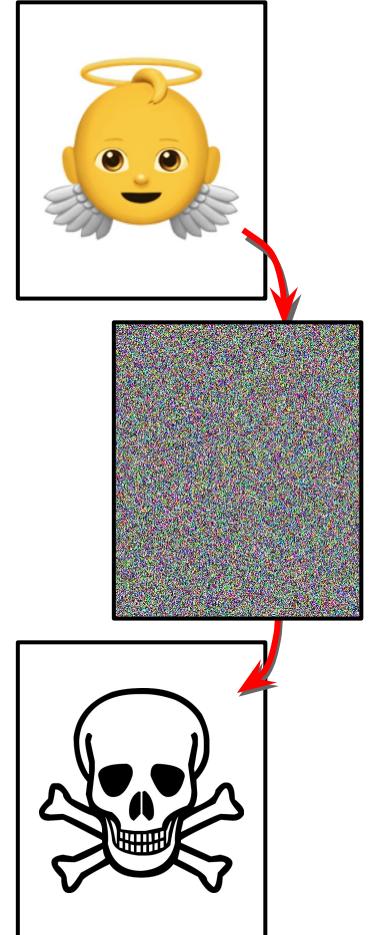
Ange Albertini



Stefan Kölbl

A.K.A.

Abusing one-time pads
with binary polyglots.



ABOUT STEFAN KÖLBL

*Our own views
and opinions.*

- Doing Cryptography research and engineering for 10 years
- Made several cryptanalysis tools ([CryptoSMT](#), [Solvatore](#))
- Contributed to several crypto designs, e.g. [Gimli](#), [Skinny](#), [SPHINCS+](#)

PROFESSIONALLY

- Now Information Security Engineer @ Google
- Worked as Postdoc and Security Consultant
- PhD on symmetric key cryptography

ABOUT ANGE ALBERTINI

- Reverse engineering since 1989
- Author of Corkami
- 6 years at PoC or GTFO*
- Occasional drawer, singer
- File Formats For Ever

PROFESSIONALLY

- 13 years of malware analysis
- 2 years of Information Security Engineer
at Google



My license plate is a CPU.
My phone case is a PDF doc.
My resume is a Super NES/Megadrive rom PDF

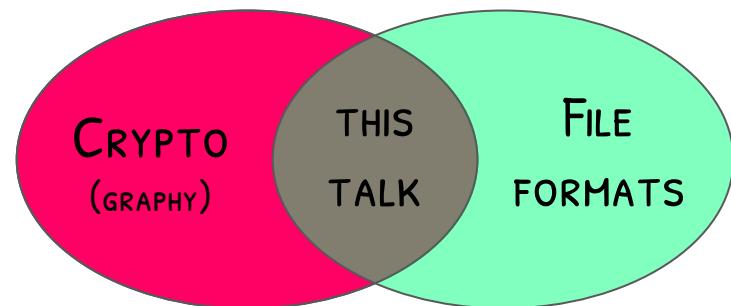
HONEST TRAILER

No new cryptographic finding.
Yet another use of file formats tricks.

GCM mode is standard
Let's raise awareness!

THE CURRENT SLIDE IS AN
HONEST TALK TRAILER

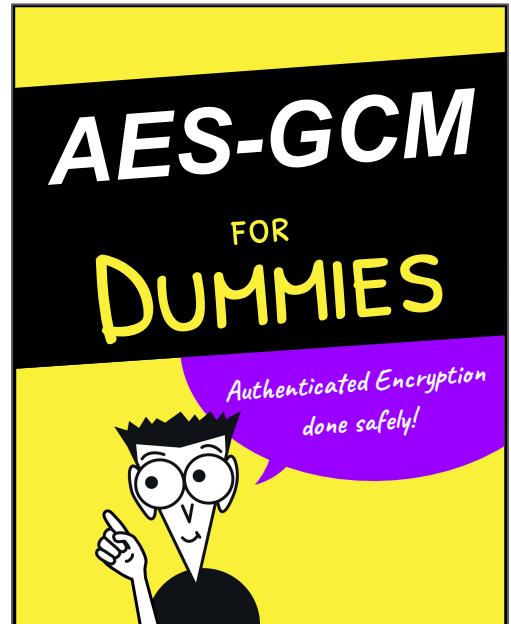
A CORKAMI ORIGINAL PRODUCTION



How to Abuse and Fix Authenticated Encryption Without Key Commitment

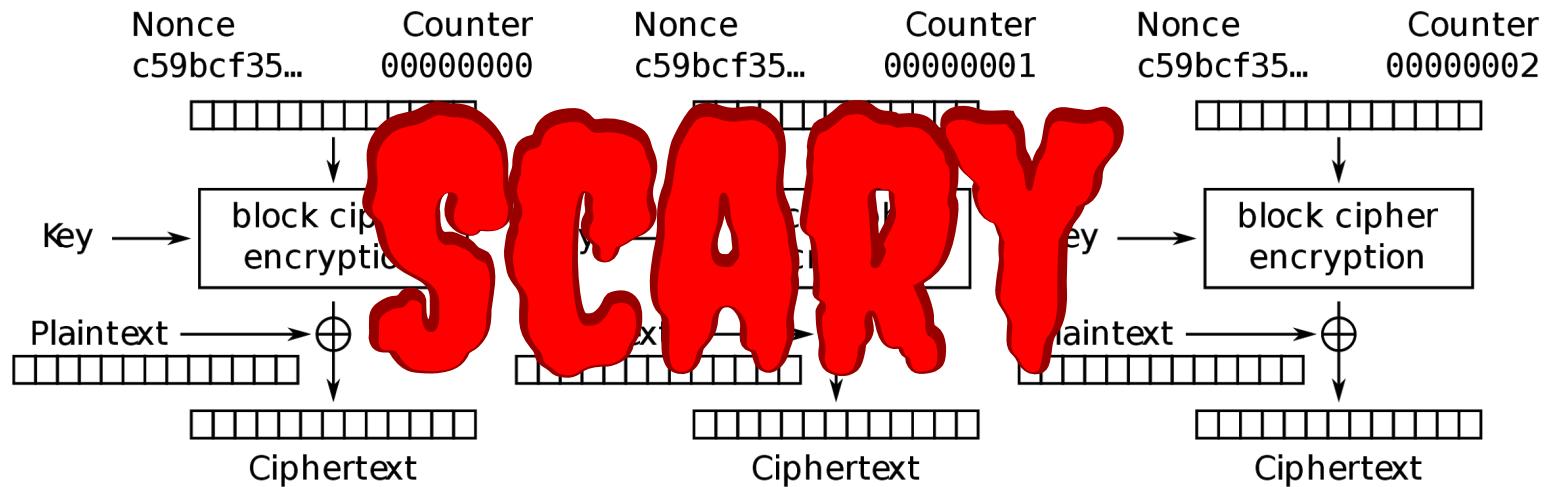
Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, Sophie Schmieg

Cryptology ePrint Archive: Report 2020/1456 – last revised 11 Jun 2021



THIS TALK IS A VULGARISATION OF OUR PAPER

<https://eprint.iacr.org/2020/1456>

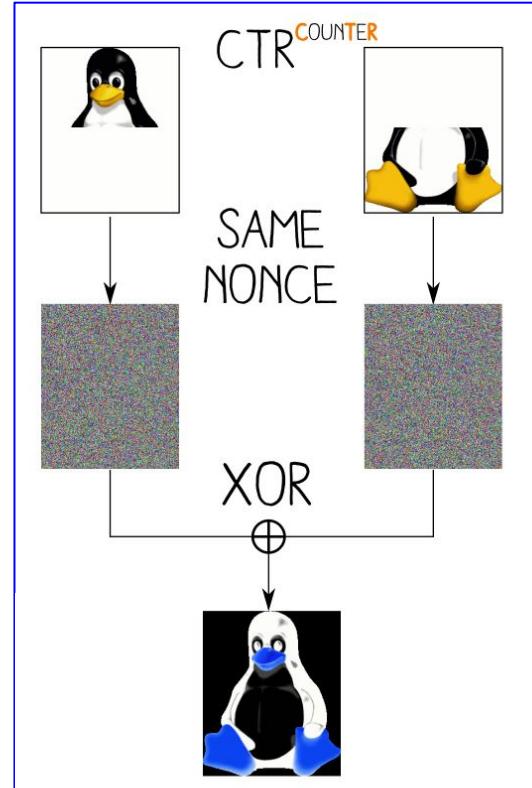


HOW YOU MAY KNOW THE COUNTER MODE (FROM WIKIPEDIA)...

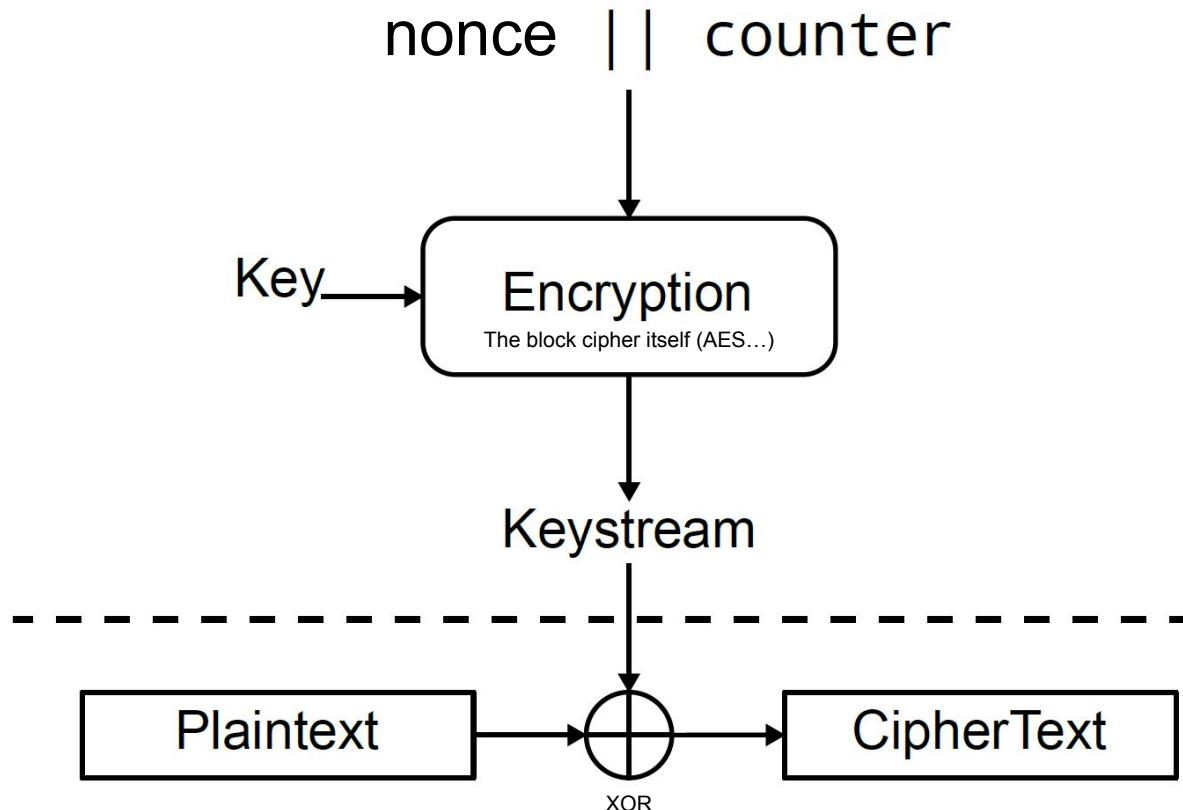
CTR MODE OF OPERATION

- Needs a “used-once” number (a nonce)
- Generates a keystream from (Nonce, Key)
 - with the same length as the plaintext
- Ciphertext = Keystream xor Plaintext
- > CTR acts as a **one-time pad**

In cryptography, the **one-time pad** (OTP) is an [encryption](#) technique that cannot be [cracked](#), but requires the use of a one-time [pre-shared key](#) the same size as, or longer than, the message being sent. In this technique, a [plaintext](#) is paired with a random secret [key](#) (also referred to as a [one-time pad](#)). Then, each bit or character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using [modular addition](#).



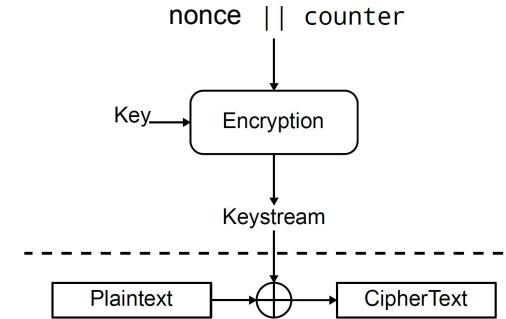
The risks of nonce reuse
(you end up xoring with the same keystream)



HIGH-LEVEL VIEW OF CTR MODE

REMARKS

- [Nonce || Counter] are the blocks being encrypted
- The block cipher is used to generate a keystream, independently of the plaintext and ciphertext
-> parallelizable



CTR mode is a xor with a keystream

CTR decryption and CTR encryption are the same operation

RECAP ON COUNTER MODE

CTR turns a block cipher into a stream cipher

Cipher decryption isn't used

Cipher encryption is just used to generate a keystream

CTR decryption is the same as CTR encryption:
a xor with this keystream

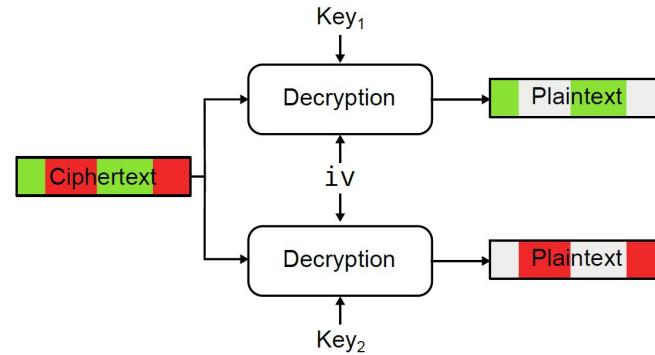
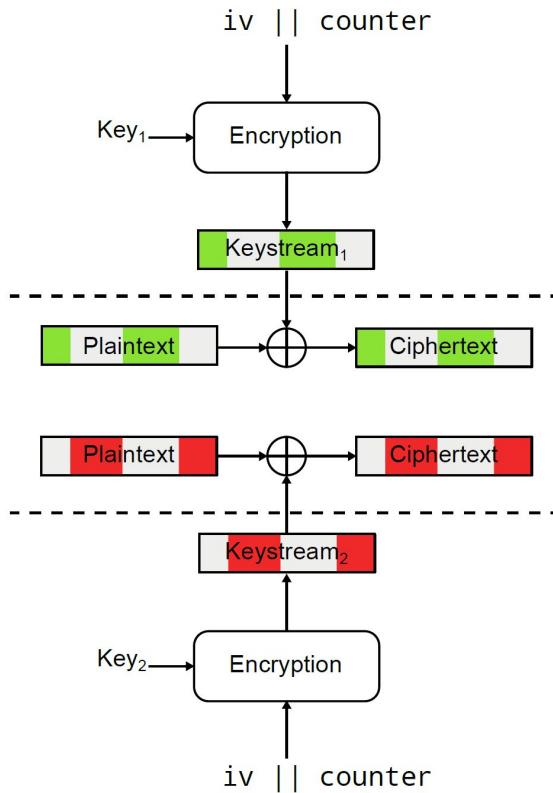
WHAT IF WE DECRYPT WITH A DIFFERENT KEY?

We just end up xor-ing
with a different keystream



CRAFT A CIPHERTEXT THAT GIVES
MEANINGFUL PLAINTEXTS
FOR DIFFERENT KEYSTREAMS

AN “AMBIGUOUS” CIPHERTEXT



MAKE VALID CONTENTS CO-EXIST BY ENCRYPTING AND SLICING

CRAFT A CIPHERTEXT?

We can freely modify the ciphertext

The keystream is set by (Nonce, Key)

Plaintext and ciphertext aren't involved

For a given keystream [which is set by (Nonce, Key)]

if we change ciphertext bytes, we set the plaintext bytes
[it's just a xor against a known keystream]

SEVERAL VALID CONTENTS
IN THE SAME FILE?

Binary polyglots !!!



MITRA: A BINARY POLYGLOT GENERATOR

<https://github.com/corkami/mitra>

*Not a parser nor a validator!
Just knows the bare minimum
about each format*

- Takes 2 files as input
- Identifies file formats (40+ supported)
- Tries different layouts: concatenation, parasites, cavities, zippers...
- Generates binary polyglots

How does it work? ->



278 FORMAT COMBINATIONS

	Delayed Any offset	Cavities start	Magic at offset zero, tolerated appended data	No appended data	Footer
Z Z A R	P I D T	P M	A B B C C E E F F G G I I I I J J N O P L P P R R T	B J P P W	I X
i Z r A	D S C A	S P	R M Z A P B L L I I Z C C D L P P E G S N E N I T I	P a C C A	D Z
p j R	F O M R	4	P 2 B I M F V a F C O 3 D 2 G S G D K G F F F	G v A A S	3
			O L c v A	F F a P P M	v
			2	N	1
Zip	. X X X	X X X X	X X	X X X X X	40
TZ	X . X X	X X X X	X X	X X X X X	40
Arj	X X . X	X X X X	X X	X X X X X	40
RAR	X X X .	X X X X	X X	X X X X X	40
PDF	X X X X	. X X X	X X	X X X X X	40
ISO	X X X X	X . X X	X X	X X X X X	40
DCM	X X X X	X X .	X X	X X X X X	36
TAR	X X X X	X X .	X X	X X X X X	29
PS	X X X X	X X X X	.		8
MP4	X X X X	X X X X	.		8
AR	X X X X	X X X X	.		8
BMP	X X X X	X X X	.		7
BZ2	X X X X	X X X	.		7
CAB	X X X X	X X X X	.		8
CPPIO	X X X X	X X X X	.		8
EBML	X X X X	X X	.		6
ELF	X X X X	X X X	.		7
FLV	X X X X	X X X X	.		8
Flac	X X X X	X X X X	.		8
GIF	X X X X	X X X	.		7
GZ	X X X X	X X X X	.		8
ICC	X X X X	X X	.		6
ICO	X X X X	X X X X	.		8
ID3v2	X X X X	X X X X	.		8
ILDA	X X X X	X X X X	.		8
JP2	X X X X	X X X X	.		8
JPG	X X X X	X X X X	.		8
NES	X X X X	X X X	.		7
OGG	X X X X	X X X X	.		8
PSD	X X X X	X X X X	.		8
LNK	X X X X	X X	.		6
PE	X X X X	X X X	.		7
PNG	X X X X	X X X X	.		8
RIFF	X X X X	X X X X	.		8
RTF	X X X X	X X X X	.		8
TIFF	X X X X	X X X X	.		8
BPG	X X X X	X X X X			8
Java	X X X X	X X X			7
PCAP	X X X X	X X X X			8
PCAPN	X X X X	X X X X			8
WASM	X X X X	X X X X			8
ID3v1					0
XZ					0

ONLY 278 ?

Several sub-formats per container format

Several layouts per combinations

=> 700+ different kinds of polyglots

(that's just how much I tried TBH)

FORMAT	SUBFORMATS
ZIP	JAR, DOCX, APK...
Riff	Wav, AVI, Ani, SfBk...
ISOBM	MP4, JP2, QT, M4V...
Ogg	Vorbis, Flac, Opus...

MITRA USE

Just run the script (Check **mini** or **tiny** PoCs for input files)

```
mitra>mitra.py in\gzip.gz in\rar4.rar
in\gzip.gz
File 1: gzip
in\rar4.rar
File 2: RAR / Roshal Archive

Stack: concatenation of File1 (type GZ) and File2 (type RAR)
Parasite: hosting of File2 (type RAR) in File1 (type GZ)
```

```
mitra>_
```

00:	1F	8B	08	08	27	B2	9D	5E	04	00	g	z	i	p	.	t
10:	x	t	00	01	04	00	FB	FF	g	z	i	p	F2	5C	E9	3A
20:	04	00	00	00	R	a	i	!	1A	07	00	CF	90	73	00	00
30:	0D	00	00	00	00	00	00	00	B9	9A	74	20	90	2D	00	04
40:	00	00	00	04	00	00	00	02	A1	34	21	98	14	99	32	50
50:	1D	30	08	00	20	00	00	00	i	a	i	4	.	t	x	t
60:	00	B0	BA	5C	90	R	A	R	4	C4	3D	7B	00	40	07	00

rar.gz

Actually named: S(24)-GZ-RAR.a7bccab6.rar.gz

the list of hex offsets where the contents change from one format to the other

A TINY GZIP/RAR POLYGLOT BY CONCATENATION

DEPENDING ON WHERE YOU START READING,
YOU'LL GET DIFFERENT RESULTS

mustangstromboneheadlinefeedbackhandrailroadsideshowdownturnoverbookcaseworkshop

CTR TOOLS FOR MITRA

```
mitra\utils\ctr>brioche.py "S(24)-GZ-RAR.a7bccab6.rar.gz" gzip-rar4.ctr
Generated output: gzip-rar4.ctr
Tests:
openssl enc -in gzip-rar4.ctr -out output1.rar -aes-128-ctr
    -iv 00000000000000000000000000000000 -K 4e6f773f000000000000000000000000
openssl enc -in gzip-rar4.ctr -out output2.gz -aes-128-ctr
    -iv 00000000000000000000000000000000 -K 4c347433722121210000000000000000

mitra\utils\ctr>_
```



output1.rar

```
0000: BF BF 9E 87 55 2C 8C 83 6F 40 7B B7 64 52 8F FF  
0010: 38 60 46 3E 2E F6 87 B9 3E 85 DD 7B E1 9E 61 47  
0020: 44 DB 84 98 52 61 72 21 1A 07 00 CF 90 73 00 00  
0030: 0D 00 00 00 00 00 00 00 B9 9A 74 20 90 2D 00 04  
0040: 00 00 00 04 00 00 00 02 A1 34 21 98 14 99 32 50  
0050: 1D 30 08 00 20 00 00 00 72 61 72 34 2E 74 78 74  
0060: 00 B0 BA 5C 90 52 41 52 34 C4 3D 7B 00 40 07 00
```

77 PçU, iâo@{ dRÅ
8 F>..÷ç||>a {ßPaG
D äyRar! Ès
Üt É-
i4!ÿ Ö2P
0 rar4.txt
||\ÉRAR4--= { @

output2.gz

```
0000: 1F 8B 08 08 27 B2 9D 5E 04 00 67 7A 69 70 2E 74  
0010: 78 74 00 01 04 00 FB FF 67 7A 69 70 F2 5C E9 3A  
0020: 04 00 00 00 9B 48 4D CE A2 B1 E8 58 4F 5D 70 86  
0030: 1D 32 32 B0 D2 30 3A E4 49 AC 26 16 6C FE 48 97  
0040: 23 F8 9C 80 2D B7 F6 3A 62 82 CE D1 82 B2 1D BA  
0050: 54 27 96 F7 22 14 F6 31 3E 79 36 16 54 45 80 DA  
0060: 76 03 B5 90 E5 CE E7 EA E4 F3 BB E7 9B 6D EF 0B
```

ÿ ' ¥^ gzip.t
xt √ gzip≥\θ:
¢HM|óΦXO]på
22 T0:ΣI%& l■Hù
°£Ç- ÷:bé|| e||
T'ü~" ÷1>y6 TEÇΓ
v Éσ||τΩΣ≤|| τ¢mø

2 VALID FILES FROM THE SAME CIPHERTEXT

MORE THAN POLYGLOTS

- 2 related files coming from the same ciphertext
- 1 format is hidden when the other is in clear
 - > hides malicious payload
 - > bypass polyglot blacklisting (Adobe Reader)

```
$ unrar vl output1.rar

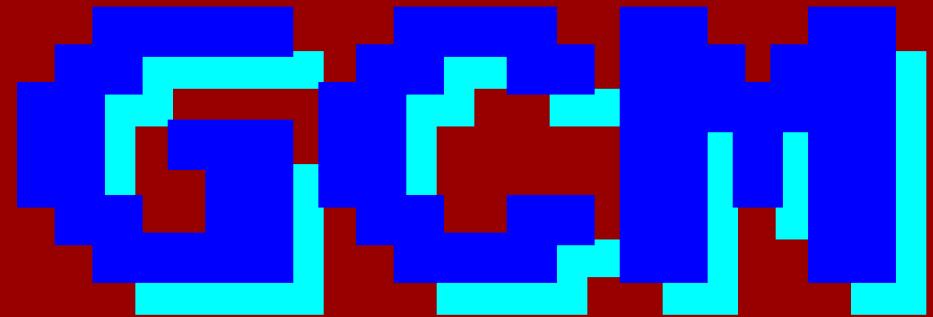
UNRAR 5.61 beta 1 freeware      Copyright (c) 1993-2018 Alexander Roshal

Archive: output1.rar
Details: RAR 4, SFX

Attributes      Size      Packed Ratio      Date      Time      Checksum      Name
-----  -----  -----  -----  -----  -----  -----  -----
..A....          4          4 100%  2020-01-18 19:08  982134A1  rar4.txt
-----  -----  -----  -----  -----  -----  -----  -----
                           4 100%                                         1
```

```
$ gzip -t output2.gz

gzip: output2.gz: decompression OK, trailing garbage ignored
```



Here comes a
new challenger!

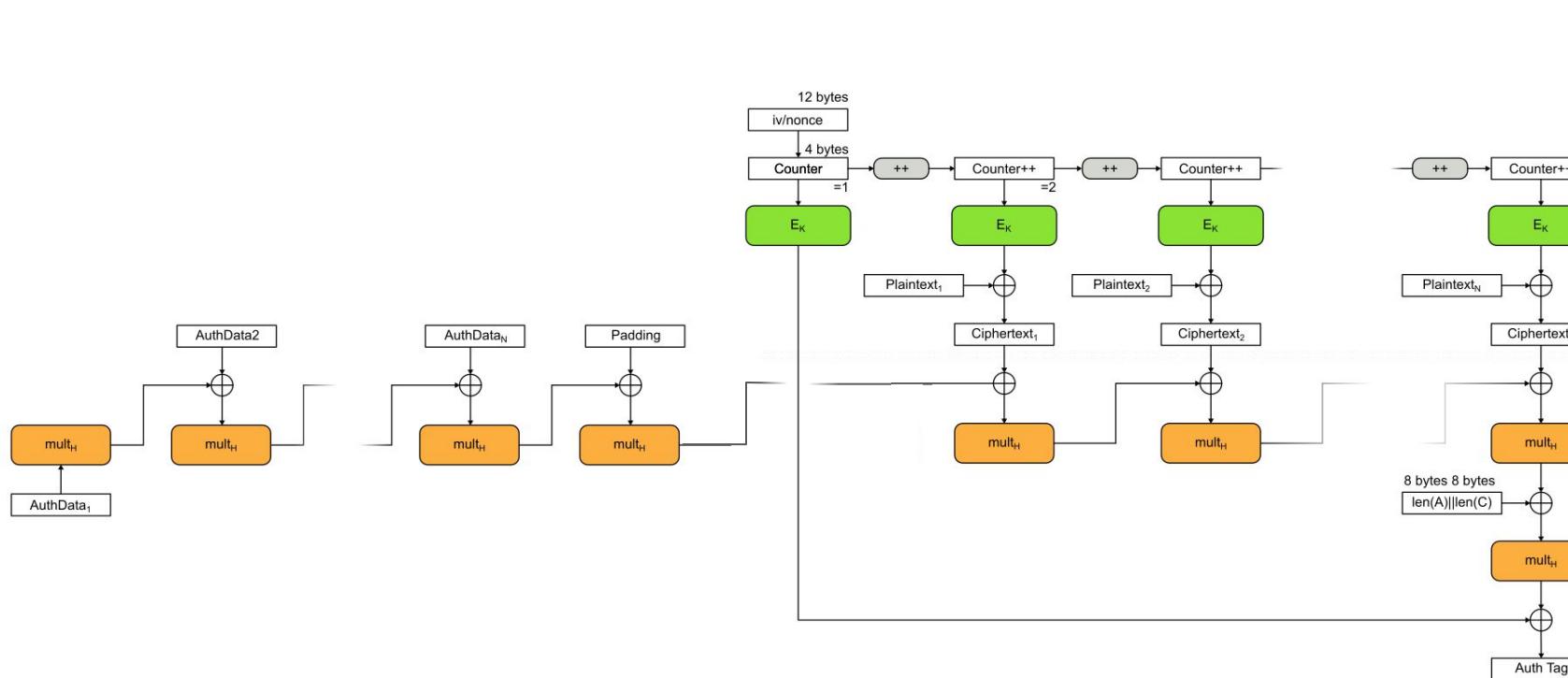
You DO NOT WANT UNAUTHENTICATED ENCRYPTION

CBC PADDING ORACLE ATTACKS ANYONE ?

https://en.wikipedia.org/wiki/Padding_oracle_attack

<https://github.com/google/security-research/security/advisories/GHSA-f5pg-7wfw-84q9>

AUTHENTICATED ENCRYPTION = AE
AUTHENTICATED ENCRYPTION w/ ASSOCIATED DATA = AEAD



GALOIS/COUNTER MODE

MODES OF OPERATION:

Electronic CodeBook

CounTeR

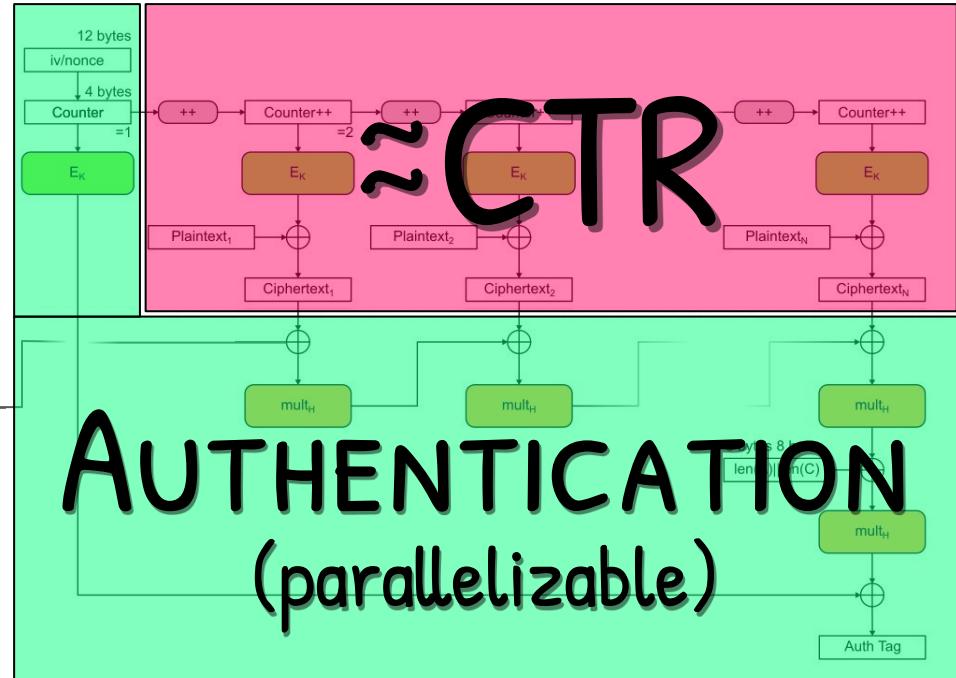
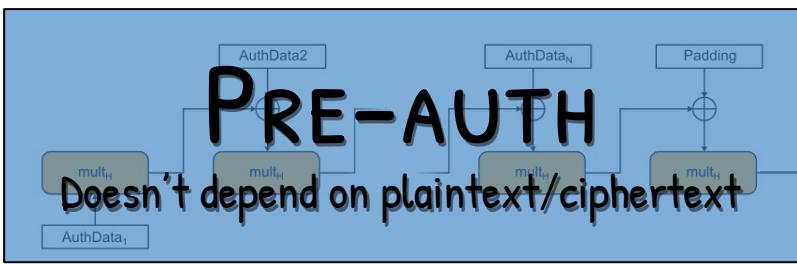
Galois/Counter Mode

Cipher Block Chaining

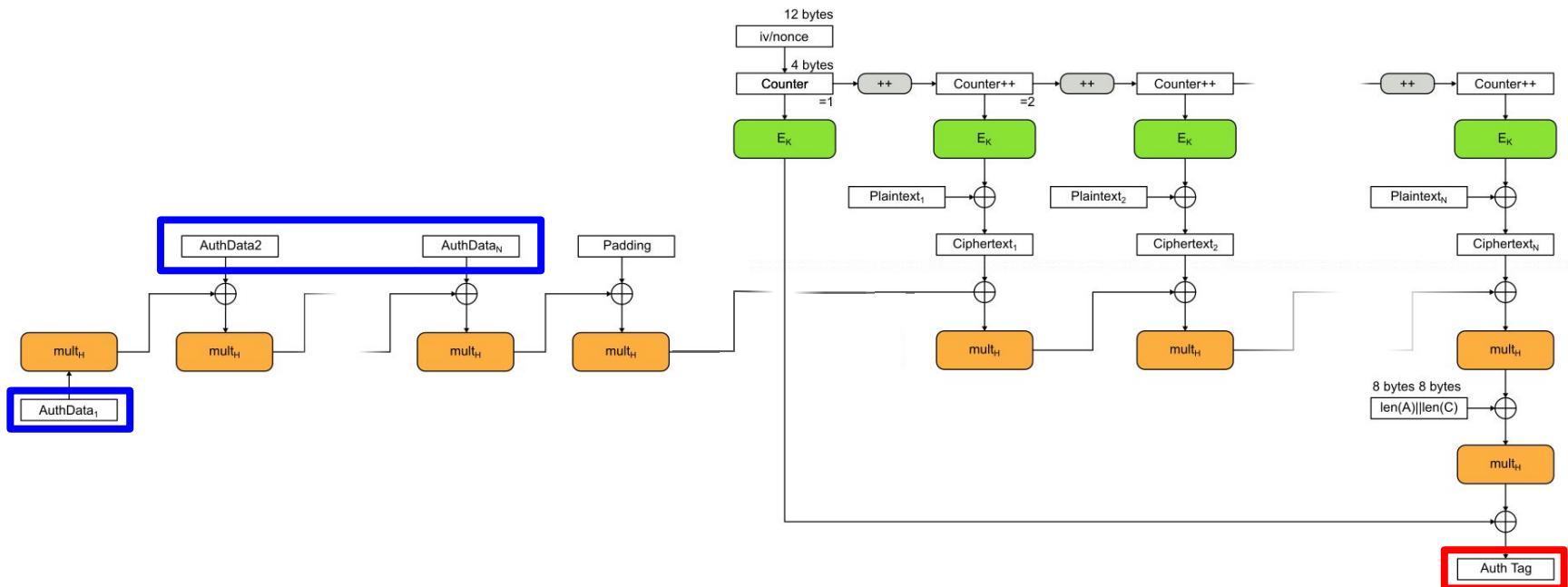
Cipher FeedBack

Output FeedBack

FTR



IT'S JUST CTR WITH AUTHENTICATION

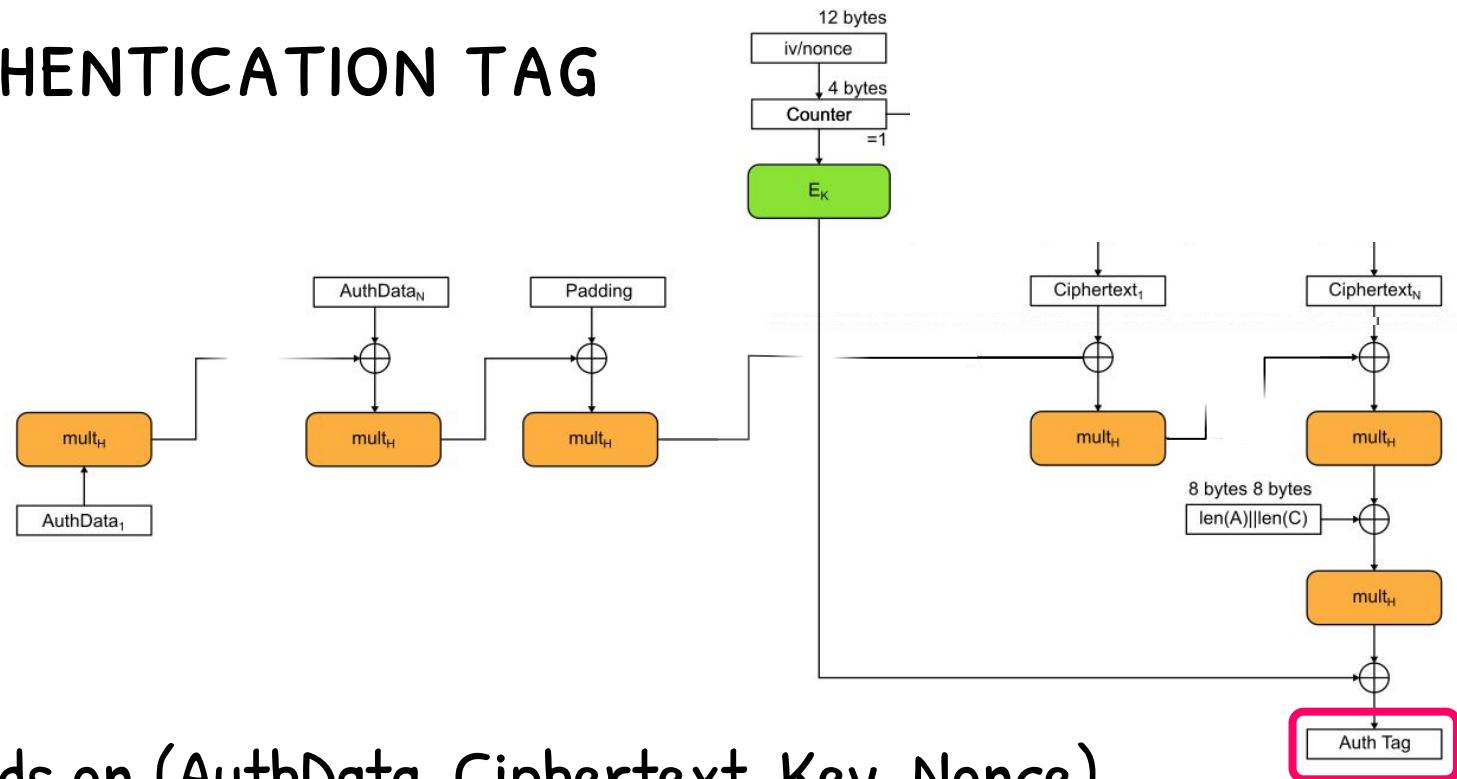


NEW INPUTS/OUTPUTS

CTR: (PLAINTEXT) \rightarrow CIPHERTEXT

GCM: (PLAINTEXT, AUTHDATA) \rightarrow CIPHERTEXT, TAG

AUTHENTICATION TAG



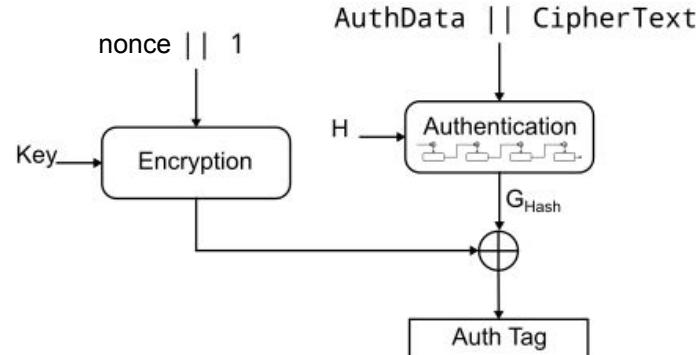
Depends on (AuthData, Ciphertext, Key, Nonce)

Re-computed after decryption, compared to the stored value

GCM MODE

CTR for encryption + G_{hash} for Authentication on Ciphertext

The authentication tag depends on (AuthData, ciphertext, Key, Nonce)
Any modification to any of these will fail the authentication



So WE CAN'T USE
A DIFFERENT KEY?

WRONG!

We can make it so two decryptions w/ different keys both pass verification

We just need to sacrifice a block and do some computation
(it was known from the beginning - in 2004 - but not seen as a risk)

GCM FAILURES IN THE WILD

Facebook, Google, Amazon...

FaceBook messenger

RECIPIENT AND ABUSE TEAM SEE DIFFERENT IMAGES

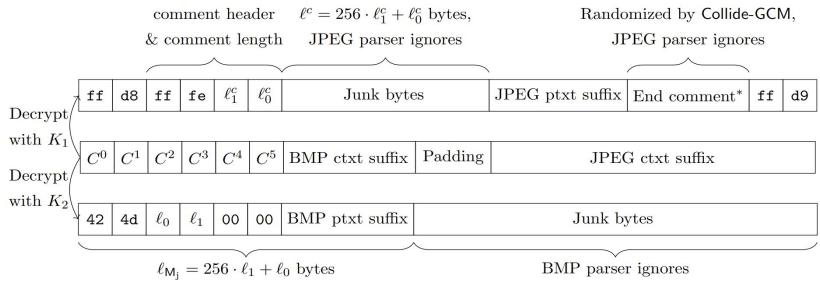
3 Invisible Salamanders: Breaking Facebook's Franking

In this section we demonstrate an attack against Facebook's message franking. Facebook uses AES-GCM to encrypt attachments sent via Secret Conversations [16], the end-to-end encryption feature in Messenger. The attack creates a "colliding" GCM ciphertext which decrypts to an abusive attachment via one key and an innocuous attachment via another. This combined with the behavior of Facebook's server-side abuse report generation code prevents abusive messages from being reported to Facebook. Since messages in Secret Conversations are called "salamanders" by Facebook (perhaps inspired by the Axolotl ratchet used in Signal, named for an endangered salamander), ensuring Facebook does not see a message essentially makes it an *invisible salamander*. We responsibly disclosed the vulnerability to Facebook. They have remediated it and have given us a bug bounty for reporting the issue.

Fast Message Franking: From Invisible Salamanders to Encryption <https://eprint.iacr.org/2019/016.pdf>
Hunting Invisible Salamanders BlackHat 2020



A BMP \leftrightarrow JPEG "collision"
(with 6b of overlap)



SUBSCRIBE WITH GOOGLE (INTERNAL)

Caught during the design phase. 😅

No key commitment -> different contents for different keys

Malicious content could be served depending on the user

Solution: store and check a hash of the key

AMAZON ENCRYPTION SDK

CVE-2020-8897

Fixed in [version 2.0.0](#) (breaking compatibility)

Is GCM BROKEN?

The property we exploit does not violate
any security goals of authenticated encryption

Many other encryption modes do not bind the key to the ciphertext

Moar impact?

Some systems with key rotations
just try all the keys of the key ring
until one decryption is authenticated
(newest one first)

RESULT

TIMECRYPTION

If you can abuse the key generation algorithm,
You can craft a tag that will be valid now for one key
and also in the future with a different key
And the new key will just be silently used - by design

What you want now. What you want later. You control both.

CONTROL THE PRESENT.

CONTROL THE FUTURE.

IT'S NOT A BUG.

TimeCrypton



How TO

Use Mitra to generate binary polyglots

Mitra/utils/GCM:

- takes a mitra polyglot, encrypts and slices
- corrects authentication (crafts collision)

STEP 1: RUN TOOL

```
mitra\utils\gcm>meringue.py S(24)-GZ-RAR.a7bccab6.rar.gz gzip-rar4.gcm
key 1: Now?
key 2: L4t3r!!!
ad   : MyVoiceIsMyPass!
blocks: /
```

Default values

```
Computing 9 coefficients.
Coef to be inverted: 4494a66081751930eed248b6fd11c74e (already computed)
```

```
mitra\utils\gcm>_
```

Craft ciphertext:
launch Meringue, and wait 1 min if computation is needed

```
Collide-GCM( $K_1, K_2, N_a, C$ ):  

 $H_1 \leftarrow E_{K_1}(0^{128}) ; H_2 \leftarrow E_{K_2}(0^{128})$ 
 $P_1 \leftarrow E_{K_1}(N_a + 1) ; P_2 \leftarrow E_{K_2}(N_a + 1)$ 
 $mlen \leftarrow |C|/128 + 1$ 
 $lens \leftarrow \text{encode}_{64}(0) \parallel \text{encode}_{64}(|C| + 128)$ 
 $acc \leftarrow lens \times_{GF} (H_1 \oplus H_2) \oplus P_1 \oplus P_2$ 
For  $i = 1$  to  $mlen - 1$ :  

 $H_i \leftarrow H_1^{mlen+2-i} \oplus H_2^{mlen+2-i}$ 
 $acc \leftarrow acc \oplus C[i] \times_{GF} H_i$ 
 $inv \leftarrow (H_1^2 \oplus H_2^2)^{-1}$ 
 $C_{mlen} \leftarrow acc \times_{GF} inv$ 
 $C_a \leftarrow C \parallel C_{mlen}$ 
 $T \leftarrow \text{GHASH}(H_1, C_a) \oplus P_1$ 
Return  $N_a \parallel C_a \parallel T$ 
```



Authentically decrypt

```
mitra\utils\gcm>decrypt.py gzip-rar4.gcm
key1: b'Now?'
key1: b'L4t3r!!!!'
ad: b'MyVoiceIsMyPass!'
nonce: 0
tag: b'6c4fd7abc224ac5323bca8b6a5f52f28'
Success!
```

```
plaintext1: b'1f8b080827b29d5e0400677a69702e74' ...
plaintext2: b'5f508c90ee9ba2b1bcb68fdb65e5ef2' ...
```

```
mitra\utils\gcm>_
```

Verify files

```
mitra\utils\gcm>file output1.gz
output1.gz: gzip compressed data, was "gzip.txt", last modified: Mon Apr 20
14:31:03 2020, max speed, from FAT filesystem (MS-DOS, OS/2, NT)
```

```
mitra\utils\gcm>unrar t output2.rar
```

```
UNRAR 5.40 beta 2 x64 freeware      Copyright (c) 1993-2016 Alexander Roshal
```

```
Testing archive output2.rar
```

```
Testing      rar4.txt
All OK
```

OK

```
mitra\utils\gcm>_
```

STEP 2: VERIFY

OTHER EXAMPLES (1/2)

```
/utils/gcm$ ./decrypt.py examples/gif-dicom.gcm
key1: Now?
key1: L4t3r!!!
ad: MyVoiceIsMyPass!
nonce: 0
tag: 819fa575fa548b4ece54d573902e8f4f
Success!

plaintext1: 47494638376119000700800100000000 ...
plaintext2: 0792c2a0fe4826efbf66896df2e7086 ...

/utils/gcm$ file output*
output1.gif: GIF image data, version 87a, 25 x 7
output2.dcm: DICOM medical imaging data
/utils/gcm$ _
```

```
/utils/gcm$ ./decrypt.py examples/pdf-exe.gcm
key1: Now?
key1: L4t3r!!!
ad: AssociatedDataForAmbiguousCipher
nonce: 59334
tag: 43cf7debd82f644c6ec3f7873c91b3c4
Success!

plaintext1: 255044462d312e330a25c2b5c2b60a0a ...
plaintext2: 4d5a1526c3461a3f30486ccfd93e4a95 ...
```

```
/utils/gcm$ pdftotext output1.pdf -
http://www.evil.com
```

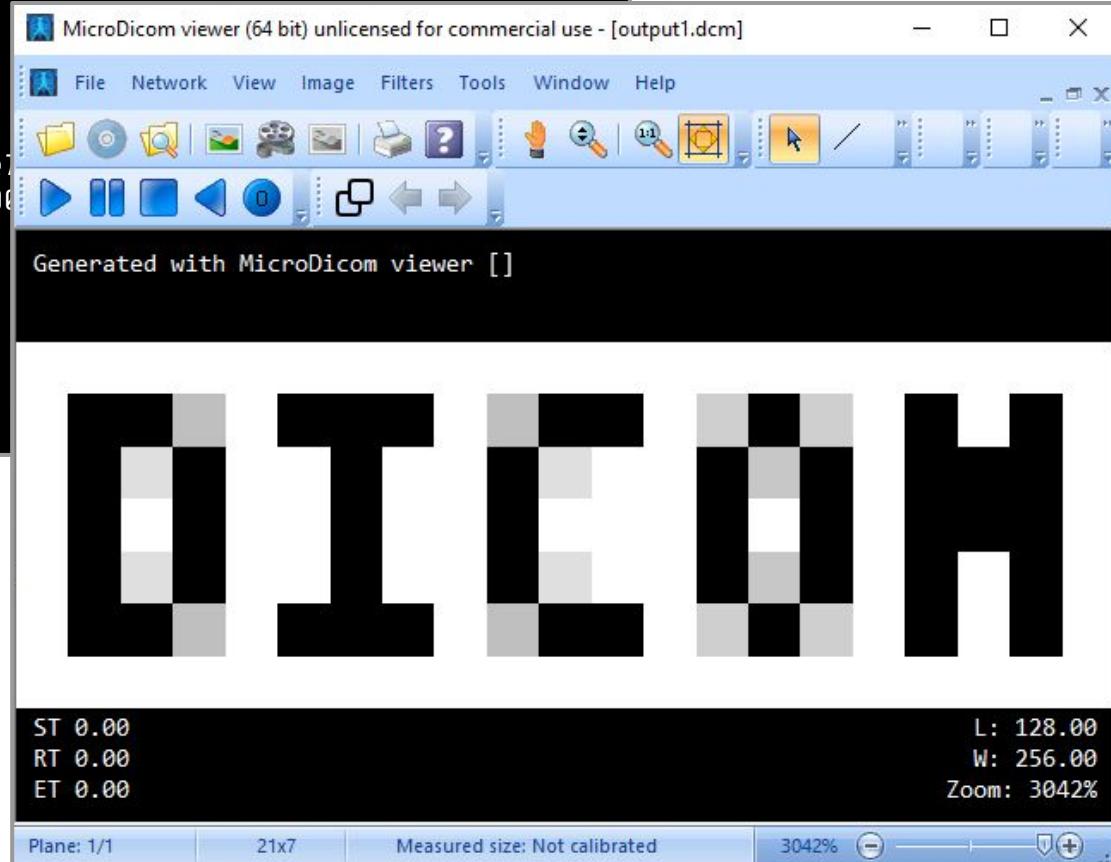
```
/utils/gcm$ wine ./output2.exe
32bit PE
/utils/gcm$ _
```

```
mitra\utils\gcm>..\decrypt.py dicom-pe.gcm  
key1: b'Now?'  
key1: b'L4t3r!!!'  
ad: b'AssociatedDataForAmbiguousCipher'  
nonce: 0  
tag: b'c1c2de2194703324afe5e78347ae1d3d'  
Success!
```

```
plaintext1: b'0d818498c9293fefb8b6e897df2e'  
plaintext2: b'4d5a000000000000000000000000000000
```

```
mitra\utils\gcm>dicom-pe-1.28b89b4c.dcm
```

```
mitra\utils\gcm>dicom-pe-2.28b89b4c.exe  
32-bit PE  
mitra\utils\gcm>
```



OTHER EXAMPLES (2/2)

OVERLAPPING BYTES?

So far, each ciphertext byte belongs to strictly one payload

CONTROL TWO OUTPUTS AT ONCE?

We know that $C = P_1 \wedge Ks_1$ and $C = P_2 \wedge Ks_2$

If we want to control some bytes of both P_1 and P_2 ,

we need that $P_1 \wedge P_2 = Ks_1 \wedge Ks_2$

We know that Keystreams depend on Keys, Nonce and Counter

-> bruteforce a nonce that gets the right xor value for both keys

CRYPTO-POLYGLOTS (w/ OVERLAPPING BYTES)

- 2 formats starting at offset zero can coexist
Ex:PDF/PE, JPG/PNG, ...
- both formats can be the same but w/ different contents
Ex: JPG/JPG

	Minimal start offset												Variable offset	Unsupported parasite																							
	1	2	4	8	9	16	20	23	28	34	40	64	94	132	12	28																					
					12			26	32	36		68	112	226		16																					
1*	P	P	J	F	M	T	F	W	G	P	R	I	R	B	C	I	P	C	J	P	E	A	P	I	I	J	W	B	O	B	E	G	L	N			
2^	S	E	P	1	P	I	L	A	Z	N	I	D	T	M	P	L	S	A	P	C	L	R	C	C	a	A	P	G	Z	B	I	N	E				
3+	G	a	4	F	V	D	G	F	3	F	P	I	D	D	B	2	A	F	A	O	C	v	S	G	G	2	M	F	K	S							
4+	C	F		F	v		O	A		P		P		a		M		L																			
															2		N																				
															G																						

(the table could go on but would take too long to bruteforce)

X: automated ?: likely possible
M: manual !: unknown

How To (1/2)

nonce.py just does what you expect

mitra/utils/gcm

PDF and PE both have short magic,
so it's very fast to bruteforce

```
mitra\utils\gcm>nonce.py
key1: b'4e6f773f' (b'Now?')
key2: b'4c34743372212121' (b'L4t3r!!!')
hdr1: b'2550' (b'%P')
hdr2: b'4d5a' (b'MZ')
start: 2 (GCM)
Results:
- nonce: 00059334 0x0000e7c6
- nonce: 00094306 0x00001002
- nonce: 00173940 0x0002a774
- nonce: 00407943 0x00063987
- nonce: 00405901 0x0006318d
- nonce: 00535752 0x00082cc8
- nonce: 00668403 0x000a32f3
- nonce: 01314505 0x00140ec9

mitra\utils\gcm>_
```

How TO (2/2)

Just use the computed nonce

```
mitra\utils\gcm>decrypt.py examples\pdf-exe.gcm
key1: b'Now?'
key1: b'L4t3r!!!'
nonce: 59334
ad: b'AssociatedDataForAmbiguousCipher'
tag: b'43cf7debd82f644c6ec3f7873c91b3c4'
Success!

plaintext1: b'255044462d312e330a25c2b5c2b60a0a' ...
plaintext2: b'4d5a1526c3461a3f30486ccfd93e4a95' ...

mitra\utils\gcm>_
```

```
mitra\utils\gcm>file output1.pdf output2.exe
output1.pdf: PDF document, version 1.3
output2.exe: PE32 executable (console) Intel 80386, for MS Windows

mitra\utils\gcm>pdftotext output1.pdf -
http://www.corkami.com

mitra\utils\gcm>output2.exe
32bit PE

mitra\utils\gcm>_
```

IT'S FULLY GENERIC, AND WORKS WITH MOST STANDARD FILES

```
mitra\utils\gcm>decrypt.py examples\pdf-viewer.gcm
key1: b'Now?'
key1: b'L4t3r!!!'
ad: b'MyVoiceIsMyPass!'
nonce: 59334
tag: b'deadbeefcafebabe0000000000000000'
Success!
```

```
plaintext1: b'4d5a1526c3461a3f30486ccfd93e4a95' ...
plaintext2: b'255044462d312e330a25c2b5c2b60a0a' ...
```

```
mitra\utils\gcm>file output1.exe output2.pdf
output1.exe: PE32 executable (GUI) Intel 80386, for MS Windows,
output2.pdf: PDF document, version 1.3
```

```
mitra\utils\gcm>output1.exe output2.pdf
```

```
mitra\utils\gcm>_
```

A PDF viewer executable and an academic document ->
(neither source is public - only the binaries are)

The screenshot shows a PDF viewer window titled "output2.pdf - SumatraPDF". The document content is as follows:

How to Abuse and Fix Authenticated Encryption Without Key Commitment

Ange Albertini¹, Thai Duong¹, Shay Gueron^{2,3}, Stefan Kölbl¹, Atul Luykx¹, and Sophie Schmieg¹

¹Security Engineering Research, Google
²University of Haifa
³Amazon

Abstract

Authenticated encryption (AE) is used in a wide variety of applications, potentially in settings for which it was not originally designed. Recent research tries to understand what happens when AE is not used as prescribed by its designers. A question given relatively little attention is whether an AE scheme guarantees “key commitment”: ciphertext should decrypt to a valid plaintext only under the key that was used to generate the ciphertext. As key commitment is not part of AE’s design goal, AE schemes in general do not satisfy it. Nevertheless, one would not expect this seemingly obscure property to have much impact on the security of actual products. In reality, however, products do rely on key commitment. We discuss three recent applications where missing key commitment is exploitable in practice. We provide proof-of-concept attacks via a tool that constructs AES-GCM ciphertext which can be decrypted to two plaintexts valid under a wide variety of file formats, such as PDF, Windows executables, and DICOM. Finally we discuss two solutions to add key commitment to AE schemes which have not been analyzed in the literature: one is a generic approach that adds an explicit key commitment scheme to the AE scheme, and the other is a simple fix which works for AE schemes like AES-GCM and ChaCha20Poly1305, but requires separate analysis for each scheme.

1 Introduction

Authenticated Encryption. Symmetric-key encryption (SKE) has been the source of many attacks over the years. The main culprit is the use of malleable, unauthenticated schemes like CBC, and their susceptibility to padding oracle [Vau02] and related attacks. Such attacks are found with as much regularity against systems designed in the 90’s as they are today; recent research [FIM20] shows that CBC continues to be an attack vector.

Beck et al. [BZG20] cite flaws in Apple iMessage, OpenPGP, and PDF encryption as examples to argue that practitioners are often only convinced that unauthenticated SKE

is insecure when they see a proof-of-concept exploit. Similar efforts are deemed necessary to demonstrate the exploitability of cryptographic algorithms such as SHA-1 [SBK⁺17].

The vast majority of applications should default to using authenticated encryption (AE) [BN00, KY00], a well-studied primitive which avoids the pitfalls of unauthenticated SKE with relatively small performance overhead. AE schemes are used in widely adopted protocols like TLS [Res18], standardized by NIST [NIS07a, NIS07b] and ISO [ISO09], and are the default SKE option in modern cryptographic libraries such as NaCl [nacl] and Tink [tink].

With AE more widely used, recent research focuses on its security guarantees in settings which push the boundaries and assumptions of conventional AE, such as understanding nonces [RS06], multiple decryption errors [BDPS13], unverified plaintext [ABL⁺14], side channel leakage [BMOS17], multi-user attacks [BT16], boundary hiding [BDPS12], streaming AE [HRRV15], and variable-length tags [RNV16]. Furthermore, constructions and security models have received additional scrutiny due to two recent competitions focusing on AE: CAESAR [CAE14] and the NIST lightweight cryptography competition [nis].

Key Commitment. Among the extended, desirable properties explored is the relatively little-studied of AE *key commitment*, which we intuitively explain as follows.

One of the defining design goals of AE is to provide ciphertext integrity: if recipient A decrypts a ciphertext with the key K_A into a valid plaintext, meaning authentication succeeds, then A knows that the ciphertext has not been modified during transmission. Intuitively, one might mistakenly assume that the integrity guarantee extends to keys, i.e., if some other recipient B decrypts the same ciphertext with their key K_B , then decryption would fail. However, this is neither an AE design goal, nor a guaranteed property, and there are secure and globally deployed AE schemes where both recipients can successfully decrypt the same ciphertext.

Key commitment guarantees that a ciphertext can only be decrypted under the same key used to produce it from some

RECAP ON OVERLAPPING POLYGLOTS

- Requires --overlap in Mitra:
 - It's disabled by default as the polyglots don't work as-is
 - The overlapping data is saved in the file **name**
0(4-84)-JPG[ICC]{**0000001C0**}.5ecbd8cf.jpg.icc
- Longer overlap takes longer to bruteforce
- That's all

NOT JUST POLYGLOTS

We can make 2 files with the same type but with different contents

Declare a content w/ its length depending on decryption:

- > different comment lengths

- > different data being parsed

Ex: DUAL-CONTENTS JPG PAIR

The minimal structure to declare
a JPG file type w/ a comment is:

FF D8 FF FE XX XX

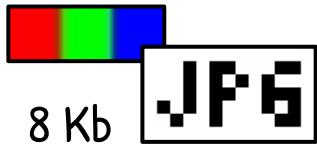
- JPEG magic signature
- comment declaration
- comment length

```
mitra\utils\gcm>nonce.py
key1: 4e6f773f ('Now?')
key2: 4c34743372212121 ('L4t3r!!!!')
hdr1: ffd8ffff ('\xff\xd8\xff\xfe')
hdr2: ffd8ffff ('\xff\xd8\xff\xfe')
start: 2 (GLM)
Results:
- nonce: 8108314280 (0x1E34B0EA8)

mitra\utils\gcm>_
```

REUSABLE WITH ANY JPG FILES PAIR.

jpg.py (no need of recomputation)



Where the magic happens: random stuff + mask

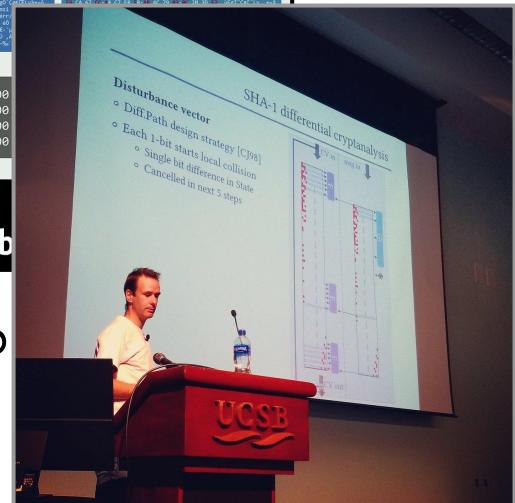
File A Collision blocks File B

File A	Collision blocks	File B
0c 00 00 02 10 00		
bc 00 00 1a 00 00		
0c 00 00 02 00 00		
bc 00 00 18 00 00		

=> generate one file from the other.

Exploiting - Ange Albertini

370 Kb



EVEN FURTHER

Unlike SHA-1, the G_{Hash} function is invertible - "it's just maths".

- > you can even set a pre-defined tag
 - by using an extra block of the ciphertext for correction.
- > a file can tell in advance which tag will authenticate the decryption ;)

You could also correct the Authentication Data instead of the ciphertext.

How to Abuse and Fix Authenticated Encryption Without Key Commitment

Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, Sophie Schmieg

Cryptology ePrint Archive: Report 2020/1456 – last revised 11 Jun 2021

Abstract: Authenticated encryption (AE) is used in a wide variety of applications, potentially in settings for which it was not originally designed. Recent research tries to understand what happens when AE is not used as prescribed by its designers. A question given relatively little attention is whether an AE scheme guarantees “key commitment”: ciphertext should only decrypt to a valid plaintext under the key used to generate the ciphertext. Generally, AE schemes do not guarantee key commitment as it is not part of AE’s design goal. Nevertheless, one would not expect this seemingly obscure property to have much impact on the security of actual products. In reality, however, products do rely on key commitment. We discuss three recent applications where missing key commitment is exploitable in practice. We provide proof-of-concept attacks via a tool that constructs AES-GCM ciphertext which can be decrypted to two plaintexts valid under a wide variety of file formats, such as PDF, Windows executables, and DICOM. Finally we discuss two solutions to add key commitment to AE schemes which have not been analyzed in the literature: a generic approach that adds an explicit key commitment scheme to the AE scheme, and a simple fix which works for AE schemes like AES-GCM and ChaCha20Poly1305, but requires separate analysis for each scheme.

Category / Keywords: secret-key cryptography / authenticated encryption, robustness, commitment, key commitment, AES-GCM

Date: received 17 Nov 2020, last revised 11 Jun 2021

Contact author: aluykx at google com

Available format(s): [PDF](#) | [BibTeX Citation](#)

Note: Changes from previous version: improved overall readability, added a subsection on how to choose a fix, further clarified contributions versus prior work, fixed an oversight in the key commitment definition, and added full explanation of proof of theorem 1 (for AES-GCM padding fix).

```

$ wget https://eprint.iacr.org/2020/1456.pdf
--2020-11-19 11:09:15-- https://eprint.iacr.org/2020/1456.pdf
Resolving eprint.iacr.org (eprint.iacr.org)... 216.184.8.41
Connecting to eprint.iacr.org (eprint.iacr.org)|216.184.8.41|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 2464928 (2.4M) [application/pdf]
Saving to: '1456.pdf'

1456.pdf      100%[=====>]   2.35M  1.65MB/s    in
2020-11-19 11:09:17 (1.65 MB/s) - '1456.pdf' saved [2464928/2464928]

```

```

$ openssl enc -in 1456.pdf -out cryptd -aes-128-ctr -iv
00000000000000000000e7c60000002 -K 4e6f773f000000000000000000000000
$ openssl enc -in cryptd -out viewer.exe -aes-128-ctr -iv
00000000000000000000e7c60000002 -K 4c347433722121210000000000000000
$ wine viewer.exe 1456.pdf

```

THE PDF ARTICLE FILE IS ALSO
A PDF VIEWER EXECUTABLE!

1456.pdf - SumatraPDF

File View Go To Zoom Favorites Settings Help

Page: 1 / 21

How to Abuse and Fix Authenticated Encryption Without Key Commitment

Ange Albertini¹, Thai Duong¹, Shay Gueron^{2,3}, Stefan Kölbi¹, Atul Luykx¹, and Sophie Schmied¹

¹Security Engineering Research, Google
²University of Haifa
³Amazon

Abstract

Authenticated encryption (AE) is used in a wide variety of applications, potentially in settings for which it was not originally designed. Recent research tries to understand what happens when AE is not used as intended by its designers. A central question is precisely how often when an AE scheme guarantees “key commitment” ciphertext should decrypt to a plain plaintext only under the key that was used to generate the ciphertext. As key commitment is not part of AE’s design goal, AE schemes in general do not satisfy it. Nevertheless, one would not expect this seemingly obscure property to have much impact on the security of actual products. In reality, however, products do rely on key commitment. We discuss three recent applications where missing key commitment is exploitable in practice. We provide proof-of-concept attacks via a tool that constructs AES-GCM ciphertext which can be decrypted to two plaintexts valid under a wide variety of file formats, such as PDF, Windows executables, and DICOM. Finally we discuss two solutions to add key commitment to AE schemes which have not been analyzed in the literature: one is a generic approach that adds an explicit key commitment scheme to the AE scheme, and the other is a simple fix which works for AE schemes like AES-GCM and ChaCha20Poly1305, but requires separate analysis for each scheme.

1 Introduction

Authenticated Encryption. Symmetric-key encryption (SKE) has been the source of many attacks over the years. The main culprit is the use of malleable, unauthenticated schemes like CBC, and their susceptibility to padding oracle [Vas02] and related attacks. Such attacks are found with as much regularity against systems designed in the 90’s as they are today; recent research [FIM20] shows that CBC continues to be an attack vector.

Beck et al. [BZG20] cite flaws in Apple iMessage, OpenPGP, and PDF encryption as examples to argue that practitioners are often only convinced that unauthenticated SKE

is insecure when they see a proof-of-concept exploit. Similar efforts are deemed necessary to demonstrate the exploitability of cryptographic algorithms such as SHA-1 [SBK+17].

The vast majority of applications should default to using authenticated encryption (AE). Known AE schemes are provably secure, which avoids the pitfalls of unauthenticated SKE with relatively small performance overhead. AE schemes are used in widely adopted protocols like TLS [Res18], standardized by NIST [NIS07a, NIS07b] and ISO [ISO09], and are the default SKE option in modern cryptographic libraries such as NaCl [nacl] and Tlsf [tlsf].

With AE more widely used, recent research focuses on its security guarantees in settings which push the boundaries and assumptions of conventional AE, such as understanding nonces [RS96], multiple decryption errors [BDPS13], inverted plaintext [ABL+14], side channel leakage [BMOS17], multi-user attacks [BT16], boundary hiding [BDPS12], streaming AE [HRRV15], and variable-length tags [RVV16]. Furthermore, constructions and security models have received additional scrutiny due to two recent competitions focusing on AE: CAESAR [CAE14] and the NIST lightweight cryptography competition [NIS18].

Key Commitment. Among the extended, desirable properties explored is the relatively little-studied of AE key commitment, which we intuitively explain as follows.

One of the defining design goals of AE is to provide ciphertext integrity. Recipient A decrypts a ciphertext with the key K_A . If the valid plaintext is P_A , then recipient A knows that the ciphertext has not been modified during transmission. Intuitively, one might mistakenly assume that the integrity guarantee extends to keys, i.e., if some other recipient B decrypts the same ciphertext with their key K_B , then decryption would fail. However, this is neither an AE design goal, nor a guaranteed property, and there are secure and globally deployed AE schemes where both recipients can successfully decrypt the same ciphertext.

Key commitment guarantees that a ciphertext can only be decrypted under the same key used to produce it from some

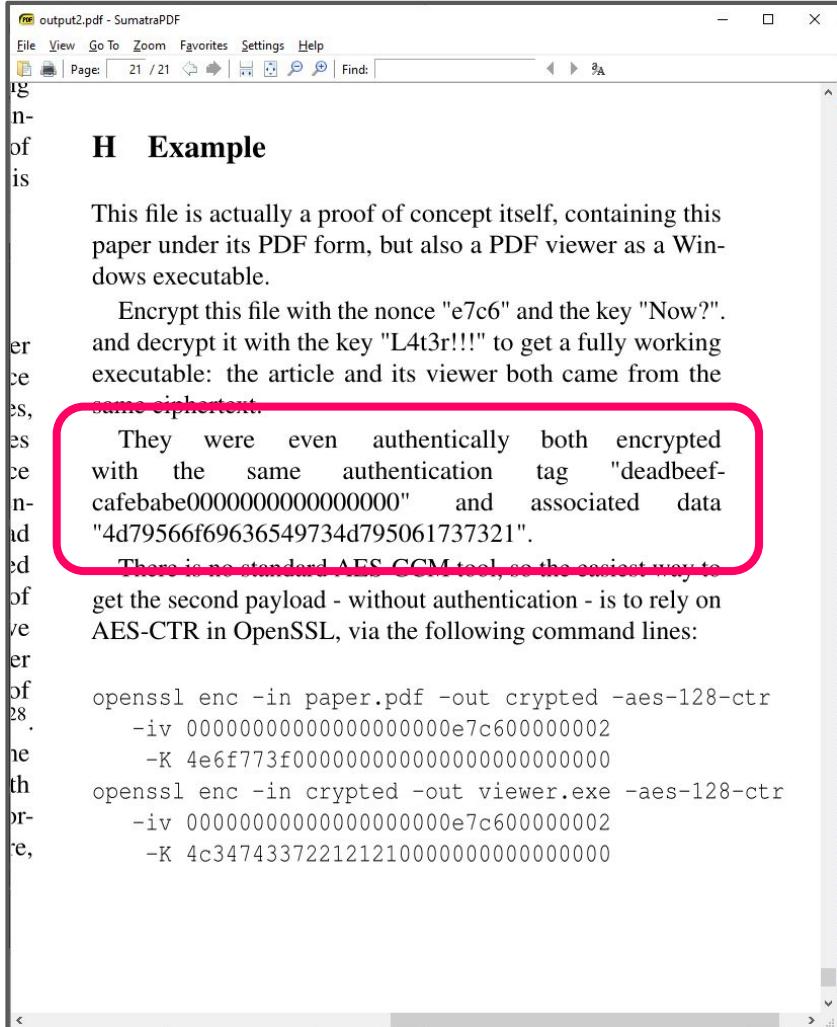
```
mitra\utils\gcm>decrypt.py examples\pdf-viewer.gcm
key1: b'Now?'
key1: b'L4t3r!!!'
ad: b'MyVoiceIsMyPass!'
nonce: 59334
tag: b'deadbeefcafebabe0000000000000000'
Success!

plaintext1: b'4d5a1526c3461a3f30486ccfd93e4a95' ...
plaintext2: b'255044462d312e330a25c2b5c2b60a0a' ...

mitra\utils\gcm>output1.exe output2.pdf

mitra\utils\gcm>_
```

SOME PoCs KNOW THEIR SECRETS IN ADVANCE 😊



The screenshot shows a Windows application window titled "output2.pdf - SumatraPDF". The menu bar includes File, View, Go To, Zoom, Favorites, Settings, and Help. The status bar at the bottom shows "Page: 21 / 21". A red box highlights the "tag: b'deadbeefcafebabe0000000000000000'" line from the terminal output, which corresponds to the "deadbeefcafebabe0000000000000000" field in the PDF viewer's hex dump. Below this, another red box highlights the "plaintext1" and "plaintext2" lines, which correspond to the two payloads in the PDF viewer's hex dump.

H Example

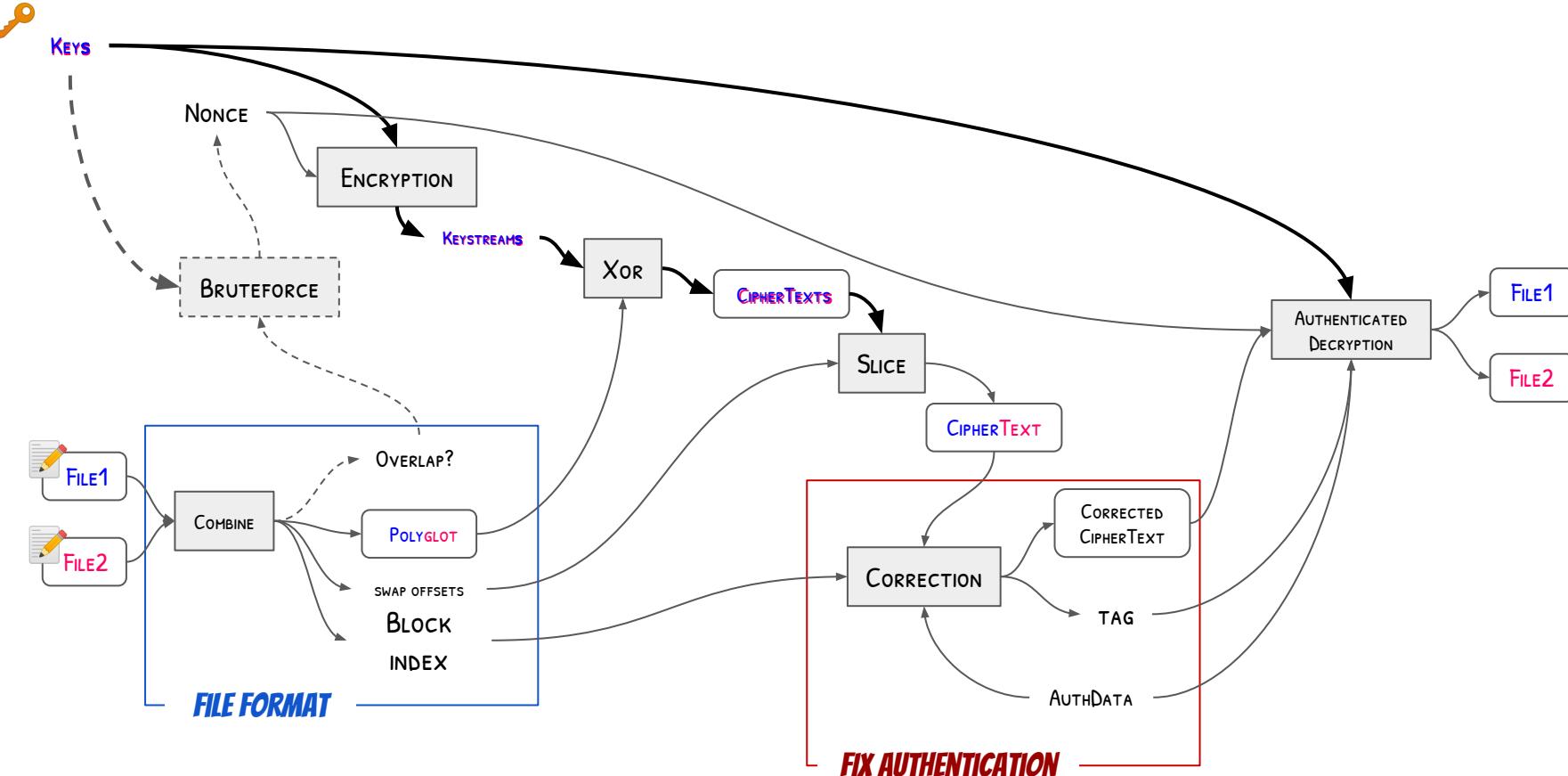
This file is actually a proof of concept itself, containing this paper under its PDF form, but also a PDF viewer as a Windows executable.

Encrypt this file with the nonce "e7c6" and the key "Now?". and decrypt it with the key "L4t3r!!!" to get a fully working executable: the article and its viewer both came from the same ciphertext.

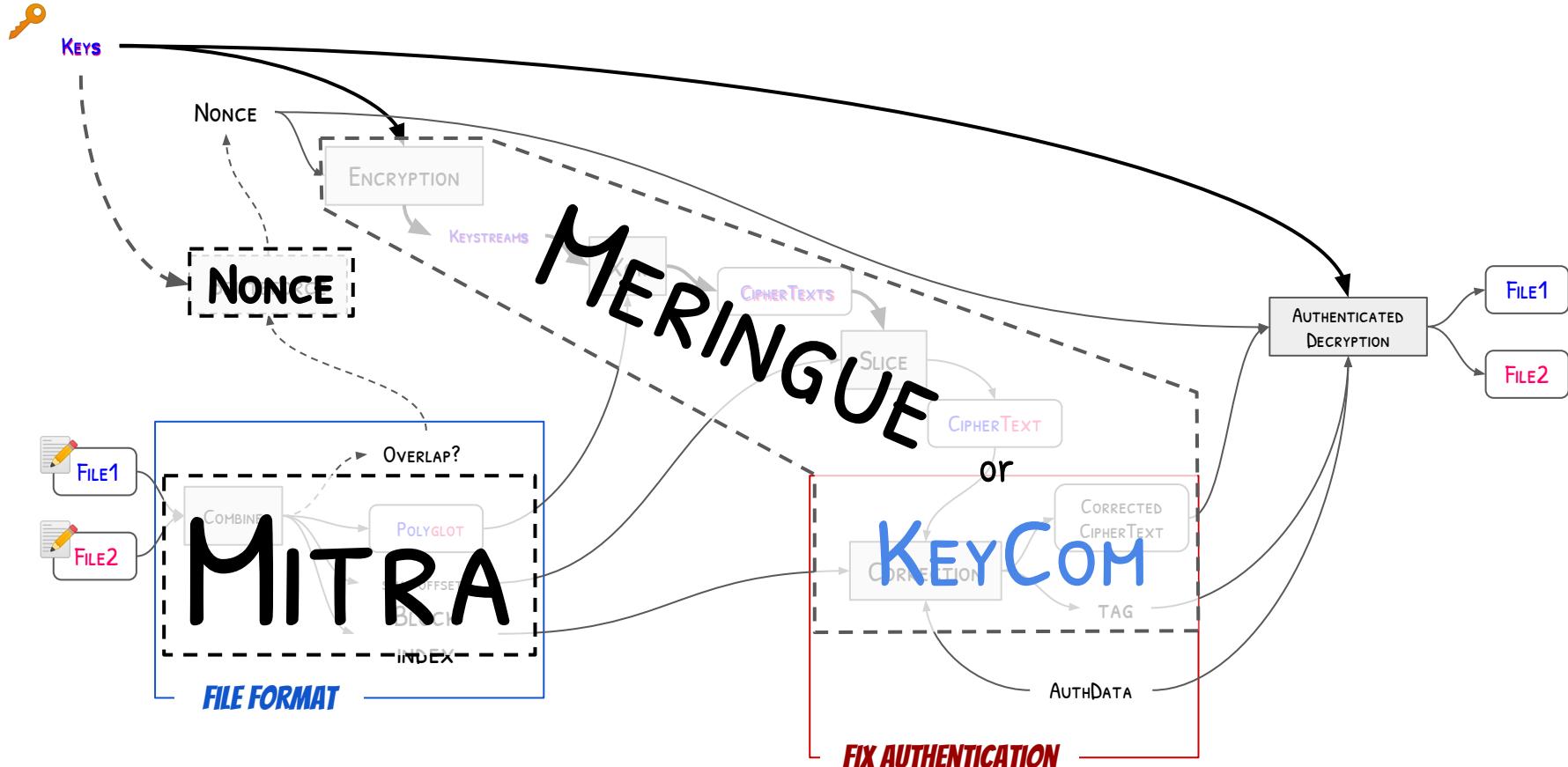
They were even authentically both encrypted with the same authentication tag "deadbeefcafebabe0000000000000000" and associated data "4d79566f69636549734d795061737321".

There is no standard AES CCM tool, so the easiest way to get the second payload - without authentication - is to rely on AES-CTR in OpenSSL, via the following command lines:

```
openssl enc -in paper.pdf -out cryptd -aes-128-ctr
    -iv 0000000000000000e7c600000002
    -K 4e6f773f000000000000000000000000
openssl enc -in cryptd -out viewer.exe -aes-128-ctr
    -iv 0000000000000000e7c600000002
    -K 4c347433722121210000000000000000
```



OVERVIEW



OVERVIEW

RELEASED TOOLS

- [Mitra](#): generates polyglots from any pair of files
`/utils/GCM`: tooling to craft custom GCM exploits
- [KeyCom](#): abuses GCM, OCB or GCM-SIV
supports Mitra polyglots

...with many lightweight and free examples!

CONCLUSION

*Authenticated
Encryption*

*A.K.A.
"Robustness".*

A.E. WITHOUT KEY COMMITMENT IS A SECURITY RISK

Pretty cool stuff

CRAFT. UPLOAD.

DOWNLOAD LATER. PWNED!

This is not the file you're looking for

Working as intended

CTR & GCM DESIGNS HAD
THESE WEAKNESSES FROM THE BEGINNING

Generic binary polyglots make them easy to exploit

SOME EASY SOLUTIONS ALREADY EXIST

- Prepend zeros and check their presence after decryption.
- Store a hash of the key

Be careful with formats with cavities (Iso, Dicom...)!

AE schemes, with one additional block output — and analyze their security. One solution simply prepends a constant block of all zero's to the plaintext and encrypts the padded plaintext as normal; decryption looks for the presence of a leading block of zero's to verify the correct key was used. This padding solution does not necessarily work for any AE scheme and must be analyzed on a case-by-case basis, which we do for AES-GCM and ChaCha20Poly1305.

Another solution applies a generic composition to any given AE: the scheme's key is first used to derive a key commitment string and an encryption key; the encryption key is then used in the underlying AE scheme; the scheme outputs the ciphertext and the commitment string.

An instantiation of our generic composition is already pub-

Our paper <https://eprint.iacr.org/2020/1456>

Key Committing AEADs

Shay Gueron

University of Haifa and AWS

Abstract. This note describes some methods for adding a key commitment property to a generic (nonce-based) AEAD scheme. We analyze the privacy bounds and key commitment guarantee of the resulting constructions, by expressing them in terms of the properties of the underlying AEAD scheme and the added key commitment primitive. We also offer concrete constructions for a key committing version of AES-GCM.

Key Committing AEADs

Useless binary polyglots?

GENERIC AND INSTANT

CRYPTO-POLYGLOTS ARE EVEN COOLER

Not a standard file™

ONE MORE THING...

Not just CTR/GCM!

GCM-SIV AND OCB3 ARE EXPLOITABLE TOO

Some extra constraints, but similar ideas

EXTRA CONSTRAINTS

Block alignment (OCB3, SIV): just need to pad to align payloads

More blocks: OCB3 requires 270 blocks, but that's only 4kb

Computing time:

- GCM-SIV require more time,
- relative to payload size: 5h30min for 300 kb

How?



Python with extra math stuff
to keep things readable

<https://www.sagemath.org/>

Just run our scripts on Mitra polyglots
That's... all!

```
$ sage mitra_gcm.sage examples/input/S\((24\)).gz.rar -p
Key1: b'010101010101010101010101010101010101010101010101010101'
Key2: b'0202020202020202020202020202020202020202020202020202020202020202'
Nonce: b'03030303030303030303030303030303030303030303030303030303030303'
AdditionalData: b'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
Ciphertext: b'26ce232179cfdf1af01e887f3f5a6e840a7'
Tag: b'0404040404040404040404040404040404040404040404040404040404040404'
$ _
```

```
$ unrar vt gcm2.bin
UNRAR 5.61 beta 1 freeware          Copyright (c) 1993-2018 Alexander Roshal
Archive: gcm2.bin
Details: RAR 4, SFX

Name: rar4.txt
Type: File
Size: 4
Packed size: 4
Ratio: 100%
mtime: 2020-01-18 19:08:40,946092200
Attributes: ..A....
CRC32: 982134A1
```

```
$ gzip -lv gcm1.bin
method crc      date   time
defla 28056779 Dec 14 07:55
$ _
```

	compressed	uncompressed	ratio	uncompressed_name
	176	144	-3.5%	gcm1.bin

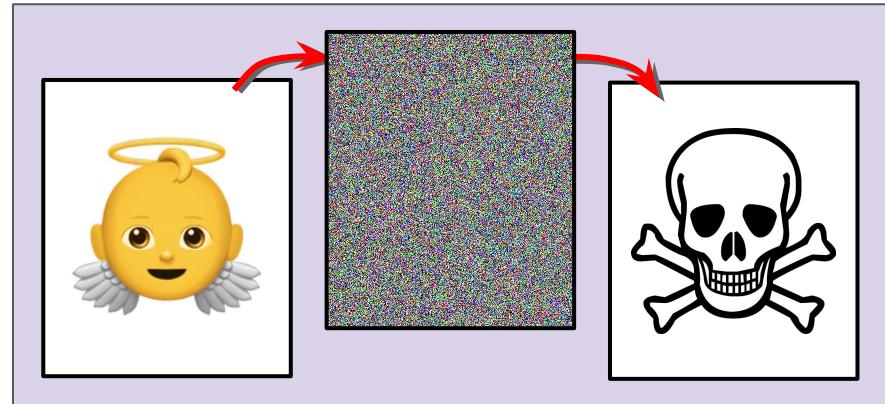
<https://github.com/kste/keycommitment>

Thank you!

Any feedback is welcome!

Special thanks to:

Daniel Bleichenbacher, Atul Luykx,
Sophie Schmieg, Thai Duong, Shay Gueron,
Philippe Teuwen, Thaís Moreira Hamasaki



ANGE ALBERTINI

reverse engineering
VISUAL DOCUMENTATIONS

@angealbertini

ange@corkami.com

<http://www.corkami.com>



BONUS

PoC WALKTHROUGH

000000	% P D F - 1 . 3 \n % C2 B5 C2 B6 \n \n
000010	1 0 o b j \n < < / L e n g t
000020	h 2 1 6 7 7 8 9 > > \n s t r e
000030	a m \n 51 96 B8 92 4D 29 DA 6A 92 04 AA 86 70
000040	3F E2 52 EF 0D 90 23 53 3A 95 AB 06 1F CB FB 83
000050	DE 04 24 64 31 4C 7F 5D 39 91 78 D1 09 9F E8 44
000060	00 1C 14 F8 96 D7 33 F1 54 F3 DD 87 29 F0 70 86
000070	41 46 EE 5C AE FB 71 8E 9D 11 59 FD 30 91 AF 59

...

2113E0	0D 4D DE 5E 59 FF 11 AD 64 7D 9B 78 A4 67 EB 92
2113F0	4F 17 C5 4C 9E EB 9E 50 CC CB 2C 08 52 CC D3 57
211400	48 22 01 AB 63 84 A6 08 86 43 72 7C 84 16 BF 68
211410	14 7E 39 F1 F9 4A 43 C2 8F 46 76 62 38 51 C5 84
211420	\n e n d s t r e a m \n e n d o b
211430	j \n \n 3 0 o b j \n < < / T y
211440	p e / C a t a l o g / P a g e s

...

259C70	t 3 0 R > > \n s t a r t x
259C80	r e f \n 2 4 6 2 3 6 5 \n % % E 0
259C90	F \n 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

M Z	15 26 C3 46 1A 3F 30 48 6C CF D9 3E 4A 95
CF 9C 39 32 CE 91 84 FB 59 61 4E 78 62 8A 31 0B	
26 D1 86 AF 85 D7 B6 E1 AE 00 4F DF 0B 35 8B 7E	
E9 91 CF 00 00 00 00 00 00 00 00 00 00 00 01 00 00	
0E 1F BA 0E-00 B4 09 CD 21 B8 01 4C CD 21 T h	
i s p r o g r a m c a n n o	
t b e r u n i n D O S	
m o d e . \r \r \n \$ 00 00 00 00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
54 89 9D 89 51 67 30 45 E9 7D 4A 7A 52 48 5A 5D	
FE FB 24 44 12 C4 92 56 B9 B9 7F FF 55 2A E9 1E	
47 FA 54 21 94 ED 81 9B 01 4B 6C 7D A5 88 B3 B4	
0A 96 B4 EC EC 10 0A F0 4B D7 D3 25 02 FC 5E 75	
66 7A 27 05 0F 4F 91 27 FE 5B C8 36 99 3D AE 28	

59 24 BB 66 8E 6E E5 83 CD 6C 64 5D 48 FE 27 4B
99 85 AD F7 86 EB 14 98 04 B6 F6 64 68 11 7E 0D
EB 70 ED C6 4A DE BC 41 B6 A6 49 04 F5 53 A1 67

Overlapping bytes
 Correction blocks

THE 2 PLAINTEXTS: 2 BYTES OF OVERLAPPING CONTENTS, 2 CORRECTION BLOCKS

000000	% P	D F - 1 3 \ % C2 B5 C2 B6 \n \n
000010	1 0 o o / L e n g t	
000020	h 2 1 0 s t r e	
000030	a m \n 51 96 B8 92 4D 29 DA 6A 92 04 AA 86 70	
000040	3F E2 52 EF 0D 90 23 53 3A 95 AB 06 1F CB FB 83	
000050	DE 04 24 64 31 4C 7F 5D 39 91 78 D1 09 9F E8 44	
000060	00 1C 14 F8 96 D7 33 F1 54 F3 DD 87 29 F0 70 86	
000070	41 46 EE 5C AE FB 71 8E 9D 11 59 FD 30 91 AF 59	
...		
2113E0	0D 4D DE 5E 59 FF 11 AD 64 7D 9B 78 A4 67 EB 92	
2113F0	4F 17 C5 4C 9E EB 9E 50 CC CB 2C 08 52 CC D3 57	
211400	48 22 01 AB 63 84 A6 08 86 43 72 7C 84 16 BF 68	
211410	14 7E 39 F1 F9 4A 43 C2 8F 46 76 62 38 51 C5 84	
211420	\n e n d s t r e a m \n e n d o b	
211430	j \n \n 3 0 o b j \n < < / T y	
211440	p e / C a t a l o g / P a g e s	
...		
259C70	t 3 0 R > > \n s t a r t x	
259C80	r e f \n 2 4 6 2 3 6 5 \n % % E 0	
259C90	F \n 00 00 00 00 00 00 00 00 00 00 00 00 00	

PDF body, XREF and trailer

M Z	15 26 C3 46 1A 3F 30 48 6C CF D9 3E 4A 95
CF 9C	39 32 CE 91 84 FB 59 61 4E 78 62 8A 31 0B
26 D1	86 AF 85 D7 B6 E1 AE 00 4F DF 0B 35 8B 7E
E9 91 CF	00 00 00 00 00 00 00 00 00 00 00 00 01 00 00
0E 1F BA	0E-00 B4 09 CD 21 B8 01 4C CD 21 T h
i s p r o g r a m c a n n o	t b u p i S D O S
m o d e	l e v e l S O D O S 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
54 89 9D 89 51 67 30 45 E9 7D 4A 7A 52 48 5A 5D	
FE FB	24 44 12 C4 92 56 B9 B9 7F FF 55 2A E9 1E
47 FA	54 21 94 ED 81 9B 01 4B 6C 7D A5 88 B3 B4
0A 96 B4 EC	EC 10 0A F0 4B D7 D3 25 02 FC 5E 75
66 7A 27 05 0F 4F 91 27 FE 5B C8 36 99 3D AE 28	
59 24 BB 66 8E 6E E5 83 CD 6C 64 5D 48 FE 27 4B	
99 85 AD F7 86 EB 14 98 04 B6 F6 64 68 11 7E 0D	
EB 70 ED C6 4A DE BC 41 B6 A6 49 04 F5 53 A1 67	

THE PDF CONTAINS THE MOST OF THE PE PAYLOAD IN A DUMMY OBJECT

```
% P D F - 1 . 3 \n % C2 B5 C2 B6 \n \n
1 0 o b j \n < < / L e n g t
h 2 1 6 7 7 8 9 > > \n s t r e
a m \n 51 96 B8 92 4D 29 DA 6A 92 04 AA 86 70
3F E2 52 EF 0D 90 23 53 3A 95 AB 06 1F CB FB 83
DE 04 24 64 31 4C 7F 5D 39 91 78 D1 09 9F E8 44
00 1C 14 F8 96 D7 33 F1 54 F3 DD 87 29 F0 70 86
41 46 EE 5C AE FB 71 8E 9D 11 59 FD 30 91 AF 59
```

```
0D 4D DE 5E 59 FF 11 AD 64 7D 9B 78 A4 67 EB 92
4F 17 C5 4C 9E EB 9E 50 CC CB 2C 08 52 CC D3 57
48 22 01 AB 63 84 A6 08 86 43 72 7C 84 16 BF 68
14 7E 39 F1 F9 4A 43 C2 8F 46 76 62 38 51 C5 84
\n e n d s t r e a m \n e n d o b
j \n \n 3 0 o b j \n < < / T y
p e / C a t a l o g / P a g e s
t 3 0 R > > \n s t a r t x
r e f \n 2 4 6 2 3 6 5 \n % % E 0
F \n 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

THE PDF IS STANDARD

The first object is unreferenced

A tiny appended data

(that could be avoided with alignment)

The rest is fully standard

THE PE IS ALMOST STANDARD

The DOS header is mostly overwritten

it's required but irrelevant nowadays anyway

However, the pointer to PE header is preserved

The rest of the PE is unmodified

The encrypted PDF is just appended data

M	Z	15	26	C3	46	1A	3F	30	48	6C	CF	D9	3E	4A	95
CF	9C	39	32	CE	91	84	FB	59	61	4E	78	62	8A	31	0B
26	D1	86	AF	85	D7	B6	E1	AE	00	4F	DF	0B	35	8B	7E
E9	91	CF	00	00	00	00	00	00	00	00	00	00	00	01	00
0E	1F	BA	0E-00	B4	09	CD	21	B8	01	4C	CD	21	T	h	
i	s	p	r	o	g	r	a	m	c	a	n	n	o		
t	b	e	r	u	n		i	n	D	O	S				
m	o	d	e	.	\r	\r	\n	\$	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
54	89	9D	89	51	67	30	45	E9	7D	4A	7A	52	48	5A	5D
FE	FB	24	44	12	C4	92	56	B9	B9	7F	FF	55	2A	E9	1E
47	FA	54	21	94	ED	81	9B	01	4B	6C	7D	A5	88	B3	B4
0A	96	B4	EC	EC	10	0A	F0	4B	D7	D3	25	02	FC	5E	75
66	7A	27	05	0F	4F	91	27	FE	5B	C8	36	99	3D	AE	28
59	24	BB	66	8E	6E	E5	83	CD	6C	64	5D	48	FE	27	4B
99	85	AD	F7	86	EB	14	98	04	B6	F6	64	68	11	7E	0D
EB	70	ED	C6	4A	DE	BC	41	B6	A6	49	04	F5	53	A1	67

PoC DIY

```
mitra/utils/extra$ pdfpe.py paper.pdf SumatraPDF18fixed.exe
  * normalizing, merging with a dummy page
  * removing dummy page reference
  * fixing object references
  * aligning PE header
  * finalizing main PDF
  * generating polyglot
Success!
```



A specially prepared executable

```
mitra/utils/extra$ ./gcm/meringue.py -i 135488 -n 59334 'Z(2-33-211420).exe.pdf' test.gcm
```

```
key 1: Now?
key 2: L4t3r!!!
ad   : MyVoiceIsMyPass!
blocks: 154060
```

Computing 154062 coefficients.

Coef to be inverted: 78a9f0686e9b7972252c8ca3796e3100 (already computed)

```
mitra/utils/extra$ _
```

RUN THE DEDICATED SCRIPT Mitra itself can't do it because of the bytes overlap via nonce

```
$ python3 ./mitra.py in/jpg.jpg ./in/pe32.exe --verbose
> Arguments parsing:
> Verbose is ON
in/jpg.jpg
File 1: JFIF / JPEG File Interchange Format
./in/pe32.exe
File 2: Portable Executable (hdr)

> Stack: JPG-PE(hdr)
> ! File type 2 (PE(hdr)) starts at offset 0 - it can't be appended.
> Parasite: JPG[PE(hdr)]
> ! File type 1 (JPG) can only host parasites at offset 0x6. File 2 should start at offset 0x0 or less.
> Zipper: JPG^PE(hdr)
> ! File type 1 (JPG) doesn't support zippers.
> Cavity: JPG_PE(hdr)
> ! File type 2 (PE(hdr)) doesn't start with any cavity.
$ _
```

BOTH FORMATS START
AT OFFSET ZERO

JPG AND PE: NO STANDARD POLYGLOT IS POSSIBLE

```
$ python3 ./mitra.py in/jpg.jpg ./in/pe32.exe --verbose --overlap
> Arguments parsing:
> Verbose is ON
> Overlap is ON
in/jpg.jpg
File 1: JFIF / JPEG File Interchange Format
./in/pe32.exe
File 2: Portable Executable (hdr)

> Stack: JPG-PE(hdr)
> ! File type 2 (PE(hdr)) starts at offset 0 - it can't be appended.
> Parasite: JPG[PE(hdr)]
> ! File type 1 (JPG) can only host parasites at offset 0x6. File 2 should start at offset 0x0 or less.
> Zipper: JPG^PE(hdr)
> ! File type 1 (JPG) doesn't support zippers.
> Cavity: JPG_PE(hdr)
> ! File type 2 (PE(hdr)) doesn't start with any cavity.
> PE Reverse overlapping parasite
> HIT JPG;PE(hdr)
> Overlapping parasite
> HIT JPG;PE(hdr)
$ _                                         -> generates OR(6-a00)-JPG[PE(hdr)]{4D5A}.e1d0b0d9.jpg.exe
```

JPG AND PE: NEAR-POLYGLOTS ARE POSSIBLE...

*Not a working polyglot
without crypto!*

```
$ sage mitra_gcm.sage OR(6-a00)-JPG[PE(hdr)]{4D5A}.e1d0b0d9.jpg.exe -p  
Overlap file found - bruteforced nonce: 11671  
Key1: b'0101010101010101010101010101010101010101010101010101010101010101'  
Key2: b'0202020202020202020202020202020202020202020202020202020202020202'  
Nonce: b'00000000000000000000000000000000d97'  
AdditionalData: b'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'  
Ciphertext: b'6a8b8127629cbc5c20d6518f6c75b31ec39c5d77b3db4314b6a5f7ff134ec266'  
Tag: b'0404040404040404040404040404040404040404040404040404040404040404'  
$ _
```

ONLY 2 BYTES OF OVERLAP!
(QUICK TO BRUTEFORCE)

Generates also gcm1.bin and gcm2.bin from the same ciphertext

```
$ file gcm*.bin  
gcm1.bin: JPEG image data, baseline, precision 8, 6x2, components 3  
gcm2.bin: PE32 executable (console) Intel 80386, for MS Windows  
$ identify gcm1.bin  
gcm1.bin JPEG 6x2 6x2+0+0 8-bit sRGB 2928B 0.000u 0:00.000  
$ wine ./gcm2.bin  
* a 'compiled' PE  
$ _
```

BOTH FORMATS START
AT OFFSET ZERO

...AND QUICK TO BRUTEFORCE

A JPEG trick

SAVING A FEW BYTES OF BRUTEFORCING

Do you feel lucky?

STRUCTURE OF A JPEG FILE

- A sequence of segments
- Each segment starts with FF then a segment marker byte
- Most segments then start with their big-endian length on 2 bytes

...the rest is complex

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	FF	D8	FF	E0	00	10	J	F	I	F	00	01	01	02	00	24
10	00	24	00	00	FF	DB	00	43	00	01	01	01	01	01	01	01
20	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01
30	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01
40	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01
50	01	01	01	01	01	01	01	01	FF	C0	00	0B	08	00	38	
60	00	68	01	01	11	00	FF	C4	00	29	00	01	01	01	01	00
70	00	00	00	00	00	00	00	00	00	00	00	00	0B	04	0A	10
80	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	FF	DA	00	08	01	01	00	00	3F	00	EF	E0	00	00	06
A0	76	80	40	21	7F	74	02	05	FB	C1	01	01	7F	70	10	08
B0	5F	DD	00	85	FD	D0	08	5F	DD	00	85	FD	C0	04	02	17
C0	F7	40	20	5F	DC	40	20	17	F7	10	0F	5F	C1	00	85	FD
D0	D0	08	5F	DC	10	08	5F	DD	00	85	FD	C6	74	04	17	F7
E0	10	08	5F	DC	04	02	05	FD	C0	00	00	07	FF	D9		

- 00: FF D8 Start Of Image (size: n/a)
- 02: FF E0 Application 0 (size: 10)
- 14: FF DB Define a Quantization Table (size: 43)
- 59: FF C0 Start Of Frame 0 (size: 0B)
- 66: FF C4 Define Huffman table (size: 29)
- 91: FF DA Start of Scan (size: n/a)
- EC: FF D9 End Of Image (size: n/a)



STRUCTURE OF A SMALL JPEG IMAGE

00	FF	D8	FF	E0	00	10	J	F	I	F	00	01	01	02	00	E	F	-
10	00	24	00	00	FF	DB	00	43	00	01	01	01	01	01	01	01	24	
...																		

- 00: FF D8 Start Of Image (size: n/a)
- 02: FF E0 Application 0 (size: 10)
- 14: FF DB Define a Quantization Table (size: 43)
- ... : FF

00	FF	D8	FF	FE	00	0E	*	*	p	a	r	a	s	i	t	e	-
10	*	*	FF	E0	00	10	J	F	I	F	00	01	01	02	00	24	
20	00	24	00	00	FF	DB	00	43	00	01	01	01	01	01	01	01	24
...																	

- 00: FF D8 Start Of Image (size: n/a)
- 02: FF FE COMment (size: 0E)
- 12: FF E0 Application 0 (size: 10)
- 24: FF DB Define a Quantization Table (size: 43)
- ... : FF

PARASITIZING: INSERT A COMMENT SEGMENT FF FE

How MANY REQUIRED BYTES TO CONTROL ?

- 6, to set an exact comment length

FF FE XX YY

- 5, to set a length rounded-up to 0x100

FF FE XX⁺¹ ??

The length is big endian: XX YY => length of 0XXYY

- 4, if our parasite fits in the random length 🤪

FF FE ?? ??

64kb more data is nothing weird for a JPG image

-> 2^{16} speed-up

Do I feel lucky?



How to...

GENERATE QUICKLY
A JPG OVERLAP POLYGLOT

```
$ mitra.py in.jpg.jpg in.icc.icc --verbose --overlap
> Arguments parsing:
> Verbose is ON
> Overlap is ON
in\jpg.jpg
File 1: JFIF / JPEG File Interchange Format
in\icc.icc
File 2: ICC / International Color Consortium profiles

> Stack: JPG-ICC
> ! File type 2 (ICC) starts at offset 0 - it can't be appended.
> Parasite: JPG[ICC]
> ! File type 1 (JPG) can only host parasites at offset 0x6. File 2 should start at offset 0x0 or less.
> Zipper: JPG^ICC
> ! File type 1 (JPG) doesn't support zippers.
> Cavity: JPG_ICC
> ! File type 2 (ICC) doesn't start with any cavity.
> Overlapping parasite
> Jpeg overlap file: reducing two bytes
> (don't forget to postprocess after bruteforcing)
> HIT ICC;JPG
Generic overlapping polyglot file created.

$ _
```

1/4 GENERATE AN OVERLAPPING POLYGLOT

USE THE
FIXED FILE!

```
$ utils/gcm/meringue.py -k 0101010101010101010101010101010101010101010101  
-i 9 -n 0x115e0000014e11ec  
"4-0(4-84)-JPG[ICC]{000001C0}.5ecbd8cf.jpg.icc" jpg-icc.gcm  
key 1: \x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01  
key 2: \x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02  
ad : MyVoiceIsMyPass!  
blocks: 1633  
  
Computing 1635 coefficients.  
Coef to be inverted: 86aabaa18fba4e751da9a6de05c6159f (not present)  
Inverting the coefficient (takes a few mins)...  
New invert: 86aabaa18fba4e751da9a6de05c6159f 013e74df0bdab42a43e625642b293fdf  
  
$ _
```

3/4 GENERATE AMBIGUOUS CIPHERTEXT FROM THE FIXED FILE

```
$ utils/gcm/decrypt.py jpg-icc.gcm  
key1: b'\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01'  
key1: b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'  
ad: b'MyVoiceIsMyPass!'  
nonce: 1251437746477470188  
tag: b'b3a546dc3faa0f3707e1ef66f6e9411d'  
Success!  
  
Key2: b'02020202020202020202020202020202'  
Nonce: b'00000000000000000002d97'  
AdditionalData: b'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'  
Ciphertext: b'6a8b8127629cbc5c20d6518f6c75b31ec39c5d77b3db4314b6a5f7ff134ec266'  
Tag: b'0404040404040404040404040404040404
```

```
$ _ $ file jpg-icc*  
jpg-icc.b3474cf7.icc: Microsoft color profile 3.2, type lcms, GRAY/Lab-prtr device by  
lcms, 448 bytes, 13-1-2009 16:10:20, no copyright tag  
jpg-icc.b3474cf7.jpg: JPEG image data, baseline, precision 8, 104x56, frames 1
```

4/4 DECRYPT & TEST

DETAILS OF A MITRA POLYGLOT FILE NAME

0(4-84)-JPG[ICC]{000001C0}.5ecbd8cf.jpg.icc

- **layout type**: Stack / Overlapping / Parasite / Cavity / Zipper
- **(Slices)**: offsets where the content change side
- **type layout**: Tells which format is the host, which is the parasite
- **{Overlapping data}**: the “other” bytes of the file start
- **partial hash** - to differentiate outputs
- **file extensions** - to ease testing

Used for mixing contents after encryption.
(Imagine two sausages sliced in blocks and mixed)

Used to bruteforce nonces.