7 More Cryptographic Coloring Books

by Philippe Teuwen

7.1 Weird crypto

In PoC||GTFO 5:3 we taught you kids why ECB is a weak encryption mode, as helpfully shown by the `ElectronicColoringBook.py` script.[12] As you may have guessed, we'll see that in some circumstances CBC deserves the same treatment!

Don't worry, though! Most of the time CBC mode is fine, but sometimes weirdos like our buddy Ange Albertini do impossibly fancy things with crypto such as *AngeCryption*. I wouldn't risk offending our PoC||GTFO's loyal readers by explaining AngeCryption all over again,[13] but please recall that it relies on the fact that you can *decrypt* plaintext to obtain ciphertext. This reverse-ciphertext *encrypts* back to the original plaintext because block encryption and decryption operations can be safely exchanged.

Let's try to reproduce the example given by Ange in his RMLL2014 presentation, available in a translated slide deck titled "Let's play with crypto."

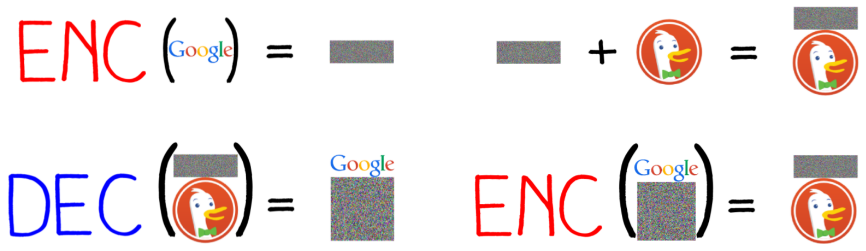


Figure 6: *"If we encrypt the final result, we get our first random data, followed by our target picture."*

This example uses PNG images, so we'll begin with two logos in PNG format and of equal width. We'll take those of Google and DuckDuckGo, with a small change: I removed subtle gradients from the original PNGs so that we get large areas of the same flat color. To better illustrate the vulnerability, we need to work on uncompressed, non-interlaced images. A tool called **advpng**[14] takes care of flattening the PNG images and minimizing the metadata by grouping all IDAT chunks into a single chunk.

```
$ advpng -z -0 google.png
$ advpng -z -0 duckduckgo.png
```

Now we can construct our AngeCryption example using Ange's *PNG-in-PNG* tool (Google for it with "corkami" and "src/angecryption/PiP/PIP.py" as search terms).

```
$ python PIP.py google.png duckduckgo.png combined.png CBC_can_fail_too
```

The resulting `combined.png` displays the Google logo and, when decrypted, displays the DuckDuckGo logo.

[12] `https://doegox.github.io/ElectronicColoringBook/`

[13] See PoC||GTFO 3:11 and its retrospectively funny quote: *"We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure when used with a random IV."*

[14] `http://advancemame.sourceforge.net/`

Figure 7: `combined.png`

Ange's `PIP.py` does the opposite of what the slide proposes, just to show that it's also possible. So, to match the tool and the rest of the article you need to swap the ENC and DEC operations. It still remains pure AngeCryption.

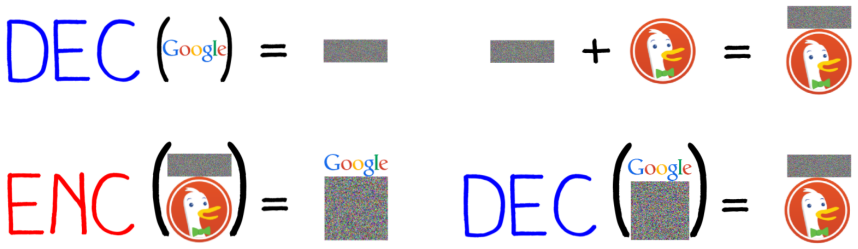


Figure 8: *"If we decrypt the final result, we get our first random data, followed by our target picture"*

7.2 Time to fire up ElectronicColoringBook.py

```
$ python ElectronicColoringBook.py combined.png -p4 -c255
```

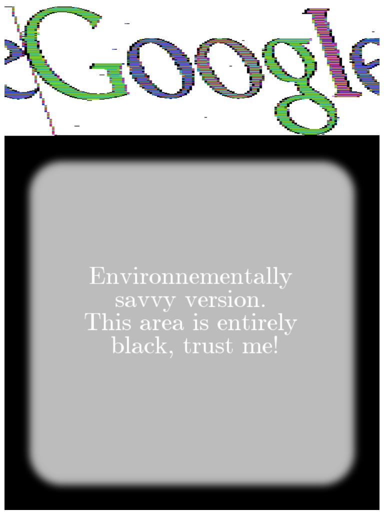


Figure 9: `combined.png` as seen through `ElectronicColoringBook.py`.

What can we see at this point?

We recovered the Google logo but it was not encrypted, so we aren't done yet. Still, we can see a few artifacts compared to what we obtained with ECB on a pure bitmap. It also looks like we couldn't recover the correct aspect ratio either. In fact, it did get correctly recovered, but the display included extra PNG metadata bytes, so the image got slightly skewed.

The artifacts in that image are due to the additional structure of the PNG format that is absent from a plain BMP. In a PNG image, each scan line is preceded by a byte of metadata describing which filter to apply to that line. In our case, those extra bytes are all null bytes indicating the absence of a filter. It is this one extra byte on each line that misaligns the blocks in our image recreation and skews it. It also breaks the uniform areas, so they are not that easy to paint over. Moreover, you can see a few blotches of gray here and there in the white area. That's because the image data, even when uncompressed, is still not raw pixels but a zlib stream encapsulating some DEFLATE data that has its own metadata[15] at the start of each 64 kB block.

Rather than adding additional complexity to our script to handle each of these specific quirks, it turns out that we can correct the misalignment due to the presence of metadata bytes by specifying a non-integer width:

```
$ python ElectronicColoringBook.py combined.png -p4 -o3 -c255 -x 600.345
```

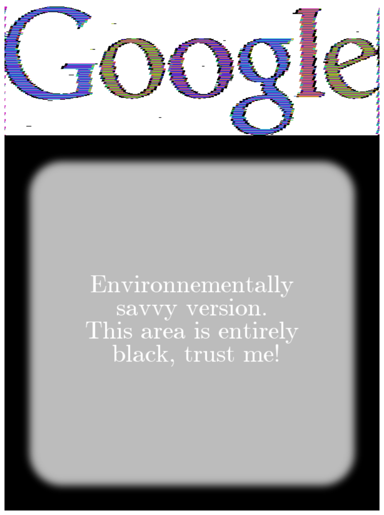
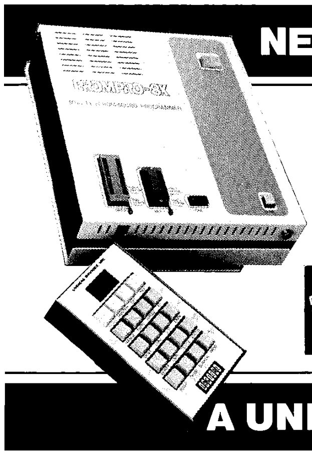

Google
Environnementally savvy version.
This area is entirely black, trust me!

Figure 10: `combined.png`, fine-tuned

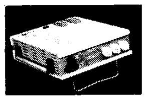
[15]See `rfc1951.txt`.



NEW FROM LOGICAL DEVICES INC:

PROMPRO-8X™ Model II

A stand-alone programmer starting at $895.00 can put you in business to program EE/EPROMs PAL/PLDs,* Single Chip micros,* and Bipolar PROMs,* + EPROM IN-CIRCUIT EMULATION* capability that can speed up your development time considerably and an RS-232 communications port that lets you integrate it with your IBM PC as a total firmware and Logic development station.



All from a company with an excellent reputation for quality and service.

A UNIVERSAL DEVICE PROGRAMMER

The bottom of the image is completely black, which is how `ElectronicColoringBook.py` represents non-repeating blocks. That's what we expect from CBC-encrypted data, as opposed to ECB.

7.3 The downside

Now we can get to the second half of the story, the decrypted `combined.png` displaying the DuckDuckGo logo. We'll use `decrypt-PIP.py`, a helper script created by `PIP.py`, and then apply `ElectronicColoringBook.py` to the output `dec-duckduckgo.png`.

```
$ python decrypt-PIP.py
```

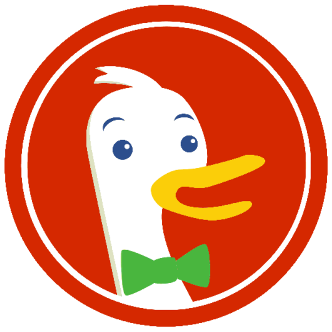


Figure 11: dec-duckduckgo.png

```
$ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3 -c255 -x 600.345
```

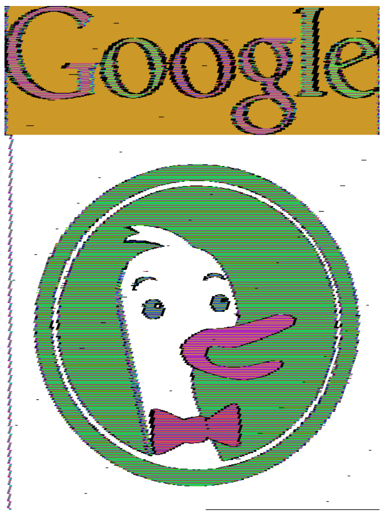


Figure 12: dec-duckduckgo.png as seen through ElectronicColoringBook.py

But what is this new devilry? Oh, no! The Google logo is still visible. Is the CBC gone all evil on us, so can't shake it off?

7.4 Why, oh why?

Recall that in the CBC mode, encryption of each block depends on all the previous blocks:

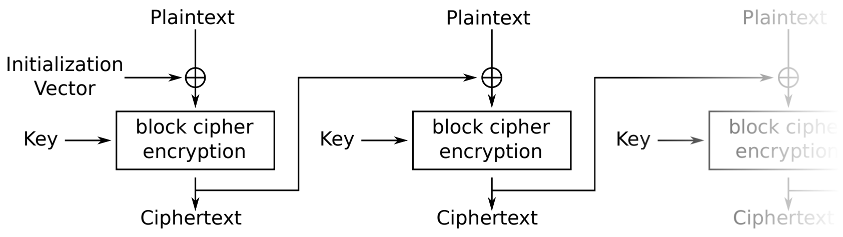


Figure 13: Cipher Block Chaining (CBC) mode encryption

But the Google part of the image is not the result of an encryption but of a decryption, remember? We must account for how these blocks feed into the CBC process.

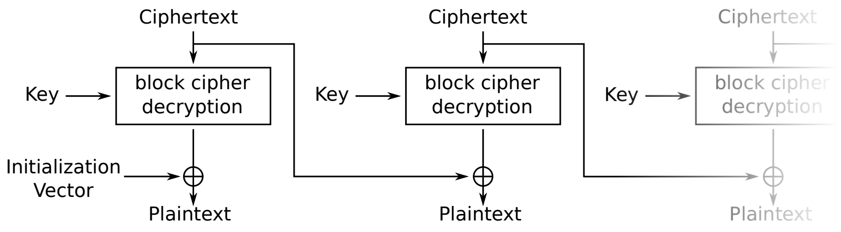


Figure 14: Cipher Block Chaining (CBC) mode decryption

Here, the ciphertext is that of the original Google image. For its image parts of constant color, we get the same ciphertext blocks over and over.

Plaintext blocks of that series will be $P_n = Dec_K(C_n) \oplus C_{n-1} \equiv Dec_K(C) \oplus C$ if all ciphertext blocks are the same.

The first plaintext block from a repetitive area depends on the previous (different) block. Thus its content is different from the following repetitive plaintext blocks.

So CBC in decryption mode is almost as bad as ECB: decrypting n repetitive blocks will give one arbitrary block followed by $n-1$ repetitive blocks (while ECB would give n repetitive blocks). That's why transitions around Google letters look slightly thicker.

In principle, we could paint over CBC when used in reverse mode as easily as we painted over ECB, but it's actually quite difficult in our example because, as you recall, the image data of PNG format is not merely raw pixels such as in the BMP or PNM formats.

In real life, decryption is usually used on data that previously went through encryption. Since the point of the CBC mode is to prevent repetitions in the ciphertext, we don't generally need to fear them, although, theoretically, they could still happen. (By a stroke of bad luck, we might get $Enc_K(C \oplus P) = C$ to occur for a given P for some combination of C and the key K.)

Let us recall another CBC fact: even if you only know the key but not the initialization vector (IV), you can still decrypt `combined.png` almost fully. Only the first block will be wrongly decrypted, which is not that hard to reconstruct; even if left corrupted, it won't prevent `ElectronicColoringBook.py` from revealing both images. Look back at Figure 14 to understand why.

So the upshot of our case study is that single-block encryption and decryption operations can still be exchanged almost safely, although the chaining mode does throw some gotchas into the process.

7.5 Exploring other chaining modes

So what about the other chaining modes that use an IV?

The CFB mode suffers of a similar problem because, in decryption mode, the block encryption depends only on the previous ciphertext. This previous ciphertext can be repeated under AngeCryption, so the resulting plaintext also repeats: $P_n = Enc_K(C_{n-1}) \oplus C_n \equiv Enc_K(C) \oplus C$.

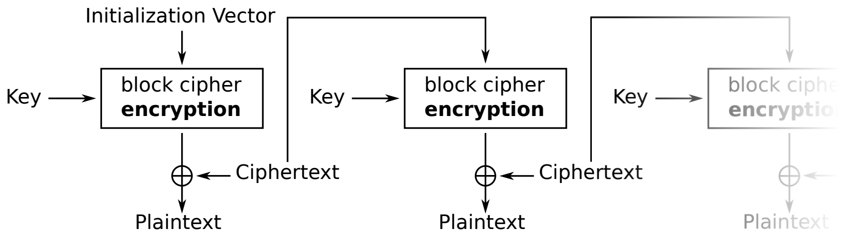


Figure 15: Cipher Feedback (CFB) mode decryption

The OFB mode makes a block cipher into a synchronous stream cipher and therefore doesn't have this issue. Encryption and decryption are just XOR with the same keystream, which only depends on the IV and the key K: $keystream_1 = Enc_K(IV)$, $keystream_n = Enc_K(keystream_{n-1})$ and $P_n = keystream_n \oplus C_n$.

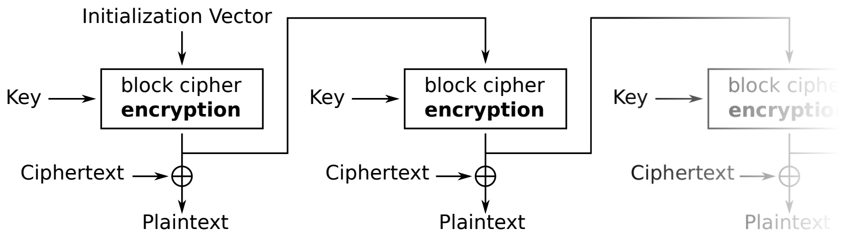


Figure 16: Output Feedback (OFB) mode decryption

Let's try this out. We modify `PIP.py` to replace `MODE_CBC` by `MODE_OFB` and inverse the order of operations to compute the IV. Indeed, if for CBC we computed $IV = Dec_K(C_1) \oplus P_1$, for OFB we must compute $IV = Dec_K(C_1 \oplus P_1)$. Then we repeat the same experiment:

```
$ python PIP_OFB.py google.png duckduckgo.png combined.png OFB_AngeCryption
$ python decrypt-PIP.py
$ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3 -c255 -x 600.345
```

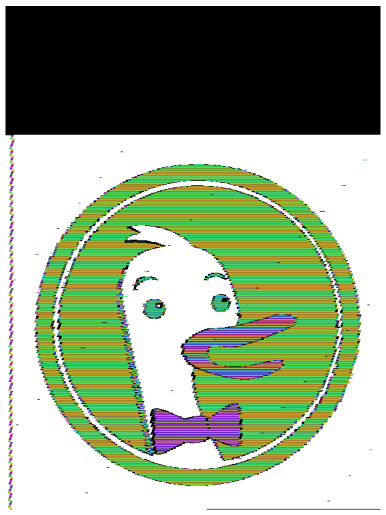


Figure 17: `dec-duckduckgo.png` (OFB version) as seen through ElectronicColoringBook.py

Finally! We get a "secure" version of AngeCryption. As a bonus, unlike CBC, if you only knew the key but not the IV, you wouldn't be able to recover anything.

Another alternative is the CTR mode, which is pretty similar to OFB: $P_n = Enc_K(counter{+}{+}) \oplus C_n$. The OFB initialization vector would play the role of the initial counter value: $counter = Dec_K(C_1 \oplus P_1)$. And, as for OFB, knowing only the key but not the initial counter value is useless.

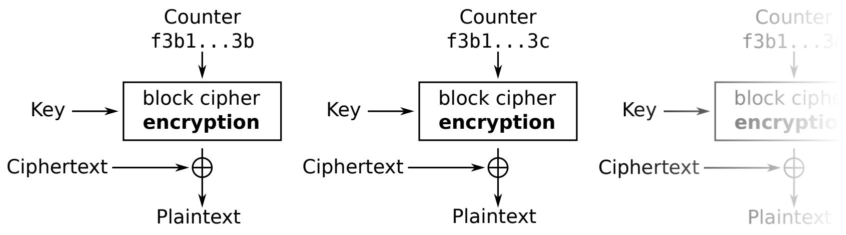


Figure 18: Counter (CTR) mode decryption

Note that both OFB and CTR have their own special limitations typical of stream ciphers: bitflipping attacks, keystream reuse, and so on. However, none of these are an issue in this unusual use case of ours.

The PCBC (Propagating CBC) mode would work as well, because each block decryption depends on the previous ciphertext *and* the previous plaintext: $P_n = Dec_K(C_n) \oplus C_{n-1} \oplus P_{n-1}$. It's not supported in PyCrypto, however, and is not very common.

7.6 Some more PoC

Before we wrap up, I'd like to circle back to a variation of AngeCryption suggested by Gynvael Coldwind, and so rightfully called GynCryption. GynCryption doesn't rely on IV forgery, but rather tries to find a key that transforms the plaintext into the ciphertext we want. For a PNG, it requires control over the first 16 bytes, but this cannot reasonably be done for an entire block. On the other hand, controlling the first 6 bytes of a JPG is enough to be able to insert a small comment section. GynCryption was originally based on ECB, but nothing prevents us from replacing ECB by CBC, CFB, OFB, or by CTR with a null IV or a reset counter respectively—as we've shown above, those are only slightly better than ECB. In this issue's polyglot archive you can find two proofs of concept, `gyncryption_ofb.pdf` and `gyncryption_cfb.pdf` that you can decrypt into a JPG with a null IV/counter and the same key "@doegox_5f32c6e5".

With OFB and CTR, once you have found such a key, you may be tempted to reuse it with any other (small) PDF and JPG, and it will work because they are similar to stream ciphers: a change in a plaintext block affects only the corresponding bits of the ciphertext, not the entire block. But remember that stream ciphers are only secure if you don't reuse the keystream—so don't reuse your key for the same mode, find another one! Otherwise a simple XOR of both files will result into the XOR of the plaintext data (and padding), and the keystream will be entirely removed.

7.7 Conclusions

Of course, since AngeCryption and GynCryption are far more likely to be used as crypto curios rather than as serious tools for serious situations, their security is not that crucial. Still, it is good to understand the risks associated with non-standard uses of block cipher modes—this understanding should serve as an antidote to their blind reuse in inappropriate contexts.

7.8 Acknowledgments

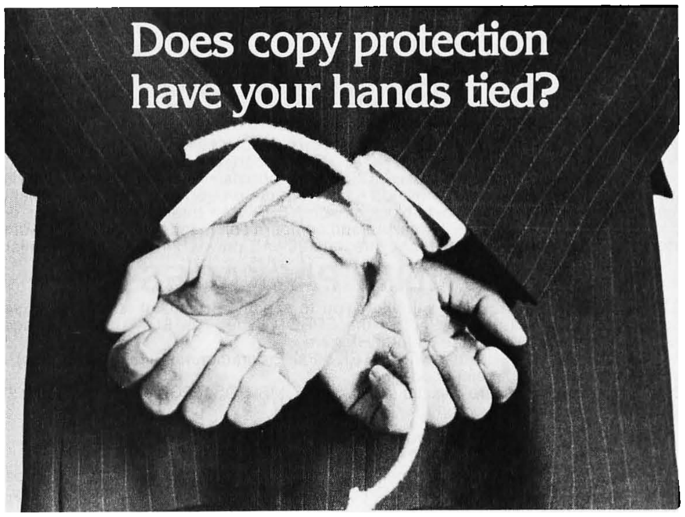
Special thanks go to Ange for his most neighborly help; without him this article would have never been possible!

BACKUP YOUR SOFTWARE WITH LOCKSMITH 6.0™.

Locksmith, the controversial copy program that took the Apple world by storm in 1981, has evolved from a powerful bit-copy programmed into a complete disk utility system, allowing the Apple user to recover crashed disks, restore accidentally deleted files, and perform hardware diagnostics on the disk drive and memory boards. The NEW Locksmith version 6.0 is now available and includes an advanced disk recovery utility, a framing-bit analyzer, an automatic boot tracer, a sector editor, many file utilities, and of course, the most powerful bit-copy program available. A fast disk backup utility copies disks in eight seconds flat. Improvements to Locksmith Programming Language have made it more powerful and easier to use for you to write your own backup and repair procedures. Includes a library disk which contains automatic procedures to copy hundreds of Apple programs.

Locksmith requires no additional hardware, but will use any additional RAM memory that it finds, including RAM boards from Applied Engineering and Checkmate Technology.

Don't get caught with your hands tied. Order Locksmith 6.0 today.



NEW LOW PRICE **$79.95**
Registered Locksmith 5.0 owners may upgrade to version 6.0 for **$29.95.**
Available from your computer dealer or directly from:

Alpha Logic Business Systems, Inc.
4119 North Union Road
Woodstock, IL 60098
(815) 568-5166

©Alpha Logic Business Systems, Inc. 1985
Locksmith and Locksmith/PC are registered trademarks of Alpha Logic Business Systems, Inc.