

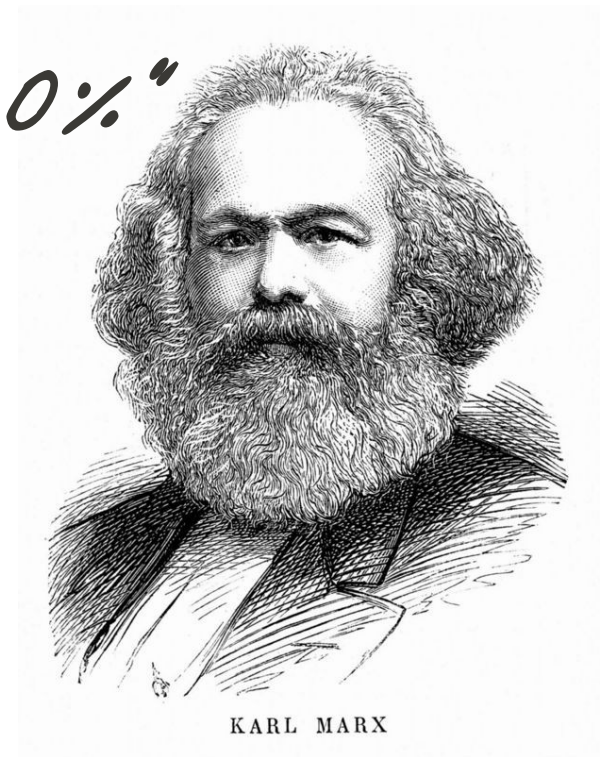
Padding Oracles for the masses

aka: "the other 60%"

Matias Soler

matias@immunityinc.com

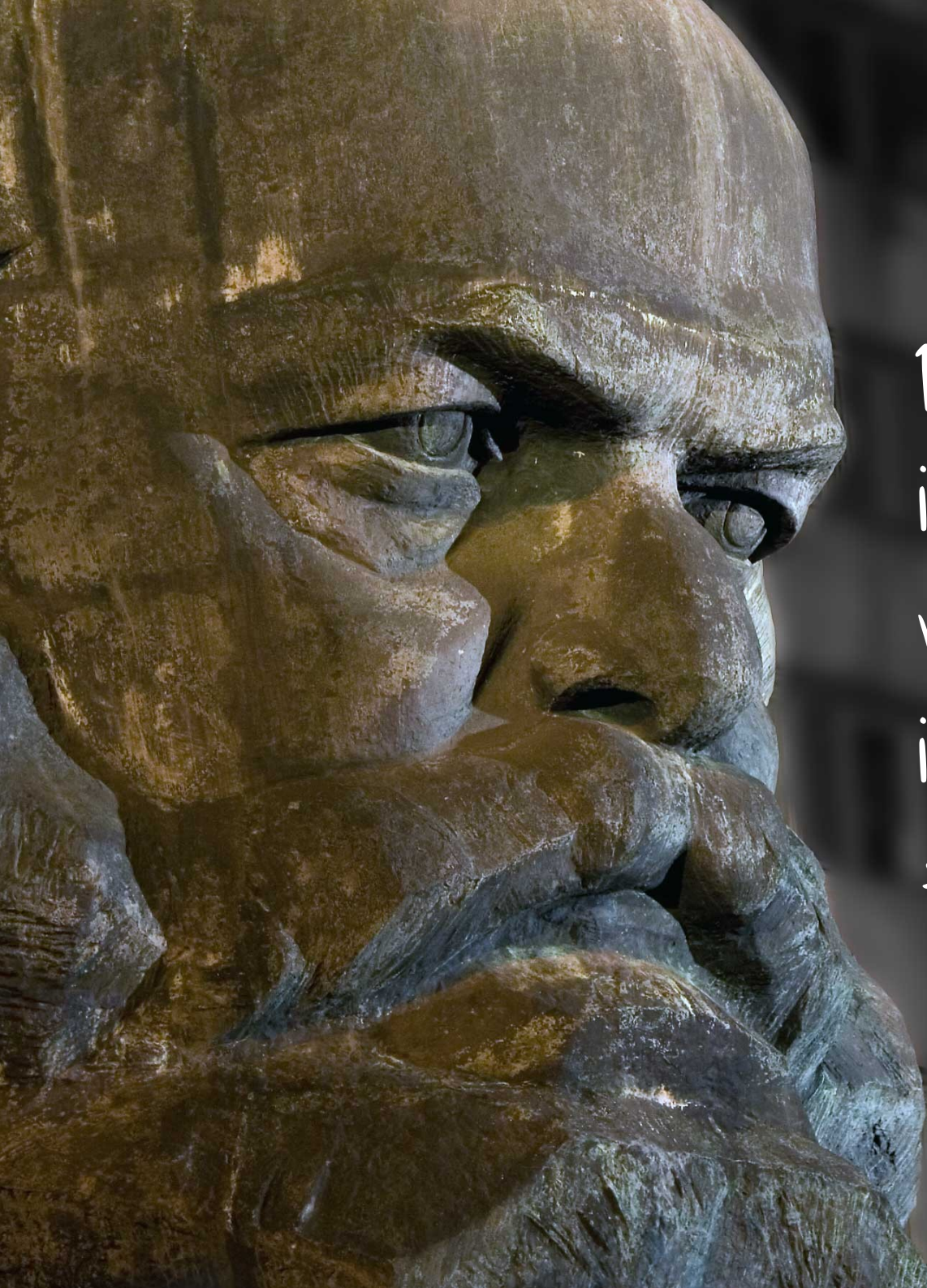
@gnuler



What is this presentation about?



- This presentation is a scrap book from our experience developing a reliable exploit against ASP.Net
- It tooks 2 people working full time to create a reliable and working exploit for this vulnerabilities
- All the kudos to Juliano Rizzo and Thai Duong for finding such a clever technique and teaching the world about the risks



Padding Oracle:
it's not a
vulnerability
it's an
Attack

What is the vulnerability?

- The vulnerability is a bad crypto implementation when using cbc mode of operation
- A block cipher by itself allows encryption of only a single data block of the cipher lengths
- To encrypt blocks that are not multiple of the block size in length we need to add some padding
- IBM came out with CBC (Cipher Block chaining), this mode of operation causes the decryption of a block of ciphertext to depend on all the preceding ciphertext blocks (ie. If you encrypt again the same block, the ciphertext will be different.)

I'm protected, I'm using AES!



- CBC encryption mode only provides **CONFIDENTIALITY**.
- Confidentiality doesn't ensure that the message has being **tampered** with, you need to apply **AUTHENTICITY** (integrity) check.
- This is what ASP.NET, JavaFaces, RubyOnRails, (your custom code perhaps), lack :>.

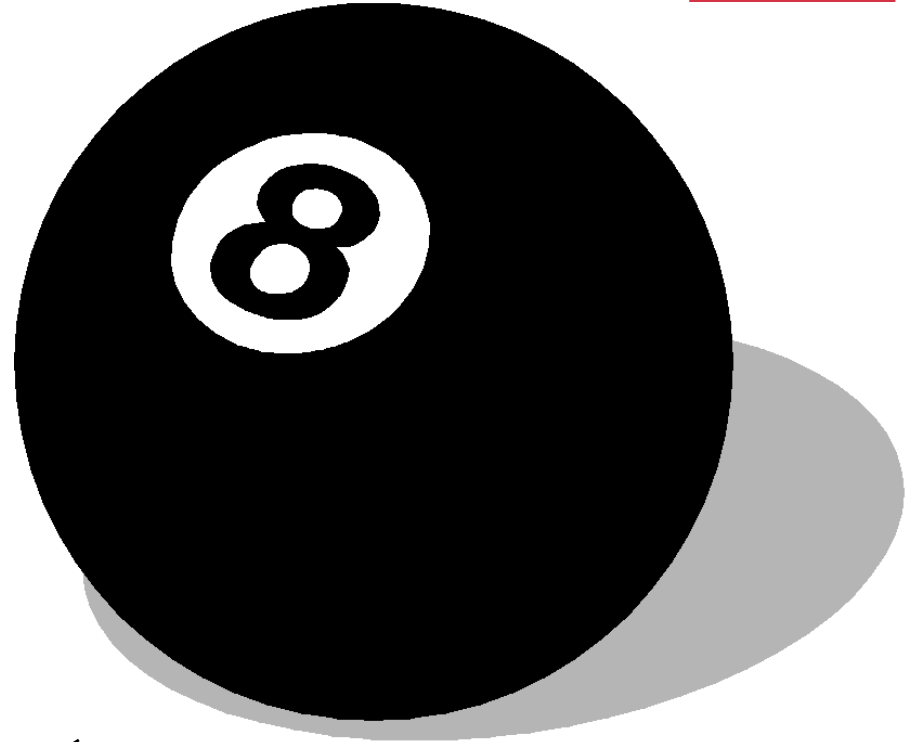
QWERTYBLAHBLAH

CORRECT DECRYPTION

WRONG PADDING



Evil Hacker



WRONG PADDING



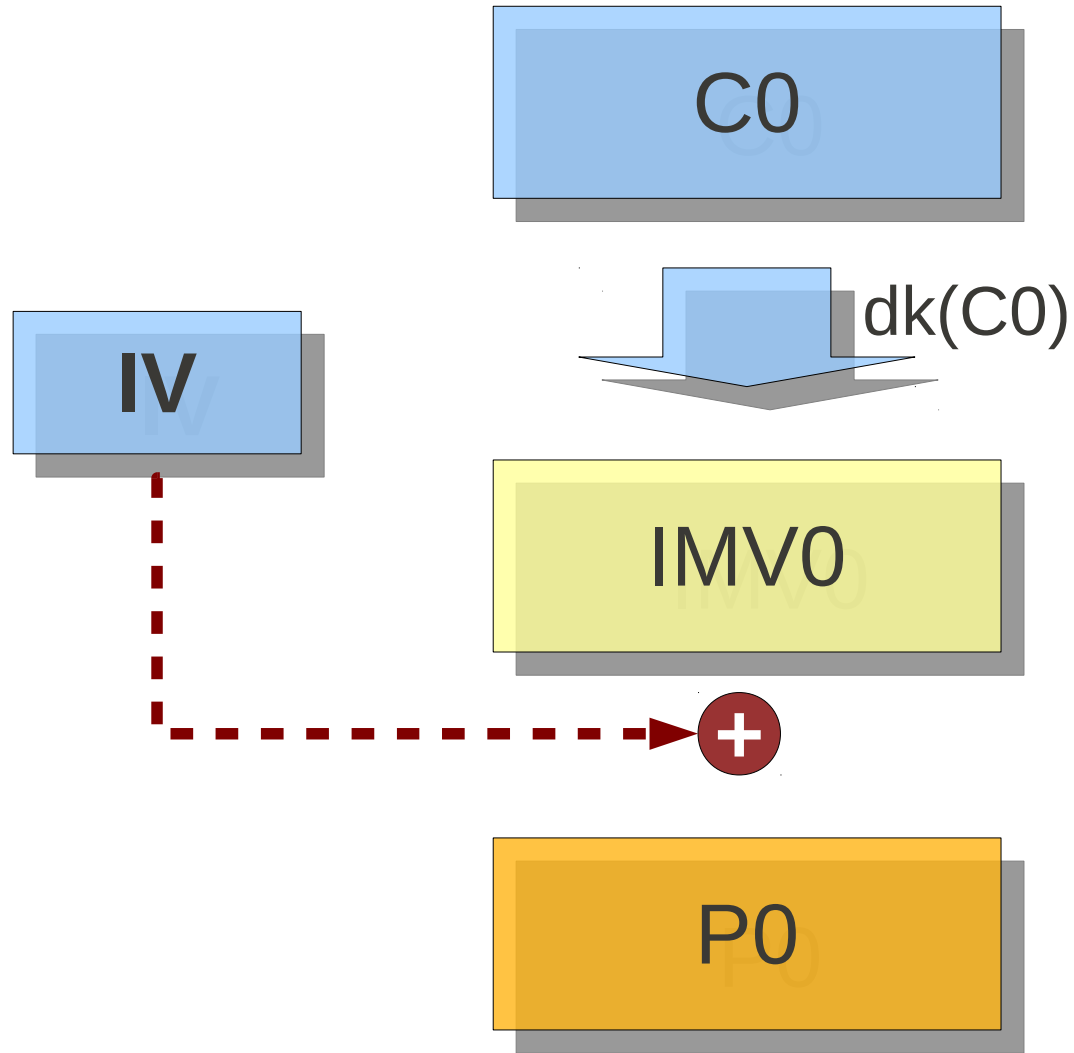
The difference between a correct decryption or a wrong padding came in different flavours:

- Time difference
- Error Code
- Stack trace
- HTML length
- Strawberry

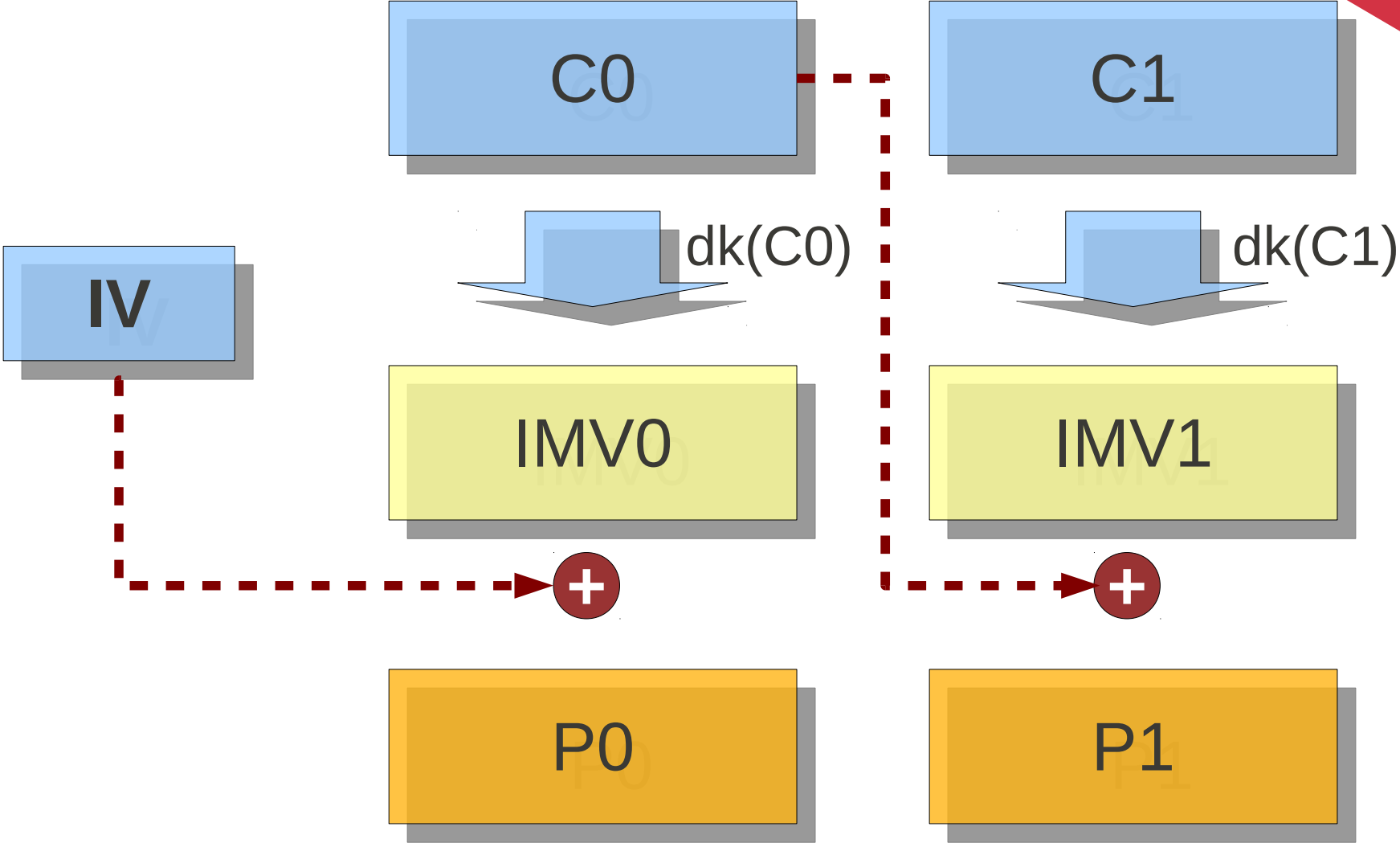


How CBC works?

Decryption process



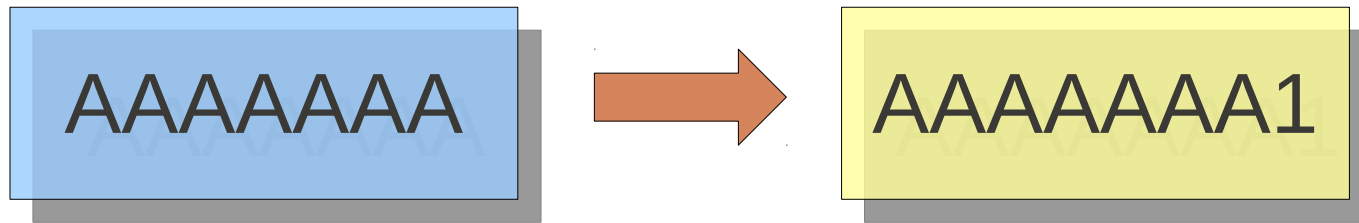
Decryption process





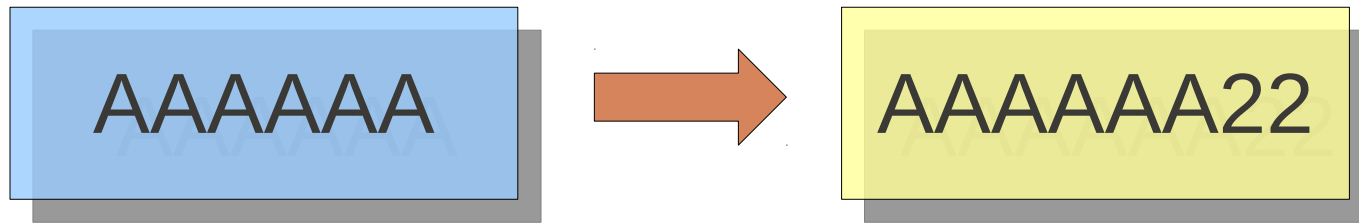
How padding works?

Ej: block-size = 8



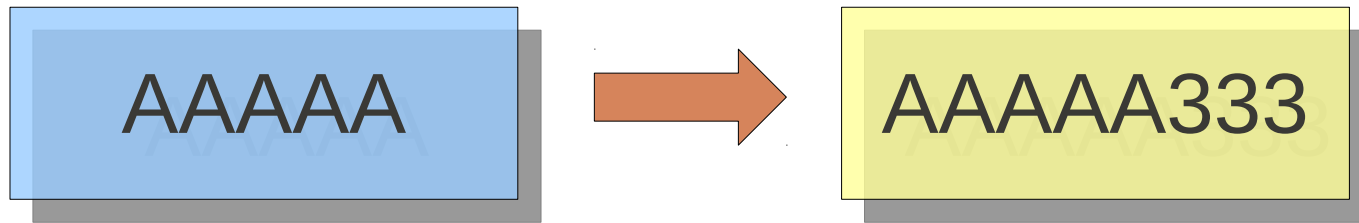
seven bytes data, padded with one `\x01` byte

Ej: block-size = 8



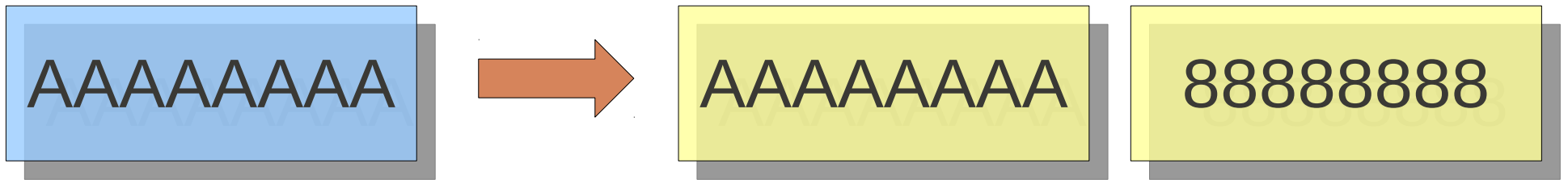
Six bytes data, padded with two `\x02` bytes

Ej: block-size = 8



Five bytes data, padded with three `\x03` bytes

Ej: block-size = 8



Eight bytes data, padded with eight `\x08` bytes



Padding Oracle 101



captcha.php?val=ABCDEFGHIJKLMNPO



IV

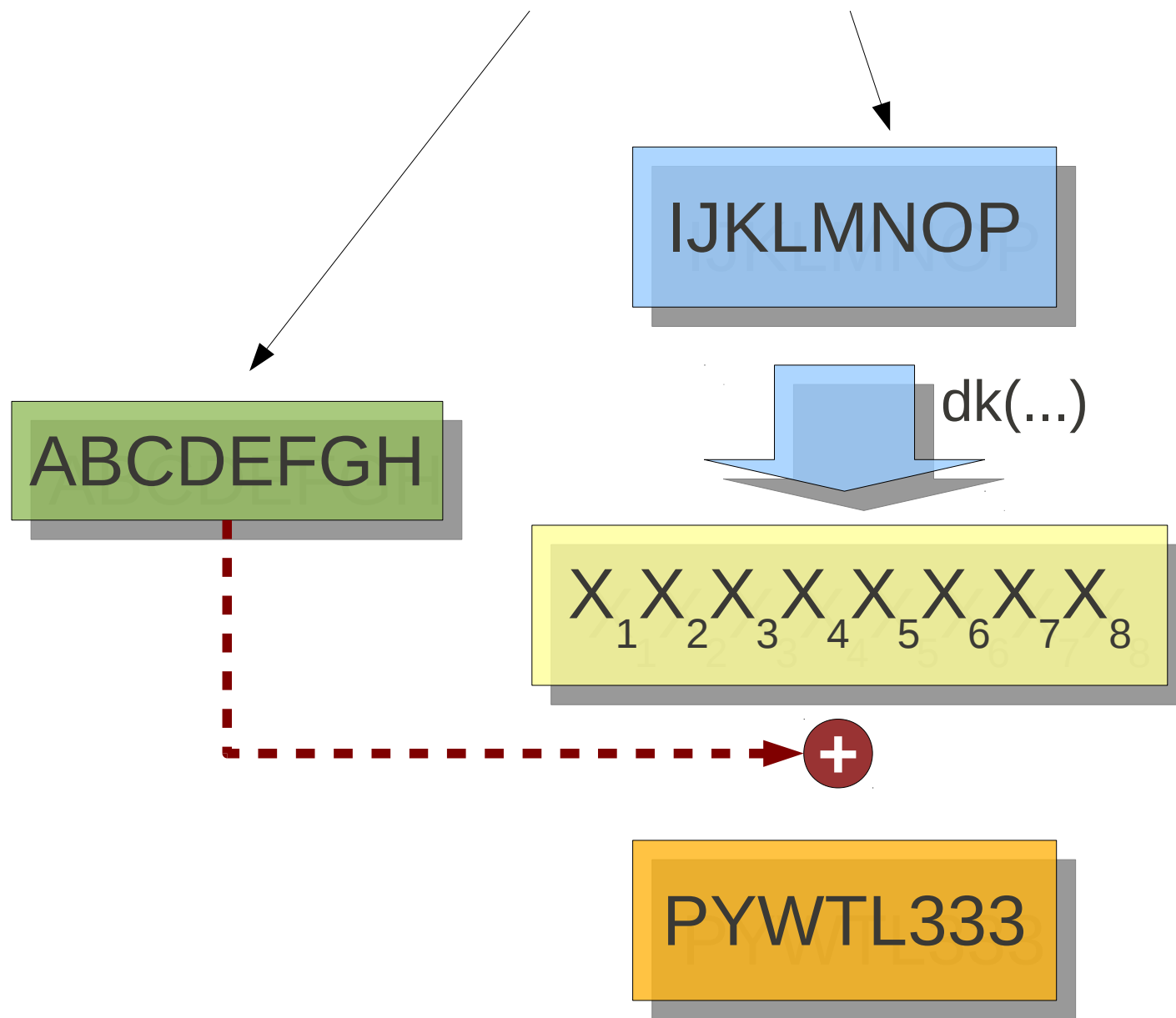
captcha.php?val=

ABCDEFGH

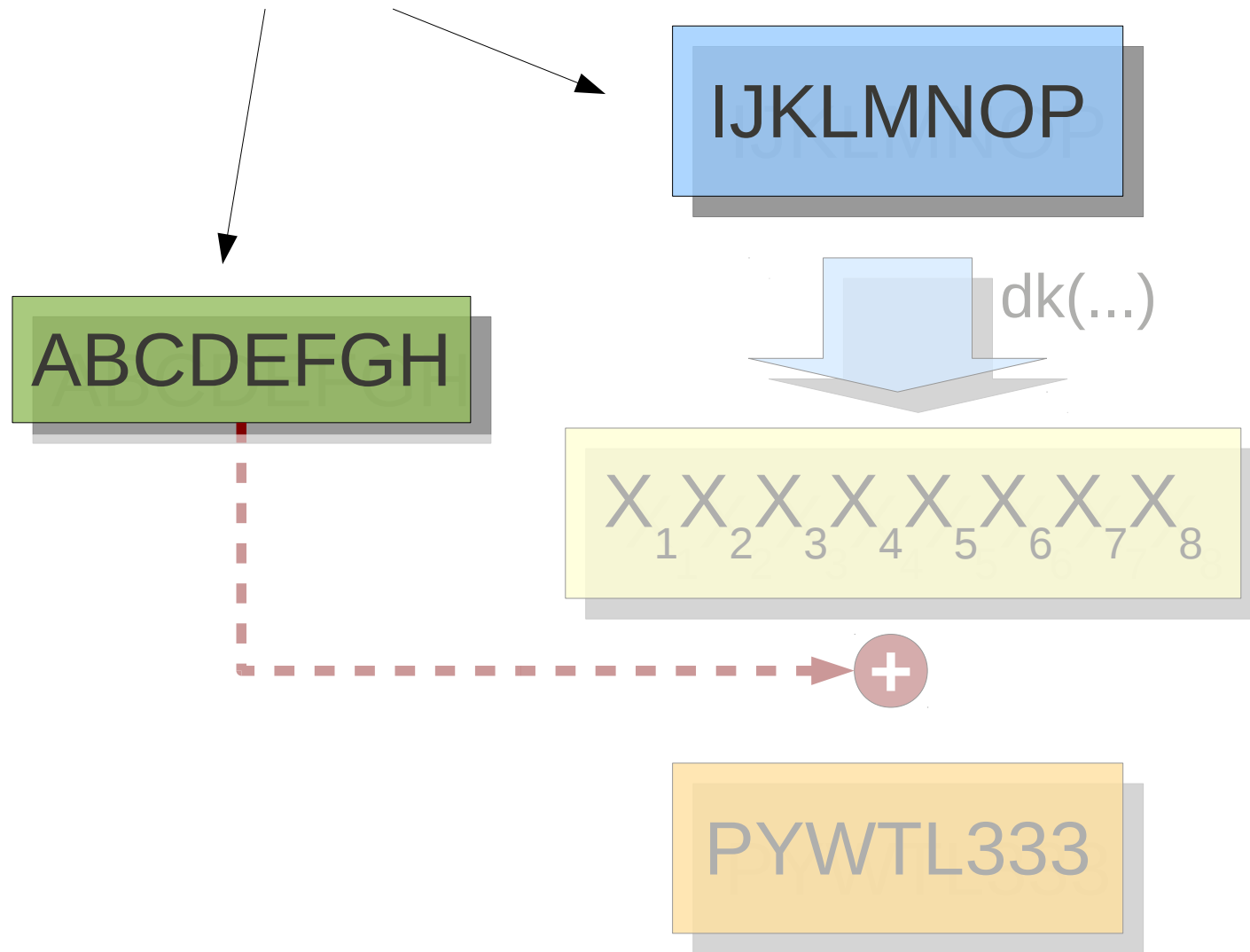
IJKLMNOP

Encrypted Data

captcha.php?val=ABCDEFGHIJKLMNOP

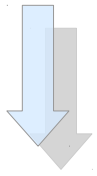


Controlled DATA





IJKLMNOP



dk(...)

X₁ X₂ X₃ X₄ X₅ X₆ X₇ X₈

IMV




ABCDEFGH

IV



PYWTL333

Plaintext

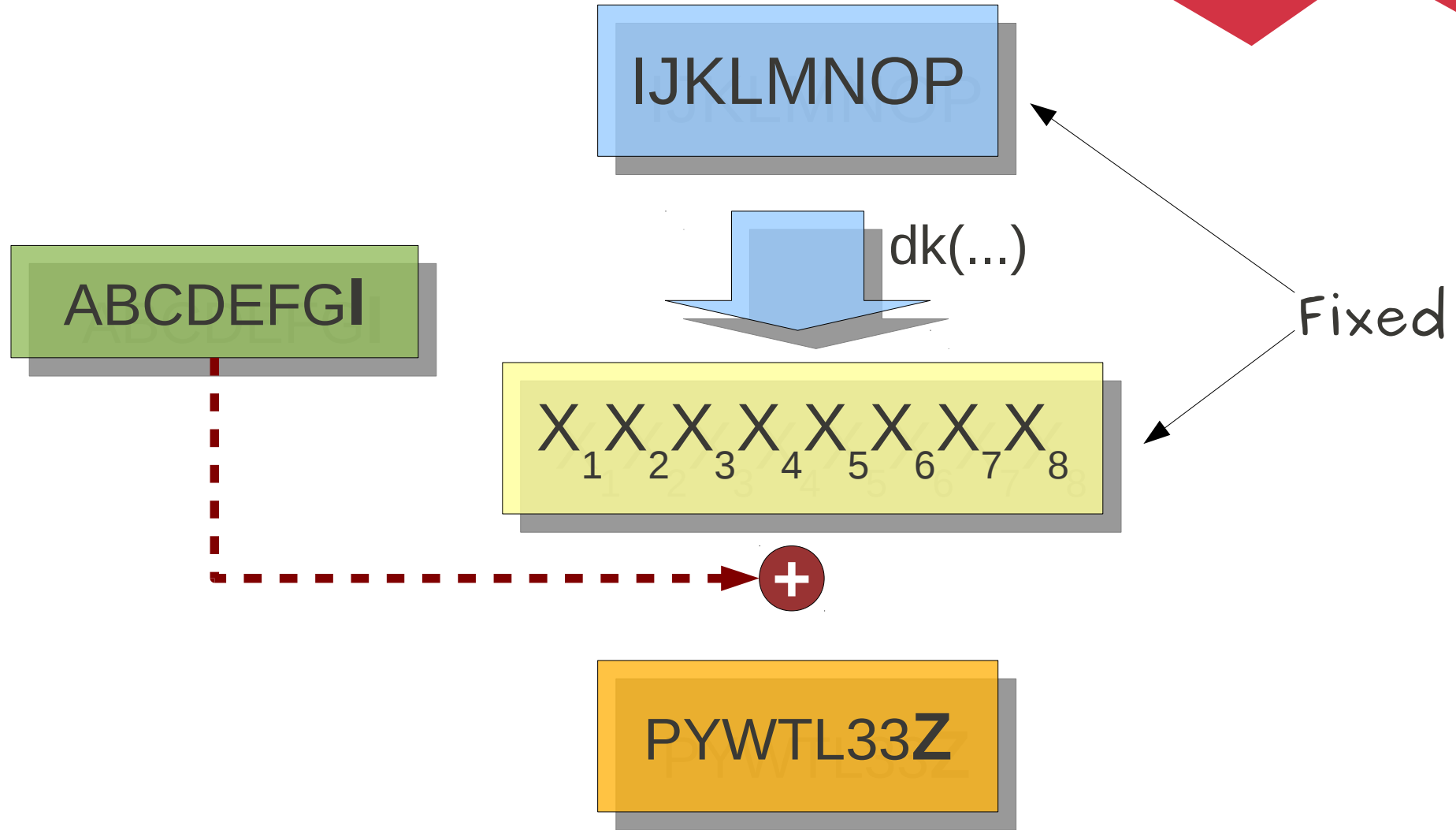


So, if by some means we can know
the IMV for a specific block, we
can forge custom captchas

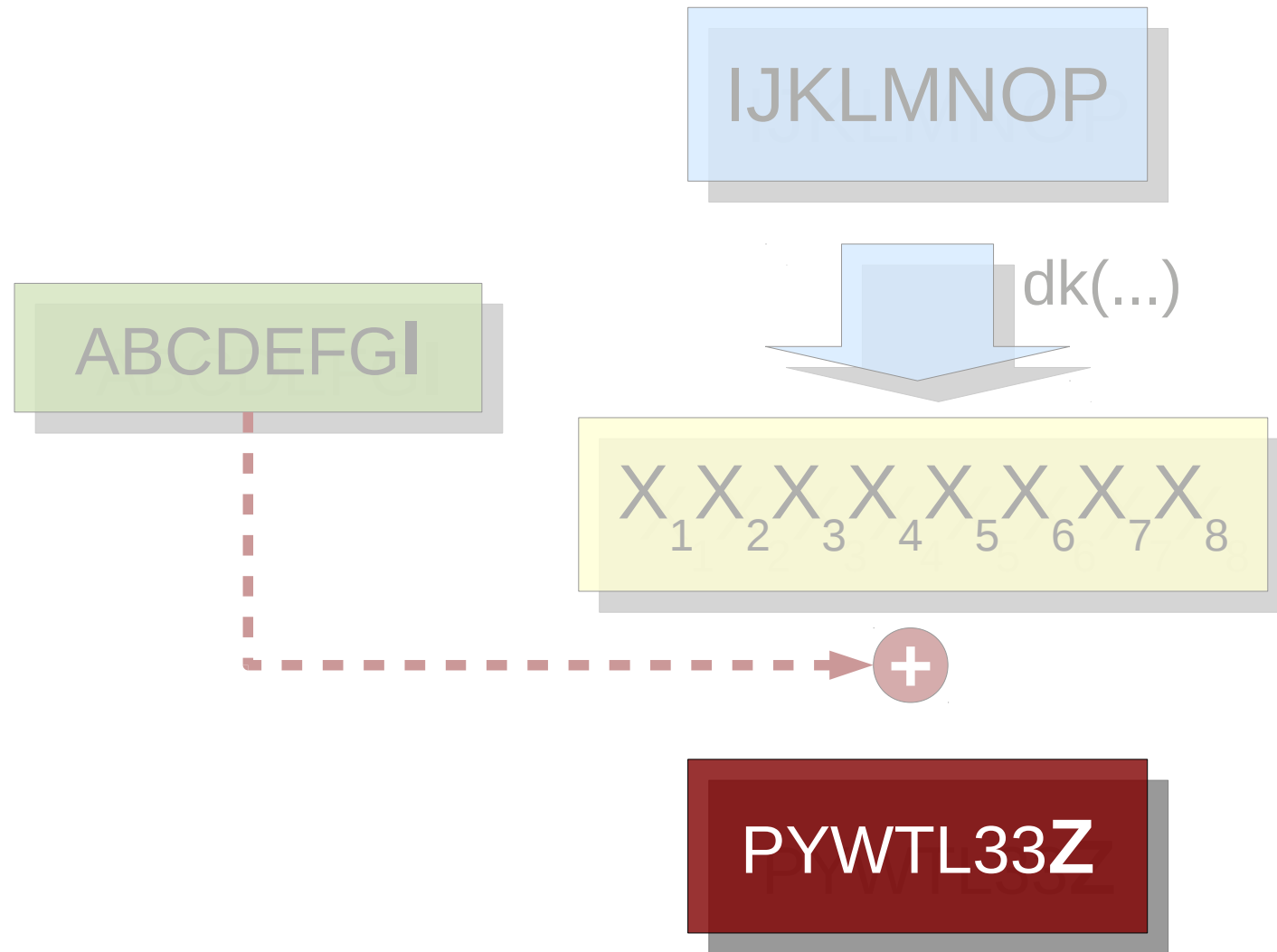


How does the attack works?

We modify the last byte from the IV
We leave the enc block Fixed

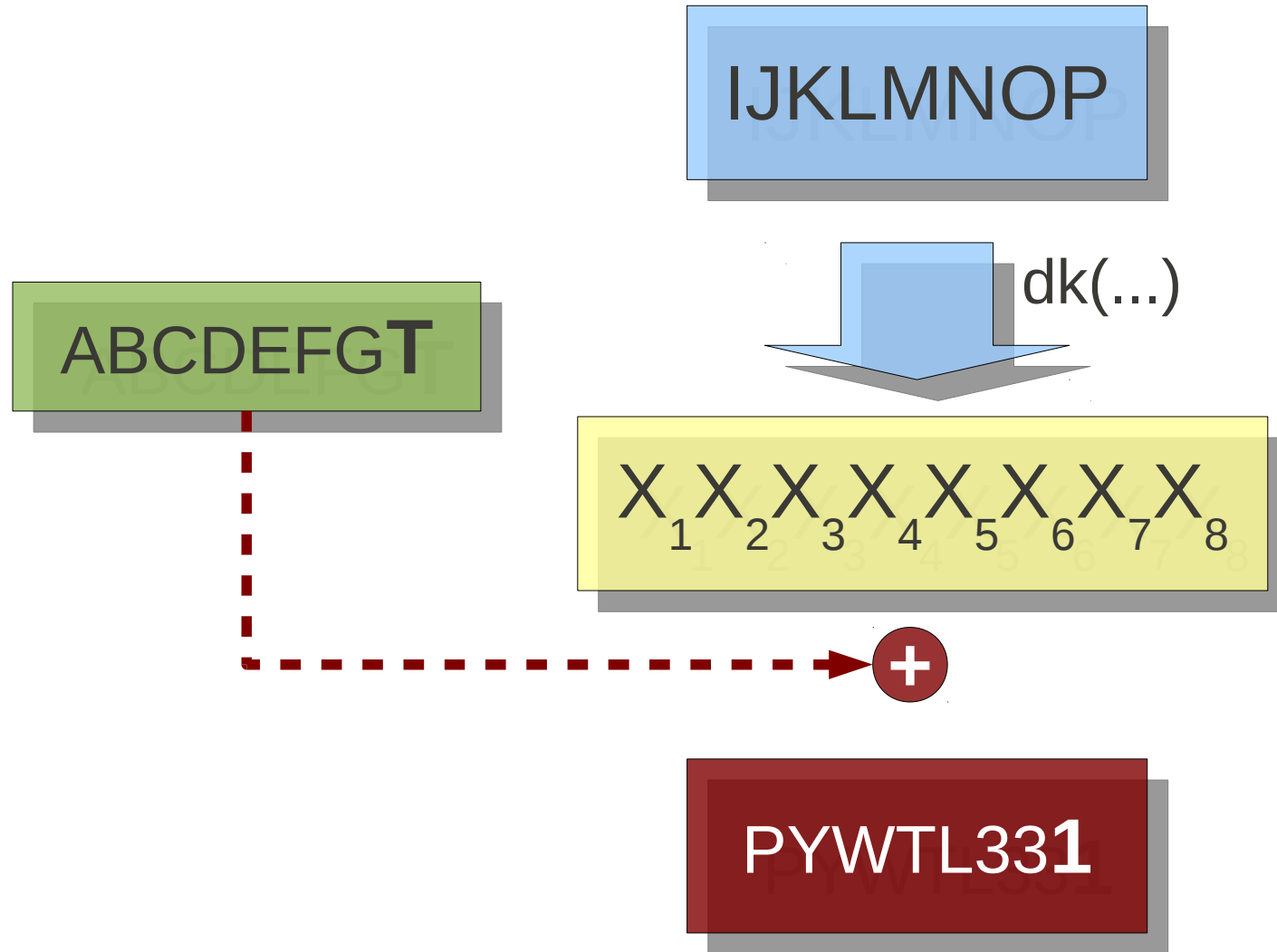


This will make the final result work or fail, in this case we can see that 33Z is not a valid padding



Wrong Padding


We keep changing the iv's last character until we find the correct padding (the web will behave differently)



Padding Ok!

$$T + X_8 = 1$$

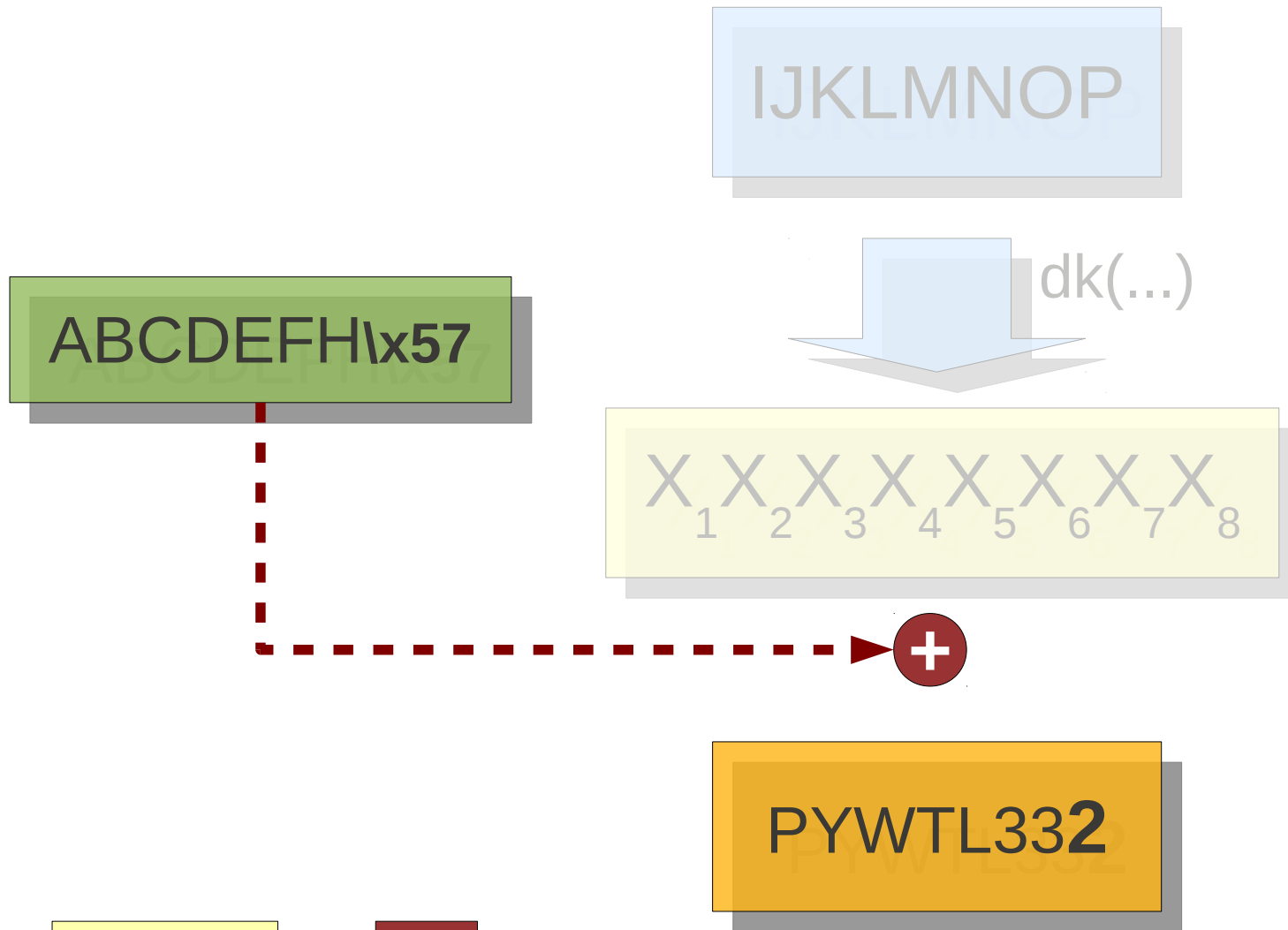



$$X_8 = 1 + T = 0x55$$



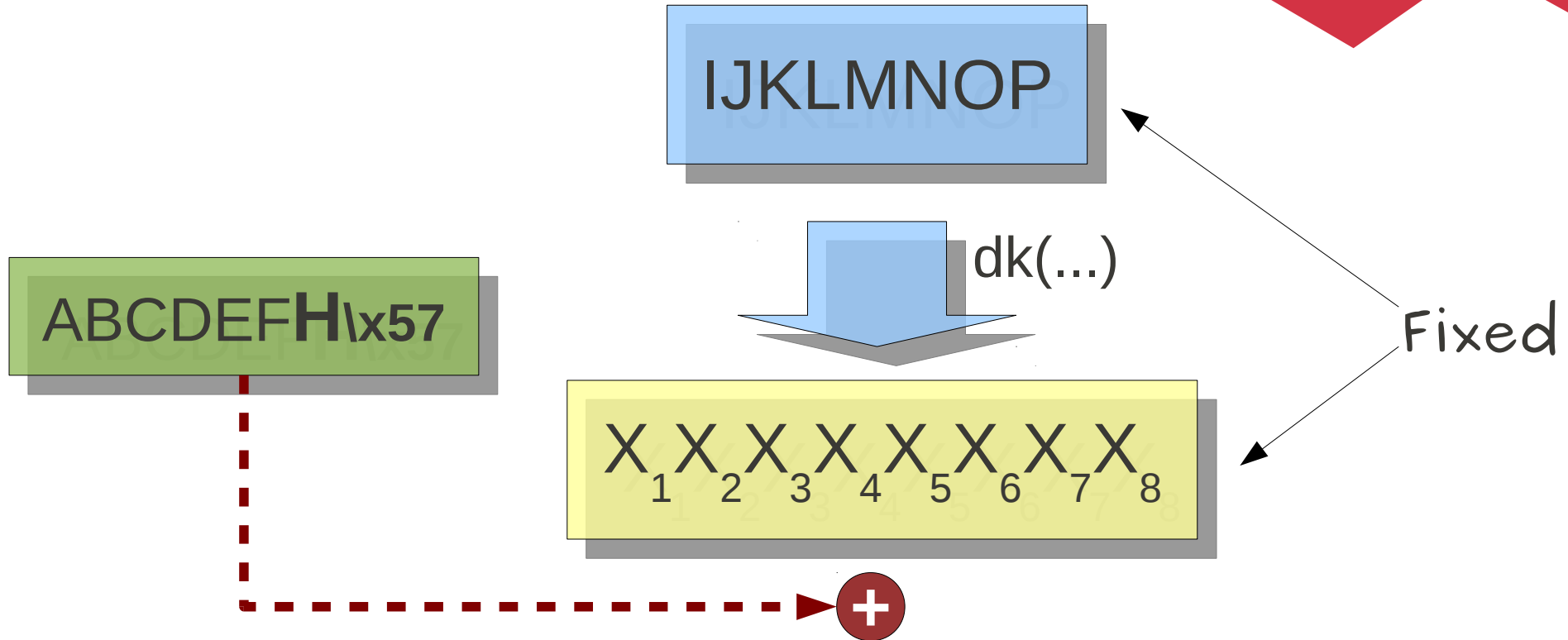
Now that we know the 8th byte of the IMV, we go for the 7th

First we set the last IV byte such as the last decrypted byte is a 2



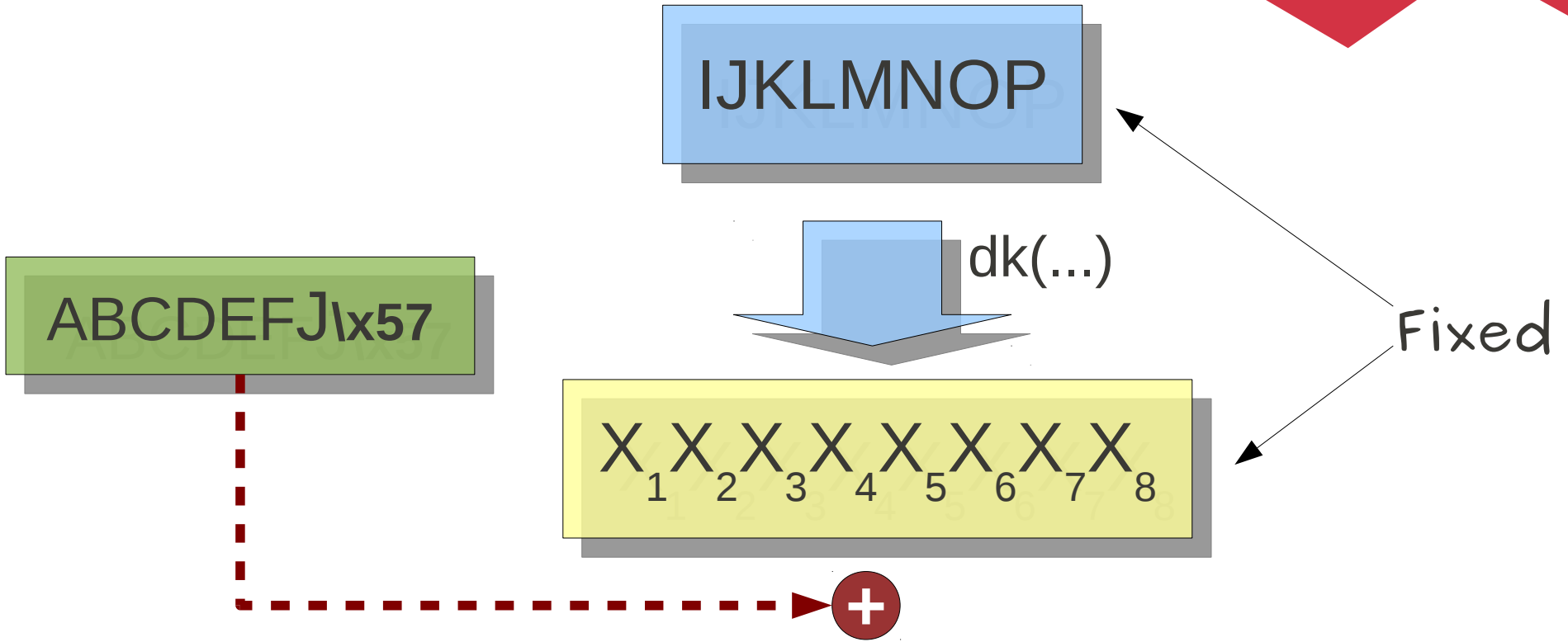
$$X_8 = 0x55 \oplus 2 = 0x57$$

Now we try to find the 7th IMV byte by keep changing the 7th IV byte



PYWTL372

Wrong Padding

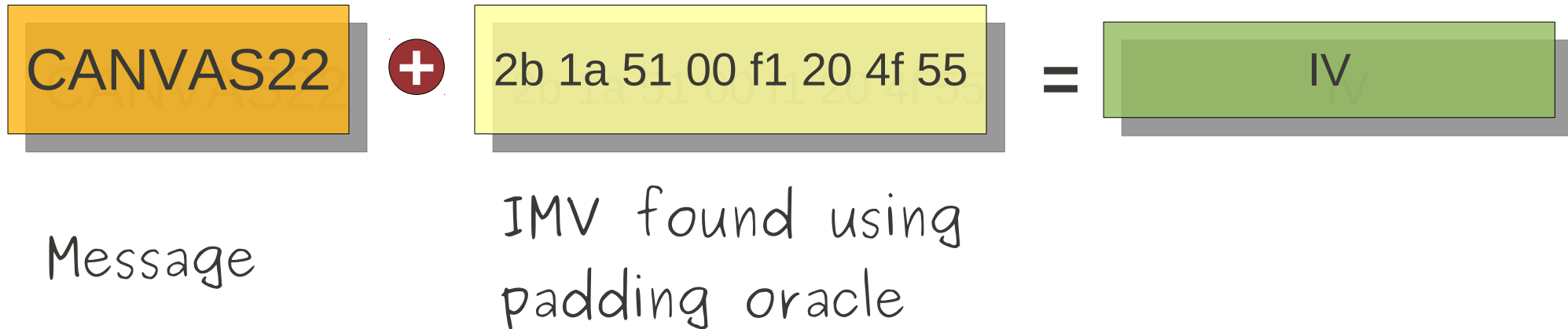


PYWTL322
Padding Ok!



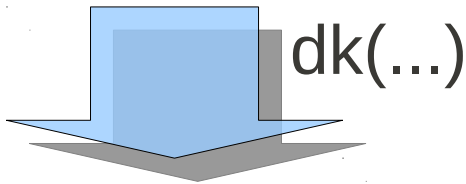
If you want to make your
encrypted buf say something,
you already have the pieces!

We calculate an IV in order to produce our message :)



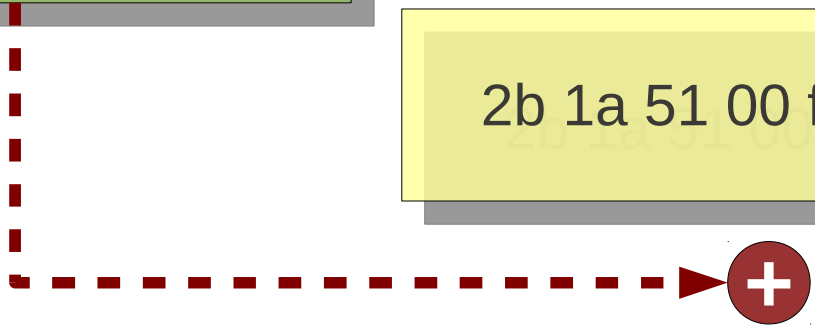


IJKLMNOP



68 5b 1f 5e b0 73 4d 57

2b 1a 51 00 f1 20 4f 55




CANVAS22



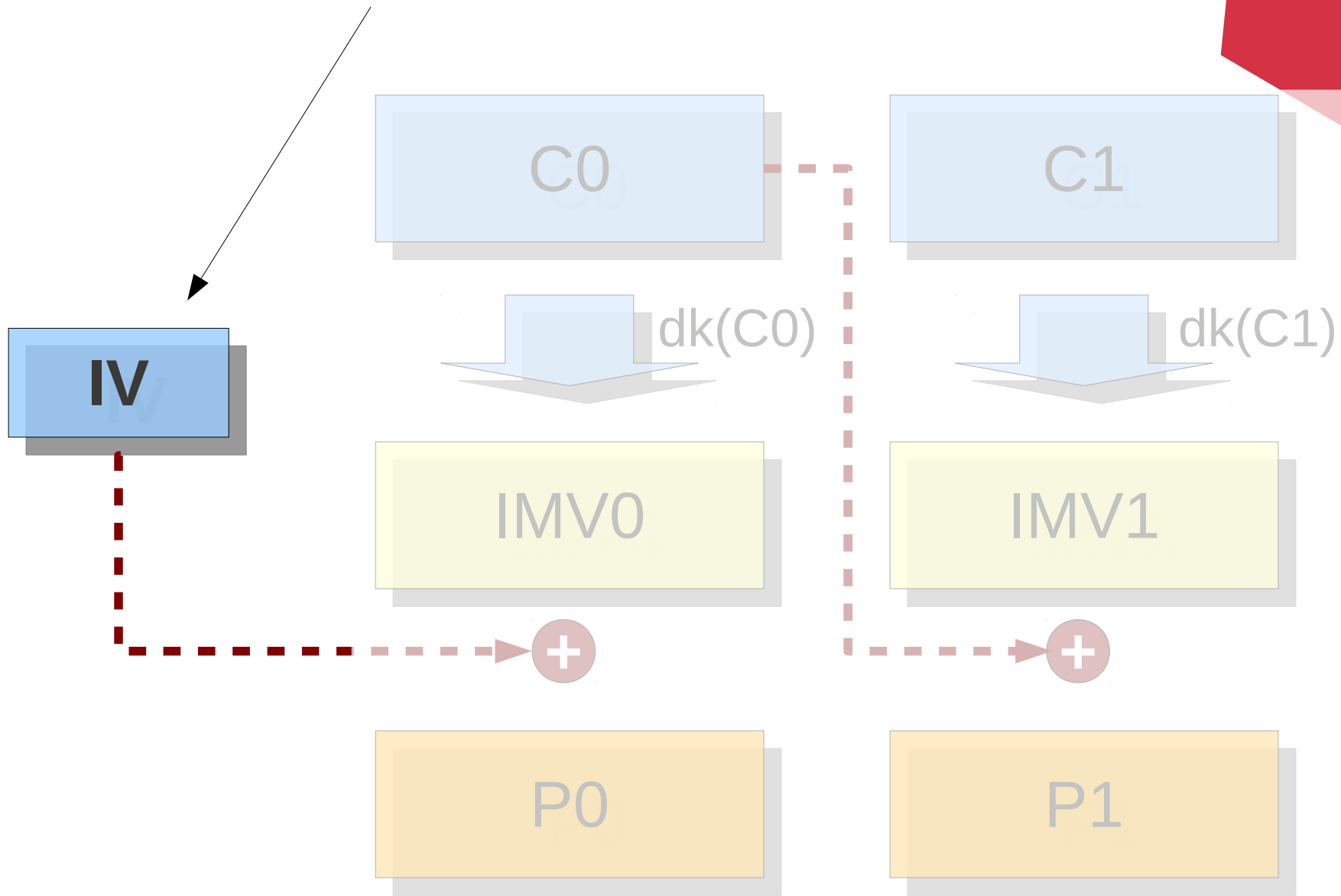
ASP.NET




- In the ASP.NET case the IV is on the server, fixed, we cannot alter it

- 
- We could still attack the Padding Oracle and do CBC-R, but we cannot fully control the content of the first block

It's Fixed on the Server



- 
- You attack `ScriptResource.axd`, that will allow you to download any file on the www root, including `web.config` (machine password, etc).



$R| \sim / \text{web.config}$



R#XXXXXXXXXX|~ /web.config

- After a lot of brain cells burning, we came out with the following:

RANDOM

IV for next

IV for next

ABCDEFGH I



R#xxxxxx

TRASHED

|||~/web

.config



RANDOM

IV for next

IV for next

ABCDEFGHJI

Obtained via randomly modifying the block
12k to 100k average (could be more).

Obtained via Padding Oracle.
Around 2k request.

R#xxxxxx

TRASHED

|||~/web

.config

Workarounds



- They don't work, the only way to fix the bug is the patch
- Examples:
 - Redirecting all the logs to the same web
 - Adding a random sleep to each request

Workarounds

- Map all the single errors to the same page
- Setting on web.config:

```
<customErrors mode="On" defaultRedirect="~/error.html" />
```
- Bypass:
 - Adding "&aspxerrorpath=url" makes the error available

Workarounds

- Adding a random sleep to each request
- Replace `Error.aspx` with something that do:

```
RandomNumberGenerator prng = new  
RNGCryptoServiceProvider();
```

```
prng.GetBytes(delay);  
Thread.Sleep((int)delay[0]);
```

- Making a lot of request and doing average
- (If you can get the error code from the trick before, you don't need to care about timing)

Workarounds



- All the workarounds could be easily bypassed by doing the Magic "T-Block"
- The trick consist into randomly modifying the first block until the decryption transforms the first byte into a "T"
- This will return the other blocks un-encrypted



Magic "T" Block

- Padding Oracle is not a vulnerability, it's an ***Attack***



RANDOM

ORIGINAL1

ORIGINAL2

ORIGINAL3

Randomly change a block, to obtain a T.
This will decrypt all the other blocks.
50-1000 hits

Txxxxxx

UNENCRYPTED

UNENCRYPTED

UNENCRYPTED



Why it works???

```
private static void ProcessRequestInternal(HttpResponse response, NameValueCollection queryString, VirtualFileReader fileReader)
```

```
{
```

```
    bool flag;
```

```
    string str = DecryptParameter(queryString);
```

```
    Throw404();
```

```
}
```

```
switch (str[0])
```

```
{
```

```
    case 'q':
```

```
    case 'Q':
```

```
        flag2 = false;
```

```
        flag = true;
```

```
        break;
```

```
    case 'r':
```

```
    case 'R':
```

```
        flag2 = false;
```

```
        flag = false;
```

```
        break;
```

```
    case 'u':
```

```
    case 'U':
```

```
        flag2 = true;
```

```
        flag = false;
```

```
        break;
```

```
    case 'z':
```

```
    case 'T':
```

```
        OutputEmptyPage(response, str.Substring(1));
```

```
        return;
```

```
    Throw404();
```

```
    return;
```

```
}
```

```
str = str.Substring(1);
```

```
if (string.IsNullOrEmpty(str))
```

```
{
```

```
    Throw404();
```

```
}
```

```
string[] strArray = str.Split(new char[] { '!' });
```

```
if (flag2)
```

MAGIC "T" Block



- The magic "T" Block will replace padding oracle.
- It will also allow you to do CBC-R
- This will bypass all types of workarounds.
- Can speed up the QR-Block lookup!!

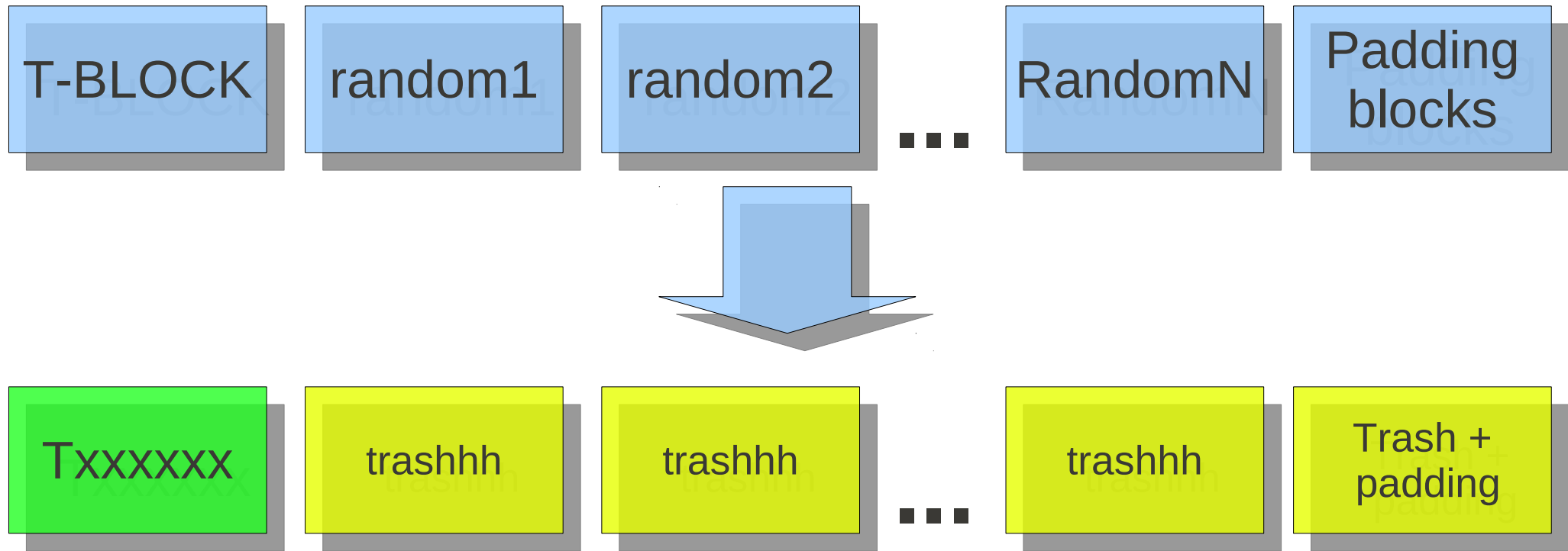


First objective: find a QRBlock

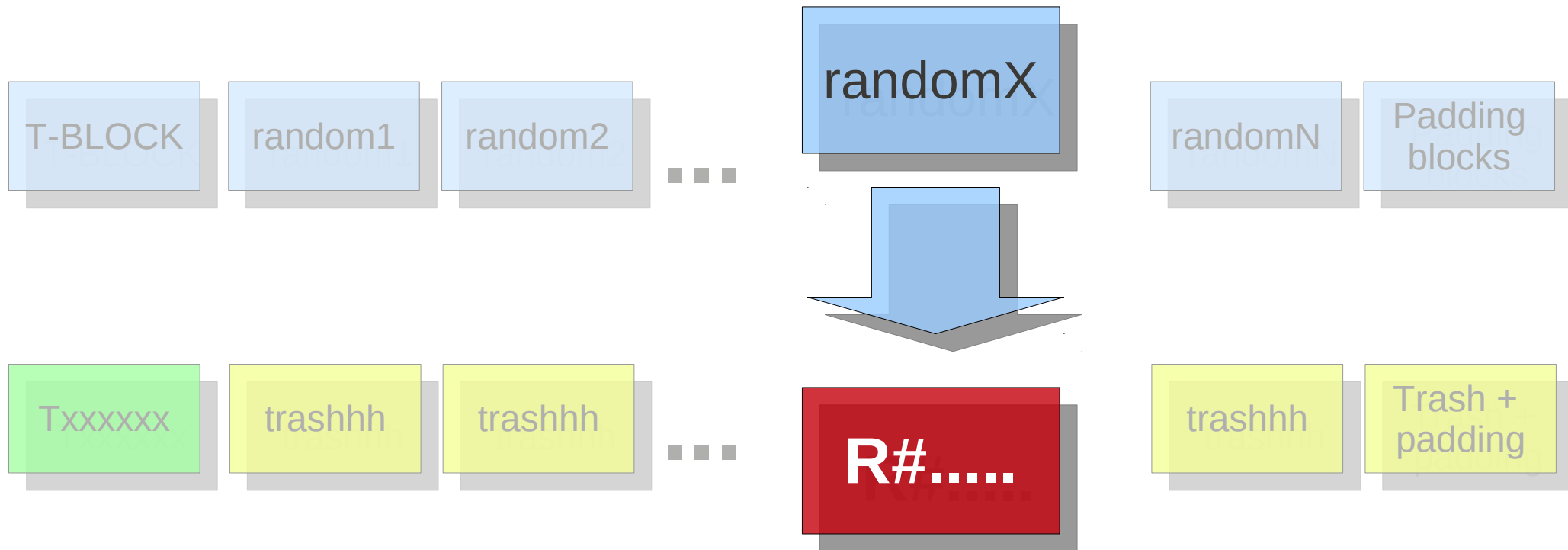
Bruteforce!!!



Send a lot of
random blocks



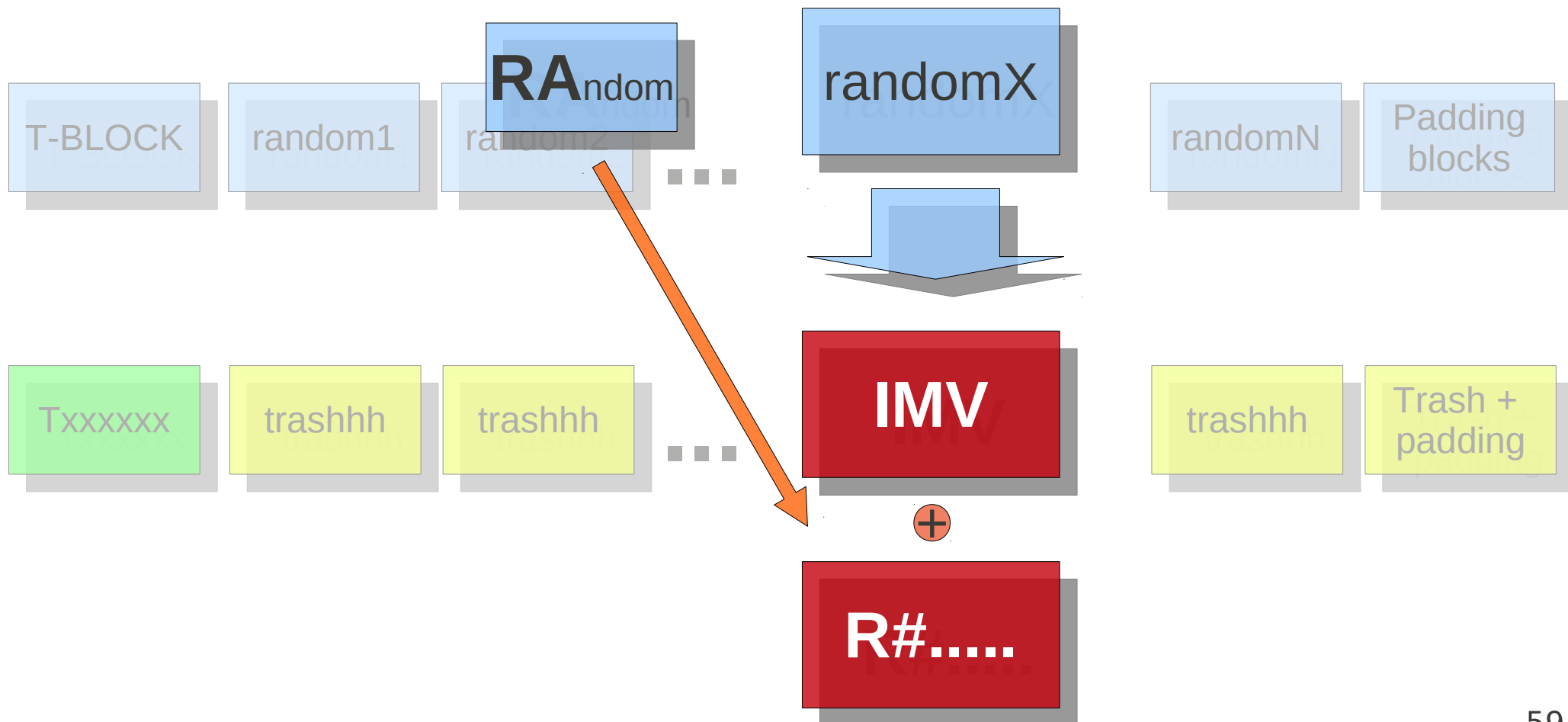
Until we find our QR-Block



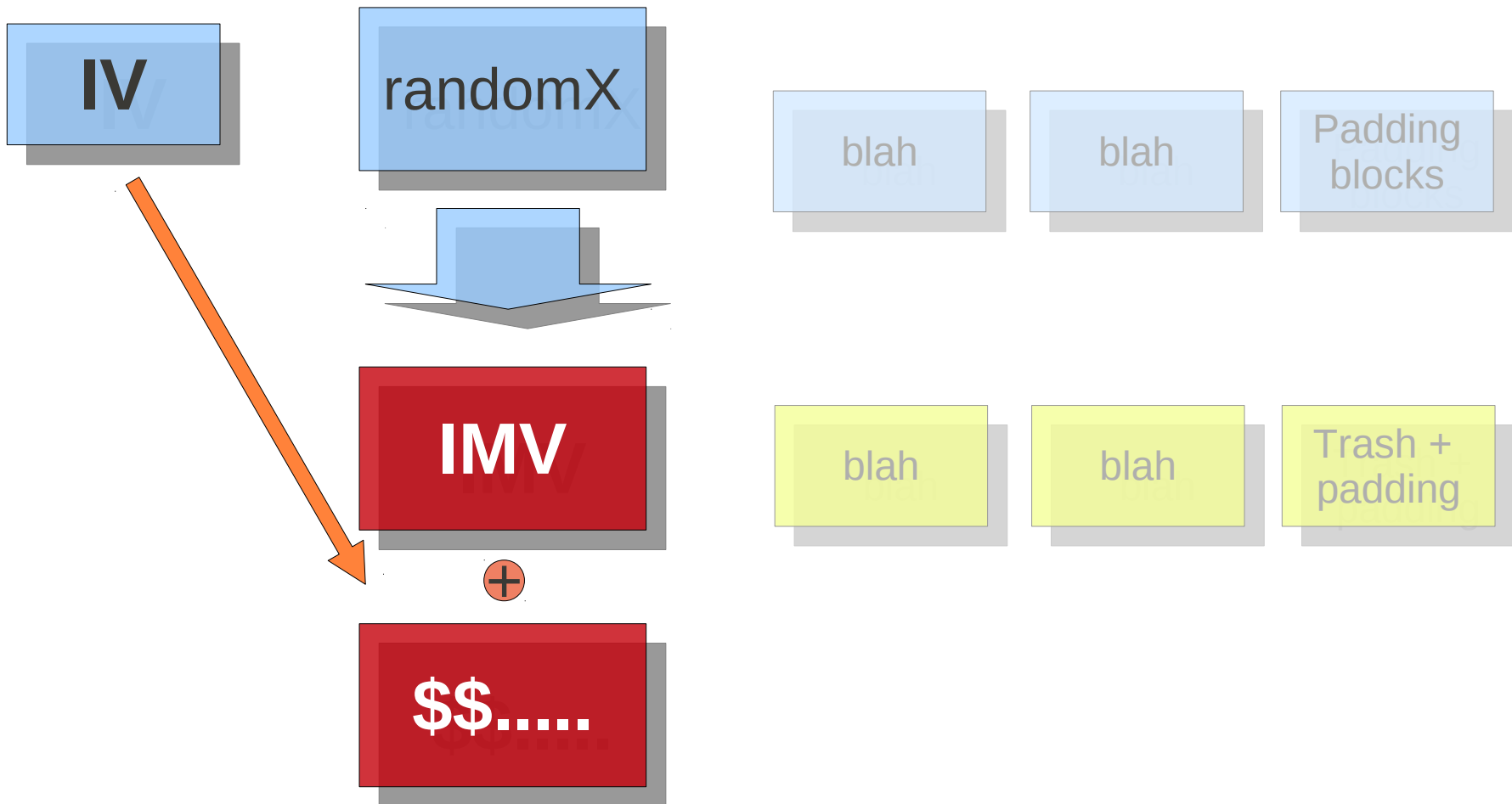


Life it's no so easy...

We really don't want an R#



We really don't want an R#



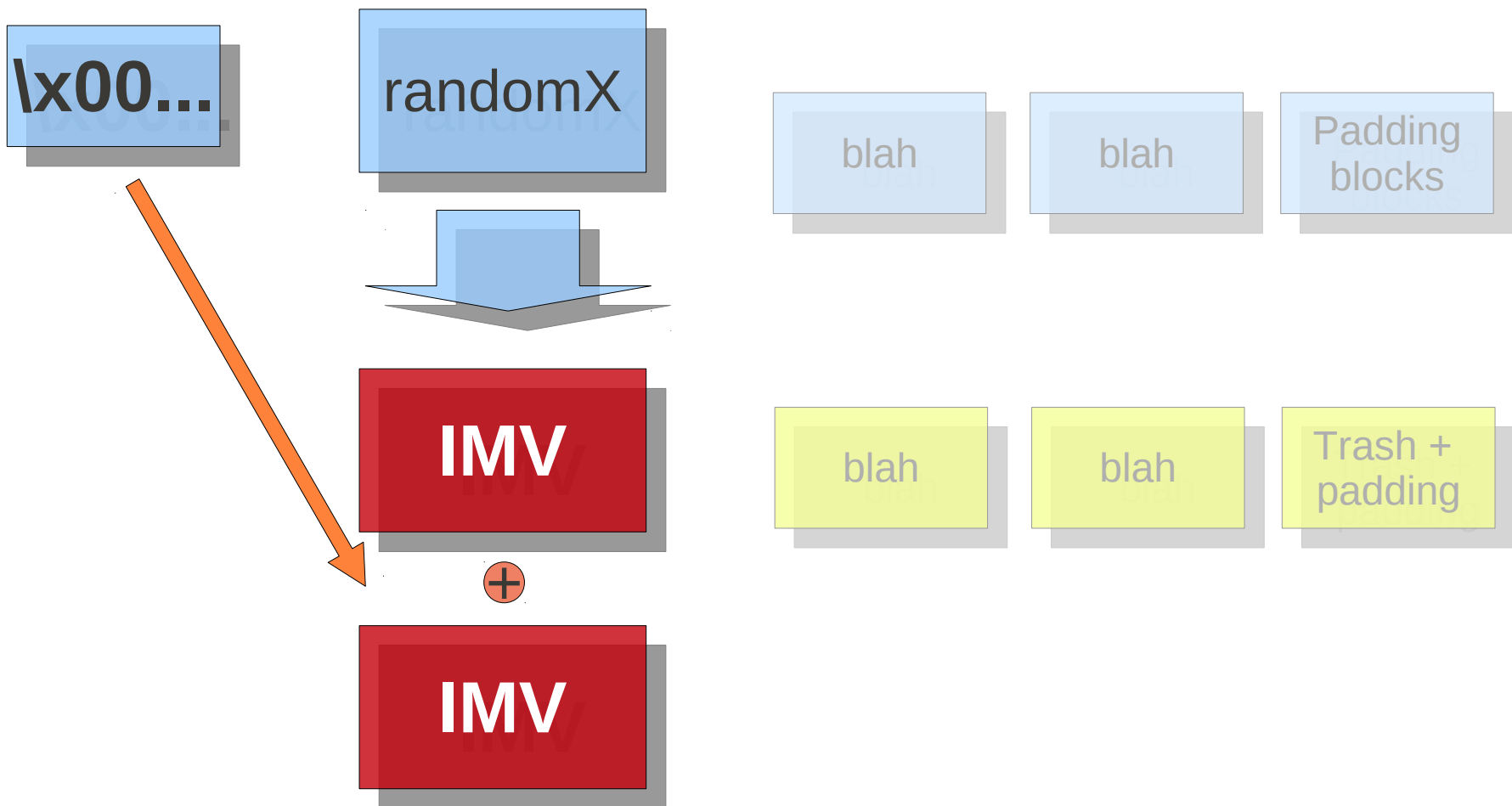


Simplification:

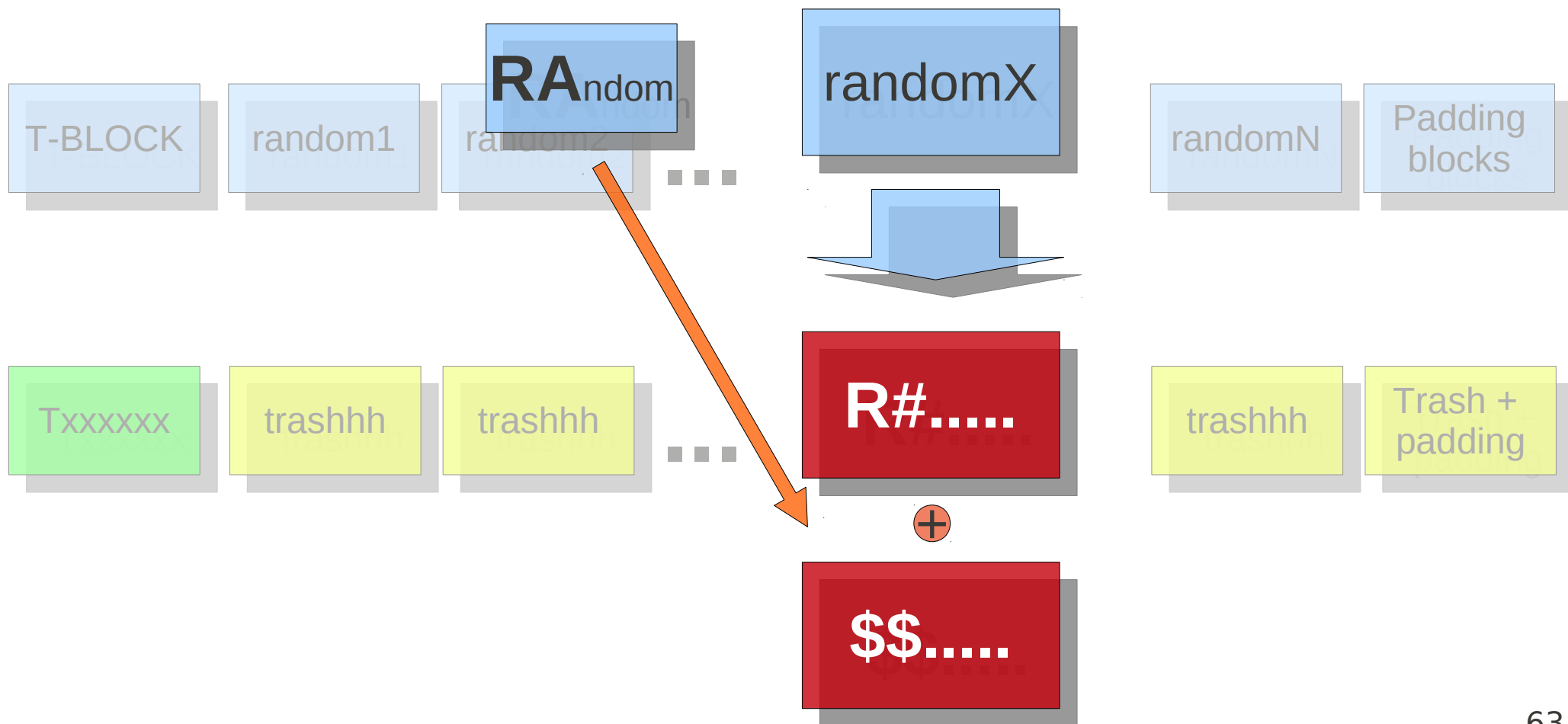
default IV =

"\x00\x00\x00\x00\x00\x00\x00\x00"

We really don't want an R#



We really want:



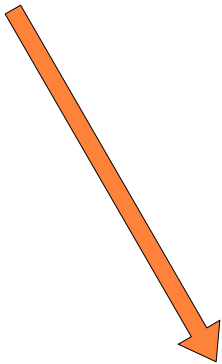
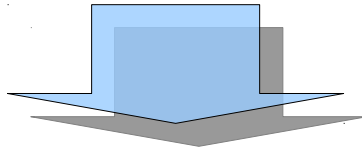


More problems...

Unicode!

CCCCCCCC

AAAAAAAA




BBBBBBBB




^G^B^ a





Each  is one, or maybe
two, invalid unicode
characters



= \xeb\x0f\xbd

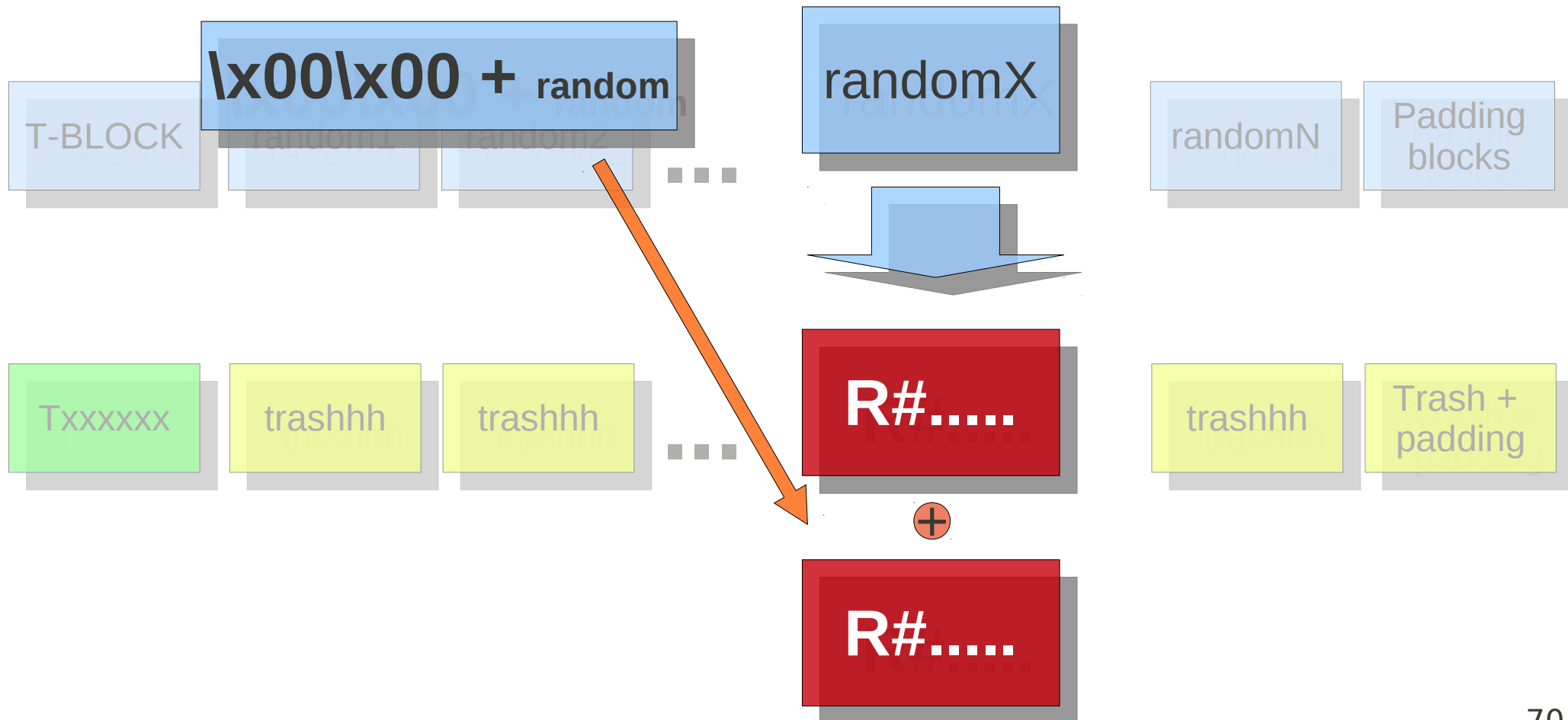


So maybe  XORed with the
Previous block is $R\#$, but
we won't know it



If instead of sending 8 random bytes you send `\x00\x00` + 6 random bytes everything is simpler!

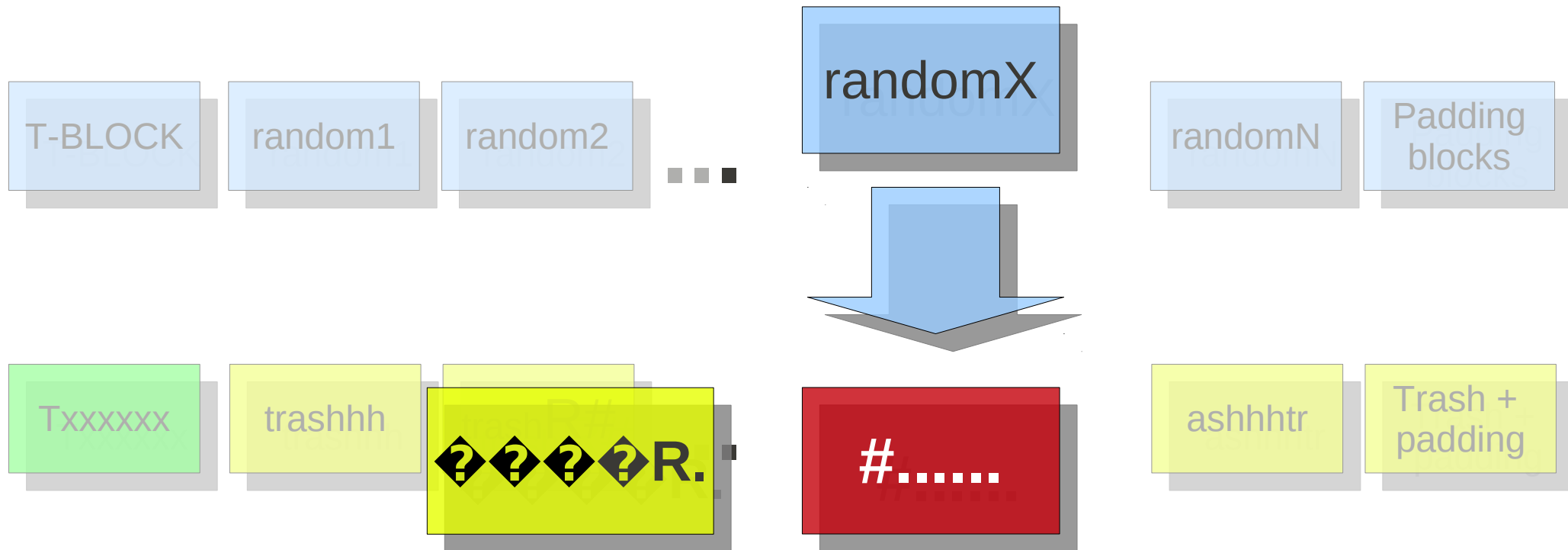
Life is easy again :)





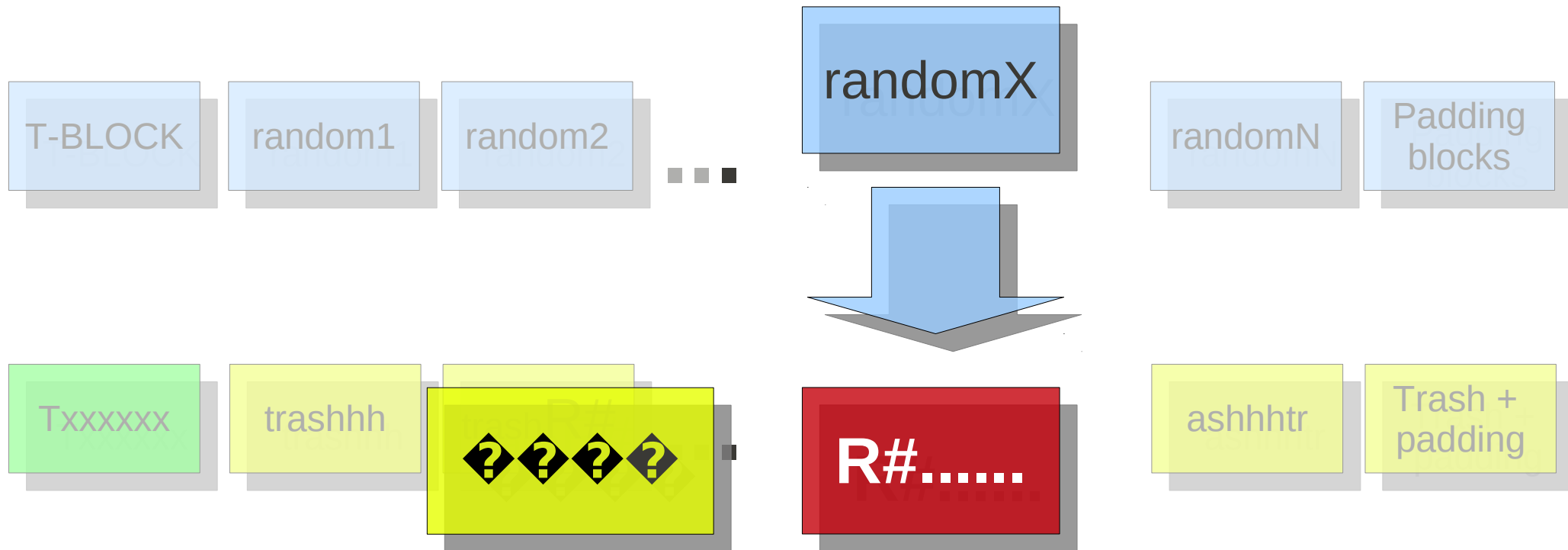
but...

we can be out of phase



False negative

we can be out of phase



False positive



Unicode: I HATE YOU!




Our solution

Every n blocks we
send a mark






This thing in numbers




Padding Oracle way took 35.000
requests in avg.



We need in average
400 requests to get a tblock

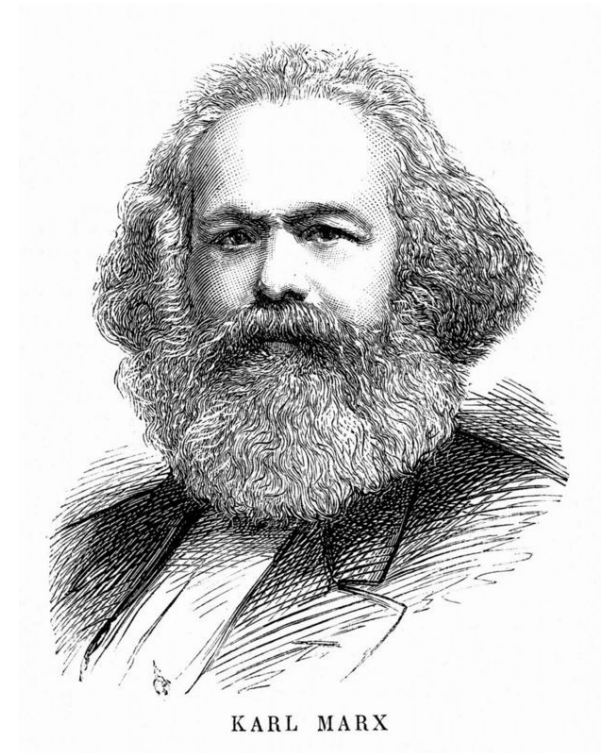


and 300 requests to
get a qrlock



Thus the complete attack
takes in avg 700 requests

DEMO



Is that all?

- ASP.net is just one wrong implementation, there are more.
- As a consultant you should be looking for:
 - Session keys that looks like base64 (ASP.NET Uses UrlEncoded base64, it is a bit different)
 - Encrypted cookies
 - Persisted information such as viewstate
 - "Any encrypted information that is stored client-side and returned to the server"

How would you fix it?



- In ASP.Net, install the **PATCH**
- The patch among many other things, introduces a MAC (Message Authenticate Code) into the CBC algorithm
- This will ensure that the encrypted blocks has not being tampered

More information



- Security Flaws Induced by CBC Padding – S. Vaudenay
- Padding Oracle Attacks on the ISO CBC Mode Padding Standard – K.G. Paterson and A. Yau
- Practical Padding Oracle Attacks – J. Rizzo and T. Duong

Conclusion

- Workarounds are useless. PATCH!
- Exploits once again show themselves to be a necessary tool to prove server risks
- This is a error of implementation, even if you fix asp.net, your own developers' software could have made their own crypto and be vulnerable

Thanks

Matias Soler

matias@immunityinc.com

@gnuler

