Week 11 — Whitebox Cryptography

Date	27.11.2026
Learning outcomes	LO.2, 3
Duration	3 hours
Prerequisites	Symmetric encryption and key management from Week 3; obfuscation rules (K-01…K-12) from Week 9; arrays and the XOR operation in C
Labs	`code/week-11` — 2 demos; build once in the `code` folder, then run from `bin/linux` (`bin\windows` on Windows)

This week's working demo

`code/week-11/01-oyuncak-tablo` — Toy whitebox: the unencoded table leaks the key, the encoded table stops the naive read. · `code/week-11/02-gomulu-anahtar` — The embedded (naive) key is physically present in the binary → the key in the software is not protected.

Run it: from the `code` folder, once `./build.sh` (on Windows `.\build.ps1`), then from the demo folder's `bin/linux` (on Windows `bin\windows`). Step-by-step commands are in the box below. Entirely synthetic and safe; it does not harm the student's computer.

Run the demo yourself — step by step (copy-paste)

The initial build is explained in `code/README.en.md`. From the **`code`** folder:

```
# Windows (PowerShell)
.\build.ps1
cd week-11\01-oyuncak-tablo
.\bin\windows\oyuncak_wb.exe
cd ..\02-gomulu-anahtar
.\bin\windows\gomulu.exe --tara
```

```
# WSL / Linux
./build.sh
cd week-11/01-oyuncak-tablo && ./bin/linux/oyuncak_wb
cd ../02-gomulu-anahtar && ./bin/linux/gomulu --tara
```

Expected output: The naive table `T[x]=S[x⊕k]` **leaks** the secret key (`0x3C`); the encoded table **does not leak it** (same encryption, different internal structure). The second demo finds the key embedded in its own binary with an **entropy scan** → "embedding a key in an array is not protection".

By the end of this week you will be able to

1. Distinguish the **black box**, **grey box**, and **white box** attacker models; define the problem that whitebox cryptography (WBC) tries to solve — "when the key and the code are in the attacker's hands at the same time".
2. Explain the WBC attacker's capabilities (reading memory, observing the flow, running a single round, fault injection) and why a software key cannot be hidden **mathematically**.
3. Analyse a **table-based** WBC implementation (Chow AES) at a conceptual level: partial evaluation, internal and external encodings, mixing bijections; and estimate its **cost** (table size, speed).
4. Read the **attack history** (BGE, DCA, DFA) of published WBC designs as a security rule: "pure software WBC is not sufficient on its own".
5. Place WBC **in its correct spot within layered defence**: alongside obfuscation, RASP, key rotation, device/version binding, and server-side risk auditing; and recognise the hardware alternatives (TEE, secure element, HSM/SoftHSM).

Class schedule (timing plan for the instructor)

Time	Part	What happens
0:00–0:25	1	Black/grey/white box models; WBC attacker capabilities; why is the problem hard?
0:25–0:50	1	The collapse of naive solutions: embedded key, entropy attack, code lifting
0:50–0:55	Break	
0:55–1:40	2	Table-based WBC (Chow AES): partial evaluation, internal/external encoding, bijections, table size and speed
1:40–1:45	Break	
1:45–2:20	3	Attack history: BGE, DCA, DFA, WhibOx competitions; "all of them were broken" and countermeasures
2:20–2:45	4	WBC's place in layered defence; hardware alternatives (TEE/SE, HSM/SoftHSM)
2:45–3:00	5+	Mistake→attack→protection example; project S8; self-check

About this week's sources

This week rests on two main sources: (1) the **secure programming technical guide** — it describes the WBC attacker model and how whitebox AES is positioned in a product (the requirement that keys be "secured and/or hidden"); (2) the field's **core academic sources** (Chow et al. 2002; BGE, DCA, DFA for cryptanalysis). We learn the WBC attacker's capabilities here **in order to test our defence**; we do not write working attack code. All examples and keys are **synthetic**. This week is the natural continuation of Week 3's "confidentiality is cryptography's job" and Week 9's "obfuscation does not protect a key" rules.

0. Basic concepts (from scratch)

This section **assumes no prior knowledge**. We define, from scratch, the terms we will use for the rest of the week. If you don't know a term, read this section first; later sections build on these.

What is encryption?

- **Encryption:** turning readable data (**plaintext**) into unreadable data (**ciphertext**) using a **key**.
- **Decryption:** reversing it with the key.

What is a key?

- **Key:** the secret number (a byte sequence) that governs the encryption.
- Same algorithm + different key = different result.
- **Security depends on the secrecy of the key**, not the secrecy of the algorithm (Kerckhoffs's principle).

Symmetric and asymmetric

- **Symmetric:** encryption and decryption use the **same** key (e.g. **AES**). Fast.
- **Asymmetric:** a public/private key pair (e.g. RSA). Slow, but key distribution is easy.

This week we will mainly work with **AES** (symmetric).

What is AES? (from a high level)

- **AES:** the most widely used symmetric encryption algorithm.
- Operates on 16-byte **blocks**.
- Mixes the data in **rounds**; each round: byte substitution, row/column mixing, key addition.

The details will not be asked in this week's exam; the **idea** is enough.

What is an S-box?

- **S-box (substitution box):** a **fixed table** that replaces one byte with another.
- This is AES's "byte substitution" step.
- Example: a 256-entry mapping such as `S-box[0x53] = 0xED`.

Keep this table in mind; it is the heart of whitebox.

Lookup table

- **Lookup table:** an array that gives "output for a given input".

- `T[x]` = the precomputed result for x.
- Instead of computing, **you read from the table**; it's fast.

Whitebox buries the computation inside tables.

XOR reminder

- **XOR** (`^`): 1 if the bits differ, 0 if they're the same.
- `a ^ b ^ b == a` → reversible.
- In AES, the "key addition" step is an XOR: `state ^ key`.

Bit security level

- "**n-bit security**" = roughly 2^n attempts are needed to break it.
- AES-128 → 128 bits; considered unbreakable today.
- Higher number = stronger.

What is a side channel?

- **Side channel:** an attack that uses not the algorithm's mathematics, but the **physical/indirect information it leaks while running**.
- Example: elapsed **time**, **power** consumption, electromagnetic emission.

This is classic on smart cards; shortly we'll see its software counterpart in whitebox.

What is DPA?

- **DPA (Differential Power Analysis):** a side-channel attack that applies statistics to a device's **power consumption** measurements to extract the key.
- It doesn't look inside the device; it measures from outside.

Keep this idea in mind; whitebox's most powerful attack (DCA) is the software form of exactly this.

Bijection (a one-to-one, onto mapping)

- **Bijection:** a transformation that maps every input to exactly one output, and that **can be inverted**.
- Example: `f(x) = x ^ 0x5A` is a bijection (its inverse is itself).

Whitebox wraps tables in **hidden bijections**.

TEE and secure element (SE)

- **TEE (Trusted Execution Environment):** a secure region of the phone's processor that is **isolated** from the operating system.
- **Secure element (SE):** a separate, tamper-resistant **hardware** chip that stores keys.

These are the strongest protection for a key; whitebox is for the situation where they are **absent**.

HSM and SoftHSM

- **HSM (Hardware Security Module):** dedicated hardware on a server that stores/operates on keys; the key **never leaves it**.
- **SoftHSM:** a **software emulation** of an HSM; the same interface (PKCS#11), but no hardware protection (for development/testing).

What is a debugger?

- **Debugger:** a tool that lets you run a program **step by step**, read every value currently in memory, and see the processor's registers (e.g. `gdb`, `x64dbg`, IDA/Ghidra's dynamic mode).
- For a programmer we use this to **find bugs**; for a white-box attacker it turns into a tool for **observing every step** of the program.

The white-box attacker's "observing execution" and "running only one round" capability is, in practice, exactly sitting down with a debugger and setting a breakpoint.

Static analysis and dynamic analysis

There are two fundamental ways to examine a program, and to understand which "box" a whitebox attack stands in, we need to distinguish between them:

- **Static analysis:** analysis performed **without running** the program, examining only the binary (or source code) (e.g. the `strings` command, reading instructions with a disassembler).
- **Dynamic analysis:** analysis performed by **actually running** the program and observing it with a debugger or the instrumentation we'll define below.

The **entropy scan** in Section 2 is static (the program is never run); **DCA** in Section 4 is dynamic (the program is run and observed repeatedly). Both are available to the white-box attacker; the black-box attacker can do neither.

What are instrumentation and an emulator?

- **Instrumentation:** modifying a program, or running it under a special tool, so that it **automatically records what is processed at specific points** while it runs (e.g. adding a layer that logs every memory read/write).
- **Emulator:** a tool that **imitates the real processor in software**; because it interprets each instruction one by one, it can record **every step** of the running program without stopping it.

If the attacker instruments the whitebox routine or runs it under an emulator, they can automatically collect the **value read/written** at every lookup-table access. They can repeat this **thousands of times** without having to manually single-step through a debugger — this is the precondition for the DCA attack we'll see shortly.

What is an execution trace?

- **Execution trace:** the **sequence** of intermediate values recorded over the course of one run — which memory address was read, which value was written, which instruction executed.
- In the black box, the attacker sees only the **final output** (the ciphertext); in the white box, the attacker can, if they wish, see and store the **entire trace** (every intermediate step of the algorithm).

In Sections 3 and 4 we will always use the word "trace" in this sense: a list of the intermediate values read from whitebox tables, recorded over and over for different inputs.

What is correlation (a statistical relationship)?

- **Correlation:** a measure of whether two variables (e.g. "an observed value" and "a predicted value computed from a key hypothesis") **change together, consistently**.
- High correlation → the two variables **appear linked**. Low/no correlation → there **appears to be no relationship beyond chance**.

Side-channel attacks (DPA, and its software form DCA) use exactly this to separate the **correct** key hypothesis from the incorrect ones: for the correct hypothesis, the predicted values turn out to be **correlated** with what is actually observed; for incorrect hypotheses, they don't.

What is a permutation?

- **Permutation:** a special kind of one-to-one, onto (bijective) mapping that **reorders** the elements of a finite set — each element takes the place of exactly one other element, exactly once.
- Example: on the set `{0,1,2,3}`, `0→2, 1→0, 2→3, 3→1` is a permutation; every value is used exactly once, none are skipped or repeated.

An S-box is a permutation of a fixed-size set (byte values); the **toy S-box** we'll build shortly is a small (4-element) example of the same idea.

What is entropy (a measure of randomness)?

- **Entropy:** a measure of how **random/unpredictable** a block of data appears to be.
- Ordinary text (e.g. an English sentence) appears to have **low** entropy: some letters (e, t, a) occur very often, others (q, z) very rarely; this uneven distribution increases predictability.
- A randomly generated cryptographic key appears to have **high** entropy: all byte values (0x00–0xFF) appear at almost equal frequency, with no pattern.
- **Entropy scan:** a static-analysis technique that walks through a binary from start to end, looking for blocks with **abnormally high entropy relative to their surroundings**; it is used to find embedded keys, compressed, or encrypted data (Shamir–van Someren's observation).

As we'll see in Section 2, this is exactly the tool that exposes the "embedding the key in a fixed array" mistake **within seconds**; the `--tara` option in this week's `02-gomulu-anahtar` demo applies the same idea.

The finite field GF(2) — very briefly

- **Finite field:** a number system with addition and multiplication defined on it that has a **limited number** of elements (unlike ordinary numbers, it doesn't "overflow"; it never leaves a fixed set).
- **GF(2):** the **smallest** finite field, consisting only of the elements `{0, 1}`; in this field, addition is exactly **XOR**, and multiplication is the **AND** operation.
- AES's internal mathematics works on bytes in a larger finite field called GF(2^8); whitebox's "mixing matrices", which we'll see shortly, are built over GF(2), i.e. with **pure XOR-based linear algebra**.

Detailed finite-field arithmetic is outside this week's scope and will not be asked in the exam; it's enough to carry the idea that "GF(2) = a world of reversible linear operations that work with XORs".

Matrix, linear transformation, non-singular matrix

- **Matrix:** a rectangular arrangement of numbers (here, only 0s and 1s).
- **Linear transformation:** a transformation that produces output bits by "multiplying" the input bits by a fixed matrix (in practice, by XOR-ing selected bits).
- **Non-singular matrix:** a matrix that **has an inverse** — i.e. the transformation it applies is **reversible** and loses no information. Whitebox's mixing matrices are always chosen to be non-singular; otherwise decryption would become impossible (the information could not be recovered).
- **Rank:** a measure of how many **independent** rows/columns a matrix has; "full rank" means the matrix is non-singular, i.e. its inverse exists.

In Section 3, Step 4, we'll make this concrete with a **small, hand-solvable GF(2) matrix example**.

Now we're ready

The terms we now know:

encryption/decryption · key · symmetric/asymmetric · AES · S-box · lookup table · XOR · bit security · side channel · DPA · bijection · TEE/SE · HSM/SoftHSM · debugger · static/dynamic analysis · instrumentation/emulator · execution trace · correlation · permutation · entropy · GF(2) · non-singular matrix/rank

These terms will be used throughout the week not **in isolation**, but in **connected chains**. Let's look at the three main chains now, because as the sections progress we'll return to **every link** in these chains:

- **Construction chain:** key → S-box → lookup table (partial evaluation) → bijection (internal/external encoding) → GF(2) matrix (mixing). This chain will be built **by hand** in Section 3, with our toy S-box.
- **Attack chain:** static/dynamic analysis → entropy scan / debugger → execution trace → correlation. This chain will become **concrete** in Sections 1, 3, and 4 — first in the white-box attacker's session, then in the DCA example.
- **Defence chain:** TEE/SE/HSM → WBC (the product of the construction chain) → key rotation + device binding + server auditing. This chain comes together in Sections 5 and 6.

Instead of memorising a term in isolation, try to remember **which link of which chain** it belongs to; the glossary at the end of the lecture (Section 8) is organised with the same logic.

Now to the real question: **what do we do when the key is in the attacker's hands?**

1. Three attacker models: black, grey, white box

In Weeks 3 and 10 we handled cryptography under the **black box** assumption: the attacker sees only the input and the output, and cannot see the key or the algorithm's internal state. The security proofs of standard algorithms such as AES, RSA, and HMAC rest on this assumption. But this assumption collapses the moment we hand the code over to the user.

A short history: where did whitebox cryptography come from?

- **1883** — Kerckhoffs's principle: security should be in the **key**, not the secrecy of the system. This is the root of white-box thinking.
- **2002** — Chow, Eisen, Johnson, van Oorschot publish the first **whitebox AES** and **whitebox DES** (for DRM). The birth of WBC.
- **2004** — Billet–Gilbert–Ech-Chatbi (the **BGE attack**) break the first WB-AES → the start of the "all of them get broken" rule.
- **2016** — Bos et al.'s **DCA** (Differential Computation Analysis): brings hardware DPA into software, breaking many WBCs **automatically**.
- **2017–2024** — every pure-software WB-AES candidate published in the **WhibOx** competitions was broken → today's rule: WBC is a **delaying layer**; hardware (TEE/SE/HSM) is preferred whenever possible.

Whitebox kriptografi: kırılma tarihi

yayımlanmış hiçbir saf-yazılım tasarım ayakta kalmadı

Sonuç: WBC bir GECİKTİRME katmanıdır; mümkünse donanım tercih edilir.

Üç saldırgan modeli: ne görüyor?

soldan sağa görünürlük artar

WBC en zor modelde anahtar çıkarmayı GECİKTİRMEYE çalışır; imkânsız kılmaz.

Visibility increases from left to right; WBC tries to **delay** key extraction in the hardest (white box) model.

Model	What does the attacker see/do?	Typical environment
Black box	Only the input and output	Remote server; network attacker
Grey box	Input/output + **side channels** (time, power, electromagnetic)	Smart card, embedded device (side-channel attacks)
White box	**Everything**: code, memory, intermediate values; can also modify	Software, on the attacker's own device

Whitebox cryptography (WBC) studies the security problems of an encryption algorithm running in a **white box** environment. In the source's own words: the WBC attacker model was defined by Chow and colleagues in 2002, and the attacker has the following capabilities (read these as **what the defence has to counter**):

- **Observing execution:** access to the instructions being processed at that moment, watching the algorithm's flow, seeing the memory in use.
- **Controlling the environment / modifying it at runtime:** tampering with program memory, running **only part** of the algorithm (e.g. one round of the cipher), changing `if` conditions, changing loop counters, **injecting faults**.

> **The source's requirement**
>
> The technical guide opens this section with a compliance statement: *"Cryptographic keys shall be secured and/or hidden to protect confidentiality and integrity."* But it immediately adds: what protects the application and its assets against these attacker capabilities is **not WBC alone, but code hardening (Week 9) and RASP (Week 6) methods together.** In other words, WBC is **not presented as a standalone solution** even in its first sentence.

A concrete example: a session of a white-box attacker (step-by-step narrative)

Let's not leave the capability list above abstract; let's follow, step by step from start to finish, **what an attacker actually does**. The scenario is entirely **conceptual and synthetic**; no real tool command or real application name is used — the goal is to see the **order and logic** of the steps.

1. **Acquisition.** The attacker downloads the target mobile application's installation file (e.g. an `.apk` / `.ipa`) onto their own computer. This is the **starting condition** of the white-box model: the code is now on the attacker's **own device**, not on a server.
2. **Static analysis.** The attacker first examines the program **without running it**: unpacks the bundle, lists functions with a disassembler, scans text constants with a `strings`-like tool. The **entropy scan** we just defined in Section 0 also comes into play here: high-entropy blocks (possible keys/constants) are flagged.
3. **Dynamic analysis — attaching a debugger.** The attacker **runs** the application on their own device and attaches a debugger. They place a **breakpoint** at the point where the encryption function is called. This is the concrete counterpart of the "observing execution" capability in the attacker model.
4. **Reading intermediate state.** When the program stops at the breakpoint, the attacker reads the current **registers** and **memory**. If the key sits in a plain variable, it is **directly visible** right here. This is the concrete counterpart of the "reading memory" capability.
5. **Partial execution / tampering.** If they wish, the attacker can run only **one round** of the encryption and stop, skip an `if` condition, or manually change a value and continue (**fault injection**). This is the precondition for the **DFA** attack we'll see in Section 4.

6. **Repetition and automation.** The attacker automates these steps for **thousands of different inputs** (the instrumentation/emulator from Section 0), recording the intermediate values each time as an **execution trace**. This is the precondition for the **DCA** attack in Section 4.

Critical observation: in none of these six steps did the attacker listen to anything **over the network**; all of it happened on **their own device, their own copy**. The black-box model's assumption that "the attacker sees only input/output" is **invalid from the start** here — because the attacker is, at the same time, **the user**.

Common mistake → consequence → rule

Mistake: designing client-side (mobile/desktop) code as though the "attacker sees only input/output" (black-box) assumption applies here too — e.g. saying "I'll keep the AES key in the C code, nobody sees the source".

Consequence: the six steps above take the attacker to the key **within minutes**; the source code being "invisible" stops nothing, because the attacker reads/runs the **compiled binary**, not the source.

Rule: treat the attacker model as **white box** for every component running on the client side. Never assume an asset (key, algorithm, business logic) will stay "hidden" on the client; either move it to hardware (TEE/SE, Section 5), raise its cost with WBC + layered defence (Sections 3–5), or never put the asset on the client at all (keep it on the server).

For comparison: what does a grey-box attacker's session look like?

Comparing the white-box attacker's six steps with the grey box makes the difference clear. The grey-box attacker works in an environment such as a **smart card or an embedded device** — the code is **not** in their hands, but the device is right at their fingertips:

1. **Access.** The attacker obtains a **physical copy** of the target device (e.g. a payment card). They cannot read the code or reach the memory (in this respect it resembles the black box) — but they **can run the device and observe it from outside**.
2. **Setting up a measurement rig.** The attacker sets up equipment that measures the **power** the card draws (or the electromagnetic signal it emits) while it runs. This is the physical counterpart of the **side channel** definition in Section 0.
3. **Many runs.** The attacker gives the card **different** inputs (e.g. different plaintexts), triggering the encryption operation for each, and records **the power trace of each run** — the physical counterpart of the white box's "execution trace", except the attacker is recording not the code but the **device's externally measurable behaviour**.
4. **Statistical analysis (DPA).** The collected traces are compared against different key hypotheses; the correct hypothesis turns out to be **statistically correlated** with a pattern in the traces (the correlation definition from Section 0). This is classic **DPA**.

Where's the difference? The white-box attacker **accesses code and memory directly** (with a debugger, in steps 3–4); the grey-box attacker **never sees the code**, only measures the device's **physical side effects** (power, time, electromagnetic emission). The common point is the logic of steps 3–4: **many runs + statistical comparison**. This shared logic explains why **DCA**, which we'll see in Sections 3 and 4 (the **software** form of DPA), is referred to as "whitebox's DPA" — DCA applies the grey box's statistical idea to the white box's **directly observable** execution traces; it needs no physical measurement rig at all.

Why can't a grey-box attacker break whitebox directly?

The grey-box attacker has only **a physical device** in hand; whitebox is pure software (e.g. a mobile application), not a separate piece of hardware whose physical "power consumption" can be measured — the software runs on the user's **own** phone, using their own operating system's resources. That's why the attack aimed at whitebox (DCA) borrows the grey box's **statistical idea** but not the grey box's **physical measurement environment**; instead, it uses the **software traces** produced with the instrumentation/emulator from Section 0.

Common mistake → consequence → rule

Mistake: thinking the three models can **substitute** for one another — e.g. "we've taken measures against side-channel attacks (grey box), so we must also be protected against whitebox attacks", or the reverse: "we've hardened our code (static analysis against white box), so we must also be safe against side-channel attacks".

Consequence: a grey-box defence (e.g. constant-time comparison, Week 3) does **nothing against static analysis**; code hardening (Week 9, against static analysis) is also **not enough on its own** against **dynamic** attacks (DCA, DFA) (Section 4). Confusing the models results in taking a measure against one threat and **forgetting another**.

Rule: for every defensive measure, explicitly ask "which capability, in **which model**, does this counter?" The "which countermeasure makes which attack harder?" table in Section 4 was built exactly to prevent this confusion.

How do these three models connect to previous weeks?

This distinction doesn't come out of nowhere; it is the **natural continuation** of the rules we built in Weeks 3, 9, and 10:

- **Week 3 — Kerckhoffs's principle.** We had learned the principle "security must rest on the secrecy of the key, not the secrecy of the algorithm". The white-box model takes this one step further: here **both the algorithm and the key** are in the attacker's hands; all that's left is the question "how expensive can I make extracting the key?"
- **Week 9 — obfuscation rules (K-01…K-08).** That week we saw rules hiding control flow (K-04), constants (K-02, K-07), and boolean values (K-08); all of them were **code hardening**, i.e. they made the white-box attacker's **static analysis** harder. WBC is a cryptography-specific, much stronger version of this: it buries not just the code, but the **key itself**, into the code's mathematics.
- **Week 10 — algorithms and key management.** The security proofs of standard AES/RSA/HMAC are written in the **black-box** model. What we'll learn this week is that when the same AES algorithm has to run in a **white-box** environment, that black-box security proof **no longer holds** — WBC tries to close this gap (partially, and expensively).

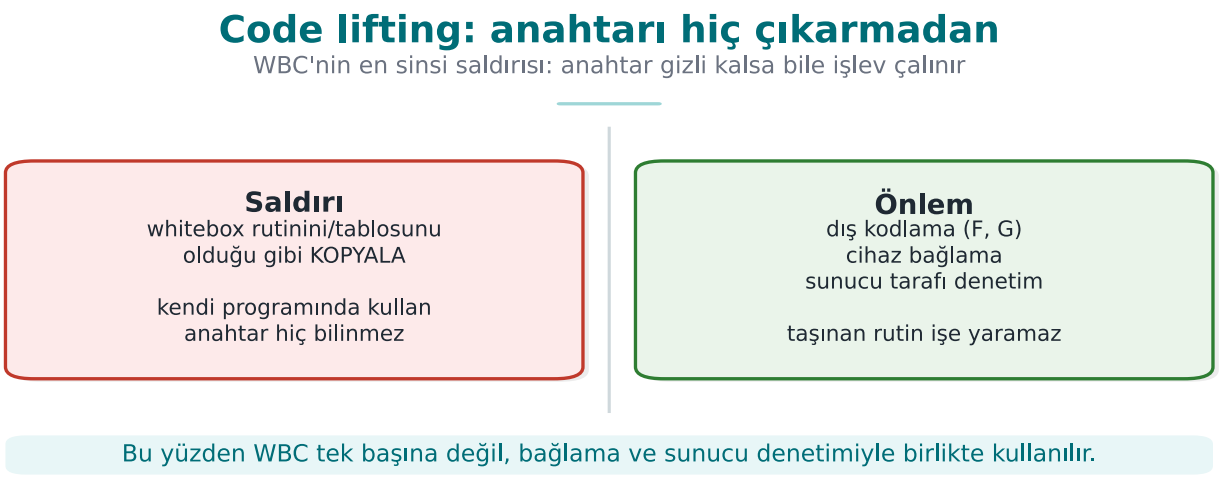
In one sentence

Black box = "you can't look inside". Grey box = "you can't look inside, but you can listen for leaks from the outside". White box = "you can see everything, and even change it". WBC is the **mathematical** answer to the third model.

2. Why is the problem so hard?

The fundamental tension in the white box is this: to perform encryption, the program **must be able to access** the key; but the attacker sees everything the program sees. So the key must, at some point, be somewhere the attacker can also see.

WBC's idea is to "bake" the key into the algorithm in such a way that it **never exists explicitly**: the key is embedded inside precomputed **lookup tables**, so that there is no byte sequence in memory of which you could say "this is the key".



This doesn't move confidentiality from cryptography to obscurity; it tries to spread the key across tables **through a mathematical transformation**. The goal is, again, a **cost rule** (as in Week 9): to make recovering the key from the tables much more expensive than storing it in plain form.

Why naive solutions collapse (rules)

To see why WBC is needed, you have to know the "easy" solutions that don't work, and the attacks that break them. Each one turns into a secure programming rule:

- **Rule — Embedding the key in a fixed array.** The most naive solution is to put the key into a `static const uint8_t key[16]` array. This is found within seconds with `strings` and an **entropy scan** (Shamir–van Someren's observation: cryptographic keys are high-entropy blocks and appear as such in the binary); it's then confirmed with a debugger. **Consequence:** a plain embedded key is not a key protection.
- **Rule — "Scrambling" the key with only XOR/whitening.** XOR-ing the key with a mask and storing it doesn't work either: the mask is also in the binary; if the program can undo it, so can the attacker.
- **The code-lifting threat.** Even if WBC buries the key in tables, the attacker can copy, as-is, the piece of code (tables + interpreter) that performs encryption/decryption **without ever extracting the key**, and use it in their own program. They steal its function without ever knowing the key. **Consequence:** it isn't enough for WBC to say "I hid the key" on its own; code lifting requires **device/version binding** (Section 4) as a counter.
- **Rule — Forgetting the "test/debug key".** A very common real-world mistake: during testing, for convenience, a developer puts a **fixed "test key"** in place of the real key (something like `static const uint8_t k[16] = {0x00, 0x01, ...}`, or plaintext in a configuration file), and then **forgets to move it out** for production, or dismisses it as "temporary anyway". **Consequence:** this is **exactly the same** vulnerability as Section 2's first rule (a plain embedded key) — except the "test-only" label doesn't **reduce** the real risk. An assessor (Week 12) scans test/debug paths with **the same rigour** as production code.

This week's main rule, up front

No published pure-software whitebox design has ever remained unbroken. So we will learn WBC not as **"magic that secures the key"**, but as **"a layer that delays key extraction and only makes sense together with the other layers"**. This is Week 9's obfuscation rules applied to cryptography.

Numeric micro-example: why doesn't "scrambling with only XOR" work?

Let's not leave the second rule above ("scrambling the key with only XOR/whitening doesn't work") hanging in the air; let's see it, **step by step**, with small but real bytes.

Setup. A developer decides that, instead of keeping the secret key byte `k = 0x3C` **plain** in the binary, they'll XOR it with a random mask and store that. Let the mask be `m = 0x5A`. The value stored in the code:

```
kodlanmis_k = k XOR m = 0x3C XOR 0x5A
```

Step 1 — Let's compute the XOR bit by bit.

```
0x3C = 0011 1100
0x5A = 0101 1010
       ----------- XOR
       = 0110 0110 = 0x66
```

So in the binary, `kodlanmis_k = 0x66`. The developer thinks "the key no longer looks plain, it's hidden".

Step 2 — The program itself must also contain the mask. At encryption time the program needs the real key; that is, somewhere it **must** perform this computation:

```
uint8_t k_gercek = kodlanmis_k ^ m;   /* m must also live somewhere in the binary! */
```

The constant `m = 0x5A` sits inside the binary too, **right next to** the constant `kodlanmis_k = 0x66` (both are static data).

Step 3 — What the attacker does. The attacker reads both bytes from the binary (`0x66` and `0x5A`) and performs the same XOR **themselves**:

```
k = kodlanmis_k XOR m = 0x66 XOR 0x5A
  = 0110 0110
  ^ 0101 1010
  -----------
  = 0011 1100 = 0x3C        ← KEY FOUND AGAIN
```

Conclusion. Masking changes the key's **bit pattern** in the binary but doesn't change the protection: if the program can undo the mask, so can an attacker who can read the same computation. This is exactly the same lesson as the limit of Week 9's **K-07 (static string/hard-coding)** rule — encoding alone gains nothing **if the decoding key sits right next to it**.

The rule this gives us

Carrying a secret value in the same binary **together with all the information needed to undo it** is not hiding — it only changes the **appearance**. The internal/external encodings we'll see in Section 3 don't fall into this trap, because the encoding bijections (`G` , `F`) are never stored anywhere as **an explicit constant**; they're spread into the code's own **structure** (how the tables connect to one another). We'll see this difference in the example we build by hand in Section 3.

Code lifting — a bit deeper

Above we defined code lifting: the attacker copies the tables and the interpreter that runs them **as-is**, without ever extracting the key, and uses them in another program. Let's make concrete, with an example, why this is a serious threat:

Conceptual scenario: code lifting

Suppose a payment application produces a token encrypted with whitebox AES to approve a transaction. The attacker, **without ever knowing the key**, copies the whitebox routine (the tables plus the code that calls it) **in its entirety** from the application and pastes it into their own program. Their own program can now also produce **valid tokens** — as if it knew the key. The attacker never "saw" the key at all; they merely **stole the function**.

The reason this is specific to WBC is this: in a normal program (key in a plain variable), the attacker can already **extract** the key and use it wherever they like; there's no need for code lifting. In WBC, the key **may be unextractable** (if the external encodings are strong), but the tables still form **a working machine** — and copying a working machine is, most of the time, **much easier** than solving its mathematics. Week 9's **K-06 (call and dependency hiding)** rule helps a little here (it makes the routine harder to find), but **isn't sufficient on its own** — once the routine is found, it can still be copied. The real solution is **device/ version binding**, which we'll see in Section 5: the tables only produce a meaningful result in a specific context (e.g. an input/output mixed with a device identifier).

Is WBC the same thing as Week 9's "code hardening"?

These two are often confused, because both work toward the goal of "making things harder for the attacker". But **what** they make harder is different:

	Week 9 — code hardening (K-01…K-08)	This week — WBC
What does it target?	Makes reading the program's **logic/control flow** harder	Hides the **key** mathematically
Typical technique	Opaque predicates, flattening, string encoding	Partial evaluation, internal/external encoding, mixing matrix
Strongest against what?	Static analysis (reading source/flow)	Directly searching for the key in memory/code
Weak against what?	Dynamic analysis (run and trace) can often get past it	Dynamic attacks such as DCA/DFA (Section 4)
Relationship to the key	Does **not** protect the key; only makes reading the code around it harder	Designed with the goal of protecting the key **itself**

Why are they used together? Because they complement each other: WBC buries the key in tables, and code hardening makes it harder to read **how** the code that calls these tables works (e.g. K-06 makes finding the whitebox routine harder, K-04 makes tracking which table is called when harder). The sentence we quoted from the source guide in Section 1 says exactly this: what protects WBC is being **together with code hardening and RASP**.

Cost balance: the attacker wins cheaply while the defender pays dearly

Let's close this section with a **cost** observation, because it's a pattern that will repeat in every section this week:

Side	Cost against a naive protection	Why
Attacker	Seconds–minutes	Entropy scanning is automatic, undoing XOR is one line of code
Developer (naive protection)	Nearly zero	A `static const` array or a single XOR is easy to add — and **that's exactly the problem**
Developer (WBC + layers)	High (design, table generation, testing, speed/size penalty)	A size/speed cost that reaches a hundredfold, as we'll see in Section 3

This asymmetry (**if the defender takes a cheap measure, the attacker also breaks it cheaply**) is the reason whitebox cryptography exists: the goal is to raise the attacker's cost as much as possible **for a cost the defender can accept** — never to bring it down to zero.

---

3. How does table-based WBC work? (Chow AES, conceptually)

The classic idea of WBC is Chow and colleagues' 2002 AES implementation. Its detailed mathematics (finite-field operations, matrix constructions) will not be asked in this course's exam; but you need to understand its **idea**, and especially its **cost** and its **limit**. We proceed step by step and conceptually.

Step 1 — Partial evaluation: "bake" the key into a table

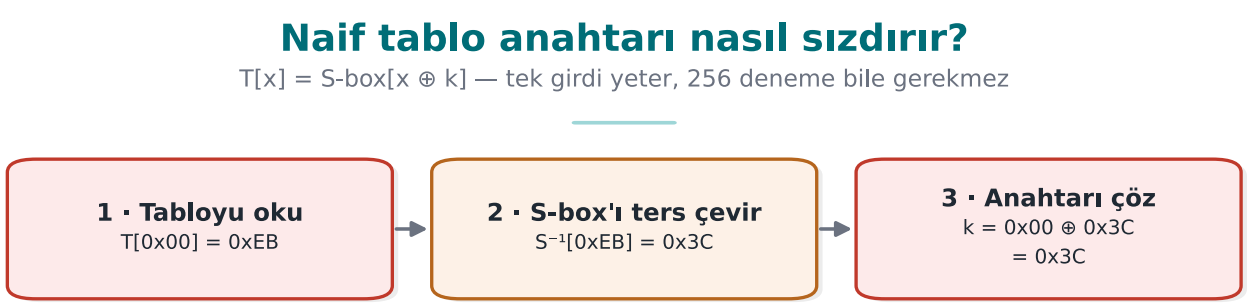
In one round, AES first XORs the state byte with the round key, then passes it through the S-box. For a fixed key, these two operations can be merged into **a single lookup table**:

```
Normal:   output = S-box[ x XOR k ]     (k is the round key's byte)
WBC:      T[x] = S-box[ x XOR k ]      → a 256-entry table, k is INSIDE the table
```

Now there's no variable called `k` in the code anymore; `k` is embedded in the table's contents. But this is **insecure on its own**: the attacker can compare the table against the S-box and recover `k` (the table is the S-box shifted by `x XOR k`). That's why tables get **encoded**.

Numeric micro-example: how does the key leak from the table? (end to end)

Let's not leave the sentence above ("the attacker can recover `k`") hanging; let's do it **step by step, with real numbers**. All we have is this: the publicly known **AES S-box**, and the **table** the attacker reads from the binary.



'Anahtar kodda görünmüyor' demek, 'anahtar korunuyor' demek DEĞİLDİR.

Setup. Let the secret key byte be `k = 0x3C` (the attacker **does not know** this). The developer has applied Step 1 and produced this table: `T[x] = S-box[x XOR k]`.

Step 1 — Read a single value from the table. The attacker looks at the entry `x = 0x00`:

```
T[0x00] = S-box[0x00 XOR 0x3C] = S-box[0x3C] = 0xEB
```

That is, the value they read in the binary: `T[0x00] = 0xEB`.

Step 2 — Invert the S-box. The S-box is public and **one-to-one**; its inverse is also known:

```
S-box⁻¹[0xEB] = 0x3C        (because S-box[0x3C] = 0xEB)
```

Step 3 — Solve for the key. Since, by definition, `S-box⁻¹[T[x]] = x XOR k`:

```
x XOR k = 0x3C
0x00 XOR k = 0x3C
k = 0x3C XOR 0x00 = 0x3C        ← KEY FOUND
```

Conclusion. The attacker found the key with **a single table entry**, without even trying all 256 possibilities. Saying "the key doesn't appear in the code" **is not** the same as saying "the key is protected".

The rule this gives us

Partial evaluation **alone** does not hide the key; because the table is the known S-box **shifted** by `k`, and this shift is reversible. What makes the table secure is the **internal encoding** from Step 3.

So what does encoding change? If we wrap the table in a hidden bijection `G`: `T'[x] = G( S-box[x XOR k] )`. The attacker can no longer compute `S-box⁻¹[T'[x]]`, because they would first need to remove `G`, but they don't know `G`. Step 2 breaks, so Step 3 doesn't work either.

See this for yourself in the demo

The `code/week-11/01-oyuncak-tablo` demo produces exactly these two tables: on the **naive** table, the three steps above find the key (`0x3C`); on the **encoded** table, the same steps fail. The run commands are in the "Run the demo yourself" box at the top of this page.

A fully hand-built example: a 4-element "toy S-box"

The real AES S-box has 256 entries; scanning **all of it** by hand is not practical (above we made do with a single entry). Now let's shrink the task down to a **4-element** toy S-box, and carry out a full attack where all **four entries** of the table are computed by hand, start to finish. The idea is exactly the same; it's just small enough to follow by hand.

Setup. Our inputs are 2 bits: `x ∈ {0, 1, 2, 3}` (binary: `00, 01, 10, 11`). Our toy S-box is a fixed table that scrambles these four values in a **one-to-one, onto** fashion (a permutation, Section 0):

x	0 (00)	1 (01)	2 (10)	3 (11)
S[x]	3 (11)	1 (01)	0 (00)	2 (10)

Check: the outputs are `{3,1,0,2}` — that is, the whole set `{0,1,2,3}`, each exactly once. So S is a bijection (Section 0), just like the real AES S-box.

Let our secret key (which the attacker **doesn't know**) be `k = 2` (binary: `10`).

Step A — Let's build the table `T[x] = S[x XOR k]` by hand for all four inputs.

```
x=0:  0 XOR 2 = 10 (2)   →  S[2] = 0    →  T[0] = 0
x=1:  1 XOR 2 = 11 (3)   →  S[3] = 2    →  T[1] = 2
x=2:  2 XOR 2 = 00 (0)   →  S[0] = 3    →  T[2] = 3
x=3:  3 XOR 2 = 01 (1)   →  S[1] = 1    →  T[3] = 1
```

Result: **the exact table the attacker reads in the binary** is `T = [0, 2, 3, 1]` (i.e. `T[0]=0, T[1]=2, T[2]=3, T[3]=1`).

Step B — Let's derive the inverse of the S-box. Since S is public, its inverse can also be computed: we read the question "which input produced which output?" backwards from the S table (`S[0]=3` means `S⁻¹[3]=0`, and so on):

y	0	1	2	3
$S^{-1}[y]$	2	1	3	0

Step C — Let the attacker verify the key across all FOUR inputs. The rule is exactly the same as Step 3 in the real AES example: `S⁻¹[T[x]] = x XOR k`, hence `k = x XOR S⁻¹[T[x]]`.

```
x=0:  T[0]=0  →  S⁻¹[0]=2   →  k = 0 XOR 2 = 2
x=1:  T[1]=2  →  S⁻¹[2]=3   →  k = 1 XOR 3 = 2      (01 XOR 11 = 10)
x=2:  T[2]=3  →  S⁻¹[3]=0   →  k = 2 XOR 0 = 2
x=3:  T[3]=1  →  S⁻¹[1]=1   →  k = 3 XOR 1 = 2      (11 XOR 01 = 10)
```

All four inputs converge on the same key (`k = 2`). This is the small-scale, exact counterpart of scanning all 256 entries in real AES: the attacker could have made do with a single entry (as we did in the real AES example above), but **consistency** (all inputs converging on the same `k`) additionally confirms the result.

The rule this gives us (again, verified at small scale)

Size doesn't matter — the 4-entry toy table and the 256-entry real AES T-box both carry **the same weakness**: if a table has the form `S[x XOR k]`, `k` is recovered **algebraically** by comparing it against the public `S`. The table's size doesn't slow the attack; it only changes how many entries need to be checked.

Once internal encoding is added: why does the same attack collapse?

Now let's apply, again by hand, the **internal encoding** we mentioned (in Step 3, which we'll get to shortly — not the next subsection) on the same toy example. We wrap the table with an unknown bijection `G`: `T'[x] = G(T[x])`.

Let's define `G` (the attacker doesn't know it):

y	0	1	2	3
G[y]	1	3	0	2

Check: the outputs are `{1,3,0,2}` — again the whole of `{0,1,2,3}`, so `G` is also a bijection.

Let's build the table `T'[x] = G(T[x])` (applying `G` on top of the earlier table `T = [0, 2, 3, 1]`):

```
T'[0] = G(T[0]) = G(0) = 1
T'[1] = G(T[1]) = G(2) = 0
T'[2] = G(T[2]) = G(3) = 2
T'[3] = G(T[3]) = G(1) = 3
```

Result: `T' = [1, 0, 2, 3]`. **This** is now what the attacker sees in the binary; not `T`.

The attacker tries the same three steps (A–C) again — they still **don't know** that `G` exists, so they proceed again under the assumption `S⁻¹[T'[x]] = x XOR k`:

```
x=0:  T'[0]=1  →  S⁻¹[1]=1   →  k_tahmin = 0 XOR 1 = 1
x=1:  T'[1]=0  →  S⁻¹[0]=2   →  k_tahmin = 1 XOR 2 = 3      (01 XOR 10 = 11)
x=2:  T'[2]=2  →  S⁻¹[2]=3   →  k_tahmin = 2 XOR 3 = 1      (10 XOR 11 = 01)
x=3:  T'[3]=3  →  S⁻¹[3]=0   →  k_tahmin = 3 XOR 0 = 3
```

Result: `1, 3, 1, 3` — the four inputs give FOUR DIFFERENT (and mutually contradictory) "keys", and not one of them even equals the real key, `2`. The "all inputs converge on the same key" consistency test from Step C **fails outright** here — the attacker realises the table they have is no longer of the simple form `S[x XOR k]`, but since they don't know `G`, they **cannot extract** `k` this way.

Why didn't it work? (the mechanism)

The attack's Step B (`S⁻¹[T[x]] = x XOR k`) is only valid if `T[x]` is **actually** equal to `S[x XOR k]`. When `T'[x] = G(S[x XOR k])`, `S⁻¹[T'[x]]` is no longer `x XOR k`, but the meaningless composition `S⁻¹(G(S[x XOR k]))`. To untangle this, the attacker would first need to know or find `G` (or its inverse) — and `G` never sits anywhere as an explicit constant (unlike the "scrambling with only XOR" example in Section 2); it is embedded in the table's **structure**. In real Chow AES this idea extends to 256-entry tables, many chained `G`s, and, on top of that, mixing matrices (Step 4).

See this for yourself in the demo (again)

The **encoded table** in the `code/week-11/01-oyuncak-tablo` demo is the real-AES-scale version of the `T'` logic above: the same three-step attack is attempted, and it produces inconsistent/wrong results — just like the `1, 3, 1, 3` we saw by hand here.

Let's see, with a small table, how DCA collects its traces

In Section 0 we defined the terms **execution trace**, **instrumentation**, and **correlation**; the **DCA (Differential Computation Analysis)** attack, which we'll cover in full in Section 4, combines them. Here let's see the attack's **data-collection mechanism**, on the same toy table, with a small tracing table. The goal isn't to carry out the attack, but to show **why it needs a strategy different from the algebraic Steps A–C**.

What would happen without encoding first? (the basic idea) DCA's logic is this: a key hypothesis (`k_h`) is chosen, and for every plaintext byte `p`, **a prediction is made of what the real computation would produce** (`S[p XOR k_h]`); this prediction is then compared against the **actually observed** intermediate value. Without encoding (i.e. while the `T` table is directly visible), this comparison can be done with an **exact match**:

Trial (p)	Observed intermediate value (`T[p]`)	Prediction, `k_h = 2` (correct) → `S[p⊕2]`	Match?	Prediction, `k_h = 0` (wrong) → `S[p⊕0]`	Match?
0	0	0	✅	3	❌
1	2	2	✅	1	❌
2	3	3	✅	0	❌
3	1	1	✅	2	❌

The correct hypothesis (`k_h=2`) matches **4/4** (by definition — because `T[p]` is exactly `S[p⊕2]`); the wrong hypothesis (`k_h=0`) matches **0/4**. Without encoding, telling the correct key apart from the wrong ones is this easy.

Now let's put the encoding (`T'`) back — and test the same two hypotheses:

Trial (p)	Observed intermediate value (`T'[p]`)	Prediction, `k_h = 2` (correct)	Match?	Prediction, `k_h = 0` (wrong)	Match?
0	1	0	❌	3	❌
1	0	2	❌	1	❌
2	2	3	❌	0	❌
3	3	1	❌	2	❌

Striking result: with the encoding in place, **the correct hypothesis also gets 0/4, and the wrong hypothesis also gets 0/4** — because of the hidden `G` , the exact-match test can no longer **tell the correct one from the wrong one**. This is proof both of why the statement "internal encoding stops the attack" is true, and of why **DCA is not a simple exact-match test**.

How does DCA actually proceed? (conceptual, details out of scope)

Real DCA doesn't settle for one trace or an exact match: it collects traces for **hundreds to thousands** of different plaintexts, and for each key hypothesis it looks for a **statistical correlation** (Section 0) between the predicted value and the observed (encoded) value — this is exactly what Bos, Hubain, Michiels, and Teuwen's 2016 study (Section 4) does. Because the encoding `G` stays **the same** (fixed) on every run, subtle statistical patterns that accumulate across many traces can reveal a signal invisible in any single trace. The mathematics of this statistic is outside this week's scope; the idea you need to carry is this: **"no instant match" does not mean "the attack is impossible"** — the defence's real strength lies in the countermeasures we'll see in Section 4.

Step 2 — Merge and expand the tables (T-box + MixColumns)

Instead of a single byte mapping, the S-box is merged with AES's linear mixing step (MixColumns); this yields larger tables (T-boxes) that produce a 32-bit output from an 8-bit input. The XOR operations between bytes are also turned into small (4-bit) **XOR tables**. This way, the entire round turns into a **network of lookup tables**; nowhere does an explicit arithmetic key operation appear.

Let's see the idea at toy scale: merging two operations into a single table

Real T-boxes (S-box + MixColumns) carry an 8-bit input and a 32-bit output; following them by hand isn't practical. But we can demonstrate the idea of **"merging two separate operations into one table"** with a small extension on top of our toy S-box.

Normally, in a non-whitebox implementation, **two separate steps** run for input `x` :

```
Step 1: y = S[x XOR k]        (our toy S-box, with the key)
Step 2: z = y XOR c            (c: a fixed "mixing" value coming from the next round, e.g. c = 1)
```

If these two steps run as **two separate instructions**, the attacker can **separately** observe the intermediate value `y` in a debugger — each step is its own observation point on its own. The T-box idea is to **pre-bake** these two steps into **a single table**:

```
Merged table: U[x] = S[x XOR k] XOR c
```

Let's take `k = 2`, `c = 1` and build `U[x]` for the four inputs (recall our earlier table `T[x] = S[x XOR k]`, `[0,2,3,1]` — we're just adding `XOR 1` at the end):

```
U[0] = T[0] XOR 1 = 0 XOR 1 = 1
U[1] = T[1] XOR 1 = 2 XOR 1 = 3      (10 XOR 01 = 11)
U[2] = T[2] XOR 1 = 3 XOR 1 = 2      (11 XOR 01 = 10)
U[3] = T[3] XOR 1 = 1 XOR 1 = 0
```

Result: `U = [1, 3, 2, 0]`. Now neither an `S[...]` call nor a separate `XOR c` instruction appears in the code — both are **collapsed** into a single table read (`U[x]`). The **number of intermediate steps** the attacker can see in the debugger goes down: where there used to be two observation points (`y` after Step 1, `z` after Step 2), there is now only **one** (`U[x]`).

> **Is this enough on its own?**
>
> No — `U[x]` still carries the same weakness as `S[x XOR k]` (only a fixed `XOR c` has been added at the end); the attacker can compute `U[x] XOR c` and apply the Step A–C attack **exactly as before** (because the constant `c`, just like the mask in Section 2, may be known or guessable somewhere). The real value of merging T-boxes is that it reduces the number of observation points; the **real protection** comes from the internal encodings of Step 3, which we'll see shortly. That's why, in Chow's design, T-box merging is used **not alone, but together with internal encoding**.

In real AES this idea is applied at a much larger scale: the **entire** round (16-byte S-box + MixColumns + round key addition) turns into a network of interconnected T-boxes; the number of points where the attacker could watch intermediate steps one by one, which was in the dozens in plain code, **drops sharply** once it's baked into tables — but it doesn't reach zero, because the transitions between tables are still observable. Full protection comes from the encodings of Steps 3–5.

Step 3 — Internal encodings

A hidden bijection (a one-to-one, onto mapping) is applied to the output of every table, and its inverse is applied at the input of the next table. The encodings of consecutive tables cancel each other out; the result is correct, but the **intermediate values are scrambled**. This aims to stop an attacker examining a single table in isolation.

Let's verify the "the result is correct" claim by hand

Let's not leave the sentence above ("the encodings cancel each other out, the result is correct") as a mere claim; let's **prove it** with our toy example. Recall the tables we built earlier: `T = [0, 2, 3, 1]` (the real intermediate values) and `T' = G(T) = [1, 0, 2, 3]` (Table 1's **encoded output**, what the attacker sees).

The next table in the chain (Table 2) takes `T'[x]` as its input. To produce the correct result, it must **first** undo the encoding, i.e. apply `G⁻¹`. Let's derive `G⁻¹` (the inverse of G — which output came from which input):

y	0	1	2	3
G⁻¹[y]	2	0	3	1

Let's compute `G⁻¹(T'[x])` for all four inputs and compare it against `T[x]`:

```
x=0:  G⁻¹(T'[0]) = G⁻¹(1) = 0   ==  T[0] = 0   ✓
x=1:  G⁻¹(T'[1]) = G⁻¹(0) = 2   ==  T[1] = 2   ✓
x=2:  G⁻¹(T'[2]) = G⁻¹(2) = 3   ==  T[2] = 3   ✓
x=3:  G⁻¹(T'[3]) = G⁻¹(3) = 1   ==  T[3] = 1   ✓
```

`G⁻¹(G(T[x])) = T[x]` is confirmed for all four of the four inputs. This is the full proof of Step 3's claim: as soon as Table 2 receives its input, it applies `G⁻¹` to recover the **real** intermediate value (`T[x]`) and continues the computation **from the right place** — the result comes out correct, as if the encoding never existed. But this recovery happens **hidden, inside Table 2**; if the attacker only sees `T'[x]` (Table 1's output) in isolation, they **cannot** perform this recovery because they don't know `G` — just as we saw in the "Once internal encoding is added" example at the start of Section 3.

✓ **Summary of the chaining idea**

There is an "encode → decode" step between every pair of tables: the outgoing table encodes with `G`, the incoming table decodes with `G⁻¹`. An attacker looking at the **middle** of the chain (someone who sees only `T'[x]` and can't reach `T[x]`) cannot see the correct result; but the real program, running from **start to finish** through the chain, reaches the **correct** result by undoing the encoding at every step. In Chow AES this idea is repeated over dozens of tables, each time with a **different** bijection.

Step 4 — Mixing bijections

In addition to internal encodings, linear mixing matrices over GF(2) (e.g. 8×8 and 32×32) are added; these **spread** information across tables, so that information leaking from a single table isn't meaningful on its own. Chow's table types (types I–IV in the literature) are the varieties of these encoded and mixed tables.

A tiny, hand-solvable GF(2) matrix example

Let's make the ideas of **linear transformation** and **non-singular matrix**, which we defined in Section 0, concrete. Consider this tiny matrix, which mixes a 2-bit value `(v₁, v₀)` (e.g. `v₁v₀ = 11` in binary is the number 3):

```
M = | 1 1 |     new_v1 = v1 XOR v0
    | 0 1 |     new_v0 = v0
```

This is a **linear transformation** over GF(2): each new bit is a selected XOR of the old bits (recall the GF(2) definition from Section 0: addition = XOR).

Example input: `v = 11` (i.e. `v₁=1, v₀=1`, decimal 3).

```
new_v1 = v1 XOR v0 = 1 XOR 1 = 0
new_v0 = v0        = 1
```

Result: `M·v = 01` (decimal 1). Someone looking at this in isolation would say "the number 3 turned into 1", but without knowing **which matrix** did it, they cannot reverse it.

Is this transformation really invertible (non-singular)? Let's test it by applying `M` a **second time** to the same input — if we return to the original value, we confirm that `M` is its own inverse (an involution):

```
input: 01  (new_v1=0, new_v0=1)
new_v1' = 0 XOR 1 = 1
new_v0' = 1
result: 11  ← we're back to the original value (3)
```

Confirmed: `M` is a non-singular matrix (it has an inverse — here, coincidentally, itself), meaning information is **not lost**, only **scrambled**. The real mixing matrices in Chow AES are 8×8 and 32×32 in size and chosen at random, but the idea is exactly the same: each is chosen to be non-singular so that decryption can **fully undo it**; for the attacker, meanwhile, the **direct** relationship between the values in a single table and the real AES intermediate values becomes invisible, because the values have been **spread** across the XOR of many bits.

> **Why isn't a single bijection (G) enough, and why is a matrix added too?**
>
> A single table-output bijection like `G` from the previous subsection only scrambles **that table's own output**. A mixing matrix, on the other hand, spreads information **across multiple bits/bytes**; this makes it even harder for an attacker to find a meaningful pattern by looking at just **one** table (without seeing the others). The two are used together — neither alone is sufficient; this is the counterpart, here, of Week 9's lesson that "a single obfuscation rule isn't enough, layer them".

Step 5 — External encodings: F and G

At the outermost layer, the entire cipher is wrapped in two hidden bijections. As the source guide describes it, the whitebox AES operation is encapsulated with two random non-singular matrices called **F and G** (rank 128, a "random bijection"). So the network no longer computes plain AES, but `G ∘ AES ∘ F⁻¹`:

Dış kodlama: en güçlü koruma, en büyük sınır

çıktı artık STANDART AES değildir

Karşı taraf F ve G'yi bilmeden sonuç anlamsız → yalnız iki ucu da sizin olduğu protokolde.

```
Input --F⁻¹--> [ network of encoded AES tables ] --G--> Output
```

This step is both the strongest protection and the biggest limitation:

- **Strength:** external encoding makes many algebraic/statistical attacks (BGE, DCA variants) that work by looking at the tables alone much harder; because the input/output the attacker sees is not standard AES's input/output.
- **Limitation:** `G ∘ AES ∘ F⁻¹` **is not standard AES.** Without the other side (a server or another endpoint) knowing `F` and `G` and undoing them, the result is meaningless. That's why externally encoded WBC can only be used in a protocol where **you control both ends**; it may not be usable in a protocol such as EMV that expects standard AES.

External encoding in the toy example: why does it make code lifting harder?

Let's connect external encoding to the code-lifting threat we defined in Section 2. Let's add an input external encoding `F` to our toy whitebox; the attacker now feeds the function `F(x)`, not `x` **directly**:

x	0	1	2	3
F[x]	2	0	3	1

The full network now works in the form `output = G(T[F⁻¹(input)])` (the input is first "decoded" with `F⁻¹`, the table is applied, the output is "encoded" with `G`). **The application calling it must** apply its own `F` encoding before sending the input, and undo its own `G⁻¹` encoding after receiving the output — that is, the routine now works only under a contract **known solely to this application**.

Now let's look through the eyes of the attacker from Sections 2/6: even if the attacker copies the whitebox table network (`T`, `G`, and indirectly the effect of `F⁻¹`) as-is into their own program, if their own program **doesn't encode the input with `F` and doesn't decode the output with `G⁻¹`**, the result they get is a **meaningless** byte sequence — not a ciphertext/token in the form the original application expects. The attacker can steal the function but **cannot use it correctly**, because without knowing what F and G are, they would also have to copy them and apply them **in the right order** in their own program — and these are usually tightly woven into the rest of the application (through the device/version binding in Sections ⅘).

Short summary

Internal encoding (Step 3) and the mixing matrix (Step 4) wear out an attacker looking at a single table; external encoding (Step 5) wears out an attacker trying to carry **the whole routine** into a different context (code lifting). The two work against **different threats** — which is why we covered two separate scenarios in Section 6.

Cost: WBC is not free

Let's close the concept with a rule: WBC's cost is high. Chow AES's typical order of magnitude (from the literature):

Metric	Approximate value (order of magnitude)
Table size	Hundreds of KB (e.g. ~770 KB) — for comparison: a standard AES key is 16–32 bytes
Table lookups per round	Thousands of lookups (e.g. ~3000)
Speed	Tens of times slower than standard AES (e.g. ~50×)

A sense of scale for the exam

You should be able to estimate a table's size: e.g. 288 tables × 1 KB ≈ 294,912 bytes. The goal isn't an exact number, but seeing the **order of magnitude** and why this is exactly why it's used only for selected, small, critical operations. This is the same cost logic as the virtualisation rule (K-10) from Week 9.

Exercise: make your own order-of-magnitude estimate

To reinforce this sense of scale, let's compute the same idea once more **by hand** — this time with different, simple numbers. Suppose a developer uses a total of **160 tables** in a whitebox AES implementation, and each table takes up an average of **900 bytes**:

```
Total size ≈ 160 tables × 900 bytes/table = 144,000 bytes ≈ 140.6 KB
```

For comparison: in a standard (non-whitebox) AES implementation, the only thing that needs to be stored is a **16–32-byte key**. So in this example, the whitebox tables take up roughly **4,500–9,000 times** more space than the plain key (`144,000 ÷ 32 = 4,500` , `144,000 ÷ 16 = 9,000`).

What doesn't this calculation show?

This is not the **actual** exact size of real Chow AES — it's just an exercise showing how quickly the product of "number of tables × table size" **grows**. What's expected of you in the exam is not a memorised number, but the ability to **set up this product** given a table count/size.

Closing Section 3: let's sum up the five steps in one sentence

Let's line up, once more, the five steps we built with our toy S-box, as a single chain: **bake the key into a table (Step 1) → merge tables and reduce observation points (Step 2) → wrap every pair of tables in a hidden bijection (Step 3, which we**

proved by hand: `G⁻¹(G(T[x])) = T[x]` **) → spread information with matrices (Step 4, which we saw with the GF(2) example) → seal it with an outer envelope that binds the whole routine to the application (Step 5).** Each step closes the point where the **previous step alone was insufficient** — that's why all five are used **together**, and none of them alone is "WBC".

> ✓ **Section 3 in one sentence**
>
> Whitebox AES doesn't **store** the key — it spreads the key **into the structure of the tables** in a way that makes recovering it (mathematically) expensive. Section 4 will describe how this "expensive" has, over time, become **cheaper** (i.e. broken).

4. Attack history: why "all of them were broken" is a rule

The most honest way to position WBC correctly is to look at its history. We give the chronology below **not to teach attacks**, but to answer the question "how much can I trust this protection?" Every row is a countermeasure takeaway for the defender.

Year	Milestone	Takeaway for the defender
2002	Chow et al.'s WB-AES and WB-DES designs	The birth of the WBC idea; the table-based approach
2004	The **BGE attack** (Billet–Gilbert–Ech-Chatbi): Chow AES broken in practical time (~2^{30})	Internal encodings alone aren't enough
2007	Goubin/Wyseur et al.: WB-DES and its externally-encoded variants broken	DES-based WBC is abandoned
2010s	Karroumi's "dual AES" strengthening also broken (~2^{22})	"More tables" doesn't mean more strength
2015	**DFA** ("Unboxing the White-Box", Black Hat EU): fault injection extracts the key without knowing the design's internal mathematics	General (grey-box) attacks apply to WBC too
2016	**DCA** (Differential Computation Analysis, CHES): DPA-like statistics applied to execution traces; ignores internal encodings	Classic side-channel defences are also needed for WBC
2017–2024	**WhibOx** competitions: **all** designs submitted by academia and industry were broken	As of today: no published pure-software WBC remains unbroken

Why does the 2007 row matter? At first glance it might be dismissed with "DES got broken, we already use AES". But the real lesson is different: when AES-based whitebox was broken in 2004, part of the field thought "maybe the problem is in AES's own structure, let's try a different cipher (DES)". The breaking, in 2007, of DES-based designs (and their variants with external encoding added) too showed that: **the problem wasn't specific to AES.** The weakness was not in the identity of the cipher used, but in **the mathematical structure of the internal encodings** — just as in Karroumi's "more tables" attempt, **changing the cipher** didn't fix the root cause either.

Why is "more tables, more security" wrong? Let's unpack the takeaway in Karroumi's row a bit, because it may seem counter-intuitive. The "dual AES" idea was to run two independent AES computations **in sequence or side by side**, **doubling** the whitebox tables — the logic was: "the attacker now has to solve twice the tables, so it's twice as hard." But security is about the **mathematical structure** of the encodings, not the **number** of tables. If the additional tables also use **the same kind** of (affine) encodings, a BGE-like algorithm solves them **the same way** too — it just needs a bit more computation ($\sim 2^{22}$), which doesn't make the design **unbreakable**. This is a counterexample to Section 3's cost discussion: here, **raising the cost** (more tables) **did not raise the security**, because what was increased was not the point where the attacker is actually constrained (the structure of the encoding).

The rule this gives us

The right way to strengthen a protection is to target **what the attack actually relies on**, not to **increase its quantity**. In the WBC context this means random/non-affine encodings, external encoding, and layered defence (the "Countermeasures" part of Section 4) — not just adding "more tables".

Why do DCA and DFA matter? Because they don't try to solve the design's hidden mathematics. DCA collects a trace of the intermediate values produced while running, and extracts the key with statistical correlation, just as in a hardware power analysis (DPA); internal encodings mostly don't disturb this statistic. DFA, meanwhile, **injects a fault** into a value at

runtime and computes the key back from the corrupted output. Both work "without opening the box" — which is why they also hit new designs.

The BGE attack: how did the first break happen? (conceptual)

Looking at it today, Chow's 2002 design might look "solid": every table is wrapped in a hidden bijection (like `G` in Section 3), and an attacker examining the tables one by one finds values that look **random**. BGE's (2004) idea was this: the internal encodings Chow used were **not random bijections**, but mathematically more structured (**linear + constant**, i.e. **affine**) transformations. When a transformation is affine, the **algebraic relationships** between neighbouring tables turn into a solvable system of equations; BGE recovered the encodings **one by one** with an algorithm that solves this system (called an "affine equivalence algorithm" in the literature) — it got past the exact point we called, in Section 3, "Step B collapses because the attacker doesn't know `G`", by exploiting `G`'s special (affine) structure.

Doesn't this contradict our toy example?

No. In Section 3 we chose the toy `G` as a **random permutation**; an attacker could only find it by brute force (trying the 4 possible permutations). What BGE showed was that the encodings in the **real Chow design** were not random but **specially structured (affine)**, and that this structure was **solvable** without needing brute force. Lesson: a protection "looking mathematically complex" does not make it **unbreakable**; fragility usually comes from the structure being **predictable** (here: affine).

DFA: how does fault injection turn into an attack? (conceptual, the software form of classic DFA)

Recall step 5 of the attacker's session in Section 1: the white-box attacker **can modify** an intermediate value at runtime. Classic (hardware) DFA is a long-known family of techniques against AES; the 2015 study in the whitebox context ("Unboxing the White-Box") carries the same idea into **software**. Its step-by-step logic:

1. **Reference run.** The attacker has a plaintext of their choosing encrypted, and records the **correct** ciphertext (no fault).
2. **Faulty run.** They have the same plaintext encrypted again, but this time, at runtime (using the "partial execution/tampering" capability from Section 1), they **deliberately inject a fault** into an intermediate value at a specific round (e.g. flipping a byte). They record the **faulty** ciphertext.
3. **Taking the difference.** The correct and faulty ciphertexts are compared. The difference between them depends on the **intermediate value at the point where the fault was injected, and hence on the key material used at that point** — the effect of the fault is not random; because of the **non-linear structure** of AES's final rounds, it is a difference that **carries information about the key**.
4. **Repeat and narrow down.** Steps 1–3 are repeated with different fault points/values; each repetition **narrows down** the possible key candidates, eventually converging on a single key.

Why is this a problem specific to whitebox? Because DFA **doesn't care at all** about the design's internal encodings (`G`, `F`, mixing matrices) — it only cares about **the difference between input/output pairs**. All the effort we put in during Section 3 (internal encoding, mixing matrix, external encoding) does **not eliminate** the "the difference carries information about the key" relationship in DFA's step 3 — because even though the encodings stay fixed, AES's own mathematics (the S-box's non-linearity) is still there.

Common mistake → consequence → rule

Mistake: the assumption "WBC was resistant to attack when it was published, so it's resistant today too" — i.e. treating a protection's strength **at birth** as **permanent**.

Consequence: as the WhibOx competitions (2017–2024) showed, **all** pure-software WB-AES designs were eventually broken; BGE brought down the 2002 design, and DCA/DFA brought down the next generation of countermeasures one after another. "Not broken today" and "unbreakable" are **not the same thing**.

Rule: design WBC like **a defence with an expiry date**: with a key-rotation schedule, with monitoring/ telemetry (is there a sign of attack?), and with a plan for "this design will fall one day too — what do we do then?" This is Week 10's crypto-period idea applied to whitebox.

WhibOx: why does an independent trial matter?

WhibOx is a series of competitions in which academics and companies submit their own WB-AES designs **publicly**, and researchers from all over the world try to break them. Here's why this matters: a design that a company's **own internal team** tested and says "wasn't broken" is **not at the same level of trust** as a design placed in front of **independent, numerous, and motivated** attackers that says "wasn't broken". WhibOx provides the latter. This is the whitebox counterpart of the "independent evaluation" idea we'll see in Week 12.

Three commonly confused abbreviations: BGE, DCA, DFA

With weak background knowledge it's very easy to mix up these three abbreviations — all three get mentioned under the heading "breaking whitebox". The table below gives **the one question that separates them**:

Attack	What does it do? (one sentence)	Static or dynamic? (Section 0)	Distinguishing question
BGE (2004)	Solves the **affine** (linear+constant) structure of internal encodings with an algebraic algorithm	**Static** — only reads the binary, doesn't run it	"Was the key found by **solving equations**?"
DCA (2016)	Applies DPA-like statistical correlation to execution **traces** (Section 0)	**Dynamic** — runs the program many times and traces it	"Was the key extracted from **the statistics of many traces**?"
DFA (2015)	**Injects a fault** at runtime and extracts the key from the correct/faulty output difference	**Dynamic** — runs the program **and corrupts it**	"Was the key extracted from **the effect of a deliberate fault**?"

The one-sentence distinguisher

BGE = mathematics (solve equations); DCA = statistics (compare many traces); DFA = sabotage (inject a fault, measure the difference). All three target whitebox, but **none of them uses the same method** — which is why no single row in Section 4's countermeasure table closed off all three at once.

Countermeasures (defence rules)

Research didn't stop; countermeasures against DCA/DFA were developed. But none of them say "now it's secure"; all of them **raise the cost**:

- **Static randomness and non-linear masking** (e.g. Biryukov–Udovenko 2018): tries to break the linear correlation that DCA relies on.
- **External encoding (F, G):** makes DCA/DFA/BGE harder — but with the limitation we saw: it breaks standard AES compatibility.
- **Layered defence:** Week 9's obfuscation rules and Week 6's RASP (anti-debug, tamper and integrity checking) are placed around WBC; the goal is to make it harder for the attacker to collect traces (DCA) and inject faults (DFA).
- **Key rotation and short lifetime:** if the key is rotated often, the value of extracting any one key drops (Week 10, crypto-period).
- **Device/version binding:** against code lifting; the tables are only meaningful on a specific device/version.
- **Server-side risk auditing:** whatever the protection at the endpoint, evaluating the transaction separately on the server (rate limiting, anomaly detection, ATC/counter checking) is the last and most reliable defence.
- **Moving to a hardware root:** if possible, never put the key in software at all — TEE, secure element (SE), HSM (Section 5).

Which countermeasure makes which attack harder? (summary table)

Let's match the list above against the attacks — a countermeasure doesn't stop "everything", it makes a **specific** class of attack harder:

Countermeasure	Makes BGE harder?	Makes DCA harder?	Makes DFA harder?	Stops code lifting?
Non-linear masking	Yes (breaks the affine structure)	Partially	No	No
External encoding (F, G)	Yes	Partially	Partially	Partially (if it binds to the application)
RASP / anti-debug (Week 6)	No (directly)	Yes (makes trace collection harder)	Yes (makes fault injection harder)	No
Key rotation	No	No	No	No — but **shortens its impact**
Device/version binding	No	No	No	**Yes**
Server-side risk auditing	No	No	No	Indirect (catches anomalous use)
Moving to a hardware root (TEE/SE/HSM)	Unquestionably — **removes the problem entirely**	Same	Same	Same

The lesson from this table

No single countermeasure row paints **every column** green. So defence is always **the sum of multiple rows** — just as we learned in Week 9 that a single obfuscation rule (K-01…K-08) isn't enough, and that all of them need to be used together.

The attack's cost from the attacker's side: what does each technique require?

In Section 2 we talked about the defender's cost; now let's build the same table **from the attacker's side**. When judging "how effective" a countermeasure is, you also have to account for the **level of access and skill the attack requires** — because the goal of a defence isn't to make the attack impossible, but to make it **unprofitable for most attackers** by **raising the required skill and effort**.

Technique	Access required	Skill required	Can it be automated?
Entropy scan (Section 2)	Read-only access to the binary (static)	Low — with off-the-shelf tools	Yes, easily
Algebraic table attack (Section 3, Steps A–C)	Read-only access to the binary (static)	Medium — understanding S-box algebra	Yes
BGE (affine equivalence)	Read-only access to the binary (static)	High — needs a specialised algorithm	Partially (specific to certain designs)
DCA	**Running** the program + instrumentation (dynamic)	Medium-high — statistical knowledge	Yes, largely
DFA	Running the program + **modifying** it (dynamic, tampering)	High — finding fault-injection points	Partially
Code lifting	Copying the binary (static, lowest effort)	Low — doesn't even require knowledge of the key	Yes

What does this table teach?

Notice: the **cheapest** attack (code lifting) doesn't require the most advanced cryptanalytic knowledge — just copying. That's why **both** of the two scenarios in Section 6 have a place in this lecture: a system can be hardened against BGE/DCA/DFA alone and left **defenceless** against code lifting. When designing a defence you also need to ask "which attack is **cheapest**?", not only "which attack is most **impressive**?"

Main rule (again): WBC is a layer, not a solution

Recall the source guide's opening sentence: what protects the application against the attacker's capabilities is WBC **together with code hardening and RASP**. Presenting WBC on its own as "I've secured the key" is the most serious assessment mistake in this lecture. The correct statement is: "I'm delaying key extraction by this much; my real assurance comes from key rotation, device binding, and server-side risk auditing."

5. WBC's place in layered defence

Let's clarify, as a secure programming rule, where to place WBC. The table below gives the **strength/cost** ordering of the options for protecting a key:

Option	Where's the key?	Strength	When?
Plain key in a fixed array	In software, exposed	None	Never (naive)
Obfuscated/split key	In software, hidden	Very low	Only for low-value, short-lived secrets
WBC + layered defence	In software, embedded in tables	Medium (delays)	When there's no hardware root; with key rotation + server auditing
TEE / Secure Element (SE)	Isolated by hardware	High	Preferred when the device supports it
HSM / SoftHSM (PKCS#11)	In a hardware (or software-emulated) module	High	Server side; the key never leaves it

Don't confuse the four abbreviations: TEE, SE, HSM, SoftHSM

- **TEE** — an isolated region **inside the processor**; on the **client** side (phone/device).
- **SE (Secure Element)** — a **separate, independent chip**; also on the client side, physically even more isolated than the TEE.
- **HSM** — dedicated hardware box/card on the **server** side that performs key operations.
- **SoftHSM** — a **software emulation** of an HSM; only the **interface** (PKCS#11) is the same, there is **no** hardware assurance (development/testing only).

In short: **TEE/SE = client-side hardware root; HSM/SoftHSM = server-side hardware (or emulated) root.**

Let's unpack every row of the table one by one, because the exam may ask "which option is correct when" **together with its justification**:

- **Plain key in a fixed array.** As we saw in Section 2: found within seconds by entropy scanning and direct reading. Not acceptable in any production system.
- **Obfuscated/split key.** Techniques like Section 2's "scrambling with only XOR" example; since the decoding information sits in **the same binary** as the key, this adds no strength. Only acceptable for **very low-value, very short-lived** secrets (e.g. a temporary token that lasts seconds), because the attacker's gain is already limited.
- **WBC + layered defence.** As we saw in Sections 3–4, this spreads the key across tables **mathematically**; used not alone but together with key rotation + device binding + server auditing, it provides a **moderate** delay. The best option when there's no hardware root.
- **TEE / Secure Element (SE).** The key never surfaces **in the open** in normal operating-system memory; so it is **structurally** resistant to all software-based attacks (entropy scan, algebraic solution, DCA, DFA, code lifting). This is the reason to prefer it when the device supports it.
- **HSM / SoftHSM.** As we saw in the subsection just before in Section 5, this is the same idea's counterpart for the **server side**; in a real HSM the key never leaves the hardware, while in SoftHSM only the **interface** is the same — its assurance level is low (should not be used outside development/testing).

The decision rule

If you don't have to put the key in software, don't. Prefer a hardware root (TEE/SE/HSM) if one is available. WBC is a **trade-off** solution for the situation where there's **no** hardware root, or where not every device can be trusted — and always together with key rotation, device/version binding, and server-side risk auditing.

Worked example: applying the decision rule to three assets

Let's not leave the rule abstract; let's decide, in turn, for **three different assets** of a hypothetical (synthetic) mobile application. For each asset we'll ask three questions: (1) How high is the asset's value? (2) Is there a hardware root (TEE/SE) on the device / can it be trusted? (3) How often/how long will the asset be used?

Asset	Value	Is there a hardware root?	Decision	Rationale
1. Session-token encryption key (short-lived, valid for minutes)	Low–medium	Unknown (not every device may support it)	An **obfuscated key** is enough (WBC not needed)	The short lifetime already limits the attacker's gain; not worth WBC's cost
2. Local database encryption key (valid for months, user data)	High	Most phones have a TEE, but **not every model**	**TEE if available; otherwise WBC + device binding + frequent rotation**	Don't rely on a single protection; prefer hardware when it exists (the cost table in Section 3)
3. Payment-signing key sent to the server	Very high	A hardware root **exists** on the server side (HSM)	**HSM — the key never reaches the client**	The strongest option for the highest-value asset; never moving it to the client removes even the need for WBC

Observed pattern: WBC appears in the table only in **row 2** — that is, at the intersection of "no reliably present hardware root" and "value isn't low". In row 1, WBC is **unnecessary cost**; in row 3, WBC is **unnecessary risk** (since there's no need to put the key on the client at all, not putting it there is always safer).

What does "a hardware root exists but isn't trustworthy" mean?

There's a hidden assumption inside the decision in the second row of the table above, "TEE if available; otherwise WBC": **the presence of a TEE does not mean it can be trusted.** Let's unpack this:

- **If the device has root/jailbreak access:** many operating-system-level protections can be disabled. Because the TEE is a **separate** security boundary at the processor level, it's generally still resistant — but since the interfaces that manage access to the TEE are called **from an untrusted operating system**, extra care is needed to make sure the application is "using the TEE correctly".
- **Old/low-cost device models:** even if they claim TEE support, they may offer a TEE implementation that hasn't been **adequately tested** by the manufacturer, or that is out of date.
- **A variable the application can't control:** a mobile application cannot individually verify the **thousands of different device models** it runs on; so treating "TEE present" as **unconditionally trustworthy** is risky.

Practical conclusion: a realistic design doesn't say "definitely secure if there's a TEE"; it **prefers** the TEE, but doesn't **entirely drop** the rest of Section 5's layers (key rotation, server-side risk auditing) **even when there is a TEE** — it may just apply them less heavily than in the pure-software (WBC) case. This is the "single layer = fragile layer" rule (Section 6) applied even to the hardware root.

Common mistake → consequence → rule

Mistake: treating saying "we used whitebox" on a project as an automatic security boost for **every** key — i.e. applying WBC not selectively, but as the **default** protection.

Consequence: applying WBC to a low-value, short-lived asset only adds **size/speed cost** (Section 3) without gaining security; "protecting" a high-value asset (e.g. row 3) with WBC instead of never putting it on the client at all creates an unnecessary attack surface (code lifting, DCA/DFA).

Rule: decide **separately** for every asset (like the three table rows above). "We buried everything in WBC" is not a security claim; it is often a sign of **design laziness**.

SoftHSM and PKCS#11 (a server-side bridge, from Week 10)

In Week 10 we mentioned PKCS#11 and SoftHSM for key storage. On the server side, the key is kept behind a standard PKCS#11 interface; the application never sees the key's value, it only says "sign/encrypt this". SoftHSM is a software emulation of a real HSM; it offers **the same interface**, so it's used for development and testing, but it provides no real hardware protection. As a counterpart to WBC's problem at the endpoint (client), this is the answer to the key problem on the server.

How is a signing request processed behind PKCS#11, step by step?

To make this concrete, let's follow, step by step, the moment a server-side application has a transaction **signed** (a conceptual flow; not a real sequence of API calls):

1. **The application prepares a request.** It says "sign this 32-byte digest with `key-id=7`". The application never sees the key's **value**, it only refers to it by an **ID**.
2. **The PKCS#11 layer forwards the request to the module.** The request goes, over the standard PKCS#11 interface, to either a real HSM or (in a development/test environment) SoftHSM.
3. **The module uses the key within its own boundary.** The module (HSM or SoftHSM) performs the signing operation with the private key corresponding to `key-id=7` **inside itself**; the private key **never leaves** the module.
4. **Only the result is returned.** Only the **signature** (the result) is returned to the application; the key itself never travels **in plain form**, over the network or between memory spaces.

This is exactly where the difference between a real HSM and SoftHSM comes into play: in a real HSM, step 3 happens **inside a tamper-resistant hardware chip that can't be reached from outside** — even an operating-system-level attacker cannot extract the key. In SoftHSM, the same step 3 runs **in the server's own memory, as an ordinary process**; the interface (API) is the same, but **the assurance level underneath is completely different** — an attacker who compromises the server can also reach SoftHSM's memory space.

Common mistake → consequence → rule

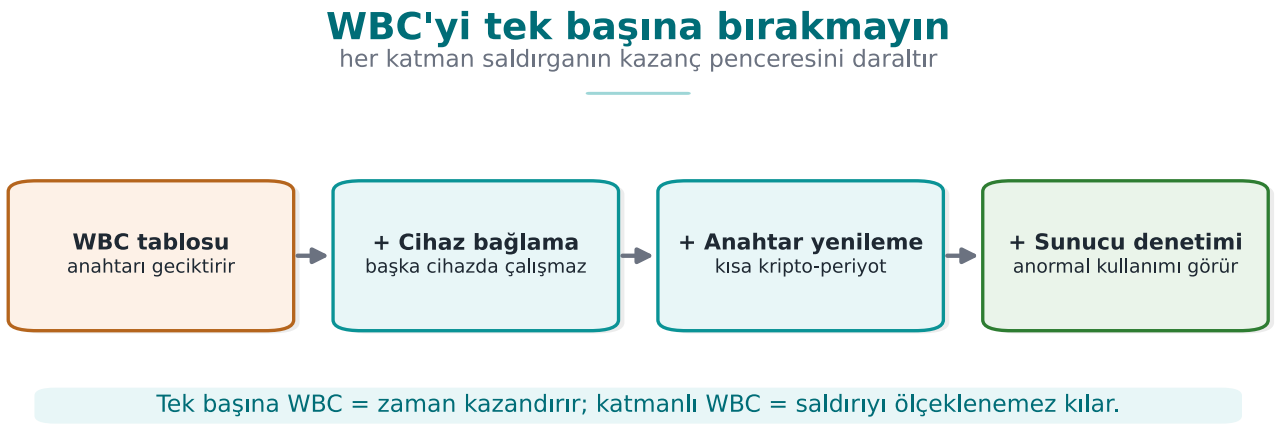
Mistake: saying "we use the PKCS#11 interface, so we're at HSM-level security" — treating SoftHSM and a real HSM as equally secure **because the interface is the same**.

Consequence: a system using SoftHSM in production can lose its key as easily as **an ordinary file/memory leak** if its server is compromised; the sentence "we use an HSM" creates a false sense of security.

Rule: use SoftHSM **only for development and testing**; move to a **real** hardware HSM (or a cloud provider's hardware-backed key management service) in production for high-value keys. The interface can stay the same (the code doesn't change); **the assurance underneath** must change.

6. Mistake → attack → protection (summary scenario)

Let's gather this week's rules into a single story (synthetic, for defensive purposes):



1. **Flawed design:** the client application embeds a data-encryption key as `static const uint8_t k[16]`.
2. **Attack (conceptual):** `strings` and an entropy scan find the high-entropy 16-byte block; a debugger confirms that this block is used as the key in the encryption call. The key comes out within minutes.
3. **Protection (layered):**
4. Move the key to **TEE/SE** if possible; otherwise bury it in **WBC tables** (no plain byte array left).
5. Wrap WBC in Week 9's **obfuscation** rules and Week 6's **RASP** (anti-debug, tamper detection); this makes it harder for DCA to collect traces and for DFA to inject faults.
6. **Bind to the device/version:** so the tables don't work when copied to another device (code lifting).
7. **Rotate the key frequently** (a short crypto-period) and apply **risk auditing on the server**.
8. **The risk that remains, honestly:** a sufficiently determined attacker can still extract the key on a single device; but now breaking one copy doesn't open up the whole system, and the extracted key's lifetime is short.

Second scenario: layers against code lifting (synthetic)

Let's work through the same structure, this time against the **code-lifting** threat we defined in Section 2 — because this threat corresponds to a **different class of mistake** than "embedding the key plainly", and requires different layers.

1. **Flawed design.** A digital content application (synthetic: a "licence verification" component) builds its whitebox AES routine **without external encoding** (only with the internal encoding + mixing matrices from Section 3). The routine is designed like an **independent function** that could be called from any application: its input/output is **the same** shape as standard AES input/output (the "without external encoding, it computes the `AES` function exactly" situation from Section 3, Step 5).
2. **Attack (conceptual).** The attacker **doesn't even try** to extract the key (no need for BGE/DCA/DFA). Instead, they copy the whitebox tables and the code that calls them **as-is**, and paste them into their own fake application. The fake application can now produce **valid licence responses** — without ever knowing the key, purely by **stealing the function**.
3. **Protection (layered).**
4. **Add external encoding (F, G)** (Section 3, Step 5): bind the routine's input/output to an **application-specific** form; the routine no longer computes "generic AES", it only produces a meaningful result with the encoded input/output **this specific application expects**.
5. **Add device/version binding** (Sections ⅘): derive the external encoding's parameters from the device identity or version information; when the tables are copied to another device, let them produce a **different** (wrong) result.
6. **Add server-side verification:** have the licence response checked **separately** on the server (e.g. flag the same licence being used from an unexpected number of devices as an anomaly).
7. **The risk that remains, honestly.** External encoding and device binding make code lifting harder **by making it device-specific**; but if the attacker compromises **that very device** (e.g. clones the entire device), they can still obtain a working copy. That's why the server-side check in step 3 remains the **last line of defence** — just as in the first scenario.

A rough timeline for both scenarios

Without claiming numerical precision, it's useful to give a **sense of scale** for "how long does each step take" — in the same spirit as the "cost of the attack" table in Section 4:

Step	Roughly how long does it take?	Why
Entropy scan / static analysis (Scenario 1)	Minutes	A single pass with automatic tools
Finding and copying the whitebox routine (Scenario 2)	Minutes–hours	Finding the routine in the codebase takes time (a bit longer if K-06 is present); copying is fast
BGE-style algebraic solving	Hours–days	Requires a specialised algorithm and tools
DCA (collecting many traces + statistics)	Hours–days	Thousands of runs + setting up analysis automation takes time
DFA (finding the fault point + repeating)	Days	Finding the right fault point requires trial and error
Designing and implementing the defence (developer)	Weeks–months	Design, testing, performance tuning, independent evaluation (Week 12)

What should you take from this table?

The **cheapest** attacks (entropy scan, code lifting) are also the **fastest** — which is why we gave them special attention in Sections 2 and 6. Designing the defence takes **much longer** than attacking; this asymmetry (the cost balance from Section 2) also holds on the time axis, and explains **why the defender needs to act early** (not wait for the project deadline).

The shared lesson of the two scenarios

First let's put the two scenarios side by side and see their **differences**:

	Scenario 1: plain embedded key	**Scenario 2: routine without external encoding**
Flawed design	Key in a `static const` array, exposed	Whitebox routine works with standard AES input/output
Initial attack	Entropy scan + debugger confirmation	Copying the routine/tables as-is
Does the attacker know the key?	Yes, **at the end** of the attack	**Never** — steals the function without ever seeing the key
Which section covers this	Section 2 (naive solutions)	Section 2 (code lifting) + Section 3, Step 5 (external encoding)
Key layer	TEE/SE or WBC tables	External encoding (F, G) + device/version binding
Last line of defence	Server-side risk auditing	Server-side risk auditing

Both scenarios start with **different mistakes** (plain embedded key vs. routine without external encoding) and proceed with **different initial attacks** (entropy scan/algebraic solution vs. direct copying), but they **end in the same place**: a single technical layer (only WBC, only device binding, only server auditing) **is never enough on its own in either case**. This confirms the pattern the lecture has been working with from the start:

- **Week 3:** "Confidentiality is cryptography's job, don't rely on the secrecy of the algorithm" (Kerckhoffs).
- **Week 9:** "A single obfuscation rule isn't enough, layer K-01…K-08."
- **Week 11 (this week):** "A single whitebox design isn't enough; layer it with key rotation + device binding + server auditing + (if possible) a hardware root."

The shared sentence of all three weeks: **"a single layer = a fragile layer."**

This week's big picture: a single look from start to finish

Before writing your project section, let's briefly review, start to finish, how this week's seven sections (0–6) form **a single story**. The goal isn't to teach anything new, but to let you see once more **how what you've learned connects together**.

1. **Section 0 — Basic concepts.** We started from scratch: encryption, key, S-box, lookup table, XOR, bijection, entropy, trace, correlation, GF(2)... None of these were specific to whitebox; they were all our **tools**. From Section 1 onward, we used these tools.

2. **Section 1 — Three attacker models.** We saw the black-box assumption from Week 3 **collapse** in client-side code. We followed the white-box attacker's six-step session (acquisition → static analysis → debugger → memory reading → tampering → automation). These six steps were the **precondition** for all the attacks in Section 4 (BGE = static, DCA/DFA = dynamic).
3. **Section 2 — Why is the problem hard?** We **computed by hand** why naive solutions (plain key, only XOR) collapse (the `0x3C XOR 0x5A` example). We met the code-lifting threat. We saw that WBC is "Week 9's obfuscation rules applied to cryptography".
4. **Section 3 — Chow AES, step by step.** We built the five steps (partial evaluation → T-box merging → internal encoding → mixing matrix → external encoding) **entirely by hand on a toy S-box**: first we extracted the key consistently from all 4 entries (Steps A–C), then, once we added the internal encoding `G`, we saw the same attack give **contradictory and wrong** results, then we traced, with a table, why DCA overcomes this not with a simple exact-match test but with **statistics**. We closed the section by computing the **order of magnitude** of the size/speed cost.
5. **Section 4 — Attack history.** We learned to **distinguish** the BGE (mathematics), DCA (statistics), and DFA (sabotage) attacks from one another; we saw why WhibOx is an independent trial; we confirmed with a table that every countermeasure only makes **specific** attacks harder.
6. **Section 5 — Place in layered defence.** Through three concrete assets (session token, local DB key, payment-signing key) we decided **when** WBC is the right choice. We saw that SoftHSM provides only an **interface**, not the real assurance.
7. **Section 6 — Two scenarios.** We applied the same "mistake → attack → protection" skeleton to two **different** mistakes (plain key, routine without external encoding); both ended in the same lesson: **a single layer is not enough**.

Next step (Section 7): now you'll apply this knowledge to **your own project** — you'll pick a key and write, in your own words, the same chain for it (asset value → threat model → chosen layers → the risk that honestly remains).

> **Looking ahead: how will this knowledge be used in Weeks 12–13?**
>
> This week we assessed, conceptually and on our own, **how much** a protection (WBC) actually withstands. In Week 12 we'll connect this to an **independent certification/penetration-testing process**, and in Week 13 to **attack potential scoring** (a method for numerically rating how "easy" an attack is). The "attack access/ skill" table we built by hand today in Section 4 is a **simplified precursor** of that scoring logic.

7. Term project: this week (S8 — key-protection justification)

Add, to the S8 (crypto and keys) section of your project, a protection decision and **rationale** for **at least one sensitive key**:

1. Where the key sits, how it's protected (if not stored plainly: TEE/SE, WBC, obfuscation — which one and why).
2. If the protection isn't sufficient (e.g. you're pure software), state this **explicitly** and name the compensating layers: the key-rotation period, device/version binding, server-side risk auditing.
3. If you don't use one (e.g. you didn't implement WBC), give the reason **why not** (cost, a hardware root exists, asset value is low). A justified "we didn't use it" is a complete answer.

Worked example: a filled-in S8 rationale

To make what you'll write concrete, let's show, step by step, how to fill out the S8 section using a **synthetic** example project (don't copy this text verbatim; write according to your own project's assets):

Example project: the 'Campus Wallet' mobile app (synthetic)

Asset: the `DEK` (data encryption key) that stores the user's balance information in the local database, used with AES-256-GCM (Week 3).

1. Where does the key sit, how is it protected? The DEK is kept in a TEE-backed key store on devices that support TEE. On older devices that don't support TEE (~15% of the application's test matrix), the DEK is embedded in a whitebox AES table network; the table network includes internal encoding + mixing matrix + external encoding, as in Section 3, Steps 3–5.

2. If the protection isn't sufficient, which layers compensate? For the pure-software whitebox path (TEE-less devices): the DEK is rotated **every 30 days** (crypto-period, Week 10); the external encoding's parameters are derived from the device identity (device binding, Sections ⅘); the client is hardened with Week 9's K-04 (control-flow flattening) and K-06 (call hiding) rules; on the server side, a request is flagged if an unexpected number of devices is making requests from the same account (server-side risk auditing).

3. Is there a protection that isn't used, and why? A hardware HSM is **not used** on the client side — because mobile devices don't have HSMs (the HSM is used only on the server side, for the payment-signing key, see the 3rd asset example in Section 5). This is a **technical impossibility**, not a cost, rationale, and it is stated as such in S8.

Notice what's done right in this example: nowhere does the text say "we used whitebox, we're secure"; every sentence says **which layer is placed against which threat** and **why it is sufficient/insufficient**. This is what actually matters to an assessor.

Self-check: is your S8 draft ready?

Before you write/submit your project's S8 section, tick off this short list:

- Is the name of at least one **sensitive key** and **what it protects** clearly written?
- Is it stated **where the key sits** (TEE/SE, WBC, obfuscated, plain — which one)?
- If a pure-software protection (WBC or obfuscation) is used, are **at least two** compensating layers (rotation, device binding, server auditing, RASP) named?
- If a protection **wasn't used**, is the rationale (cost/no hardware root/low value) written **explicitly** — or was it silently skipped?
- Is there nowhere in the text a sentence like "WBC = secure" that **contradicts** this week's main rule?
- Is there **no** real key, password, or secret in the repo/examples (see the warning below)?

A weak and a strong S8 rationale, side by side

The most common feedback this week is that rationales are written **too short and generic**. Let's put a weak and a strong answer for the same asset side by side, and see the difference:

Weak rationale (avoid this)

"We protected our key with whitebox, so it's secure."

Why is it weak? (1) It's unclear which key, which asset. (2) "Whitebox = secure" **directly contradicts** this week's main rule (Section 4). (3) No compensating layer (rotation, binding, server auditing) is mentioned at all. (4) There's no cost/limitation discussion — it's written as if WBC were free.

Strong rationale (aim for this)

"Our local database encryption key (DEK) is embedded in a whitebox AES table network (with internal/ external encoding + a mixing matrix, Section 3) on older devices that don't support TEE. We know this alone isn't sufficient (Section 4); so the key is rotated every 30 days, the external encoding is tied to the device identity, and the server side monitors for an abnormal device count. On TEE-capable devices, TEE is used directly instead of WBC."

Why is it strong? The asset is **named**, the protection is **justified**, the limitation is **acknowledged**, the compensating layers are **listed**, and the alternative preferred when hardware is available is **stated**.

Short formula: *asset name + where it's protected + against which threat + what its limitation is + compensating layers.* If any one of these five elements is missing, the rationale is probably in the **weak** category.

Frequently asked questions (about S8)

We didn't implement WBC at all; will our project score drop?

No — recall item 3 of Section 7: **a justified 'we didn't use it' is a complete answer.** What's evaluated is not the presence of WBC, but **whether the right decision was made and justified**. A sentence like "our asset value is low, a hardware root (TEE) already exists, so we didn't see a need for WBC" is a **stronger** answer than an unjustified "we used WBC".

Do we really have to build our whitebox table with Chow's five steps?

No, this is not **expected** within the scope of this course. The five steps in Section 3 (partial evaluation → T-box → internal encoding → mixing matrix → external encoding) should be understood **at the conceptual level**; you're not required to write a full Chow AES implementation in your project. The simple encoded-table idea in the demo (`01-oyuncak-tablo`) is an example at the **level** expected of you in your project.

We keep our key in an environment variable; is that enough for S8?

An environment variable is **a bit better than a key hard-coded into the code** (it isn't found directly by an entropy scan in the binary), but it still appears as **plaintext** to anyone who can read the process environment (e.g. another process running on the same machine, a debugger). In S8, state this explicitly as not "key management" per se, but **which category** it falls into alongside the table in Section 5 (probably somewhere between "obfuscated/plain" and "WBC").

No real key should be present in the repository

All key files, tables, and examples must be **synthetic**. Finding a real key in the repository is one of the most serious findings in a certification assessment; it counts the same way in the project.

8. Self-check

1. Distinguish the black, grey, and white box attacker models in one sentence each. Which one does standard AES's security proof assume?

Black box: the attacker sees only input/output. **Grey box:** they also observe side channels (power/ time). **White box:** they have full access to memory, code, and intermediate values. Standard AES's security proof assumes the **black box** model.

2. What ground do the WBC attacker's 'run only one round' and 'inject a fault' capabilities lay for which attacks (DCA/DFA)?

Tracing intermediate values and solving them statistically lays the ground for **DCA** (Differential Computation Analysis); injecting a deliberate fault and comparing the output lays the ground for **DFA** (Differential Fault Analysis).

3. Why isn't embedding the key in a fixed array a protection? What role does an entropy scan play here?

In the binary, the key sits as a **high-entropy** byte block; putting it in the code only changes its location, it doesn't hide it. An entropy scan finds high-entropy (key-like) regions in the binary; the demo's `--tara` shows this.

4. What is code lifting? Why can WBC be defenceless against an attacker who never learns the key at all? Which countermeasure closes this?

Code lifting: the attacker copies the whitebox table/routine as-is and uses it elsewhere — performing encryption/decryption without ever extracting the key. **External encoding** closes this: the function's input/output is bound to the application, so the lifted routine is useless in another context (plus device binding).

5. Why is partial evaluation, `T[x] = S-box[x ⊕ k]`, insecure on its own?

The table depends directly on `k`; the attacker compares the table against the known S-box and **recovers** `k` from the `x ⊕ k` mapping. To be secure, the table's inputs/outputs must be masked with hidden internal encodings.

6. What is the difference between internal encodings and external encodings (F, G)? What is external encoding's biggest limitation?

Internal encodings mask the intermediate values between tables with hidden bijections (internal to the component). **External encodings (F, G)** transform the function's input and output and bind it to the application (making code lifting harder). Biggest limitation: because the input/output changes, **the caller must also match** → the output is no longer standard AES, it can't be used just anywhere.

7. State the approximate table size and speed cost as an order of magnitude. Why does this limit WBC to only selected operations?

Tabulated WB-AES tables are on the order of **megabytes** and **tens to hundreds of times** slower than black-box AES. Because of this size/speed penalty, WBC is applied only to selected, critical operations, not the entire crypto flow.

8. How does the statement 'every published pure-software WBC design has been broken' affect your decision to use WBC in your project?

We don't treat WBC as key protection **on its own**; we use it as a layer that **delays** key extraction, together with key rotation + device binding + server-side auditing. Hardware-backed protection (TEE/SE/HSM) is preferred whenever possible.

9. Which classic hardware attack does DCA resemble? Why don't internal encodings always stop it?

It resembles **DPA** (Differential Power Analysis); DCA replaces the power trace with a **software trace** (memory accesses/intermediate values). If the internal encodings are linear or weak, statistical correlation can get past the encoding; that's why DCA doesn't always stop it.

10. Rank the options for protecting a key (plain → hidden → WBC → TEE/SE → HSM) on the strength/cost axis.

From weak/cheap to strong/expensive: **plain embedded key < obfuscated/encoded table (pure-software WBC) < WBC + binding/rotation < TEE/secure element (SE) < HSM/hardware**. Strength and cost rise together; the choice depends on the asset's value.

11. If you're using WBC, name at least three layers you need to add so it isn't left standing alone.

Key rotation, **device/application binding**, and **server-side usage auditing/revocation**. RASP/ integrity checking is added to these as well.

12. Write the S8 protection rationale for a key in your project in three sentences.

Example: 'This DEK protects data at rest with AES-256-GCM. The key is generated and stored with SoftHSM/PKCS#11 and never held in memory in plain form. Because pure-software whitebox designs have been broken and the asset's value is high, hardware-backed protection was chosen.'

13. In Section 3's toy S-box example, the real key is k=2, and the table is T=[0,2,3,1]. Using input x=2, recompute the key step by step.

`T[2]=3` → `S⁻¹[3]=0` (because `S[0]=3`) → `k = x XOR S⁻¹[T[x]] = 2 XOR 0 = 2`. Matches the real key (`k=2`).

14. In the same example, once internal encoding `G` has been added (`T'=[1,0,2,3]`), what 'key' does the naive attack predict for input x=3? Why doesn't this match the real key?

`T'[3]=3` → `S⁻¹[3]=0` → `k_estimate = 3 XOR 0 = 3`. Not the real key, `2`, because the attacker computes `T'[x] = G(S[x XOR k])` as if it were `S[x XOR k]`; they don't account for `G`'s presence, so the estimate becomes meaningless.

15. In Section 3's DCA example table, what are the match rates of the correct/wrong key hypotheses in the unencoded (`T`) and encoded (`T'`) cases? What does this tell us?

Unencoded: the correct hypothesis matches **4/4**, the wrong one **0/4** (a clean separation). Encoded: both the correct and the wrong hypotheses match **0/4** — the exact-match test can no longer tell them apart. This shows why DCA works not with a single trace, but with **statistical correlation over many traces**.

16. In the GF(2) mixing-matrix example (`yeni_v1 = v1 XOR v0`, `yeni_v0 = v0`), compute the output for input `10` (v1=1, v0=0). How is it confirmed that the matrix is non-singular?

`yeni_v1 = 1 XOR 0 = 1`, `yeni_v0 = 0` → output `10` (unchanged for this input, because v0=0). Non-singularity: confirmed by showing the original value is recovered when the same matrix is applied a second time (in this example the matrix is its own inverse) — i.e. the transformation **loses no information**.

17. In one sentence, what was the fundamental reason the BGE attack broke Chow's 2002 design?

The internal encodings were not random but had an **affine** (linear + constant) structure; this special structure could be recovered by an algorithm (the affine equivalence algorithm) that solves the algebraic relationships between neighbouring tables.

18. List the four steps of the DFA attack in order.

(1) Get the correct ciphertext from a correct (reference) run. (2) Re-run the same plaintext, injecting a fault into an intermediate value, and get the faulty ciphertext. (3) Examine the difference between the two outputs; the difference carries information about the key. (4) Repeat with different fault points, narrowing down the key candidates.

19. Why is the WhibOx competition considered more trustworthy than a claim of 'a design a company tested itself'?

WhibOx exposes a design publicly to attack by **independent, numerous, and motivated** outside researchers; this is a much wider and more impartial trial than the limited testing a company does with its own internal team.

20. SoftHSM and a real HSM offer the same PKCS#11 interface; what is the fundamental security difference between them?

In a real HSM, the key operation is performed inside a **tamper-resistant, isolated hardware chip**. In SoftHSM, the same operation runs in the **server's normal memory/process**; even though the interface is the same, if the server is compromised, the key in SoftHSM is also at risk.

21. In Section 6's second scenario (code lifting), unlike the first scenario (plain embedded key), which step does the attacker never take, and why?

The attacker never takes the step of **extracting the key** (no need for BGE/DCA/DFA); they steal the function by copying the whitebox routine/tables **as-is**, without ever knowing the key.

22. In Section 5's three-asset example (session token, local DB key, payment-signing key), in which row is WBC preferred? Why not in the other two?

WBC is preferred only in **row 2** (the local DB key, where a hardware root isn't guaranteed on every device). In row 1 (the short-lived session token), WBC adds unnecessary cost; in row 3 (the server signing key), never putting the key on the client at all (HSM) is safer than WBC.

23. How does the cost asymmetry 'the attacker wins cheaply while the defender pays dearly' explain the reason WBC exists?

Naive protections (plain key, a single XOR) are broken within seconds-to-minutes; WBC tries to reverse this asymmetry, **raising** the attacker's cost in exchange for an extra cost the defender can accept — it doesn't zero out the asymmetry, it only shifts the balance.

24. Is an entropy scan a static or a dynamic analysis technique? Which is DCA?

An entropy scan is **static** — performed on the binary without ever running the program. DCA is **dynamic** — it requires actually running the program (multiple times) and collecting execution traces.

25. In Section 2's 'scrambling with only XOR' example, k=0x3C, m=0x5A. What is the encoded value, and how does the attacker recover the key?

`kodlanmis_k = 0x3C XOR 0x5A = 0x66`. The attacker finds both `0x66` and the mask `0x5A` in the binary; computing `0x66 XOR 0x5A = 0x3C` recovers the key — because the decoding information (the mask) sat in **the same place** as the key.

26. What is the shared sentence that unites this week's two scenarios (Section 6) with the rules of Weeks 3 and 9?

"A single layer = a fragile layer." Neither following Kerckhoffs's principle alone (Week 3), nor obfuscation rules alone (Week 9, K-01…K-08), nor WBC alone (this week) is sufficient on its own; security always comes from **the sum of the layers**.

27. In Section 3, Step 3's verification `G⁻¹(T'[x]) = T[x]`, show the computation for x=1.

`T'[1] = 0` → `G⁻¹[0] = 2` → result `2`. The real value is `T[1] = 2`. They match (`2 == 2`); this confirms that Table 2 undoes the encoding and recovers the correct intermediate value.

28. Why was Karroumi's 'dual AES' strengthening broken? What is the general lesson from this?

Because the additional tables also used the same kind of (affine) encodings, a BGE-like algorithm solved them too, the same way (with a bit more computation). Lesson: the way to strengthen a protection is not to **increase its quantity**, but to change the **weak structure** the attack actually relies on.

29. Group TEE, SE, HSM, and SoftHSM by client-side vs. server-side.

Client side: TEE (an isolated region inside the processor), SE (a separate secure chip). **Server side:** HSM (hardware), SoftHSM (a software emulation of an HSM, for development/testing only).

30. Why is forgetting to move a 'test/debug key' out for production the same vulnerability as Section 2's first rule?

In both cases the key sits in the binary in **plain, fixed** form; the "test-only" label doesn't **stop** an attacker from finding it with an entropy scan. An assessor scans test/debug paths with the same rigour as production code.

31. Why is the sentence 'we've taken measures against side-channel attacks, so we must also be protected against whitebox attacks' mistaken?

It treats the three attacker models (black/grey/white box) as if they could substitute for one another. A grey-box defence (e.g. constant-time comparison) **does nothing against static/dynamic whitebox analysis**; every defence must be assessed by **explicitly stating** which capability, in which model, it counters.

32. Which layer is common to both scenarios in Section 6 as the 'last line of defence'?

Server-side risk auditing. In both the plain-embedded-key scenario and the routine-without-external- encoding (code lifting) scenario, even if every client-side layer is bypassed, anomalous-usage monitoring on the server remains the last line of defence.

33. Summarise Section 3's five steps (partial evaluation → T-box → internal encoding → mixing matrix → external encoding), each in order, stating what gap in the previous step each one closes.

Step 1 buries the key in a table, but the table alone can be compared against the S-box and solved; Step 2 reduces the number of observation points but isn't sufficient alone; Step 3 (internal encoding) masks the table against an attacker looking at it in isolation; Step 4 (mixing matrix) spreads information across multiple bits/bytes, making it even harder to look at just one table; Step 5 (external encoding) makes code lifting harder by binding the whole routine to the application. All five are used together.

34. In Section 5's 'a hardware root exists but isn't trustworthy' discussion, why can the TEE itself still be resistant on a device that has been rooted/jailbroken?

The TEE is a security boundary **separate and isolated** from the normal operating system; a root-level compromise at the OS level doesn't automatically **bypass** this boundary. But because the interfaces that manage access to the TEE are called from an untrusted operating system, the application still needs to use this interface **correctly and carefully**.

35. Briefly summarise the distinction between the 'construction chain', the 'attack chain', and the 'defence chain' from Section 0; in which sections was each one covered?

The **construction chain** (key → S-box → table → internal/external encoding → GF(2) matrix) was built by hand in Section 3. The **attack chain** (static/dynamic analysis → entropy scan/debugger → execution trace → correlation) became concrete in Sections 1, 3, and 4. The **defence chain** (TEE/SE/HSM → WBC → rotation + device binding + server auditing) came together in Sections 5 and 6.

36. Which of this week's hand-worked examples can you match the naive and encoded tables of the `01-oyuncak-tablo` demo to?

The **naive table** matches the idea of Section 3's `T[x] = S[x XOR k]` (Steps A–C, which give the real key consistently from all 4 entries). The **encoded table** matches the idea of `T'[x] = G(T[x])` (the same attack now giving contradictory, wrong results such as `1, 3, 1, 3`).

A quick review before the exam: glossary of terms

Before solving the self-check questions or sitting the exam, use the table below to review all of this week's terms **at a glance**. Each term points to the section where it was first defined — if you get stuck on a term, go back to that section and read it **together with its context**.

Term	Short definition	Where defined/used
Black/grey/white box	Three threat models based on the attacker's level of visibility/access	Section 1
WBC (whitebox cryptography)	The field studying the security of encryption algorithms running in a white-box environment	Section 1
Debugger / static-dynamic analysis	A tool for tracing a program step by step; examining it without running it / by running it	Section 0
Instrumentation / execution trace	Automatically watching a program and recording intermediate values; the recorded sequence of intermediate values	Section 0
Entropy / entropy scan	A measure of randomness; searching a binary for high-entropy (key-like) blocks	Sections 0, 2
Partial evaluation	Precomputing the key + S-box into a single lookup table	Section 3, Step 1
Permutation / bijection	A one-to-one, onto mapping with an inverse; the basis of our toy S-box	Sections 0, 3
Internal encoding	Masking the intermediate values between tables with hidden bijections	Section 3, Step 3
GF(2) / mixing matrix / non-singular matrix	XOR-based linear algebra; an invertible matrix that spreads information across bits	Sections 0, 3 (Step 4)
External encoding (F, G)	The outermost layer that binds the function's input/output to the application	Section 3, Step 5
Code lifting	Copying the whitebox routine without knowing the key, stealing its function	Sections 2, 6
The BGE attack	The first break, solving affine internal encodings algebraically (2004)	Section 4
DCA	An attack applying statistical (DPA-like) analysis to execution traces	Sections 3, 4
DFA	An attack that injects a fault and extracts the key from the output difference	Section 4
Correlation	A measure of whether the predicted value and the observed value change together, consistently	Sections 0, 3, 4
WhibOx	A competition series in which whitebox designs are tested publicly and independently	Section 4
TEE / SE	An isolated, secure region of the processor/hardware, suitable for storing keys	Sections 0, 5

Term	Short definition	Where defined/used
HSM / SoftHSM / PKCS#11	Dedicated hardware that stores keys / its software emulation / the standard interface	Sections 0, 5
Crypto-period / key rotation	A key's usage lifetime; shortening it reduces the value of an extracted key	Week 10, Sections 4, 5
Device/version binding	Making the encodings device-specific, rendering code lifting useless	Sections 4, 5, 6

How should you use the glossary?

If you see a term and can't **immediately** recall its meaning, that's not a shortcoming — it's normal. Go to the "Where defined" column in the table and **re-read** that section; remembering a term **in its context** (which example, which attack it appeared in) sustains learning better than memorising the definition alone.

9. Resources and further reading

- **The secure programming technical guide** (course source) — the WBC attacker model and whitebox AES's position in a product; the "keys shall be secured and/or hidden" requirement.
- S. Chow, P. Eisen, H. Johnson, P. van Oorschot — "White-Box Cryptography and an AES Implementation" (SAC 2002) and WB-DES (DRM 2002) — the foundation of table-based WBC.
- B. Wyseur — "White-Box Cryptography: Hiding Keys in Software" (short introduction) — a student-friendly overview.
- Billet, Gilbert, Ech-Chatbi — the BGE attack (SAC 2004).
- Bos, Hubain, Michiels, Teuwen — "Differential Computation Analysis" (CHES 2016).
- Sanfelix, Mune, de Haas — "Unboxing the White-Box" (Black Hat EU 2015) — DFA/DPA.
- **WhibOx** competition results (2017–2024) — the status of published designs.

How should you read these sources?

You aren't expected to read **all** of these sources for the exam or the project. The priority order should be this: (1) the relevant section of the technical guide — because your project's S8 rationale rests on it; (2) Wyseur's short introduction — to reinforce the five steps from Section 3 with **a different telling** than this week's notes; (3) Chow et al.'s original paper and the attack papers (BGE, DCA, DFA) only for those who are **curious** or want to go deeper into the subject later (e.g. in a capstone project). This week's notes are written to be **self-sufficient** even if you read none of them.

Next week

Week 12 — Certification and penetration-test planning. This week we saw the importance of measuring how much a protection "actually withstands"; in Week 12 we'll connect this to the process and reporting of an independent evaluation (attack potential scoring, Week 13).