Week 9 — Advanced Obfuscation and Diversification

Date	13.11.2026
Learning outcomes	LO.3
Duration	3 hours
Prerequisites	The white-box attacker model from Week 1; compilation, `objdump` and reverse engineering concepts from Week 4; control flow in C (`if`, `switch`, loops)
Labs	`code/week-09` — 2 demos; build once in the `code` folder, then run from `bin/linux` (`bin\windows` on Windows)

This week's working demo

`code/week-09/01-manuel-gizleme` — Manual obfuscation + cost measurement: same behaviour, the secret stays hidden, cost measured with `objdump`. · `code/week-09/02-cesitlendirme` — Diversification: the same source compiled with two seeds → binaries that are behaviourally equivalent but structurally different.

Run with: `sh demo.sh` (Linux/WSL), or build with CMake and run from under `bin/`. Fully synthetic and safe; it does not harm the student's computer.

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

Setup and the first build are in `code/README.en.md` (Visual Studio or the command line). After cloning the repository and entering the `code` folder:

```
# Windows (PowerShell) — 'code' klasöründe
.\build.ps1                                  # tüm demoları bir kez derle
cd week-09\01-manuel-gizleme
.\bin\windows\erisim_temiz.exe CEN429-OK     # korumasız sürüm
.\bin\windows\erisim_gizli.exe CEN429-OK     # gizli sürüm — çıktı AYNI
```

```
# WSL / Linux — 'code' klasöründe
./build.sh
cd week-09/01-manuel-gizleme
sh demo.sh                                   # açıklamalı tam akış
```

Expected output: both versions print the **same** "access granted" line for `CEN429-OK` (behaviour preserved). In the `strings` output, `CEN429-OK` **is visible in the clean build and not visible in the obfuscated build** (K-07 string encoding). `objdump` measurement: `erisim_ver` goes from **~28 instructions / 3 branches → ~51 instructions / 6 branches** (cost increased). The second demo (`02-cesitlendirme`) compiles the same source with two seeds → **same behaviour, different binary**.

By the end of this week you will be able to

1. Define **code obfuscation** as a security **rule/countermeasure**, and write each technique in the form "what does it protect, which requirement does it satisfy, what is its cost, what is its limit."
2. Explain the obfuscation **taxonomy** (layout, data, control flow, anti-analysis, virtualisation) and the **white-box / MATE** (man-at-the-end) threat model.
3. Analyse, at a conceptual level, advanced **control-flow** and **data** obfuscation rules (opaque predicates and loops, arithmetic encoding, bogus operations and dead branches, control-flow flattening, random exit, string and constant encoding).
4. Explain why **diversification** is a holistic resilience rule, and what **virtualisation-based**, **compiler-based**, and **dynamic** obfuscation buy you and at what cost.
5. **Measure the effectiveness** of obfuscation using the potency, resilience, stealth, and cost metrics to justify a protection decision, and write a **protection table** for the S9 section of your own project.

Class schedule (timing plan for the instructor)

Time	Section	What happens
0:00–0:20	1	Why is obfuscation a rule? White-box / MATE threat model; what obfuscation gives and does not give
0:20–0:45	2	Obfuscation taxonomy and the "protection rule" template (what it protects / requirement / cost / limit)
0:45–0:50	Break	
0:50–1:35	3	Control-flow rules: opaque predicate and loop, arithmetic encoding, bogus operation/dead branch, flattening, random exit
1:35–1:40	Break	
1:40–2:10	4	Data rules: string/constant encoding, variable splitting/merging; virtualisation, compiler-based, dynamic obfuscation (concept)
2:10–2:40	5	Diversification; measuring obfuscation (potency, resilience, stealth, cost); deobfuscation and resilience
2:40–3:00	6+	Its place in layered defence; project S9; self-check

About this week's sources

This week draws on two main sources: (1) the **secure programming technical guide** — which lists the **code hardening countermeasures** of a certified mobile payment library in a "description / implementation / limit" format; (2) the **Secure Programming Cookbook** (Viega & Messier), Chapter 12, "Anti-Tampering." The countermeasures in the guide are given here as **rules**, stripped of the product and of any real values; all code and values are **synthetic**. Week 4 introduced these techniques (symbol/string hiding, an introduction to flattening); this week treats the same techniques at an **advanced level** and adds **how to measure** them. Week 14 shows their **automated** counterpart (Tigress).

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; the later sections build on it.

Why does this section exist?

This week's subject is **code obfuscation**. But first let's pin down a few basic terms.

Without these terms, the subject **stays up in the air**.

We assume you know nothing — that's a good starting point.

What is source code?

- The **human-readable** program text that you write (e.g., a C file).
- Example: `int topla(int a, int b) { return a + b; }`

Source code is **compiled** and turned into the form the machine runs.

Compiler and binary

- **Compiler:** the program that translates source code into **machine code** (gcc, clang).
- **Binary:** the result of compilation; the file the computer runs directly (`.exe` , `.so`).

Machine code and assembly

- **Machine code:** the numeric instructions the processor understands.
- **Assembly:** a somewhat more human-readable form of machine code (`mov` , `cmp` , `jmp`).

This is what you see when you open a binary file.

What is reverse engineering?

Reverse engineering: looking at a binary file and trying to work out **what the program does**.

This is the attacker's basic job.

Decompiler

- **Decompiler:** a tool that turns a binary file back into a form close to readable code.
- Examples: Ghidra, IDA.

Goal: someone with no access to the source can infer the logic just by looking at the binary.

What does decompiler output look like? Imagine we only have a `.exe` / `.so` file and never saw the source code. When we open the function `int topla(int a, int b) { return a + b; }` in a decompiler, the typical output looks like this (the exact format varies from tool to tool, but the idea is the same):

Decompiler output (conceptual example — Ghidra/IDA-like)

```
undefined4 FUN_00401020(int param_1, int param_2)
{
    return param_1 + param_2;
}
```

Notice: meaningful names such as `topla`, `a`, `b` **have been lost** (they were never in the binary the compiler produced — the source-level names simply aren't stored there); the tool falls back on its own generated names, **generic** ones such as `FUN_00401020` (address-based) and `param_1`, `param_2`. But **the logic is preserved**: the addition is still plainly visible. This is concrete proof of why changing only **names** (the layout family) is not enough for obfuscation — a decompiler already operates having lost the names; the real fight is over making the **logic** (control flow and data) unreadable, which is what sections 5 and 6 are about.

The `strings` command

- **`strings`:** a simple tool that lists the **readable text** inside a binary file.
- `strings program` → text such as `"Lisans gecersiz"`, `"http://..."`.

This is usually the attacker's **first step**.

Symbol and symbol table

- **Symbol:** the **name** of a function or variable inside a binary file (e.g., `lisans_dogrula`).
- **Symbol table:** the list of these names.

If the name `lisans_dogrula` is visible, the attacker knows where to look.

Debugger

- **Debugger:** a tool that runs a program **step by step** and shows its variables (gdb, lldb).
- An attacker can stop the program, read memory, and change values.

Bit, byte, hexadecimal

- **Bit:** 0 or 1.
- **Byte:** 8 bits.
- **Hexadecimal (hex):** written with a `0x` prefix; `0x2A` = 42.

We will see values like `0x5A` in the code; these are just numbers.

Let's tie the hex-binary-decimal conversion together in one table. Each hexadecimal digit corresponds to exactly **4 bits** (because $2^4 = 16$); that's why a byte (8 bits) is always written with **two** hexadecimal digits:

Hex digit	Binary (4 bits)	Decimal
`0`	`0000`	0
`5`	`0101`	5
`A`	`1010`	10
`F`	`1111`	15

Let's use this table to decode `0x5A`: digit `5` → `0101`, digit `A` → `1010`; written side by side, `0101 1010` (8 bits, 1 byte). To convert to decimal, we sum the place values: `5 × 16 + 10 × 1 = 80 + 10 = 90`. So `0x5A = 90` (decimal) `= 0101 1010` (binary) — every `0x..` value you will see in this week's code examples is just a different way of writing the **same number** in these three notations (hex/binary/decimal); none of them is "more hidden" than another, they only differ in how easy they are for a human to read (hex is short, binary shows bit operations explicitly).

What is XOR?

- **XOR** (`^`): 1 if the two bits differ, 0 if they are the same.
- Property: `a ^ b ^ b == a` → **it undoes itself**.

```
c = a ^ 0x5A;   /* encode */
a = c ^ 0x5A;   /* decode */
```

It is heavily used in obfuscation because it is reversible.

Let's trace XOR step by step: hiding and recovering the byte `0x41`

Let's not leave XOR's "self-undoing" property abstract; let's trace a single byte bit by bit. The letter `'A'` has the binary (ASCII) value `0x41` (hex) = 65 (decimal) = `0100 0001` (binary). Let's use `0x5A` = `0101 1010` as the key.

Encoding: `0x41 ^ 0x5A`

Bit position	7	6	5	4	3	2
`0x41`	0	1	0	0	0	0
`0x5A`	0	1	0	1	1	0
XOR	0	0	0	1	1	0

Result: `0001 1011` = `0x1B`. The rule is simple: matching bits (`0^0=0`, `1^1=0`) give 0, differing bits (`0^1=1`, `1^0=1`) give 1 — exactly the bit-level counterpart of the "What is XOR?" definition above.

Decoding: XOR again with the same key: `0x1B ^ 0x5A`

Bit position	7	6	5	4	3	2
`0x1B`	0	0	0	1	1	0
`0x5A`	0	1	0	1	1	0
XOR	0	1	0	0	0	0

Result: `0100 0001` = `0x41` — exactly the byte we started with. The algebraic rule that guarantees this is: `a ^ b ^ b = a ^ (b ^ b) = a ^ 0 = a` (any value XORed with itself gives 0; a value XORed with 0 is unchanged). In rule K-07 in section 6 we will encode and decode a string using exactly this mechanism; there you will rebuild the same bit table, only for an array of bytes instead of a single byte.

A numeric entropy example

Let's make "entropy" concrete. Shannon entropy, a simple metric, looks at the **probability of occurrence** of each distinct byte value in a sequence: `H = -Σ p(x) · log2 p(x)`. Let's work through two small examples by hand:

- **Sequence A:** `[0x41, 0x41, 0x41, 0x41]` — all four bytes are identical. The probability of the single value is `p = 1`, `log2(1) = 0`, so `H = -(1 · 0) = 0` bits/symbol. Completely **predictable**: see the first byte and you know the rest.
- **Sequence B:** `[0x3F, 0xA1, 0x08, 0xC7]` — all four bytes differ, each with probability `p = 1/4`. `H = -4 · (1/4 · log2(1/4)) = -4 · (1/4 · (-2)) = 2` bits/symbol — the **highest** entropy value reachable in this four-symbol alphabet.

A real cryptographic key, at the scale of hundreds of bytes, looks like "sequence B": there is no repeating pattern among the bytes, and every byte value occurs with almost equal probability. When an attacker scans a binary with a tool, they can statistically tell low-entropy blocks like "sequence A" (fixed text, zero padding, repeating structures) apart from high-entropy blocks like "sequence B" (keys, compressed or encrypted data) — this is precisely the numeric counterpart of the sentence in the "What is entropy (randomness)?" definition below: "if the attacker sees a high-entropy block, they say there might be a key here."

What is entropy (randomness)?

- **Entropy:** how "random" a piece of data looks.
- Cryptographic keys look **high-entropy** (irregular bytes).

If an attacker sees a high-entropy block in a binary, they say "there might be a key here."

Function, branch, condition

- **Function:** a block of code that does one job (`erisim_ver`).
- **Branch:** a fork in the road, like an `if` .
- **Condition:** the expression that decides which way the branch goes.

Basic block

- **Basic block:** a sequence of instructions that runs straight through, with no branch.

- When an `if` is reached the block ends, and two new blocks begin.

Program = basic blocks wired together.

Control flow graph (CFG)

- **CFG** (Control Flow Graph): a diagram that makes basic blocks its **nodes** and the transitions between them its **edges**.
- It is the program's "road map."

Düzleştirme bu iskeleti tek bir dağıtıcıya indirger: 'hangi blok ne zaman' bilgisi kaybolur.

Let's tie the "basic block" and "CFG" ideas together with a concrete example. Take the small function below, which we will meet again in section 5:

denetim: a small function with three basic blocks

```
int denetim(int girdi) {
    int a = girdi + 1;      /* Block 1 */
    if (a > 10) {            /* Block 2 */
        return HATA;
    }
    return a * 2;             /* Block 3 */
}
```

Let's split this function into basic blocks: **Block 1** (`a = girdi + 1` and evaluating the `if` condition) runs straight through and ends the moment it reaches the `if` . **Block 2**, when the condition is `true` , goes to `return HATA` — a separate one-line block (**Block 2a**). When the condition is `false` , control moves to **Block 2b** (`return a*2`). The CFG's nodes are these three blocks (1, 2a, 2b), and its edges are the transitions between them:

Node (basic block)	Contents	Outgoing edges
Block 1	`a = girdi+1` ; evaluate `if (a>10)`	→ Block 2a (condition true) or Block 2b (condition false)
Block 2a	`return HATA`	(exit — no edge)
Block 2b	`return a*2`	(exit — no edge)

Total: **3 nodes, 2 edges**. When a decompiler draws this graph, it reads the logic — "error if the input is greater than 10, otherwise return double it" — **within seconds**, because there are few nodes, the edges are direct, and the branching decision is a single comparison. In section 5, when we flatten this exact function with K-04, we will see through this same example how the node/edge count grows and how this readability is destroyed — here you learn the definition, there you learn "why it exhausts the attacker."

Compiler flag

- **Compiler flag:** an option passed to the compiler (e.g., `-O2`, `-DGUNLUK_ACIK`).
- The same source can be compiled differently with different flags (e.g., with logging / without logging).

What is a log?

- **Log:** the informational messages a program writes while it runs (`printf("...")`).
- Problem: if sensitive information or internal state ends up in the log, the attacker reads it.

Now we're ready

We now know the following terms:

source · compiler · binary · decompilation · `strings` · symbol · debugger · XOR · entropy · branch · basic block · CFG · log

We will use these **constantly** for the rest of the week. Come back to this section whenever you get stuck.

Three frequently confused terms: "encoding," "encryption," "obfuscation"

These three are used interchangeably in everyday speech, but they are **technically different** things; mixing them up is one of the most common conceptual mistakes on the exam and in the project:

- **Encoding:** turning data into another representation (e.g., a simple XOR transform, Base64). The goal is **representation, not secrecy**; there is no mathematical security claim. If the key (e.g., `0x5A`) is found, it is reversed **instantly**.
- **Encryption:** a mathematically studied, key-based transformation providing provable security (e.g., the AES-GCM from week 3). Without the correct key it **cannot be broken** in a reasonable time with today's computers.
- **Obfuscation:** making code harder for **a human** to understand. There is no mathematical security claim; it only raises the cost in effort and time ("Rule 1" in section 1).

This distinction matters because the "string **encoding**" we meet in K-07 is exactly **encoding** (via XOR), not **encryption** — the decoding key also sits inside the binary, so no mathematical secrecy is claimed; it only slows down static tools such as `strings`. We will meet this again in section 3 as the rule "obfuscation does not store keys."

Quick glossary: this week's terms

This week's tools (Ghidra, IDA, KLEE, Tigress, Obfuscator-LLVM) and academic sources use a fixed set of English terms. The table below lists the terms that will come up throughout the week, alongside the fuller or acronym-expanded form you will meet in that tool documentation or literature:

Term (as used this week)	Full form / as seen in tools & papers
Obfuscation	Obfuscation
Reverse engineering	Reverse engineering
Decompiler	Decompiler
White-box attacker / attacker-at-the-end	White-box attacker / MATE (Man-At-The-End)
Control-flow flattening	Control flow flattening
Opaque predicate	Opaque predicate
Bogus operation	Bogus (dead) operation
Dead branch	(Bogus) death branch
Random exit point	Random exit point
Virtualisation-based obfuscation	Virtualisation-based obfuscation
Diversification	Diversification
Symbolic execution	Symbolic execution
Potency (metric)	Potency
Resilience (metric)	Resilience
Stealth (metric)	Stealth

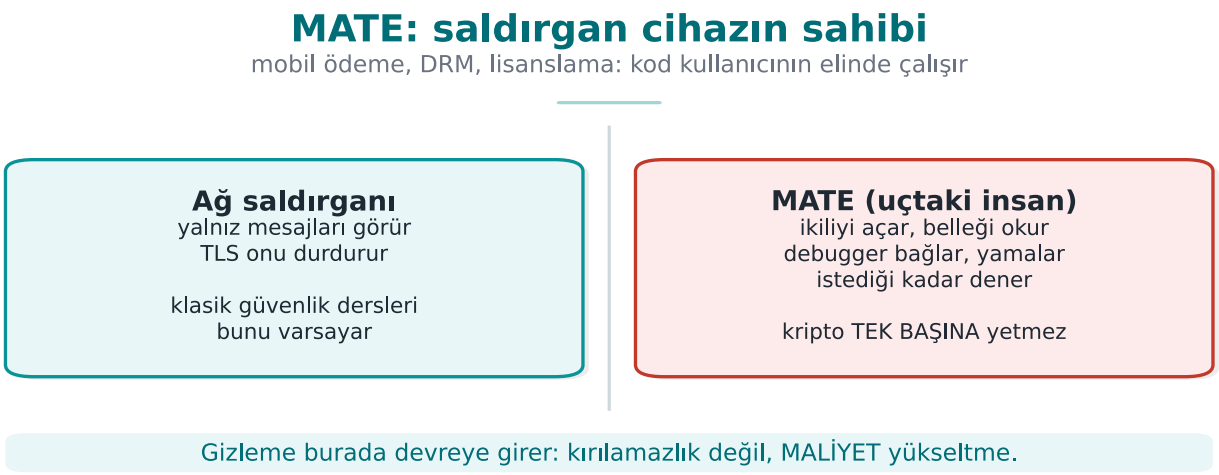
This week's question in one sentence

> I've handed the program to the user; **the attacker now owns it.** How do I make it **harder** for them to read and modify my code?

Answer: **code obfuscation rules.** Let's begin.

1. Why is obfuscation a security rule?

In the first week of the course we drew a line between attacker models. Against an attacker who sends input over the network, our defence was input validation, memory safety, and cryptography. But once we **deliver** a mobile app, a desktop program, or an embedded library to the user, the attacker now owns the program itself. In the literature this threat model is called **MATE** (Man-At-The-End) or the **white-box** model: the attacker can open the binary in a decompiler, single-step through it, read and modify memory, and try as many times as they like. No server-side security check applies here, because the code runs on the attacker's own machine.



A short history: where does code obfuscation come from?

- **1976** — Diffie & Hellman first discuss the idea of making a program "incomprehensible but working" (protection through **effort**, not through secrecy).
- **1997** — Collberg, Thomborson, and Low publish the first **obfuscation taxonomy** (layout · data · control flow · anti-analysis) and the *potency–resilience–stealth–cost* framework. The **five families** in this lecture come from here.
- **2001** — Barak et al. prove that "perfect (black-box) obfuscation is **impossible** in general" → obfuscation is therefore taught not as "unbreakability" but as **cost**.
- **2002** — Chow et al. launch the idea of embedding a key in software with **whitebox AES** (week 11).
- **2010s** — DRM, mobile banking, and the games industry make obfuscation mainstream; **Tigress** (Collberg) and **Obfuscator-LLVM** turn it into tooling (week 14).

So obfuscation is a **practical delaying discipline** built on top of an academic **impossibility** result.

What does "impossibility" mean? Understanding the Barak et al. result from scratch

Let's unpack the 2001 item in the timeline, because it is the foundation of why every rule we learn this week is **never marketed as "unbreakable."** "Perfect black-box obfuscation" means: turning a program into such a form that the attacker can learn **nothing** beyond what they could learn by feeding it inputs and observing its outputs — even if they open up the program and look inside, they gain no more knowledge than someone who only observes the input–output relationship. Barak and colleagues proved mathematically that such a perfect transformation cannot exist for **every** program: there exist specially constructed programs where looking inside **necessarily gives more** information than observing them from outside — no matter how good the transformation is.

This has two concrete consequences for our course:

1. No K-01–K-12 rule (nor any combination of them) can **claim** "the attacker can never learn anything" — such a claim has been mathematically disproved.
2. That's why this week we write every rule as "what it protects / **cost / limit**" (the template in section 4): the goal is not perfect secrecy, but the practical goals of the three rules in section 1 (raise the cost, break automation, buy time).

> ✓ **Rule: never use the word 'unbreakable' in a protection description**
>
> Describing a protection as "unbreakable" or "impenetrable" in your S9 or in a production document is both academically wrong (the theorem above) and it **damages your credibility** with an evaluator. Use measurable statements instead: "not measured against symbolic execution," "delays static scanning by X," "cannot be bypassed without modifying two independent points," and so on.

Honesty starts with one fact in this model: **an attacker with enough time, skill, and motivation eventually breaks every obfuscation.** The Cookbook says this outright (Recipe 12.1): anti-tampering measures are not "unbreakable"; their purpose is to **make breaking them uneconomical**. That is why we treat obfuscation not as a way of keeping a secret, but as a **cost-raising rule**:

- **Rule 1 — Push the cost above the value.** Make the attack more expensive than the value of the asset it protects. Guarding a 10-lira secret with 1000 lira of effort is irrational; so is the reverse.
- **Rule 2 — Break automation.** Make sure an attack that works on one copy does not automatically work on every copy. This is called **diversification** (this week's section 5, tooled in week 14).
- **Rule 3 — Buy time for the other layers.** Obfuscation is not a defence on its own; it is a layer that **buys time** for RASP (week 6), integrity checks, server-side checks, and key rotation. It only has to hold until the key is renewed or the version is updated.

> ❞ **The source's own language**
>
> The technical guide writes every protection measure under three headings: **Description** (what it protects), **Implementation** (how it's done), and **Example use** (where it is used), most often together with a **compliance requirement ID** (e.g., a payment scheme's "sensitive data is not logged in plaintext" item). We will write every technique this week with the same discipline, as a **rule**: *what it protects · how · cost · limit*.

A numeric example: what does "push the cost above the value" mean?

Let's not leave Rule 1 hanging in the air; let's make it concrete with a simple cost-benefit calculation. The figures are entirely **hypothetical** (not a real contract or product number); the point is only to show the reasoning:

1. Assume an experienced reverse engineer's "market value" per hour is **500 TL**.
2. In a binary with no obfuscation applied, finding a function whose name gives it away — such as `lisans_dogrula` — and working out its logic takes **10 minutes (⅙ of an hour)**. The attacker's cost: `500 × (1/6) ≈ 83 TL`.
3. When we apply the three rules we'll learn this week together (control-flow flattening K-04 + opaque predicate K-01 + string encoding K-07), the same analysis now takes **40 hours** (times like this are **measured** in real projects, not guessed — section 9). The attacker's cost: `500 × 40 = 20,000 TL`.
4. Now let's compare two scenarios:
   - If the protected asset (e.g., preventing a license key from being copied) is worth avoiding an **annual revenue loss of 5,000 TL**: `20,000 TL > 5,000 TL` — the attack is economically **pointless**, the rule has achieved its purpose.
   - If the protected asset is worth **100,000 TL a year**: `20,000 TL < 100,000 TL` — the attack is still **profitable**; these three rules alone are not enough, additional layers are needed (RASP — week 6, whitebox — week 11).

Common mistake: treating this calculation as a one-off, exact formula

The numbers above are meant to **illustrate a decision framework**, not to serve as real security proof. Concluding "the attacker's hour costs 500 TL, so we're safe" is wrong: a real attacker's skill, motivation (e.g., a competing company or a curious student), and available automated tools (section 10) radically change this time. **Rule:** use this calculation only to start the conversation about "which layers are worth adding"; never present it as a firm security guarantee. Week 13's attack-potential scoring turns this kind of estimate into something systematic.

This reasoning also lines up with the **risk = probability × impact** framework from week 1: here, "raising the cost" is a way of lowering the **probability** — the likelihood that a rational attacker bothers to try.

2. Obfuscation taxonomy: five families

The classification we bring into this course from Collberg and colleagues' software-protection body of work divides obfuscation into five families by **what it hides**. This map was introduced in week 4; here we make clear when each family "becomes a rule."

Family	What it hides	Example rules	Its place in this course
Layout	Names, format, metadata	Hide symbols, obfuscate function/file names, strip logging from release	Week 4 (introduction), this week (depth)
Data	Constants, strings, variables	String encoding, constant transforms, variable splitting/merging, opaque booleans	This week, section 6
Control flow	The algorithm's structure	Control-flow flattening, opaque predicates, bogus/dead branches, random exit	This week, section 5
Anti-analysis	The analysis tools' own work	Structures that mislead a decompiler, runtime code decoding	This week (concept), week 6 (RASP)
Virtualisation	The machine code itself	Turning a function into a custom virtual machine's bytecode	This week (concept), week 14 (Tigress)

In the field these families are used together

The code-hardening section of the source guide lists almost all of the families above **inside a single product**: switching off symbol visibility in a shared library; manually obfuscating function, file, and parameter names (the real names kept in internal documentation); transforming arithmetic instructions; encoding static strings before build and decoding them only at the moment of use, then wiping them immediately; opaque booleans that return only "success/failure"; bogus operations that don't change the result and dead branches that never run; control-flow flattening and a random exit point on failure; stripping logging from the release build. **None of these is strong on its own; their strength comes from being used together, and together with RASP.** The Cookbook's recommendation is the same (Recipe 12.1): rather than perfecting a single advanced technique, it's more effective to combine several simple techniques together with data hiding.

Worked example: seeing all five families in one function

Let's not leave the taxonomy abstract. In the small (synthetic) function below, let's mark which line belongs to **which family** with comments:

Five families in one function (concept demonstration)

```
/* LAYOUT: the function name isn't 'esik_hesapla' but the meaningless 'f7' */
int f7(int x) {
    /* DATA: the 0x2A constant isn't written directly, it's produced from two parts (K-02, section 6)
*/
    uint8_t esik = (uint8_t)(0x37 ^ 0x1D);

    /* CONTROL FLOW: branching via an opaque predicate (K-01, section 5) */
    if (((x * (x + 1)) & 1u) == 0u) {
        return esik;          /* real path: since x*(x+1) is always even, this is ALWAYS entered */
    }
    return 0;                 /* ANTI-ANALYSIS dead branch: never runs, just distracts the analyst */
}
```

The fifth family — **virtualisation** — is absent from this small example, because virtualisation is applied not to a single line but to **an entire function's machine code** (section 7, K-10). This example exists to show one thing: in real code the five

families live **intertwined**; being able to ask, while reading a line, "whose job is this?" is what makes the source's "they should all be used together" warning concrete.

Common mistake: applying one family and thinking you've 'obfuscated' the code

A beginner developer typically applies only the **layout** family (obfuscating names) and considers the job done. But even with symbol names hidden, the function's **logic** (control flow and data) is still plainly readable — even if you don't see the name `f7` in a decompiler's output, you'll understand its `if` / `return` structure within seconds. **Rule:** a protection decision must cover at least **control flow + data** together; the layout family alone was taught in week 4 as an "entry-level" measure — this week is where you learn that it is **insufficient** on its own.

Why this order? Why do potency and cost rise from bottom to top?

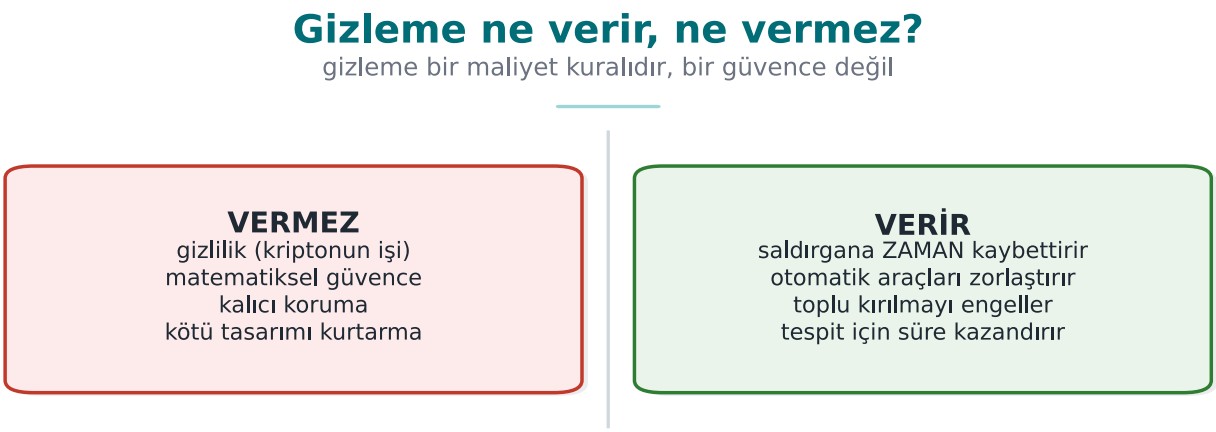
Let's unpack the "potency and cost rise from bottom to top" note on the diagram (h09-01) — this is not an arbitrary ordering, it is tied directly to **what each family changes**:

- **Layout** changes only **names/metadata**; it touches **nothing that produces the program's behaviour**. That's why it's cheap to apply (a compiler flag or a name list) but also low in potency — as we just saw, the logic stays exactly the same.
- **Data** and **control flow** change **what** the program processes and **in what order**; that requires both more engineering effort (reworking every constant, every branch) and more runtime cost (extra instructions) — their potency rises accordingly.
- **Anti-analysis** techniques (such as the function substitution in K-06) target the analysis **tools themselves**; this is more laborious than just hiding data/flow, because you have to guess which tool will be used.
- **Virtualisation** changes the program's **machine code itself** — the original instructions no longer exist at all, replaced by an entirely different instruction set (VM bytecode). This is the most radical change; that's why it is both the strongest and (as we'll see in K-10, an increase too large to state in percentages, more a multiplier) the most expensive option.

In short: the closer a family sits to **the mechanism that produces the behaviour**, the stronger — and the more expensive — hiding it becomes. Recall the four metrics from section 9: this ordering is really the taxonomy's reflection of the **potency ↔ cost trade-off**.

3. What does obfuscation give, and what does it not give?

Applying a rule correctly requires knowing its limits. Here is what obfuscation gives, and what it does **not** give:



What obfuscation gives	What obfuscation does not give
A delay against static analysis (`strings`, symbol table, decompilation)	The mathematical secrecy of a secret — a key needs **cryptography**
Resistance against automated attack (via diversification)	Protection against reading a running program's memory — that needs **RASP** and **whitebox** (week 11)
Time for the other layers	Permanent security — there is no such thing as "unbreakable"
A measurable increase in cost	Free protection — every obfuscation comes with a **maintenance and performance** cost

The most common mistake: mistaking obfuscation for storing a key

A key embedded or "hidden" in code — hidden or not — is still **there**. The program has to use the key while it runs; at that moment it is exposed in memory, and the decoding logic is inside the program too. Obfuscation slows down **static** searches like `strings`; real key protection needs **whitebox cryptography** (week 11) or hardware (an HSM, a secure element). We stressed this same distinction in weeks 3 and 10: **secrecy is cryptography's job, not obfuscation's.**

Step by step: an attacker's first 10 minutes — before and after obfuscation

Let's turn the "gives/does not give" table into a timeline. Consider both versions: **unprotected** and **protected with this week's rules**.

An attacker's first 10 minutes with an unprotected binary:

1. They run `strings program.exe` → strings such as `"CEN429-OK"`, `"Erisim reddedildi"` are directly visible (minute 0).
2. They open a decompiler and see names like `erisim_ver`, `lisans_dogrula` in the symbol table (minute 2).

3. They read the function's CFG: a single `if/else`, and it is obvious at a glance which branch means "allow" and which means "deny" (minute 5).
4. They find the comparison value (`"CEN429-OK"`), test it with their own input, and confirm it (minute 10). **Job done.**

The same 10 minutes with a binary protected by this week's rules:

1. They run `strings` → thanks to K-07 the strings are invisible; they come away with no clue (minute 0).
2. They look at the symbol table → thanks to K-06 and the layout family there is no recognisable name (minute 2).
3. They open the CFG → because of the K-04 flattening they see a single `switch` dispatcher; it is not clear which `case` means "allow"; K-01's opaque predicates hide which branch is real and which is dead (minute 10, **still unsolved**).
4. At this point the attacker either switches to an automated tool such as symbolic execution (section 10 — this takes not minutes but **hours**) or gives up.

These two timelines are the **time-spread** version of the table's four rows: obfuscation doesn't say "never breakable," it says "hours or days instead of 10 minutes."

Common mistake: treating obfuscation as a substitute for a penetration test

A team might say "we obfuscated our code, so we no longer need security testing." This is wrong: obfuscation only makes **static** reading harder; **dynamic** analysis that examines a running program's behaviour (input/output, network traffic, memory) and an independent **penetration test/assessment** (weeks 12–13) are still required. **Rule:** obfuscation is planned not as a **replacement** for security testing, but as **a layer that delays the weaknesses testing will find**.

4. The "protection rule" template

We will evaluate every technique this week with the same four questions. We expect you to write the S9 section of your own project with this same template:

```
RULE K-xx: <name of the technique>
  What does it protect? : which asset / which code section (e.g. licence check, key derivation)
  Against which threat? : static analysis / dynamic analysis / automated attack / tampering
  How?                  : concept-level application (one sentence)
  Cost                  : size, speed, maintenance, debugging difficulty
  Limit                 : what it doesn't protect, which attack it doesn't withstand
  Measurement           : how we verify its effectiveness (see section 5)
```

The value of this template is this: obfuscation is not a "yes/no" attribute. An evaluator (week 13, attack potential scoring) asks **how much** each layer delays the attack. Without the "Measurement" line above, you cannot defend a protection decision.

Worked example: let's fill the template with this week's demo

So this doesn't stay abstract, let's fill the template exactly, using the **real measurement** numbers given in the "This week's working demo" box at the start of this section (K-04 as applied in the demo):

```
RULE K-04-applied: control-flow flattening applied to the erisim_ver function
  What does it protect? : access-check flow order (where/how the CEN429-OK comparison is done)
  Against which threat? : static analysis (reading the CFG in a decompiler), single-byte/single-
branch patch attack
  How?                  : switch-based dispatcher (K-04) + opaque predicate (K-01) + random exit (K-
05)
  Cost                  : instruction count 28 → 51 (+82%), branch count 3 → 6 (+100%) — measured
with objdump
  Limit                 : flattening only; can be undone via symbolic execution (section 10); data
still needs K-07
  Measurement           : instruction/branch count comparison in objdump output (this week's demo, in
the step-by-step guide)
```

Let's verify the percentages in the cost line: the increase in instruction count is `(51 - 28) / 28 × 100 ≈ 82.1%`; the increase in branch count is `(6 - 3) / 3 × 100 = 100%` (exactly doubled). This is concrete proof that the "Measurement" line must be **a number, not a claim** — we will reuse these figures again in section 9.

Common mistake: filling in only the template's 'How' line and leaving 'Measurement' blank

Students typically write which technique they applied (How) but skip the **cost** and **measurement** lines. The evaluator's note in section 12 says this explicitly: an unmeasured protection counts as a "claim," not evidence. **Rule:** a protection is not considered written into S9 until **all six** lines are filled in; at least one line, "Measurement," must contain a concrete number or observation (e.g., "string X no longer appears in `strings` output").

A second worked example: let's fill the template for a different asset

To see that the template fits any kind of asset, let's fill it this time for a **data** protection rule we'll meet in section 6 (a string, not a function):

```
RULE K-07-applied: string encoding applied to the error-message string
  What does it protect? : readable text sitting in the binary, such as "Erisim reddedildi"
  Against which threat? : strings scanning (section 3's "attacker's first 10 minutes")
  How?                  : encode with XOR 0x5A, decode at the moment of use, wipe immediately with
memset_s (section 6)
  Cost                  : low — a few extra instructions, no measurable speed loss
  Limit                 : the decoding key sits in the binary; not real key protection (section 3's
rule)
  Measurement           : count of the relevant string in strings output: 1 before encoding, 0 after
```

Placing the two examples (K-04-uygulama and K-07-uygulama) side by side makes the difference clear: **control-flow** rules are usually measured with "instruction/branch count," **data** rules are usually measured with "visible string/constant count" — but in both cases the template's six lines are filled with **the same discipline**. In your S9 you are expected to fill out this template separately for at least one control-flow rule and one data rule.

---

5. Control-flow rules (advanced)

A compiled function's **control flow graph** (CFG) shows the skeleton of the algorithm: which check happens in what order, which branch leads to success. The white-box attacker's first job is to extract this skeleton. Control-flow obfuscation rules make this skeleton unreadable. Week 4 introduced flattening; here we add the rules that **strengthen** it. All examples are synthetic and are told through a single small check (`erisim_ver`).

RULE K-01 — Opaque predicates and opaque loops

What does it protect? Which condition a branch is taken under; how many times a loop runs. **How?** Branching is done on a value that the programmer knows but that an analysis tool cannot easily resolve (an **opaque predicate**). A classic example is an arithmetic identity that is always true but hard to prove statically:

Opaque predicate: always true, but static analysis can't easily prove it

```c
/* x*(x+1) is always even → (x*(x+1)) % 2 == 0 is always true. */
static int opak_dogru(unsigned x) { return ((x * (x + 1)) & 1u) == 0; }

if (opak_dogru(sayac)) {          /* real path: this is ALWAYS entered */
    durum = gercek_adim(durum);
} else {
    durum = sahte_adim(durum);    /* dead path: never runs, but looks real under analysis */
}
```

Why is `x*(x+1)` always even? (let's see it step by step)

Of two consecutive integers, **one is always even**; anything multiplied by an even number is even. In numbers:

`x`	`x + 1`	`x * (x+1)`	Even?
0	1	0	yes
1	2	2	yes
2	3	6	yes
3	4	12	yes
4	5	20	yes
7	8	56	yes

So `(x * (x+1)) & 1` is **always 0**, which means `opak_dogru(x)` returns `true` **for every x**. While the program runs it always takes the "real path."

So why do we call it 'opaque'? Because *we* know it; **the static analysis tool doesn't**. When the tool looks at the code without running it, it has to prove for every value of `x` that this product is even — that can't be solved by a simple constant-folding step. So the tool has to treat **both branches as possible** and is forced to analyse the bogus branch too. The analyst's work doubles, while the program's behaviour never changes.

Know its limit too

This is a **classic** pattern; modern tools recognise it in their libraries. That's why in week 14 we generate opaque predicates that are **seed-diversified** with Tigress and harder to crack — repeating the same pattern in every build reduces resilience.

The guide writes opaque loops as a separate rule: loops (for, while) are never shown directly; they are **hidden inside control-flow flattening**, the body is split into small blocks, and iteration is driven by a state variable. That way, the information "this loop runs N times" cannot be read off the CFG.

Cost: low–medium (extra arithmetic and a branch). **Limit:** well-known opaque predicate patterns (libraries) can be recognised automatically; hence they need to be diversified. **Measurement:** basic block count before/after flattening.

Worked example: let's trace an opaque loop bound by hand. Let's verify the guide's claim that "the loop bound cannot be read off the CFG" with numbers. Let our real iteration bound be `3`, but instead of writing it directly, let's wrap it in a second **always-true** opaque predicate:

Opaque loop: the real bound (3) isn't explicitly visible in the code

```
static int hep_dogru(unsigned y) { return ((y * y + y) & 1u) == 0u; }  /* y*(y+1) is always even — see above */

unsigned i = 0;
while (hep_dogru(i) && i < GERCEK_SINIR) {   /* GERCEK_SINIR = 3, defined in a separate constant (can be encoded with K-02) */
    isle(i);
    i++;
}
```

Let's trace it by hand (`GERCEK_SINIR = 3`):

Step	`i`	`hep_dogru(i)` calculation	Result	`i < 3`?	Does the loop body run?
1	0	`0*0+0=0`, `0 & 1 = 0`	true	yes	yes → `isle(0)`, `i=1`
2	1	`1*1+1=2`, `2 & 1 = 0`	true	yes	yes → `isle(1)`, `i=2`
3	2	`2*2+2=6`, `6 & 1 = 0`	true	yes	yes → `isle(2)`, `i=3`
4	3	`3*3+3=12`, `12 & 1 = 0`	true	**no** (`3 < 3` is false)	**no**, the loop ends

Result: the loop runs exactly **3 times** (`i = 0, 1, 2`), exactly as expected. But notice: `hep_dogru(i)` **always** returns `true` — so an analysis tool, to understand when the loop terminates, has to track not `hep_dogru` but the comparison `i < GERCEK_SINIR`; and to **prove** that `hep_dogru` is always true, it also has to solve the integer identity above. Two separate analysis burdens land on the same line.

RULE K-02 — Encoding arithmetic instructions

What does it protect? Constants and simple computations; directly readable hints such as "this number is 0x1F, this operation is an XOR." **How?** An operation is replaced with an equivalent expression that produces the same result but looks more complex (mixed boolean-arithmetic, MBA). For example, `a + b` is equal to `(a ^ b) + 2*(a & b)`; `x * 2` equals `x << 1`. Constants are never written directly; they are produced at runtime by a small computation:

Constant transform: 0x2A isn't directly visible

```
/* Produce it from two parts instead of 0x2A; a '0x2A' search in the binary returns nothing. */
static uint8_t esik(void) { uint8_t a = 0x37, b = 0x1D; return (uint8_t)(a ^ b); } /* = 0x2A */
```

Cost: low. **Limit:** modern decompilers and tools like `arybo` / `msynth` can simplify many MBA expressions; it is weak on its own and gains meaning together with control-flow obfuscation. **Measurement:** a `strings` /byte search for the constant should return nothing.

Worked example: let's verify two MBA identities by hand. Let's confirm the rule's claim that `a + b` ↔ `(a ^ b) + 2·(a & b)` **with numbers** (`a = 5` , `b = 3`):

Expression	Calculation	Result
`a + b`	`5 + 3`	`8`
`a ^ b`	`0101 ^ 0011 = 0110`	`6`
`a & b`	`0101 & 0011 = 0001`	`1`
`(a ^ b) + 2·(a & b)`	`6 + 2·1 = 6 + 2`	`8`

Both give `8` — the identity is confirmed. This is a valid algebraic identity for every 4-bit `a,b` pair (the `2·(a&b)` term compensates for the carry bit); if the compiler produces this instead of `a+b` , the binary shows no plain "addition" — it shows three separate instructions (XOR, AND, shift+add).

Using the same logic, let's also verify the `0x2A` constant produced by the rule's `esik()` function (`a = 0x37` , `b = 0x1D`):

Bit	`0x37`	`0x1D`	XOR
7–4	`0011`	`0001`	`0010`
3–0	`0111`	`1101`	`1010`

Result: `0010 1010` = `0x2A` — exactly matching the code block's comment. Inside the binary, the `0x2A` value never sits **anywhere** as a single byte; only `0x37` and `0x1D` sit there, so a `strings` /byte search for `0x2A` **returns nothing**.

Common mistake: applying the transform only to compile-time constants

If you write the `esik()` example as a preprocessor macro like `#define ESIK ((0x37) ^ (0x1D))` , the compiler will **constant-fold** this expression at compile time and embed `0x2A` directly into the binary — the MBA transform exists in the source, but **not in the binary**. **Rule:** verify in the compiled binary (`objdump` / `strings`) that the transform is really computed **at runtime**, not simplified away at compile time; the "compiler flag" distinction from section 0 applies here too — test with optimisation turned on.

RULE K-03 — Bogus operations and dead branches (the two are different)

The guide writes these two as **separate** rules, and the difference matters:

Sahte işlem ≠ ölü dal

ikisi de sahte iş üretir ama farklı biçimde

SAHTE İŞLEM (bogus op)
ÇALIŞIR
sonucu kullanılmaz

analizciyi ve aracı yorar
maliyet: hafif yavaşlama

ÖLÜ DAL (dead branch)
HİÇ ÇALIŞMAZ
opak yüklemle korunur

analizde gerçek görünür
maliyet: boyut

Sınav sorusu: hangisi çalışır? → Sahte işlem çalışır, ölü dal çalışmaz.

	Bogus operation	**Dead (death) branch**
Does it run?	**Yes**, it runs but doesn't change the result	**No**, it never runs (protected by an opaque predicate)
Purpose	Hide real operations in a crowd	Show the analyst a fake execution path
Example	Adding `x ^ x` to a CRC computation (result unchanged)	The code inside `if (opak_yanlis()) { ... }`

Bogus operation: doesn't change the result, just crowds the code

```
crc = crc32_guncelle(crc, veri, n);
crc ^= (sabit ^ sabit);   /* = crc; bogus operation, result unchanged */
```

What does it protect? The real logic, by surrounding it with meaningless-but-plausible-looking code. **Cost:** low (bogus operation)–medium (dead branches grow the CFG). **Limit:** an analysis tool that performs dead code elimination can drop a dead branch once the opaque predicate protecting it is solved; so it depends on the predicate's quality. **Measurement:** how long it takes to tell dead branches apart from real ones.

Worked example: does the bogus operation really leave the result unchanged? Let's verify the `crc ^= (sabit ^ sabit)` line above with a number (let `sabit = 0x11`): `sabit ^ sabit = 0x11 ^ 0x11 = 0x00` (any value XORed with itself always gives `0` — the same XOR identity from section 0). The operation `crc ^= 0x00` leaves `crc` **unchanged** (`x ^ 0 = x`). So no matter what `sabit` is, this line is mathematically a no-op every time; its crowding effect lives only in the **compiled instruction count**, not in `crc`'s value.

Common mistake: letting the compiler silently delete the bogus operation

An optimising compiler (`-O2` , `-O3`) may spot an operation whose result is used nowhere and can be **proven** mathematically to have no effect, classify it as dead code, and **remove it entirely**. A line such as `crc ^= (sabit ^ sabit)` — if the compiler constant-folds it — becomes `crc ^= 0` , which it then sees does nothing and deletes — the obfuscation vanishes without you noticing. **Rule:** values used in bogus operations must be determined **at runtime** (e.g., tied to an input or an opaque predicate), never fixed at compile time; also always verify obfuscation on a binary compiled **with optimisation on** (using the compiler flag defined in section 0) — "present in the source, gone from the binary" is a common surprise.

RULE K-04 — Control-flow flattening (in depth)

What does it protect? The algorithm's **structure**: which blocks naturally neighbour each other and "which block leads to which." **How?** In the guide's words, **vertical** control flow is turned **horizontal**: all basic blocks are moved into a `switch` dispatcher inside a single loop; the sequence is read only from a **state variable**.

Flattening template (from week 4; here we strengthen it)

```
int durum = BASLA;
for (;;) {
    switch (durum) {
    case BASLA:  durum = ADIM1; break;
    case ADIM1:  durum = kosul ? ADIM2 : HATA; break;
    case ADIM2:  durum = BITIR; break;
    case HATA:   return RED;
    case BITIR:  return IZIN;
    default:     return RED;          /* random exit lands here (K-05) */
    }
}
```

We said back in week 4 that this template alone can be worked out fairly quickly by an experienced analyst. The "on the native side this flattening is strengthened with several security features" the guide talks about is exactly the other rules:

- **State values are hidden** (K-02): instead of `case 1, 2, 3` , scattered, arithmetically produced values; no visible sequence.
- **Bogus and dead blocks** are added (K-03): telling which `case` is real becomes harder.
- **Random exit** (K-05): failure exits through the `default` branch, with an unpredictable state.
- **Branching on opaque predicates** (K-01): `case` transitions depend on opaque conditions, not fixed values.

Cost: medium–high (block and branch counts grow a lot, performance drops). **Limit:** flattening alone can be undone by symbolic execution (section 5); its strength comes from the reinforcements. **Measurement:** CFG node/edge count in a decompiler, time required to trace a single check.

Worked example: let's flatten a three-block function by hand. Let's start with the following ordinary (unflattened) function:

Before flattening: natural (vertical) flow

```c
int denetim(int girdi) {
    int a = girdi + 1;      /* Block 1 */
    if (a > 10) {            /* Block 2 */
        return HATA;
    }
    return a * 2;             /* Block 3 */
}
```

Let's move these three blocks by hand into the flattened form using K-04's template:

After flattening: the same three blocks, inside a single switch

```c
int denetim(int girdi) {
    int durum = B1, a = 0;
    for (;;) {
        switch (durum) {
        case B1: a = girdi + 1; durum = B2; break;
        case B2: durum = (a > 10) ? B_HATA : B3; break;
        case B3: return a * 2;
        case B_HATA: return HATA;
        default: return HATA;      /* random exit lands here (K-05) */
        }
    }
}
```

Let's trace `girdi = 4` by hand: `durum=B1` → `a = 4+1 = 5`, `durum=B2`. `durum=B2` → `a(5) > 10` is false, `durum=B3`. `durum=B3` → `return 5*2 = 10`. **Both versions return the same `10` for the same `girdi`** — the behaviour is preserved, only the **shape** of the flow has changed: in the natural version the three blocks read one after another, while in the flattened version they all look like equal-distance branches of a single `switch`, and the **natural adjacency** information between them (B1 is always followed by B2) has been erased from the CFG; it can now only be read from `durum`'s value at runtime.

Common mistake: leaving state values sequential

Using sequential constants like `B1=1, B2=2, B3=3, B_HATA=4` makes the flattening **look** present but actually weak: an analyst can read the `case` order and easily guess the original flow. **Rule:** state values should be produced with K-02 (arithmetic encoding) or with random, scattered constants; they must not be consecutive integers — otherwise the flattening has only added **visual noise**, not real protection.

RULE K-05 — Random exit from control flow

What does it protect? Where a check fails. Attackers often look for the "failure branch" and try to turn it into the "success branch." **How?** As the guide describes it: when a check fails, the program does not go straight to a `return RED` line; instead the state variable is set to an **unpredictable, large value** and the function exits through the `default` branch. Successful exit can use the same trick, so success and failure look alike in the flow.

Random exit: the failure point can't be read off the flow

```c
if (!imza_gecerli(p)) {
    durum = kararsiz_deger();   /* a value not defined in the switch → default */
    break;
}
```

Cost: low. **Limit:** meaningful only together with flattening, not on its own. **Measurement:** time needed for a "find and flip the failure branch" attack; ideally a single byte patch should not be enough to bypass the check.

Common mistake: using the same sentinel value for every failure state

If `kararsiz_deger()` always returns the same constant (e.g., always `0xDEADBEEF`), an attacker can `grep` for that constant once in the binary and find **all** the failure points at once — the random exit's purpose is defeated. **Rule:** the sentinel value must be **different** at every call site (either generated from a seed as in K-11, or a separate constant per call site); otherwise a "random exit" is really just **one fixed signature**, violating the diversification principle from section 8.

RULE K-06 — Hiding function calls and external library dependencies

What does it protect? Hints like "this function calls `memcmp`, so it must be doing a comparison." Standard library calls tell the attacker directly what a function does. **How?** On critical paths, standard library functions are replaced with **your own internal versions** (e.g., your own constant-time `esit_mi` function), so the linker never shows a recognisable name. The guide also lists **hiding function parameters** and **adding bogus parameters** as separate rules: unused fake parameters are placed alongside the real ones, making the signature misleading to an attacker.

Cost: low–medium (your own versions need maintenance). **Limit:** behavioural analysis can still reveal what it does; this is a **delaying** rule. **Measurement:** count of recognisable library calls in the binary.

Worked example: your own constant-time comparison instead of `memcmp`. The rule tells us to replace the standard `memcmp` because it is recognisable; but the real payoff is closing a familiar side channel from week 3:

Constant-time comparison (concept)

```
static int sabit_zamanli_esit_mi(const uint8_t *a, const uint8_t *b, size_t n) {
    uint8_t fark = 0;
    for (size_t i = 0; i < n; i++) {
        fark |= (uint8_t)(a[i] ^ b[i]);   /* every byte is processed, does NOT stop at the first
mismatch */
    }
    return fark == 0;                      /* a single comparison, at the end of the array */
}
```

Let's trace it for `a = {0x41,0x42,0x43}`, `b = {0x41,0x00,0x43}`: `i=0`: `0x41^0x41=0x00`, `fark = 0x00`. `i=1`: `0x42^0x00=0x42`, `fark = 0x00 | 0x42 = 0x42`. `i=2`: `0x43^0x43=0x00`, `fark = 0x42 | 0x00 = 0x42`. The loop **always runs all three steps**, never returning early even though the mismatch is found at `i=1`; at the end `fark = 0x42 ≠ 0`, and `esit_mi` returns `0` (false). A standard `memcmp` or a hand-written `for (...) if (a[i]!=b[i]) return 0;` would **return early** at `i=1` — a classic timing channel that can leak **where** the mismatch occurred through runtime (the same `CRYPTO_memcmp` rule from week 3). K-06's payoff is twofold: the `memcmp` name no longer appears in the binary, and the comparison is constant-time.

A bogus parameter example. The rule's second part — bogus parameters — makes the signature misleading:

Bogus parameter: the signature becomes misleading

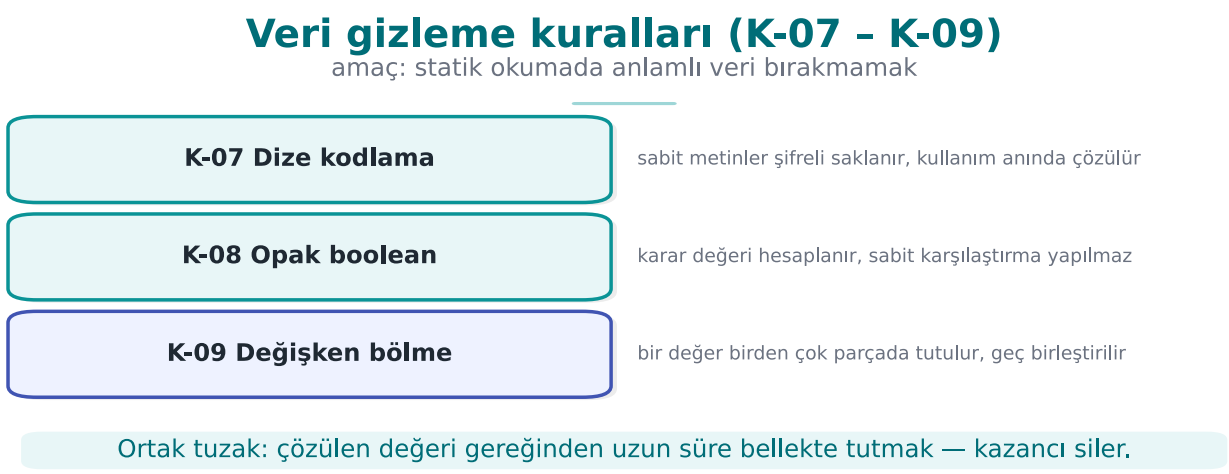
```
/* Real signature: int erisim_kontrol(const char *kod); */
int erisim_kontrol(const char *kod, int gunluk_seviyesi, void *ayarlar) {
    (void)gunluk_seviyesi;   /* unused — bogus parameter */
    (void)ayarlar;           /* unused — bogus parameter */
    return sabit_zamanli_esit_mi((const uint8_t *)kod, GERCEK_KOD, GERCEK_KOD_UZUNLUK);
}
```

A decompiler will show this function as taking three parameters; the attacker wastes time asking "is the second parameter really a log level, is the third really a settings pointer used?" — when in fact neither is **ever** used. The `(void)parametre` lines silence the compiler's "unused parameter" warning; in real projects, unless someone searches for them, these lines don't give away that they're bogus.

> ⚠ **Rule: don't break constant-timeness**
>
> When you write your own comparison/crypto version, make sure it stays **constant-time** (week 3). Writing an early-exit comparison for the sake of obfuscation opens up a **side channel**; don't create a new hole while adding a protection.

6. Data obfuscation rules

Control flow hides the algorithm's structure; **data obfuscation** hides the **values** the program processes: strings, constants, tables, and variables. The Cookbook's anti-tampering chapter and the technical guide list the same rules here.



RULE K-07 — Encoding static strings

What does it protect? The readable text inside a binary. The cheapest first step of reverse engineering is running `strings`; `"Lisans gecersiz"` or a URL tells the attacker exactly where to look. **How?** Sensitive strings are **encoded before compilation** (e.g., encrypted with an XOR key or a generation script), sit encoded in the binary, are **decoded only at the moment of use**, and are **wiped from memory** the instant the job is done.

String encoding: use then wipe immediately (synthetic)

```
static const uint8_t GIZLI[] = { 0x3B,0x2A,0x2E,0x2E,0x2D };   /* not "merhaba"; synthetic */
void kullan(void) {
    char tmp[sizeof GIZLI];
    for (size_t i = 0; i < sizeof GIZLI; i++) tmp[i] = GIZLI[i] ^ 0x5A;  /* decode */
    isle(tmp, sizeof GIZLI);
    memset_s_benzeri(tmp, sizeof tmp);   /* wipe immediately after use (week 3) */
}
```

Cost: low. **Limit:** once the program is running, the decoded string is visible in memory; this is a **static-scan** countermeasure. The decoding key also lives in the binary — that's why in the field the key is **split and distributed**, and the decoding function is protected with additional checks. **Measurement:** the sensitive string **must not be found** in `strings` output.

Worked example: let's decode the `GIZLI` array above by hand. Let's decode the array `GIZLI = {0x3B, 0x2A, 0x2E, 0x2E, 0x2D}` from the code block one byte at a time, XORing each byte with `0x5A` (the same method as the bit table in section 0):

Index	`GIZLI[i]`	`^ 0x5A` (bit computation)	Result (hex)	ASCII
0	`0x3B` = `0011 1011`	`0011 1011 ^ 0101 1010`	`0110 0001` = `0x61`	`'a'`
1	`0x2A` = `0010 1010`	`0010 1010 ^ 0101 1010`	`0111 0000` = `0x70`	`'p'`
2	`0x2E` = `0010 1110`	`0010 1110 ^ 0101 1010`	`0111 0100` = `0x74`	`'t'`
3	`0x2E` = `0010 1110`	(same computation)	`0x74`	`'t'`
4	`0x2D` = `0010 1101`	`0010 1101 ^ 0101 1010`	`0111 0111` = `0x77`	`'w'`

The decoded byte sequence is `"apttw"` — consistent with the code comment "not `merhaba`; synthetic": we're showing the **mechanism**, not a real string. What matters is this: the bytes sitting in the binary are `{0x3B, 0x2A, 0x2E, 0x2E, 0x2D}`; `strings` never runs XOR on these bytes, it just tries to read them as-is and finds no readable text (these five bytes may fall outside the printable ASCII range, or simply look meaningless) — the attacker's first and cheapest step (the "first 10 minutes" from section 3) comes up **empty** here.

Common mistake: assuming `memset` 'wiped' the secret

The code example's `memset_s_benzeri(tmp, sizeof tmp)` line specifically calls a **non-ordinary** function, not a plain `memset`; the reason is compiler optimisations. If a variable goes out of scope right after it's used (the function ends), an optimising compiler may perform **dead store elimination**, deciding "this `memset` call has no observable effect" and deleting it — the decoded secret stays **unwiped** in memory. **Rule:** use a call the compiler cannot delete when clearing sensitive memory (`memset_s`, `SecureZeroMemory`, `explicit_bzero`, or a hand-written loop over a `volatile` pointer); using plain `memset` and assuming "I wiped it" is as real a mistake as the obfuscation itself.

Rule: don't confuse string obfuscation with storing a key

String obfuscation stops static scans like `strings`; it is not meant to protect a **key**. This is the concrete form of the "obfuscation does not store keys" rule from section 1. For real keys, use whitebox (week 11) or hardware.

RULE K-08 — Constant transforms, opaque booleans, and function boolean returns

The guide makes a separate rule out of how security results are represented. Returning a check's result as a plain `0` / `1` is dangerous: an attacker can flip a single byte and turn "failure" into "success." The **opaque boolean** rule keeps a true/false value tied to **multiple values** in a way that cannot be flipped by patching a single byte. Functions likewise return this kind of **opaque return code** instead of a plain boolean.

Opaque boolean: a single-byte patch can't flip the result (concept)

```
/* The "allow" value isn't a single constant; it's derived from two states, and the caller verifies
both. */
typedef struct { uint32_t a, b; } Karar;
static Karar izin_ver(void)  { return (Karar){ 0xA3C1u, 0x5C3Eu }; } /* a ^ b == 0xFFFF */
static int  karar_izin_mi(Karar k) { return (k.a ^ k.b) == 0xFFFFu; }
```

Cost: low. **Limit:** it can be solved given enough scrutiny; its purpose is to block a single-byte patch and crude branch-flipping. **Measurement:** number of independent points that must be changed together in a "flip the result" attack, > 1.

Worked example: is `0xA3C1 ^ 0x5C3E` really `0xFFFF`? Let's verify the code example's claim nibble by nibble:

Nibble	`0xA3C1`	`0x5C3E`	XOR
15–12	`A` = `1010`	`5` = `0101`	`1111` = `F`
11–8	`3` = `0011`	`C` = `1100`	`1111` = `F`
7–4	`C` = `1100`	`3` = `0011`	`1111` = `F`
3–0	`1` = `0001`	`E` = `1110`	`1111` = `F`

All four nibbles come out `F` → `a ^ b = 0xFFFF`, the claim is confirmed. Every nibble is `F` for a simple reason: the two values were chosen to be each other's **bitwise complement** (`A=1010` ↔ `5=0101`, `3=0011` ↔ `C=1100`, etc.).

Now let's test the rule's claim that "it cannot be flipped by a single-byte patch": suppose an attacker changes only the **low byte** of `a` from `0xC1` to `0x00` (`a` becomes `0xA300`). New result: `0xA300 ^ 0x5C3E`:

Nibble	`0xA300`	`0x5C3E`	XOR
15–12	`A` = `1010`	`5` = `0101`	`F`
11–8	`3` = `0011`	`C` = `1100`	`F`
7–4	`0` = `0000`	`3` = `0011`	`3`
3–0	`0` = `0000`	`E` = `1110`	`E`

Result: `0xFF3E` — **not equal** to `0xFFFF`. The `karar_izin_mi` function's `(k.a ^ k.b) == 0xFFFFu` check now returns `false`: a crude single-byte patch **fails** to trigger "allow." For the attack to work, both `a` and `b` must be changed **consistently with each other** — meaning the attacker has to find and change at least two independent points at once, not one; this is exactly the numeric meaning of the "Measurement" line's "number of independent points > 1."

Common mistake: leaving the comparison constant (`0xFFFF`) the same everywhere

Strong as this mechanism looks on its own, if **every** opaque boolean check in the project compares against the same `0xFFFFu` constant, an attacker can find that constant once in the binary and get a sense of the pattern behind **all** the allow points — the same mistake as "using the same sentinel everywhere" in K-05. **Rule:** use a **different** target constant for each function/check (e.g., `0xA5C3` , `0x3D71` , ...); the constant itself can also be encoded with K-02.

RULE K-09 — Variable splitting, merging, and array restructuring

What does it protect? The recognisable footprint of a sensitive variable in memory. Patterns like "a 32-byte block = an AES key" point the attacker straight at it. **How?** A variable is **split** into two parts (e.g., computing a 32-bit value from two 16-bit shares), several variables are **merged** into a single word, arrays are rearranged; buffer sizes, order, and layout are changed, bogus padding is inserted between them, and buffers are filled with random values after use.

Cost: low–medium. **Limit:** it only slows down static/pattern analysis. **Measurement:** how long it takes to recognise the sensitive buffer from its memory pattern.

Worked example: let's split a 32-bit value and merge it back. Say the value we want to store is `0x1234ABCD` (32-bit). Instead of keeping it as a single 4-byte block, let's split it into two separate 16-bit **shares**:

Variable splitting and re-merging

```
uint16_t yuksek = 0x1234;   /* upper 16 bits — in a separate variable, perhaps a separate struct */
uint16_t dusuk  = 0xABCD;   /* lower 16 bits — kept somewhere else in the code */

uint32_t deger = ((uint32_t)yuksek << 16) | dusuk;   /* re-merged at the moment of use */
```

Let's verify: `yuksek << 16` shifts `0x1234` left by 16 bits → `0x12340000` . OR-ing this with `dusuk = 0x0000ABCD` gives: `0x12340000 | 0x0000ABCD = 0x1234ABCD` — exactly the original value, recovered.

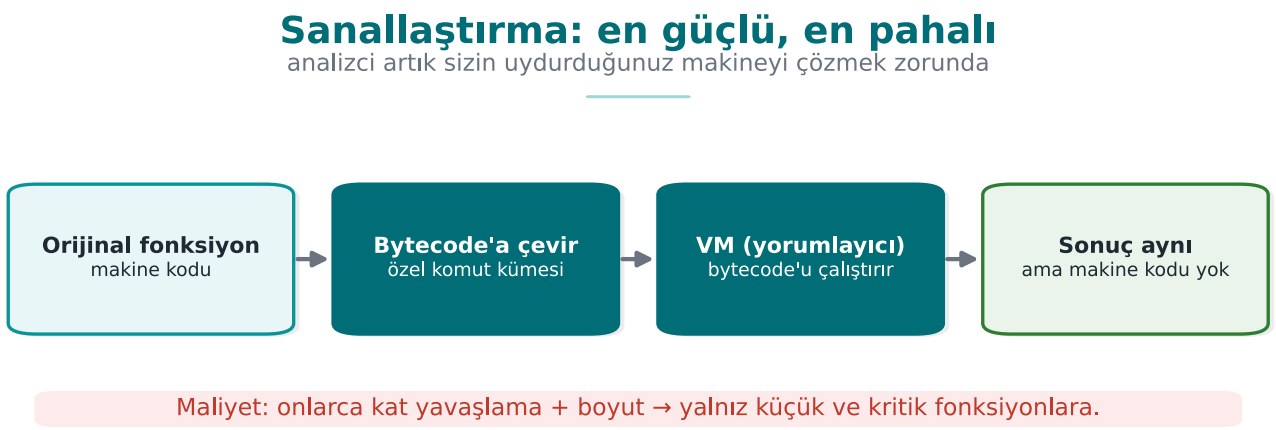
Let's think about why this works from the point of view of a memory-scanning tool: such a tool usually looks for "a contiguous, high-entropy block of 4 (or 16/32) bytes" — this is where the entropy definition from section 0 gets used in practice (e.g., an AES key looking like "sequence B" in memory). `yuksek` and `dusuk` sit in separate variables, and depending on the compiler's choices may even end up at **non-adjacent** addresses in memory; the scanned block is no longer 4 bytes but two separate 2-byte pieces, and the information "these two combine to form a key" is **never written explicitly** in the code, it hides only in the merge expression.

Common mistake: producing the merged value early and keeping it in memory for a long time

Splitting/merging only pays off as long as merging happens **at the moment of use**. If the `deger` variable is computed once at the start of the function and kept around for the rest of the function (or stored permanently in a struct), you again end up with a single, whole, recognisable 32-bit block — K-09's benefit is lost. **Rule:** merge only at the **exact moment it's needed**, then wipe it immediately after use (as in K-07, with `memset_s`); keeping the parts (`yuksek` , `dusuk`) separate is always safer than keeping the merged form around.

7. Whole-program-level rules

RULE K-10 — Virtualisation-based obfuscation (concept)



What does it protect? A function's **machine code itself**. **How?** The function is turned into **a custom virtual machine's (VM) bytecode** instead of real machine instructions; a small interpreter that reads this bytecode is embedded in the binary alongside it. The attacker no longer sees familiar machine code — they see a **custom instruction set** they first have to solve.

Cost: high — expensive both in performance (interpreted code is slow) and in maintenance; so it is only applied to the most critical, small, and rarely-changing functions. **Limit:** once the VM itself is solved, all the functions it protects can be unlocked; and if a single VM design is identical across every copy, there is no diversification. **Measurement:** the protected function's size/speed cost; how long it takes an analyst to work out the VM's instruction set. We'll see this conceptually in week 14 with Tigress's `Virtualize` transform.

Worked example: a small, conceptual bytecode and its interpreter. Real virtualisation tools are far more complex; but to see the mechanism, let's build a tiny three-instruction "virtual machine" — our goal is to compute `5 + 7` **without** a real addition instruction:

Virtualisation idea (small, conceptual example — not real Tigress output)

```
enum { OP_PUSH, OP_ADD, OP_RET };
uint8_t kod[] = { OP_PUSH, 5, OP_PUSH, 7, OP_ADD, OP_RET };   /* bytecode for 'compute 5 + 7' */

int yorumla(const uint8_t *kod, size_t n) {
    int yigin[8], sp = 0, pc = 0;
    for (;;) {
        switch (kod[pc++]) {
        case OP_PUSH: yigin[sp++] = kod[pc++]; break;
        case OP_ADD:  { int b = yigin[--sp], a = yigin[--sp]; yigin[sp++] = a + b; } break;
        case OP_RET:  return yigin[--sp];
        }
    }
}
```

Let's trace it by hand (stack `sp`, program counter `pc`):

Step	`pc` before	Instruction	Operation	Stack (after)	`sp`
1	0	`OP_PUSH`	push `5`	`[5]`	1
2	2	`OP_PUSH`	push `7`	`[5, 7]`	2
3	4	`OP_ADD`	pop `b=7`, `a=5`, push `a+b=12`	`[12]`	1
4	5	`OP_RET`	return `12`	—	—

Result `12 = 5 + 7` — correct. Now let's look at what a reverse engineer actually sees: in the binary they find a generic, `switch`-driven loop called `yorumla`, and somewhere separate a byte array like `{0, 5, 0, 7, 1, 2}`. There is **nowhere** an instruction that says "5 plus 7"; the addition is a **side effect** produced by `yorumla` processing these six bytes in a particular order. To understand the real logic, the analyst first has to solve `yorumla` itself (the VM), then interpret each bytecode sequence separately — this is exactly where K-10's limit, "cost is high but once the VM is solved everything opens up," comes from: `yorumla` itself is a single place, and once it is solved every bytecode sequence becomes readable.

Let's see the cost, roughly, as an order of magnitude. Look again at the `yorumla` function above: a single addition like `5 + 7` needs just **one** instruction (`add`) in real machine code, but in the interpreted version every step requires: reading the bytecode (`kod[pc++]`), deciding which operation it is with `switch`, popping value(s) off the stack, doing the operation, pushing the result, advancing the program counter. A single "addition" ends up costing at least 5–6 real machine instructions; if the addition is called **repeatedly inside a loop**, this multiplier repeats too. In the literature, bytecode interpretation causing **up to a 10x** slowdown (sometimes more, in some designs) is a common order of magnitude — the exact ratio depends on the VM's design (how many stack operations, how many indirections) and **must be measured in your own project**; the number given here is only a rough answer to "why is it this expensive." That's why K-10's "Cost: high" line isn't just rhetoric; it means **every** instruction can cost 5-10x more, which is why it is only applied to small, critical functions that aren't called thousands of times per second (e.g., a license key derivation step, a check called once a second) — it is **not applied** to an image processing loop or a frequently called helper function.

> ⚠ **Two frequently confused ideas: 'virtualisation-based obfuscation' is not the same thing as a Java/CLR virtual machine**
>
> In week 5 you saw that Java/Kotlin applications run on a **virtual machine** (JVM/ART), and that ProGuard/R8 shrinks DEX bytecode. K-10's "virtualisation-based obfuscation" is **an entirely different** thing: - **JVM/ART:** a **general-purpose, universally known** virtual machine that the whole application runs on; its bytecode format (DEX) is standard and can be decoded by anyone with a public tool (`baksmali`, `jadx`). - **The VM in K-10:** a **project-specific, non-standard** bytecode and interpreter, built only for **a handful of selected critical functions**; no general-purpose tool can read it directly, because **you** design the instruction set. So K-10 does **not** mean "I wrote my app in Java, it already runs in a VM, so it counts as obfuscated" — quite the opposite: a standard bytecode format like DEX is easily read by standard tools; K-10's strength comes precisely from the VM design being **non-standard**.

RULE K-11 — Compiler-based obfuscation (the O-LLVM family)

What does it protect? The control/data rules above — applied **automatically by the compiler** rather than by hand. **How?** LLVM-based obfuscators (Obfuscator-LLVM and its derivatives) automatically apply flattening, bogus control flow, and instruction substitution passes at compile time. Advantage: the source code stays readable, the protection is a compile option; diversifying by giving a different seed (**seed**) per build is easy. **Cost:** medium; the performance impact depends on

which passes are selected. **Limit:** the patterns well-known passes produce are recognisable; you need to stay current with the tool's version. **Measurement:** the difference between binaries produced by the same passes with different seeds (the diversification metric, section 5).

Conceptual example: same source, two seeds, two different binaries. Suppose we compile the same `erisim_ver` source twice with an O-LLVM-based compiler, changing only the `--Seed` value. The source code is **byte-for-byte identical**; what changes is the compiler's **random choices**:

What changes?	Seed A	Seed B
Flattening's state constants (K-04)	e.g. `{0x7A1, 0x3F0, ...}`	e.g. `{0x22C, 0x9E4, ...}` (completely different)
Which opaque predicate pattern is chosen (K-01)	one from the `x*(x+1)` family	a **different** identity from the same family
Order of basic blocks inside the `switch`	one permutation	another permutation

What doesn't change? When `erisim_ver("CEN429-OK")` is called, both binaries return the **same** result — the source code and the behaviour are identical, only the compiled **shape** differs. This is exactly the phenomenon we'll see under "diversification" in section 8; O-LLVM just produces it automatically at compile time instead of by hand.

Common mistake: treating a single O-LLVM flag as 'secure' without measuring it

Adding a flag like `-mllvm -fla` (flattening) and calling it "obfuscated now" is a common mistake. How many instructions/branches the pass really adds, how much it slows execution down, and which sub-passes (opaque predicate, bogus control flow) got enabled together cannot be known **without measuring it**. **Rule:** evaluate every O-LLVM configuration with the protection rule template from section 4 too (especially the "Cost" and "Measurement" lines); "I added a flag" is not evidence.

RULE K-12 — Self-modifying code and dynamic encryption (concept, handle with care)

What does it protect? The most sensitive code sections, by keeping them **encrypted** in the binary. **How?** The section sits encrypted in the binary; it is only decoded into memory right when it's about to run, runs, and is then re-encrypted or wiped. **Cost:** high and **risky**: it directly conflicts with modern operating systems' memory protections (DEP/NX, the separation of writable-and-executable pages, W^X); done wrong it both crashes and opens a new hole. **Limit:** at some point while running, the code sits exposed in memory; a memory dump can catch it. **Measurement:** how short the exposure window is.

Worked example: a timeline of the decode-run-re-encrypt cycle. The conceptual flow has these steps:

1. t_0**:** The function is called; the relevant memory page is still encrypted.
2. t_1**:** The page is decoded (exposed in memory), page permissions are temporarily made **writable+executable**.
3. t_2**:** The code runs (the real work happens here) — this is the window where an attacker taking a memory dump could catch the code **exposed**.
4. t_3**:** The work is done; the page is re-encrypted and permissions return to their original state (execute-only, not writable).

Critical point: step `t₃` must run on **every exit path** — not only on a successful return, but also on error, on an exception, or on an early `return`. Otherwise the code stays **exposed indefinitely** in memory, and the "exposure window" is no longer short as defined, but infinite.

Common mistake: re-encrypting only on the success path

A developer will usually add the `t₃` step only to the function's **normal** (successful) exit; it gets **skipped** on early `return`s in error paths or when an exception is thrown. Result: if an attacker deliberately triggers an error (e.g., by giving invalid input), the code stays decoded permanently and is exposed in the next memory dump. **Rule:** tie the re-encryption step to the language's "always runs" construct (a `goto cleanup:` label in C, RAII/a destructor in C++, `finally` in Java/Kotlin); don't handle success and error paths **separately** and forget one of them.

Rule: don't blindly disable a technique that conflicts with OS protections just for the sake of security

Self-modifying code requires making executable memory writable; this weakens the DEP/NX and W^X principles from week 4. If adding one protection means turning off another, measure the net gain and record the trade-off. In practice this technique is used very selectively, on small sections only.

The Java/managed side: ProGuard/R8 and DEX (a bridge from week 5)

This week's rules are for the native (C/C++) side. In managed languages (Java/Kotlin, Android) the equivalent is name obfuscation, dead code elimination, and shrinking with **ProGuard/R8**; `-keep` rules and APIs called via reflection were covered in week 5. The two sides are used together: the Java side is obfuscated with R8, the native side with this week's rules; the native side checks the Java side's integrity, the Java side checks the native library's (cross-checking — week 6, RASP).

Sections 5–7 summary table: twelve rules at a glance

One weak point is holding this many rules together at once. The table below is not a **copy-paste list**, it is a reminder; behind every row is a full explanation above, a worked example, and a common-mistake/rule box:

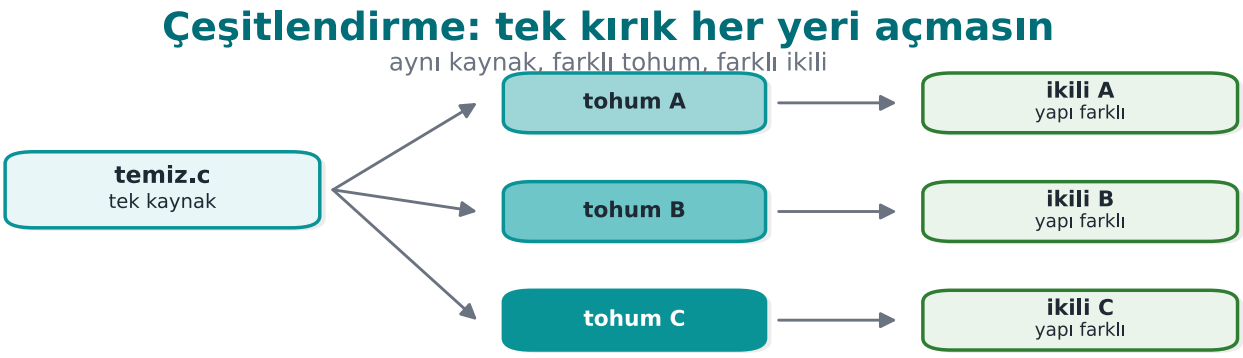
Rule	Family	What it protects (summary)	Cost	Its biggest limit
K-01 Opaque predicate/loop	Control flow	Branch/loop condition	Low–medium	Known patterns are exposed to symbolic execution
K-02 Arithmetic encoding	Data	Constants, simple computations	Low	Can be undone by MBA simplifiers
K-03 Bogus operation/dead branch	Control flow	Hides real logic in a crowd	Low–medium	Can be deleted by compiler optimisation
K-04 Flattening	Control flow	Block adjacency/order	Medium–high	Symbolic execution + pattern recognition
K-05 Random exit	Control flow	Success/failure point	Low	Weak alone, meaningful with flattening
K-06 Function/parameter hiding	Layout	Library call hints	Low–medium	Behavioural analysis can still show what it does
K-07 String encoding	Data	Readable text	Low	Exposed in memory while running
K-08 Opaque boolean	Data	Decision result (allow/deny)	Low	Can be solved with enough scrutiny
K-09 Variable splitting/merging	Data	Memory pattern/footprint	Low–medium	Only slows pattern/static analysis
K-10 Virtualisation	Virtualisation	The machine code itself	High	Once the VM is solved, everything it protects opens
K-11 Compiler-based (O-LLVM)	Mixed	Applies the above automatically	Medium	Known pass patterns are recognisable
K-12 Self-modifying code	Anti-analysis	Critical code section	High, risky	Conflicts with OS memory protections (DEP/NX, W^X)

For a quick answer to "which rule do I pick when?" when preparing for the exam or S9: for **low-value, frequently changing** code, pick from the rows near the top (K-01, K-02, K-05, K-07 — the low-cost ones); for **high-value, rarely changing, small** code sections, pick from the rows near the bottom (K-10, K-12) — exactly what the "Decision rule" in section 9 says.

8. Diversification: one crack should not open every door

"Rule 2 — break automation" from section 1 becomes concrete here. If an attacker breaks one copy, and the patch or script they produce can be distributed to **all** users, then a single crack opens every door. **Diversification** means producing

different-but-behaviourally-equivalent binaries from the same source; that way an automated attack developed against one copy does not work against the others.



- **Diversification in space:** each build or each distribution is obfuscated with a different **seed**; opaque predicates, bogus blocks, and state values change from copy to copy.
- **Diversification in time:** every release comes with a new arrangement; an attack found against an old release breaks in the new one. This works together with your key/version renewal policy (week 10).

Diversification does not raise obfuscation's **potency** — a single copy can still be broken — but it prevents **scaling**. In week 14 we'll produce this with tooling using Tigress's `RandomFuns`, `--Seed`, and transform-combination features.

A simple metric for diversification effectiveness

Compile the same source with two different seeds. Compare the protected functions of the two binaries: the higher the byte-level difference ratio, the lower the odds that an attack written against one copy works against the other. Write this metric into your S9.

Worked example: let's compute the difference ratio numerically. Let's turn the metric above into a concrete number. Say the flattened version of the protected `erisim_ver` function takes up **512 bytes** in the binary. Comparing this function byte-by-byte between binaries produced with two different seeds, we measured **340 bytes** coming out different (a hypothetical but realistic demo result):

```
diff_ratio = changed_byte_count / total_byte_count × 100
           = 340 / 512 × 100
           ≈ 66,4 %
```

Interpretation: **66.4%** of the two copies are **different**; only **33.6%** stays common (likely fixed parts the compiler couldn't change, such as the function's prologue/epilogue). If an attacker applies a byte patch found in Copy A (e.g., "write this value at this offset, so the check always allows") directly to Copy B, that offset now holds a **different instruction**, so the patch either does nothing or crashes the program. This is the numeric proof of "Rule 2 — break automation" from section 1.

Common mistake: mistaking a change in compiler optimisation level for diversification

Saying "I compiled with `-O3` instead of `-O2`, so now I have two different binaries" is **not** diversification. Changing the optimisation level (a) produces the **same** result on every compile (it isn't tied to a seed, so it's a repeatable, predictable single "version B," not many different copies) and (b) even though it strongly guarantees the behaviour stays the same, that isn't the goal here. **Rule:** real diversification must be produced with a **different and unpredictable** seed on every build (K-11's `--Seed`); switching between just two fixed configurations (`-O2` and `-O3`) gives you at most **two** fixed variants, which does not reach section 1's "break automation" goal.

The diversification decision is thought about together with RASP (week 6) and key/version renewal (week 10): even if one copy is broken and a bypass script spreads, that script doesn't work on the other copies (diversification), and the old script also stops working in the next release (diversification in time + key renewal).

A dynamic known from the field: why is diversification a continuous race, not a one-off?

Digital rights management (DRM) and anti-cheat systems in games are a well-known, publicly visible field that shows **why diversification has to be continuous, not one-off**: when a version of a protection is broken (a "crack" or bypass spreads), the vendor usually releases a **different** protection arrangement in the next version, and the old crack stops working. This is the concrete way of working behind the sentence "2010s — DRM, mobile banking, and the games industry make obfuscation mainstream" in the history box in section 1: the defence is not one single "unbreakable" version, but a series of **continuously renewed** versions. The takeaway for the course: in your S9 writeup in section 12, describe diversification not as "I applied it once" but as "a process reapplied in every release."

9. Measuring obfuscation

Defending a protection decision requires **measuring** it. Collberg's framework evaluates obfuscation along four dimensions; these same four metrics are also the foundation of week 13's **attack potential** scoring ("time required," "expertise required").

Metric	Its question	How it's measured (concept)
Potency	How incomprehensible is it to a human?	Complexity metrics: CFG node/edge count, cyclomatic complexity, nesting depth — before/after obfuscation
Resilience	How much does it withstand an automated tool?	How long/successfully a deobfuscation tool (symbolic execution, a simplifier) takes to undo the protection
Stealth	Does the obfuscation give itself away?	How statistically distinguishable the obfuscated code is from normal code (if it's distinguishable, the attacker knows where to look)
Cost	How much does it cost us?	Size increase, runtime increase, memory, maintenance, and debugging difficulty

These four metrics form a **trade-off**: raising potency and resilience raises cost, and stealth usually drops (more complex code draws more attention). A good engineering decision balances these four against the **value of the protected asset**.

Worked example: filling in the four metrics with numbers for `erisim_ver`

Let's spread the real demo measurements we met in section 4 (**28 → 51 instructions**, **3 → 6 branches**) across the four metrics here:

Potency. Let's use McCabe's **cyclomatic complexity** idea as a rough approximate metric: for a function with one entry/one exit, `complexity ≈ number_of_branches + 1`.

```
Before  : 3 branches + 1 = 4
After   : 6 branches + 1 = 7
Increase: (7 - 4) / 4 × 100 = %75
```

Complexity rises from `4` to `7`, i.e. by **75%**: a human analyst tracing the function by hand now has to follow 75% more "paths."

Cost. Let's compute the rise in instruction count the same way:

```
(51 - 28) / 28 × 100 ≈ %82,1
```

So applying this one rule (K-04 + K-01 + K-05) has made the function **82%** bigger and probably a little slower — this is the concrete number to write into the "Cost" line of the protection rule template from section 4.

Resilience and stealth. These two cannot be measured just by counting instructions/branches; resilience requires actually running a real deobfuscation tool (section 10) and measuring how long/successfully it "undoes" the protection, and stealth requires checking whether the obfuscated code is statistically distinguishable from "normal" code (e.g., does an abnormally high branch density draw attention). This week's demo **directly** measures only potency and cost; measuring resilience is the subject of the next section.

Common mistake: measuring only cost and calling it 'strong'

The conclusion "the binary grew 82%, so it must be well protected" is wrong — growth only shows **cost**, not **resilience** against being broken (as we'll see in section 10, a weak opaque predicate can add a lot of instructions and still be solved by a solver in seconds). **Rule:** report the four metrics separately; don't single out just one (usually "cost," the easiest to measure) and skip the others.

Decision rule (write this into S9):

- Asset value is **low** → light, cheap obfuscation (K-01, K-02, K-07) is enough.
- Asset value is **high** → layered protection (K-04 + K-05 + K-08 + diversification); the cost is accepted.
- For every layer, the potency/resilience gain is measured against the size/speed cost, and written down.

Worked example: applying the decision rule to two different assets. Suppose the same application has two different code sections:

- **Asset A — a helper function that picks the text of a "welcome" screen.** Its value is low (even if it leaks, it causes no major loss). Decision: K-07 (string encoding) alone is enough; an expensive transform like K-04 is **unnecessary** — here section 9's "cost" metric also works in the direction of "don't add unnecessary cost."
- **Asset B — the `lisans_dogrula` function where the license key is verified.** Its value is high (tied to 100,000 TL/year of revenue in the numeric example in section 1). Decision: K-04 + K-01 + K-05 + K-08 + diversification applied together; the total cost (the **82%** instruction increase we measured in section 9) is accepted because the value of the protected asset justifies it.

This comparison shows that "applying the same level of obfuscation to every function" can be **both needlessly expensive and insufficiently protective**: applying K-04 to Asset A wastes performance for nothing; applying only K-07 to Asset B leaves the CFG completely open. Make a **separate** decision for each code section in your S9.

10. Deobfuscation: the other side's tools, and the resilience rule

Measuring resilience correctly requires knowing the attacker's automated tools at a conceptual level. We learn these in order **to test our own defence**:

- **Symbolic / concolic execution (tools like KLEE):** program paths are solved as mathematical constraints; opaque predicates and flattening can be undone this way. **Resilience rule:** to defeat the solver, tie opaque predicates to structures that are expensive to solve (e.g., based on factorization or hash values), and grow the state space; but measure the cost.
- **Expression simplification (MBA solvers):** undoes arithmetic encoding (K-02). **Rule:** don't treat arithmetic encoding as a secrecy layer on its own; combine it with control-flow obfuscation.
- **Pattern recognition:** recognises known library/pass patterns. **Rule:** diversify; don't rely on a single opaque predicate pattern or a single VM design.

Banescu and colleagues' work (Tigress + KLEE) does exactly this: it measures how much each transform withstands symbolic execution. The takeaway for the course is clear: **resilience is not a claim, it's a measured quantity.** Instead of saying a protection is "strong," you need to say "it withstood this much against this tool."

How does symbolic execution break the opaque predicate in K-01? Step by step

Let's return to the `opak_dogru(x) = ((x*(x+1)) & 1) == 0` predicate from section 5; the warning box there said it is "a classic pattern, recognised in modern tools' libraries." Now let's see **why**.

A symbolic execution tool (like KLEE) runs the program not with **concrete** values (like `x = 5`) but by keeping `x` as an unknown **symbol**. When it reaches the line `if (opak_dogru(x))`, the question it needs to ask is:

> "Is `((x*(x+1)) & 1) == 0` true for **every** value of `x`, or is there an `x` that makes it false?"

It hands this question to a **constraint solver** (an SMT solver). The solver splits `x` into two cases, odd and even (a standard solving strategy in integer arithmetic), and quickly proves both:

- If `x` is even: `x = 2k` → `x*(x+1) = 2k*(2k+1)` → the expression is a multiple of `2` → `& 1 = 0`. ✅
- If `x` is odd: `x = 2k+1` → `x+1 = 2k+2 = 2(k+1)` is even → `x*(x+1)` is again a multiple of `2` → `& 1 = 0`. ✅

In both cases the result is `0`; the solver reaches, **within seconds**, the conclusion "this predicate is always true, the second (bogus) branch never runs," and automatically eliminates the dead branch. That's why the warning box for K-01 doesn't consider this **particular** identity strong enough on its own, and recommends seed-diversified predicates in week 14 that tire the solver out more (e.g., based on factorization or on inverting a hash function) — now we've also seen **why** that recommendation makes sense.

How do MBA simplifiers undo K-02?

An expression like K-02's `(a ^ b) + 2·(a & b) = a + b`, complex as it looks to a human, can be mechanically checked with a finite **truth table** (`2⁴ × 2⁴ = 256` rows for every 4-bit `a,b` pair, `65,536` for the 8-bit case). Tools like `arybo` / `msynth` do exactly this: they **brute-force** the expression at small bit widths, find the shortest standard expression producing the same truth table (`a+b`), and report the two as **equivalent**. That's why K-02's limit line said "weak on its own": if arithmetic encoding stays a single layer, this simplification step can undo it in a few seconds; but if you embed the result **inside** K-04's flattened `switch`, the simplifier first has to solve the flattening just to **find** which expression to simplify — two weak layers together are stronger than one.

How does pattern recognition target K-04/K-11?

A "pattern recognition" tool scans a binary looking for known **structural signatures**: a large `switch` inside a single loop, non-sequential state values drawn from a fixed set, an error path exiting through `default` — exactly K-04's template. The flattening shape produced by popular tools like O-LLVM has itself become a "recognisable" signature over time (there are

even academic studies that guess which obfuscation tool a piece of software was compiled with). This is the source of the "known passes' patterns are recognisable" warning in K-11's limit line.

Common mistake: assuming every technique withstands every tool equally

Before declaring a protection "strong," you need to state **which class of tool** it was measured against. The table below summarizes which deobfuscation tool this week's rules mainly threaten; a rule measured only against the tool in its own column may **still be weak** against the tools in the other columns:

Rule	The tool it mainly threatens	This week's countermeasure
K-01 (classic opaque predicate)	Symbolic/constrained execution (KLEE)	Solver-resistant predicates in week 14
K-02 (MBA)	Expression simplifier (arybo/msynth)	Combine with control-flow obfuscation
K-04/K-11 (flattening)	Pattern recognition, CFG signature matching	Diversification (section 8), reinforcement (K-01/K-03/K-05)

Rule: also write "which tool it was measured against" into the protection rule template from section 4 (implicitly, into the "Which threat?" line); a claim of being "generally strong" means it hasn't been measured against any tool.

Resilience and cost are not the same thing — don't mix them up

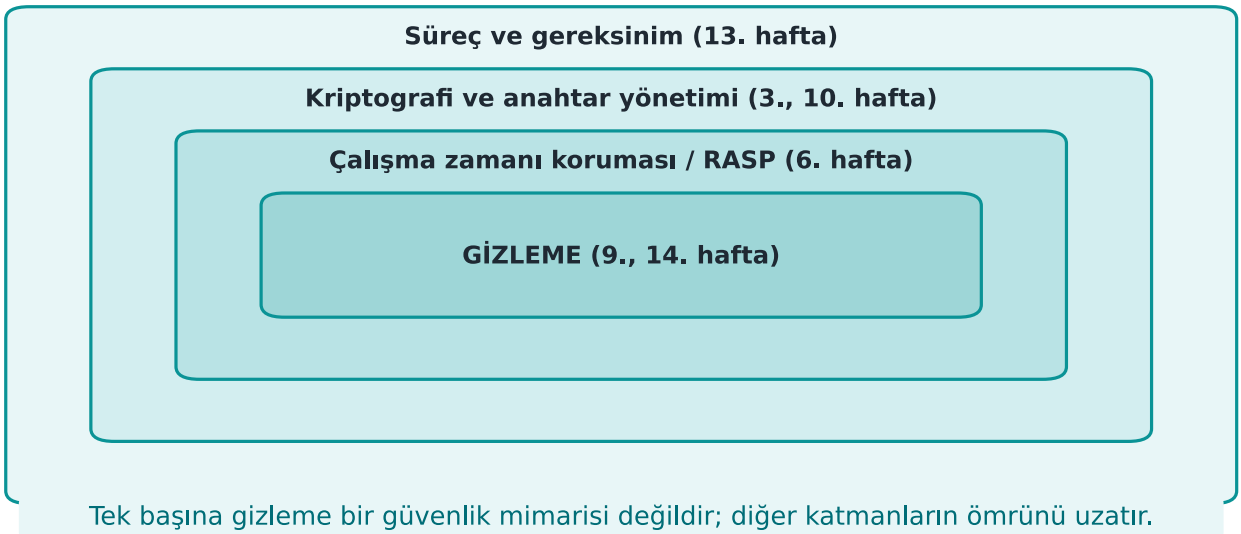
In section 9 we defined "resilience" and "cost" as two separate metrics; after this section the difference should be clearer. **Cost** is how much applying the protection costs *us* (instruction count, speed, memory) — we measure this directly in our own build process (the `objdump` examples in sections 4 and 9). **Resilience** is how much the protection withstands *the attacker's tool* — measuring this requires actually running the attacker's tool (symbolic execution, an MBA simplifier, pattern recognition) and observing the result; just saying "I added a lot of instructions" does **not prove** resilience. Recall the "82% cost increase" example from section 9: that number alone does not say how much K-04 withstands symbolic execution — that question can only be answered by actually trying a deobfuscation tool, as in this section. Report these two metrics **separately, without mixing them up**, in your S9.

11. Obfuscation's place in layered defence

Obfuscation is not a fortress on its own, it is one layer of **defence in depth**. Its relationship to the other weeks:

Katmanlı savunmada gizlemenin yeri

gizleme en dıştaki kabuk değil, diğerlerini örten bir katmandır



Tek başına gizleme bir güvenlik mimarisi değildir; diğer katmanların ömrünü uzatır.

Layer	Its week	Relationship to obfuscation
Memory safety, secure compilation	4	Obfuscation doesn't replace these; the two work together
RASP (tamper/debugger detection)	6	Obfuscation also hides RASP's own checks; RASP protects obfuscation while it runs
Whitebox cryptography	11	The real protection for the key; obfuscation makes the shell around it harder
Crypto and key renewal	10	Obfuscation buys time "until the key is renewed"
Certification / attack potential	12–13	Every layer's "delay time" feeds into the scoring

Tying the whole term together through one example: a mobile payment app's license check

To make the connections between weeks concrete, let's follow the same imaginary function — a mobile payment app's `lisans_dogrula` function — through the eyes of different weeks across the term:

- **Week 1:** the threat model is drawn up; `lisans_dogrula` is flagged as an asset that needs protecting against a **MATE** attacker (the user who holds the app).
- **Week 3:** the keys the function uses are encrypted with **AEAD**, random numbers come from a **CSPRNG**; but this is the security of **storing** the data, not the security of **reading** the code.
- **Week 4:** the first obfuscation steps are taken: symbols are hidden, a basic flattening is tried.
- **Week 6:** RASP is added; whether a debugger is attached is checked at runtime.

- **Week 9 (this week):** K-01–K-12 are applied: control flow is flattened and reinforced (K-01, K-03, K-05), data is encoded (K-07, K-08, K-09), diversification is added (section 8), effectiveness is measured (section 9).
- **Week 10:** the **lifecycle** (generation, distribution, renewal) of the keys the function uses is managed with PKI — obfuscation does **not store** the key, it only makes its surroundings harder (the rule from section 3).
- **Week 11:** the most critical key-derivation step is protected with **whitebox cryptography** — the cryptographic counterpart of K-10's (virtualisation) "only apply it to the most critical small function" principle.
- **Weeks 12–13:** an independent evaluator scores all of these layers with the attack potential framework; the "Measurement" lines from the protection rule template in section 4 are presented here as evidence.
- **Week 14:** all this hand-done work is **automated** with Tigress; the same source is regenerated with a different seed on every release, with a single command.

This chain shows which **different** threat a single function is protected against in each week of the term: no week replaces another, each one closes the gap the previous one left behind.

> **An observation from the field**
>
> The native protection measures listed in a certified product cover almost all of this week's rules; but the document attaches a **"not strong on its own"** note next to every one of them. What convinces an evaluator is not a single technique, but the rules being used **together**, being **diversified**, and each one being presented with a **measured** delay value.

In practice, what do you look at to estimate "delay time"?

The attack potential assessments we'll meet in weeks 12–13 generally look at **how many factors** an attack is made harder by; the rules we learned this week affect those factors as follows (exact scores vary by certification scheme, here we only show the **direction**):

Factor	Its question	Which of this week's rules affects it?
Time required	How many hours/days does the attack take?	All of them — especially K-04 (grows the CFG) and K-10 (solving the VM takes time)
Expertise required	Does it need general skill or specialist knowledge?	K-01/K-02 (MBA, opaque predicate knowledge), K-10 (VM reverse engineering)
Knowledge of the target	How familiar is the attacker with the source code/design?	K-06 (hiding internal function names/signatures lowers "target knowledge")
Window of opportunity	How long can the attacker access the binary?	Diversification (section 8) + K-12 (short exposure window) narrow this
Equipment needed	Does it need a general tool or a special/expensive one?	Solver-resistant predicates (section 10) push it toward "a general tool isn't enough"

These five questions are the common skeleton of the **attack potential** assessment approaches widely used in the certification world. For example, if you apply **only string encoding (K-07)**, you raise the required time by a few minutes but barely change the required expertise (anyone can search for a string in a decompiler instead of using `strings`). If **control-flow flattening + opaque predicate + diversification** are applied together, the required time (solving the CFG by

hand takes hours), the required expertise (a generic tool isn't enough, a custom script/analysis has to be written), and the window of opportunity (a script has to be rewritten for every copy since each is different) all rise together — this is the technical explanation for why **layers**, not a single rule, score higher in an assessment.

Short summary of this section

Obfuscation's place in layered defence can be summed up in three sentences: (1) obfuscation does not **replace** memory safety (week 4), cryptography (week 3), or RASP (week 6) — it makes their **surroundings** harder; (2) every layer is counted as a separate "delay" in the attack potential assessment (weeks 12–13), none is sufficient alone; (3) every protection line you write in the next section (S9) should be traceable back to this table and to the template in section 4 — if you cannot answer "which layer does this protection belong to, which other layer does it work with," that protection has probably ended up isolated and insufficient.

12. Term project: this week (S9 — advanced hardening)

Take the S9 section of your post-midterm project to an **advanced** level:

1. **Protection table:** for the two–three most critical code sections (e.g., licensing, key derivation, integrity check), fill in the **"protection rule" template** from section 1: what it protects, against which threat, how, cost, limit, measurement.
2. **Measurement:** for at least one technique, give a before/after obfuscation metric (e.g., the count of sensitive strings in `strings` output; CFG node count; binary size and an operation's duration).
3. **Diversification decision:** will you apply diversification? If not, **why not** (a justification is also an answer).
4. **Limit and remaining risk:** for every protection, state explicitly what it does **not** protect. Something like "this string obfuscation does not protect the key; the key uses the whitebox/hardware from S8."

Worked example: an S9 draft from start to finish

Let's combine this week's examples into a short S9 draft (replace the function names and numbers with your own measurements in your own project):

```
S9 — Advanced hardening (example draft)

1) Protection table
   RULE: K-04 (flattening) + K-01 (opaque predicate) + K-05 (random exit) applied to the
lisans_dogrula function
     What does it protect? : the order in which the licence key is verified
     Against which threat? : static CFG reading, single-byte patch
     How?                  : switch dispatcher + x*(x+1)-based opaque predicate + unpredictable
failure exit
     Cost                  : instructions 28→51 (+%82), branches 3→6 (+%100) [measured with objdump]
     Limit                 : weak against symbolic execution (section 10); data still needs K-07
protection
     Measurement           : objdump output (appendix A), CFG node/edge count

   RULE: K-07 (string encoding) applied to the key string
     What does it protect? : sensitive strings in the binary
     Against which threat? : strings scanning (section 3's "first 10 minutes" attack)
     How?                  : encode with XOR 0x5A, decode at the moment of use and wipe with memset_s
     Cost                  : low (a few extra instructions)
     Limit                 : the decoding key sits in the binary; real key protection is
S8/whitebox's job
     Measurement           : count of sensitive strings in strings output: 3 → 0

2) Measurement: the "Measurement" field is filled in on both lines above (objdump + strings)

3) Diversification decision: YES — a random --seed parameter was added to the build script;
   every release is compiled with different state constants (section 8, with K-11).

4) Limit and remaining risk: this layer doesn't protect the key (it only delays static scanning);
   it hasn't been measured against symbolic execution (future work); permanent protection of the key
is left to S10/S11.
```

This draft combines the template from section 4 with the four metrics from section 9 and the diversification decision from section 8; you are expected to replace every line with numbers **you measured yourself** — the numbers above come from this week's demo and should not be copy-pasted directly into another project.

⚠ Checklist before submitting S9

- For at least two code sections (one control-flow rule, one data rule), is the six-line template from section 4 filled in **completely**?
- Does every "Measurement" line contain a concrete number or observation (not just an adjective like "strengthened")?
- Is it stated which **family** (layout/data/control flow/anti-analysis/virtualisation — section 2) each technique belongs to, or was only a single family used (section 2's "common mistake" applies here again)?
- Is the diversification decision (applied/not applied) written down together with its justification (section 8)?
- Does the "Limit" line for every protection clearly state what it does **not** protect — in particular, hasn't the core warning from section 3, "this does not protect the key," been forgotten?
- Has an absolute claim like "unbreakable" or "impenetrable" been **avoided** everywhere (the Barak et al. warning from section 1)?
- Was it tested on a binary compiled with optimisation on (`-O2` / `-O3`) (against the "the compiler silently deletes it" mistake from K-02/K-03)?

Through an evaluator's eyes

What S9 is graded on is not "I used a lot of techniques" but **the justification and measurement of every technique**. An unmeasured protection counts, in an assessment, as a "claim," not as evidence.

13. Self-check

1. What is source-to-source obfuscation? In what way does the MATE threat model differ from a network attacker, and why does that not leave cryptography sufficient on its own but make obfuscation necessary?

MATE (Man-At-The-End) has full possession of the device the app runs on: they can read memory, attach a debugger, modify code; a network attacker only connects from outside. Cryptography is still necessary but not sufficient on its own, because the key/data is exposed in memory **while running**. That's why obfuscation is needed: it delays and raises the cost of extracting the key and understanding the logic.

2. 'Obfuscation doesn't make something unbreakable, it makes it expensive.' Explain this with an asset-value example.

Obfuscation doesn't make an attack impossible, it makes it require more **time/skill/tools**. If breaking content worth 100 TL takes 10,000 TL of effort, a rational attacker gives up. That's why protection strength is measured relative to the value of the protected asset.

3. What is the difference between a bogus operation and a death (dead) branch? Which one runs?

A **bogus operation** really runs, but its result is never used (it wears out the analyst and the tool). A **dead branch** is protected by an opaque predicate and **never runs at all**. So a bogus operation is code that runs but has no effect, while a dead branch is code that never runs.

4. Why is control-flow flattening weak on its own? Name at least three rules that strengthen it.

Flattening leaves behind a dispatcher (switch) pattern; this pattern is recognisable, and the flow can be reconstructed by tracking the state variable. Reinforcements: **opaque predicates**, **encrypting the state variable**, **bogus states/blocks + random exit** (also name/string obfuscation).

5. Which attack does a random exit point make harder? Why does a single-byte patch stop being enough?

It makes pattern-matching, automated scripting, and symbolic execution attacks harder (there is no single fixed exit pattern). Because there is no single fixed exit, a single byte patch cannot close off every path; the attacker has to deal with every path separately.

6. Does string obfuscation protect a key? Why or why not? Which weeks should you look at for protecting a key?

No. String/table obfuscation only makes static `strings` scanning harder; the key is exposed **in memory while running**. Real key protection requires the methods from week 10 (PKI, HSM/PKCS#11) and week 11 (whitebox cryptography).

7. What are the two biggest costs of virtualisation-based obfuscation? Why is it applied only to small, critical functions?

A large **performance** penalty (a virtual machine interpreting bytecode is slow) and a **size** increase (VM + bytecode). Because of this cost it is applied only to functions that are high-value, small, and critical; it cannot be applied to the entire codebase.

8. Which operating system protection does self-modifying code conflict with? Why can this create a new risk?

W^X / DEP-NX: a memory page cannot be writable **and** executable at the same time. Modifying code requires temporarily making the page writable-and-executable; this is blocked or suspicious in most environments, and it can also open a window for code injection by an attacker (a new risk).

9. Define the four dimensions that measure obfuscation (potency, resilience, stealth, cost). Which of them trade off against each other?

Potency: how much it hinders a human analyst. **Resilience:** resistance to automated deobfuscation tools. **Stealth:** how unnoticeable the obfuscation is (looking normal). **Cost:** the performance/size penalty. The main trade-offs: resilience ↔ cost, and potency ↔ stealth (strong obfuscation usually looks more 'abnormal').

10. Does diversification prevent obfuscation's potency, or its ability to scale? Propose a simple effectiveness metric.

It prevents **scaling**: it doesn't raise a single copy's potency, but it prevents a crack from **spreading** to all copies/users. Metric: the difference ratio (percentage of different bytes/instructions) between binaries produced from the same source with two seeds, plus a behaviour test showing both binaries give the same output for the same input.

11. Which obfuscation rules does symbolic execution (like KLEE) challenge? What do you do to raise resilience, and how do you keep the cost in check?

KLEE can break simple opaque predicates and flattening by automatically solving paths. For resilience, you make predicates **resistant to symbolic execution** (input-dependent, cryptographic, causing path explosion). You control the cost by applying these expensive transforms only to critical functions and measuring before/after.

12. Fill in the 'protection rule' template (six lines) for a code section in your own project.

Example — **Asset:** the license verification function. **Threat:** a MATE attacker bypassing verification with a patch. **Rule:** control-flow flattening + opaque predicate. **How:** Tigress `Flatten` + `AddOpaque`, applied only to this function. **Measurement:** size +X%, speed −Y%, instruction count N→M. **Remaining risk:** strong but not virtualised; diversify with a seed per release.

13. If you XOR the byte `0x41` with `0x5A`, and XOR the result with `0x5A` again, what do you get? Which algebraic rule guarantees this?

`0x41 ^ 0x5A = 0x1B`, then `0x1B ^ 0x5A = 0x41` — you're back to the byte you started with. The rule that guarantees this is `a ^ b ^ b = a ^ (b ^ b) = a ^ 0 = a`: any value XORed with itself gives `0`, and a value XORed with `0` stays unchanged. This is the mathematical basis of K-07's string encoding/decoding and of the general claim that "XOR is reversible."

14. Why is the Shannon entropy of `[0x41,0x41,0x41,0x41]` 0 bits/symbol, while `[0x3F,0xA1,0x08,0xC7]`'s is 2 bits/symbol? What is the practical meaning of this for an attacker?

In the first sequence, a single value has probability `p=1`, `log2(1)=0`, so `H=0` — completely predictable. In the second sequence, four distinct values each occur with equal probability (`p=1/4`): `H = -4·(1/4·log2(1/4)) = -4·(1/4·(-2)) = 2` bits/symbol, the highest value reachable in this four-symbol alphabet. In practice: when scanning a binary, an attacker can statistically distinguish low-entropy (repeating) blocks from high-entropy blocks (potentially keys/encrypted data); high entropy is a clue that "there might be a secret here."

15. Verify that `(a ^ b) + 2·(a & b)` equals `a + b` for `a=5, b=3`. What are expressions of this kind called, and why aren't they strong enough on their own?

`a+b = 5+3 = 8`. `a^b = 0101^0011 = 0110 = 6`; `a&b = 0101&0011 = 0001 = 1`; `(a^b)+2·(a&b) = 6+2 = 8`. The two are equal — the identity is confirmed. Expressions of this kind are called **mixed boolean-arithmetic (MBA)** (K-02). They aren't strong enough alone because expression simplifiers like `arybo` / `msynth` can reduce expressions like this back to their original form (`a+b`) quickly by building a truth table at small bit widths; they only gain strength combined with control-flow obfuscation (K-04).

16. In K-08's opaque boolean example, with `a=0xA3C1, b=0x5C3E`, why is `a^b` equal to `0xFFFF`? If the attacker only changes the low byte of `a` (`0xC1→0x00`), what does `karar_izin_mi` return?

Because all four nibbles are each other's bitwise complement (`A=1010` ↔ `5=0101`, `3=0011` ↔ `C=1100`, `C=1100` ↔ `3=0011`, `1=0001` ↔ `E=1110`), their XORs always give `1111=F`, totalling `0xFFFF`. If `a`'s low byte is set to `0x00`, the new `a=0xA300`; `0xA300^0x5C3E = 0xFF3E ≠ 0xFFFF`, so `karar_izin_mi` returns **false (deny)** — the single-byte patch fails to trigger the attack, because `a` and `b` must change **together, consistently**.

17. How is the value `0x1234ABCD` split into two 16-bit shares and merged back as in K-09? Why does this technique work against memory scanning?

Splitting: `yuksek=0x1234`, `dusuk=0xABCD`. Merging: `(yuksek<<16)|dusuk`; `0x1234<<16 = 0x12340000`, OR-ing this with `0x0000ABCD` gives `0x1234ABCD` — the original value comes back. Why it works: memory-scanning tools usually look for "a contiguous, fixed-size, high-entropy block" (e.g., a 16/32-byte key pattern); once the value is split into two separate, non-adjacent variables, this "single block" pattern disappears.

18. In section 7's small bytecode example, how does the stack change when `{OP_PUSH,5,OP_PUSH,7,OP_ADD,OP_RET}` runs, and what is the result? Why can't an analyst see this result directly 'in the source code'?

In order: `OP_PUSH 5` → stack `[5]`; `OP_PUSH 7` → stack `[5,7]`; `OP_ADD` → pops `7` and `5`, pushes `5+7=12` → `[12]`; `OP_RET` → returns `12`. Result: `12`. The analyst can't see this in the source because there is no "addition" instruction anywhere — the addition is a **side effect** of the `yorumla` function processing these six bytes in a particular order; the interpreter (the VM) itself has to be solved first.

19. Why can a bogus operation in K-03 (`crc ^= sabit^sabit`) be completely deleted by an optimising compiler (`-O2`)? What do you do to prevent this?

`sabit^sabit` is always `0` (the XOR identity from section 0: a value XORed with itself gives `0`), and `crc^=0` leaves `crc` unchanged. The compiler can spot this by constant folding and remove it via **dead code elimination**, saying "this has no observable effect"; the obfuscation vanishes without you noticing. Prevention: decide the values used in the bogus operation **at runtime** (tied to an input/opaque predicate), never fix them at compile time; always verify obfuscation on a binary compiled with optimisation on.

20. In section 8's example, if a 512-byte function's binaries produced with two seeds differ by 340 bytes, what is the difference ratio as a percentage? What does this ratio show, and what does it not show?

`fark_orani = 340/512×100 ≈ %66.4`. This shows how **different** the two copies look at the code level — it implies a high likelihood that a byte-level patch written for one copy will not work at the same offset in the other. What it does not show: it does not raise a single copy's **potency/resilience**; it only makes a crack harder to **scale** (section 8's main claim).

21. In this week's demo, `erisim_ver` goes from 28→51 instructions, 3→6 branches after flattening. What is the simple cyclomatic complexity (branches+1) before/after, and by what percentage does it increase? Which metric (section 9) does this represent?

Before: `3+1=4`. After: `6+1=7`. Increase: `(7-4)/4×100=%75`. This represents **potency** among section 9's four metrics — a rough indicator of how many paths a human analyst has to trace to understand the function; the `%82` increase in instruction count in the same example represents the **cost** metric.

22. Why does symbolic execution make K-01's `((x*(x+1))&1)==0` opaque predicate 'classic and weak'? How does a solver prove it?

Symbolic execution keeps `x` symbolic rather than concrete and asks a constraint solver "is this expression true for every `x`?" The solver splits `x` into even/odd (if even, `x=2k`; if odd, `x=2k+1`) and proves in both cases that `x*(x+1)` is a multiple of `2`, so `&1=0`. Because the proof takes seconds, the dead branch is eliminated automatically; that's why week 14 recommends predicates more resistant to solvers (based on factorization/hash values).

23. In attack potential frameworks, which five factors are generally evaluated? Which of these does applying only string obfuscation (K-07) affect, and by how much?

Generally: **time required**, **expertise required**, **knowledge of the target**, **window of opportunity**, and **equipment required**. Applying only K-07 (string obfuscation) raises the required time by only a few minutes (searching for the string by hand in a decompiler instead of running `strings`), but barely changes the required expertise — it's a step anyone can do. Layers like control-flow flattening + opaque predicate + diversification raise time, expertise, and window of opportunity together.

24. In K-12, what happens if the re-encryption step is skipped on an error/exception exit? What is the rule?

If the code falls into an error outside the normal (successful) path, it stays **permanently exposed (decoded)** in memory; the "exposure window" is no longer short as defined, it becomes infinite, and a memory dump can catch it at any time. Rule: tie the re-encryption step to the language's "always runs" construct (`goto cleanup:` in C, RAII/a destructor in C++, `finally` in Java/Kotlin); don't handle success and error paths separately and forget one.

25. What is the difference between 'encoding,' 'encryption,' and 'obfuscation'? Which of these is K-07's 'string encoding'?

Encoding: converts data into another representation, with no mathematical secrecy claim (e.g., XOR). **Encryption:** key-based, provides provable security (e.g., AES-GCM, week 3); cannot be broken without the correct key. **Obfuscation:** makes code harder for a human to understand, carries no mathematical security claim. K-07's "string encoding" is an **encoding** (via XOR); since the decoding key also sits in the binary, it is not **encryption**, it only slows down static scans like `strings`.

26. Why are the five families in the taxonomy ordered 'potency and cost rise from bottom to top'? Why is the layout family the weakest?

A family's potency is proportional to **how close it sits to the mechanism that produces the program's behaviour**. The layout family only changes names/metadata, touching nothing that produces the behaviour — that's why it's the cheapest but weakest layer (as we saw in section 0's decompiler example, names are already lost but the logic remains). Virtualisation, at the other end, changes the machine code itself, making it both the strongest and the most expensive.

27. Why is K-10's 'virtualisation-based obfuscation' not the same thing as the JVM/ART virtual machine that Java/Kotlin applications run on?

JVM/ART is a **general-purpose, standard** virtual machine that the whole application runs on; its bytecode format (DEX) is known and easily read by general tools like `jadx`. The VM in K-10 is a **project-specific, non-standard** bytecode and interpreter designed only for a handful of selected critical functions; no general tool can decode it directly. An application being written in Java does not mean it is "virtualised" in K-10's sense.

28. Why is diversification in DRM and game anti-cheat systems described as a 'continuous process' rather than a 'one-off'?

When a version of a protection is cracked (a bypass spreads), that crack is **specific to that version only**; the vendor usually releases a different protection arrangement (a new seed, new opaque predicates) in a new version, and the old crack stops working. That's why diversification in S9 should be described not as "I applied it once" but as "a process reapplied in every release" — the version-level concrete form of "Rule 3: buy time for the other layers" from section 1.

29. What does Barak et al.'s 2001 'perfect black-box obfuscation is impossible' result mean? Does this mean this week's rules are useless?

The result mathematically proves that for **every** program there cannot exist a perfect transformation under which an attacker looking inside the code **learns nothing extra** beyond observing input-output behaviour. This does **not** mean the rules are useless; it only means that no protection can be marketed as "absolute/unbreakable secrecy." This is the academic reason why we write every rule this week not as "unbreakable" but as "this is the cost it imposes, this is its limit, this is how much it withstands" (the template in section 4).

30. In section 9's two-asset example, why is it the wrong decision to apply an expensive transform like K-04 to Asset A (low-value text)?

K-04 (flattening) carries a significant cost (an 82% instruction increase, 100% branch increase in this week's demo). Paying this cost for a low-value asset does exactly the opposite of Rule 1 in section 1 ("push the cost above the value"): you needlessly raise the **defender's** cost, not the attacker's. The right decision is to settle for a low-cost rule like K-07.

31. In K-06's bogus parameter example, what is the purpose of the line `(void)gunluk_seviyesi;`?

The `gunluk_seviyesi` and `ayarlar` parameters are never used inside the function (they were added only to make the signature misleading); most compilers issue an "unused parameter" warning. The `(void)parametre` line silences this warning and signals that the parameter is **intentionally** unused (a design choice, not a bug); this way the bogus parameter escapes notice even under real code review.

14. Sources and further reading

- **Secure Programming Cookbook for C and C++** (Viega & Messier) — Chapter 12, Anti-Tampering: the purpose and limit of obfuscation, a recommendation to combine several simple techniques.
- **Secure programming technical guide** (course source) — native code hardening countermeasures: opaque loop, arithmetic/string/parameter obfuscation, opaque boolean, bogus operation/dead branch, control-flow flattening, and random exit.
- C. Collberg, J. Nagra, *Surreptitious Software* — the obfuscation taxonomy and the potency/resilience/stealth/cost framework.
- S. Schrittwieser et al., "Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?" (ACM Computing Surveys, 2016) — an overview and deobfuscation tools.
- S. Banescu et al., measuring transform resilience with Tigress + KLEE — a bridge to week 14.
- Obfuscator-LLVM (O-LLVM) — an example of compiler-based obfuscation.

Which source should you open, and when?

These sources don't replace each other; each answers a different question:

- **While writing your S9**, if you get stuck on "what is this technique called, how is it defined" → look at the **technical guide** and **Cookbook Chapter 12**; these two are the direct source of this week's K-01–K-12 rules.
- **"Who proposed this taxonomy, when, and why these five families?"** → look at **Collberg & Nagra** (the detailed version of the short history box in section 1).
- **"How resilient is this protection really, and against which tool is it measured?"** → look at **Schrittwieser et al.** (a general survey) and **Banescu et al.** (a concrete measurement with Tigress+KLEE); this is the academic foundation of the deobfuscation discussion in section 10.
- **"How do I apply these rules with a tool instead of by hand?"** → look at the **Obfuscator-LLVM** documentation; week 14 will show this applied in practice through Tigress.

> **Next week**
>
> **Week 10 — Certificates and cryptographic methods.** This week we said "obfuscation does not protect the key"; week 10 covers choosing keys correctly, their lifecycle, and protecting them with PKI. Week 11 covers whitebox cryptography, and week 14 covers this week's rules' automated counterpart (Tigress).