Tigress and Diversification

CEN429 Secure Programming — Week 14

Asst. Prof. Dr. Uğur CORUH · 18.12.2026

Today's Plan (3 Hours)

Hour	Section	Topic
1	0–1	Basic concepts · source-to-source obfuscation · Tigress · license · basic flow
2	2	Transform families · week 9 mapping · transform pipeline · step-by-step example
3	3–4	Diversification · measurement · in-class flow · build pipeline (S15) · project

A Brief History — Source-to-Source Obfuscation and Diversification

- **1993** — Cohen: the idea of **diversification** (same function, different binary)
- **1997** — Collberg et al.'s obfuscation taxonomy (the foundation of week 9)
- **2013** — **Obfuscator-LLVM**: compiler-based obfuscation
- **2010s** — **Tigress**: source-to-source + virtualisation + diversification for C
- **2016–17** — Banescu et al. **measure resilience** with Tigress+KLEE

> Main rule: **resilience ↔ cost**; protection is chosen in proportion to the value of the asset.

Where Does This Week Fit?

- **Week 9:** we learned obfuscation rules **by hand**
- **Week 11:** whitebox for the key
- **This week (14):** applying the same rules **with a tool, automatically**, and **diversified** → Tigress

All three form a whole; today is the automation step.

Learning Outcome

This week is about **LO.3** (binary application protections).

By the end, you will be able to:

- Explain source-to-source obfuscation
- Describe how to set up a **transform pipeline**
- **Diversify and measure** obfuscation
- Put obfuscation into the **build pipeline** (S15)

A Reminder from the Start

Week 9's main rule still applies this week:

> Obfuscation does not grant unbreakability, it raises **cost**.

Tigress is not **magic**; it automates what we did by hand and makes **diversification** easier.

0. Basic Concepts (From Scratch)

Why This Section?

Today terms such as "source-to-source," "transform," "seed," and "build pipeline" will come up.

Let's first define all of them **one by one**.

Reminder · Source, Compiler, Binary

- **Source code:** the program text a human writes (a C file).
- **Compiler:** the program that translates source into machine code (gcc/clang).
- **Binary file:** the resulting executable.

Reminder · Obfuscation

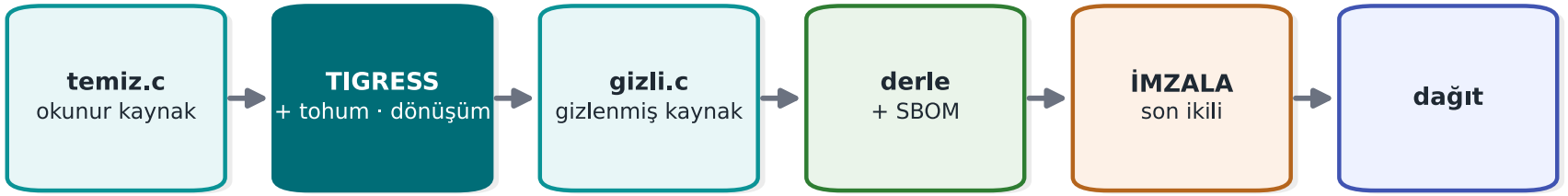
- **Code obfuscation:** making code **hard to understand** without changing its behaviour.
- Goal: raise the cost for the attacker (week 9).

Source-to-Source

- Input: C source. Output: **C source again** — but obfuscated.
- It is then compiled with your normal compiler.

Kaynaktan kaynağa gizleme hattı (S15)

imza EN SON: önce gizle, sonra imzala



Her hattan sonra: 1) davranışı test et · 2) maliyeti ölç (boyut/hız).

What Is a Transform?

- **Transform:** a single obfuscation operation applied to the source (e.g., flattening).
- The tool offers many transforms; you choose which one, and where.

Transform Pipeline

- **Pipeline:** applying several transforms **in sequence**.
- Each transform is applied to the output of the previous one.
- Order matters.

What Is a Seed?

- **Seed:** a starting number that governs randomness.
- Same transform + different seed = **different** obfuscated output.
- This is the key to diversification.

Diversification

- **Diversification:** producing **behaviourally equivalent, structurally different** binaries from the same source.
- An attack written against one copy does not work on another (week 9 Rule 2).

What Is a CLI (Command Line)?

- **CLI (Command-Line Interface):** an interface where you run things by typing commands.
- Tigress is a CLI tool: `tigress --Transform=... dosya.c`.

Unit Test

- **Unit test:** a small test that automatically checks that a function works correctly.
- The same tests must pass **after** obfuscation (behaviour must be preserved).

CFG Reminder

- **CFG (Control Flow Graph):** a diagram that turns basic blocks into nodes and transitions into edges.
- We measure the **potency** of obfuscation by the number of nodes/edges (week 9).

Symbolic Execution (Briefly)

- **Symbolic execution:** automated analysis that solves program paths as mathematical constraints (e.g., KLEE).
- It is a **deobfuscation** method against obfuscation; we test resilience with it.

Now We're Ready

Terms:

source-to-source · transform · pipeline · seed · diversification · CLI · unit test · CFG · symbolic execution

Now: what is Tigress and how does it work?

1. Source-to-Source Obfuscation and Tigress

Three Problems with Manual Obfuscation

In week 9 we applied the rules **by hand**. It was instructive, but:

1. **Error-prone** — can break behaviour
2. **Hard to maintain** — the source becomes unreadable
3. **Cannot be diversified** — you cannot differentiate every copy by hand

The Solution: Let a Tool Do It

- Let a **tool** do the obfuscation.
- You keep the **readable** source.
- Obfuscation becomes an **automatic** step in the build pipeline.

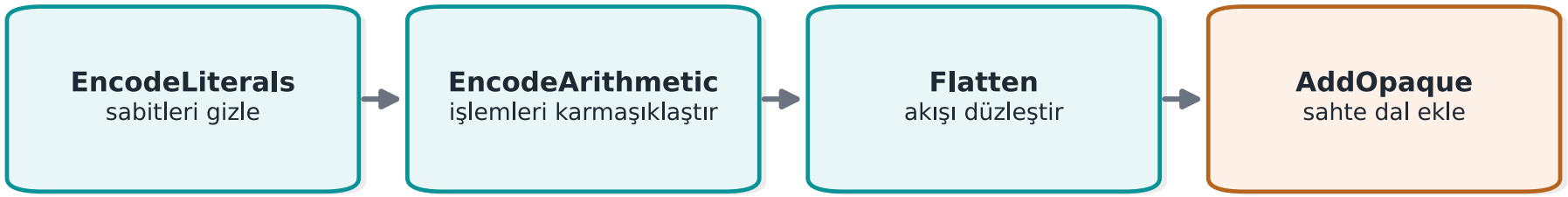
The Source-to-Source Idea

The maintenance cost stays with the **readable source**; the distributed source is obfuscated.

Transform Pipeline — Diagram

Dönüşüm hattı: sıra önemlidir

her dönüşüm bir öncekinin çıktısı üzerinde çalışır



Yanlış sıra bazı dönüşümleri etkisizleştirir ya da maliyeti gereksiz artırır.

What Is Tigress?

- A **source-to-source C obfuscator/diversifier** developed by Collberg and his team at the University of Arizona.
- The **automatic** counterpart of week 9's manual rules.

Tigress · Features

- Cross-platform: Linux, macOS, Windows, Android
- Architectures: Intel, ARM, WebAssembly
- Compilers: GCC, Clang, MSVC
- Current version: **v4**

⚠️ License (Important)

- **Non-profit** (academic/research) use: **free**.
- **Commercial** use: requires a **license** from the University of Arizona.
- The source code is not open; researchers may request access to an encrypted source.

⚠️ Rule for Use in Class

- The **Tigress binary or an old version is not distributed** in class.
- Students download the current version themselves from the **official site** (`tigress.wtf`).
- They verify the license terms **there**.
- You apply it only to **your own** code, on **your own** machine.

Why a Tool, Not by Hand? (1)

By Hand (Week 9)	With a Tool (Tigress)
Instructive	Scales
Error-prone	Preserves behaviour (tested)
Source becomes unreadable	Readable source stays with you

Why a Tool, Not by Hand? (2)

By Hand	With a Tool
Diversification isn't feasible by hand	Automatic via seed
Uniform, recognisable	Varies with transform+seed

But a Tool Is Not Magic Either

- Tigress also does **not** grant unbreakability, it raises cost.
- The patterns of a well-known tool can become recognisable over time.
- That's why **diversification** and **layered defence** (RASP, server) are still essential.

Basic Flow · a Single Transform

```
# girdi: temiz.c → çıktı: gizli.c
tigress --Transform=Flatten --Functions=erisim_ver \
        --out=gizli.c temiz.c
cc -o program gizli.c
```

Basic Flow · Three Ideas

1. `--Transform=...` which transform (Flatten = flattening, week 9 K-04)
2. `--Functions=...` which functions (only the sensitive ones — the cost rule)
3. The output is C again; you compile it **with your own compiler**

Why Do We Target with `--Functions`?

- If obfuscation is applied to every function, the program **slows down/bloats a lot**.
- We target only sensitive functions (license, key derivation, integrity).
- This is the tool's version of week 9's "cost rule."

Section 1 — Quick Check

1. What is source-to-source obfuscation?
2. Three advantages over manual obfuscation?
3. Why isn't Tigress distributed in class?

Section 1 — Answers

1. A transform is applied to the source code, producing **new source (still C)**, which is then compiled normally. Tigress takes C, gives obfuscated C.
2. **Repeatable/automatic** (every build), **consistent and scalable** (many functions), the **original source stays clean** + diversification via seed.
3. **License/terms of use**; everyone downloads it themselves. If Tigress isn't available, demos work with a **clean derivative/fallback**.

2. Transform Families and Pipeline

Transforms = Week 9's Rules

Tigress transforms correspond to week 9's obfuscation families.

Let's look at them group by group; we'll tie each one to a K-rule.

Transform ↔ Rule Mapping — Diagram

Tigress dönüşümleri ↔ 9. hafta kuralları

araç yeni bir şey icat etmiyor — aynı kuralları otomatikleştiriyor

Flatten	K-04 kontrol akışı düzleştirme
AddOpaque / InitOpaque	K-01 opak yüklem · K-03 ölü dal
EncodeArithmetic	K-02 aritmetik kodlama
EncodeLiterals	K-07 sabit / dize gizleme
Virtualize	K-10 sanallaştırma (en pahalı)

Hangi dönüşümü seçtiğinizi S9'da kuralla eşleyerek gerekçelendirin.

Control-Flow Transforms

- **Flatten:** flattens control flow → **K-04**
- **InitOpaque / AddOpaque / UpdateOpaque:** opaque predicate + bogus branch → **K-01, K-03**

Data Transforms

- **EncodeArithmetic:** arithmetic into an equivalent complex expression (MBA) → **K-02**
- **EncodeLiterals:** encodes constants and strings → **K-07, K-08**
- **EncodeData:** encodes variable representation → **K-09**

Function Transforms

- **Split / Merge:** splits/merges functions → **K-06**
- **Virtualize:** turns the function into custom VM bytecode → **K-10**
- **Jit:** generates code at run time → **K-12 (dynamic)**

Anti-Analysis Transforms

- **AntiBranchAnalysis / AntiAliasAnalysis / AntiTaintAnalysis:** make static analysis techniques harder → preventive family

Diversification Transforms

- **RandomFuns:** random bogus functions
- **RndArgs:** bogus parameters
- Combined with a seed, every build differs → diversification, **K-06**

Diversification in Space/Time — Diagram

Uzayda ve zamanda çeşitlendirme

ikisi birlikte saldırının yeniden kullanımını kırar

UZAYDA
her kullanıcı/kopya farklı

bir kırık TEK kopyayı açar
ölçeklenemez saldırı

ZAMANDA
her sürüm farklı

bir kırık KALICI olmaz
saldırgan tekrar başlar

Çeşitlendirme tek kopyanın gücünü artırmaz; saldırının ÖLÇEKLENMESİNİ engeller.

Transform ↔ Rule · Table

Tigress	What It Does	Week 9
Flatten	Flattening	K-04
AddOpaque	Opaque predicate/bogus	K-01, K-03
EncodeArithmetic	Arithmetic encoding	K-02
EncodeLiterals	Constant/string	K-07, K-08
Virtualize	Virtualisation	K-10
RandomFuns/RndArgs	Diversification	K-06

Families and Cost

- **Cheap:** EncodeArithmetic, EncodeLiterals
- **Moderate:** Flatten, opaque predicates
- **Expensive:** Virtualize (tens of times slower)

That's why Virtualize is targeted only at **small, critical** functions; via `--Functions`.

Transform Pipeline

"Not One Technique, Together"

Week 9's most important rule.

In Tigress, this means applying transforms **in sequence**.

Conceptual Pipeline

```
tigress \
  --Transform=EncodeLiterals   --Functions=erisim_ver \
  --Transform=EncodeArithmetic --Functions=erisim_ver \
  --Transform=Flatten          --Functions=erisim_ver \
  --Transform=AddOpaque        --Functions=erisim_ver \
  --out=gizli.c temiz.c
```

What Does the Pipeline Do?

The single check (`erisim_ver`):

1. Encode constants/strings
2. Encode its arithmetic
3. Flatten
4. Strengthen with opaque predicates

= the automatic version of what week 9's K-04 called "strengthened flattening."

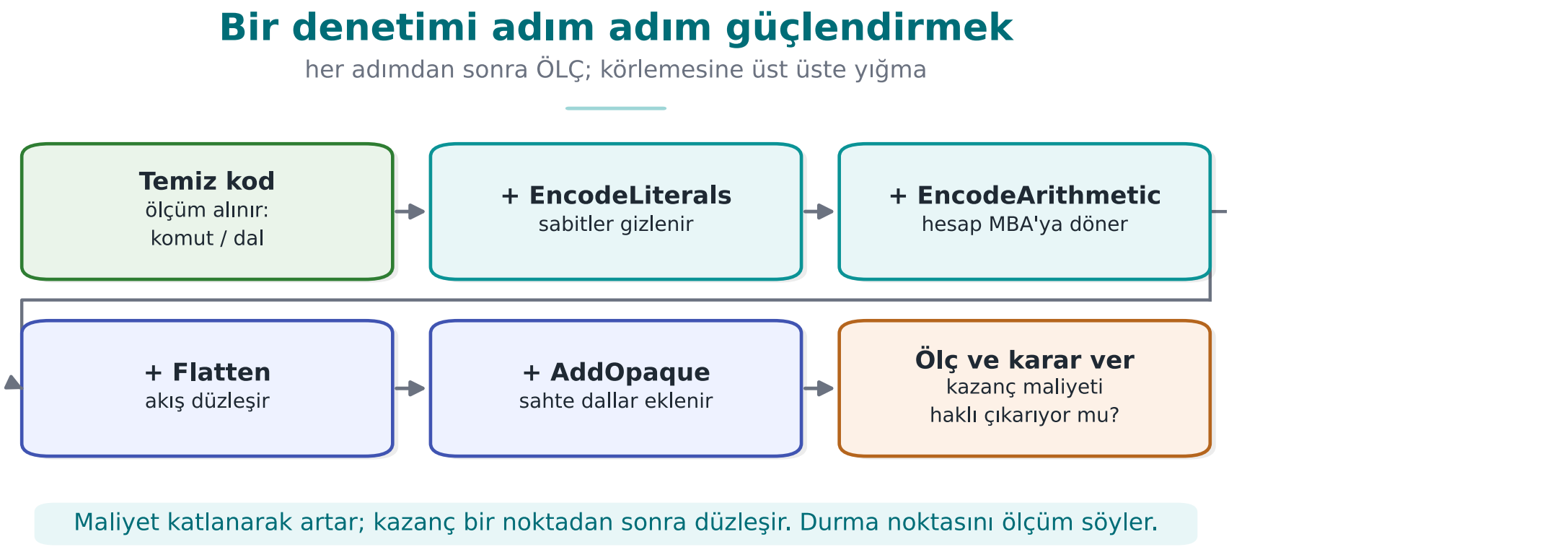
⚠ Order and Testing Rule

- Transform order **affects** the result; some orders needlessly hurt performance.
- **Rule:** after every pipeline, run the **unit tests** (was behaviour preserved?) and measure size/speed.
- Obfuscation must **never** change behaviour; if it does, the pipeline is wrong.

This connects to week 12's "S16 results."

Step-by-Step Example

Step-by-Step Strengthening — Diagram



Maliyet katlanarak artar; kazanç bir noktadan sonra düzleşir. Durma noktasını ölçüm söyler.

Starting Point · temiz.c

```c
int erisim_ver(const char *jeton) {
    if (jeton_gecerli(jeton)) return IZIN;  /* tek dal */
    return RED;
}
```

Unprotected: a single branch, bypassed with `strings` and a single-byte patch.

Step 0 · The Attacker

- `strings` → `IZIN` / `RED`
- Reverse-compile → a single `if`
- Turn the failure branch into success

Time: minutes.

Layer by Layer

Step	Transform	What Changes	Cost
0	—	Bypassed with a single-byte patch	—
1	EncodeLiterals	IZIN/RED no longer plain	Very low
2	EncodeArithmetic	Comparison complex	Low
3	Flatten	Single branch not visible	Moderate
4	AddOpaque	Bogus branches	Moderate
5	Virtualize (if needed)	VM bytecode	High

Takeaway

- Every step makes the attack a bit more **expensive**.
- Every step adds a **cost**.
- Where you stop: **asset value** + **the cost you measured**.

Step 5 (Virtualize) is unnecessary for most checks.

This Table → S9 Itself

This "step / transform / what changes / cost" table

is the direct counterpart of your project's **S9 protection table**.

Section 2 — Quick Check

1. Which K-rules do Flatten, EncodeArithmetic, Virtualize correspond to?
2. Why do you target with `--Functions`?
3. Two things you **must always** do after every pipeline?

Section 2 — Answers

1. **Flatten** = control-flow flattening; **EncodeArithmetic** = arithmetic/data obfuscation; **Virtualize** = virtualisation (strongest, most expensive).
2. Obfuscating every function **blows up** cost/performance; targeting only **sensitive** functions concentrates resilience there.
3. (1) **Test behaviour** (unit tests must pass), (2) **measure cost** (size/speed/instruction count, before-after).

3. Diversification and Measurement

Recall Rule 2

> An attack that works on one copy should not work on all copies.

If an attacker can crack one copy and distribute the patch to **everyone**, one crack opens every door.

What Is Diversification?

Behaviourally equivalent, structurally different binaries from the same source.

Tigress does this with a **seed**:

- Same transforms + a different seed → opaque predicates, bogus branches, state values **change**.

Two Seeds, Two Different Binaries

```
tigress --Seed=1001 --Transform=Flatten --Transform=AddOpaque \
        --Functions=erisim_ver --out=gizli_a.c temiz.c
tigress --Seed=2002 --Transform=Flatten --Transform=AddOpaque \
        --Functions=erisim_ver --out=gizli_b.c temiz.c
```

`gizli_a` and `gizli_b` do the same job; their machine code is **different**.

Two Types of Diversification

- **In space:** a differently seeded copy for each user/device → one copy's attack doesn't work on another.
- **In time:** each version gets a new seed → an old attack breaks on the new version.

Works together with week 10's key/version renewal.

What Does Diversification Prevent?

- It does not increase the **strength** of obfuscation (cracking one copy is still possible).
- But it prevents it from **scaling**.

`RandomFuns`, `RndArgs` also change structure when combined with a seed.

Diversification Effectiveness · Metric

- Compile the same source with two seeds.
- Measure the **byte difference** of the protected functions.
- The higher the difference, the lower the chance that one attack works on the other.

Write the metric into S9.

Measuring Obfuscation

Tigress's Teaching Value

In week 9 we learned to measure obfuscation on four dimensions (potency, resilience, stealth, cost).

Tigress lets you make these measurements **concrete**: obfuscate the same program, measure before/after.

Cost Measurement (Easy, Mandatory)

Measure for every pipeline and write it into S9/S15:

Metric	How
Binary size	`size` / `ls -l`
Running time	Same input, N times, average
CFG complexity	Basic block count in the decompiler

Cost Measurement · Command

```
size ./program_temiz ./program_gizli   # boyut farkı
# süre: aynı girdiyle N kez çalıştır, ortalamayı karşılaştır
```

Simple, but it produces **evidence**.

Cost · Example Table

Metric	Before	After
Size	100 KB	130 KB
Time	1.0×	1.8×
Basic blocks (target fn.)	5	40

The figures are an example; you write **your own** measurements.

Resilience Measurement (Advanced, Concept)

- Resilience = resistance to an automated tool (symbolic execution).
- Week 9: "not a claim, a measured quantity."
- Tigress is at the **centre** of this research.

Banescu et al. · Tigress + KLEE

A study measuring how much Tigress transforms resist **symbolic execution**. Findings:

- A single transform (Flatten alone) → usually unwound.
- **Combining** transforms + growing the state space → forces the solver **exponentially**.
- But **cost** also increases.

Resilience ↔ Cost — Diagram

Banescu'nun kuralı: dayanıklılık ↔ maliyet

KLEE ile ölçülen deneysel sonuç

Güçlü dönüşüm
Virtualize, çok katman

dayanıklılık YÜKSEK
performans/boyut CEZASI yüksek

Hafif dönüşüm
EncodeLiterals, opak yüklem

dayanıklılık düşük
maliyet düşük

Koruma, varlığın değeriyle ORANTILI seçilir — her şey en güçlüyle gizlenmez.

Takeaway for the Course

> Resilience and cost are a **trade-off** (like week 9).

Don't say "strong"; say **"it didn't let this tool solve a problem of this size within this time."**

Measurement Rule (S9/S15)

Defend an obfuscation decision with measurements:

> "The Flatten + AddOpaque pipeline grew the size by 30%, slowed the process by 1.8×; the target function's basic block count grew 8-fold."

Unmeasured obfuscation = a **claim** for the evaluator (week 12).

Section 3 — Quick Check

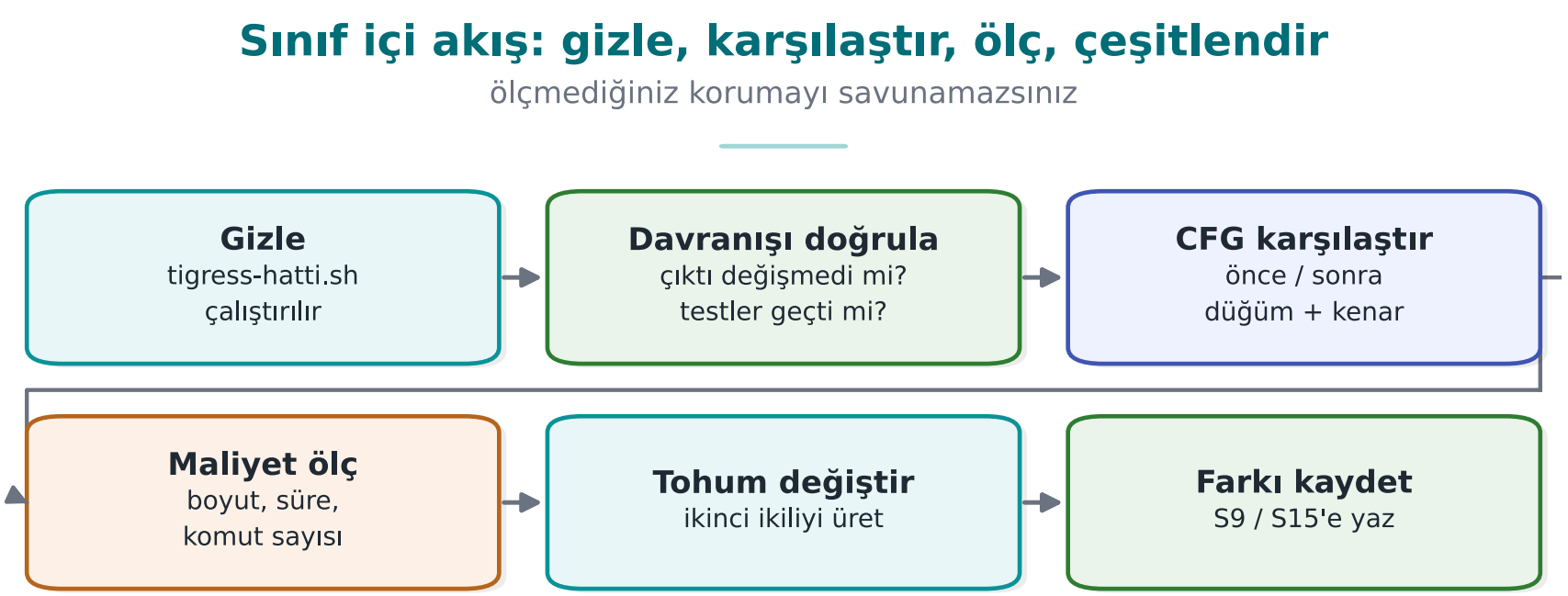
1. How does a seed provide diversification?
2. Difference between diversification in space and in time?
3. Banescu et al.'s main rule? (resilience ↔ cost)

Section 3 — Answers

1. Obfuscating the same source with a different **seed** makes the tool make different random choices → every build is a **different binary** (same behaviour).
2. **In space:** different copies/users differ (one crack doesn't crack everyone). **In time:** it changes across versions (a crack doesn't stay valid).
3. Stronger obfuscation means more **resilience** but more **cost**; protection is chosen **in proportion to the value** of the asset.

4. In-Class Flow (Step by Step)

In-Class Flow — Diagram



Davranış doğrulaması atlanırsa gizleme sessizce programı bozar — en sık yapılan hata budur.

What Will We Do?

Take your own small, synthetic program:

obfuscate → verify behaviour → compare → measure → diversify → report

Let's go step by step.

Step 1 · Preparation

- Write a small, synthetic program (e.g., `erisim_ver`).
- Show with unit tests that it **works correctly**.
- This is your "clean" baseline.

Step 2 · Obfuscate

```
tigress --Transform=EncodeLiterals --Transform=EncodeArithmetic \
        --Transform=Flatten --Transform=AddOpaque \
        --Functions=erisim_ver --out=gizli.c temiz.c
cc -o program_gizli gizli.c
```

Apply a transform pipeline.

Step 3 · Verify Behaviour

- Run the same unit tests on the **obfuscated** version.
- The result must be **exactly the same**.
- If different: the pipeline is wrong → fix it.

> Obfuscation never changes behaviour.

Step 4 · Compare

- Open the clean and obfuscated versions in a decompiler (Ghidra/objdump).
- Compare the control flow.
- Note the **increase** in basic block count.

Step 5 · Measure

```
size ./program_temiz ./program_gizli
# süre: aynı girdiyle N kez çalıştır, ortalamayı karşılaştır
```

Write down the size and time difference.

Step 6 · Diversify

- Run the same pipeline with **two different seeds**.
- Show that the two binaries are different (byte difference).

```
tigress --Seed=1001 ... --out=gizli_a.c temiz.c
tigress --Seed=2002 ... --out=gizli_b.c temiz.c
```

Step 7 · Report

Put your findings into a table:

Transform	Size	Time	Blocks	Seed Difference
…	…	…	…	…

This table goes directly into **S9/S15**.

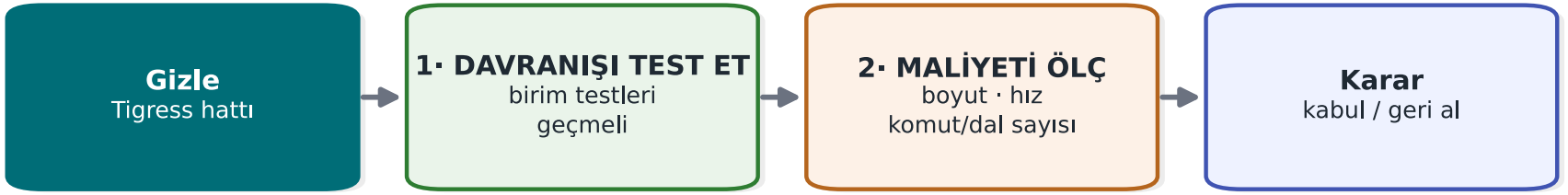
⚠ Ethical and Safe Use

- Tigress only on **your own** code.
- The example program does not **harm** the computer: it does not change system settings, does not touch the network.
- All values are **synthetic**.

In-Class · Summary Flow

Her hattan sonra mutlaka iki şey

ölçmeden 'güçlendirdim' denemez



Örnek ölçüm: erisim_ver 28 komut/3 dal → 51 komut/6 dal.

Step-by-Step Flow · Why This Order?

- First **behaviour** (tests) — make sure nothing broke.
- Then **evidence** (CFG, measurement) — show the gain.
- Then **diversification** — break scalability.

Don't say "strong" without measuring.

Section 4 — Quick Check

1. The first thing you **must always** do after obfuscating?
2. How do you show diversification?
3. Which project sections does the report table go into?

Section 4 — Answers

1. **Test that behaviour hasn't changed** (unit tests must pass); obfuscation must not break function.
2. Produce with two different seeds; compare the binaries (**hash/size/instructions differ**) but **same input → same output** (diff/objdump).
3. **S9** (code hardening) and **S15** (build/deployment pipeline); before/after measurement as evidence.

5. Build Pipeline (S15) and Project

Obfuscation Is Not "By Hand at the End"

Obfuscation should be a step in the build **pipeline**.

Your project's **S15** section, after the midterm, documents exactly this.

Build Pipeline — Diagram

Sıra bozulursa imza geçersizleşir; TOE kimliği (12. hafta) buradan çıkar.

Pipeline Rule 1 · Sensitive Functions Only

- Obfuscate only sensitive functions with `--Functions`.
- Write down the rationale (which function, why).
- The cost rule (week 9).

Pipeline Rule 2 · Every Version Is Diversified

- Every version is diversified with a **seed**.
- The seed and build identity are **recorded**.
- Diversification in time (section 3).

Pipeline Rule 3 · Signing After Obfuscation

- First obfuscate, **then** sign.
- The build identity and hash values (week 12's TOE) must be **consistent**.
- Why afterwards? The signature must cover the **final** distributed binary.

Pipeline Rule 4 · Automatic in CI

- The pipeline runs in continuous integration (CI).
- Every version: unit tests + size/speed measurement **automatic**.
- If behaviour breaks, the build **stops**.

9, 11, 14 Together

- **9:** obfuscation rules (code)
- **11:** whitebox (key)
- **14:** automation + diversification (this week)

Common rule: protection = **delay**; strength comes from layers and measurement.

Project · S9 Extended + S15 Pipeline — 1

1. Transform pipeline (S9): which functions, which transforms, in what order, **why**.

Project · S9 + S15 — 2

2. Measurement table: size, time, (if available) block count — before/after.

Project · S9 + S15 — 3

3. Diversification: production with two seeds + a difference metric; if you won't apply it, a **rationale**.

Project · S9 + S15 — 4, 5

4. S15 pipeline: document the obfuscation + signing + build identity steps with a **diagram**.

5. Behaviour evidence: show that the obfuscated version passes the unit tests (the S16 link).

Through the Evaluator's Eyes

- Not "I used a lot of transforms."
- **A rationale and a measurement for every transform.**
- Unmeasured obfuscation = a claim (week 12).

Section 5 — Quick Check

1. Why is signing **after** obfuscation?
2. The rationale for the "sensitive functions only" rule?
3. The steps of the S15 pipeline?

Section 5 — Answers

1. The signature protects the **final binary**; if you sign first and obfuscate afterwards, the binary changes → the signature becomes invalid. Order: build → obfuscate → **sign**.
2. Obfuscating everything blows up performance/size and most code isn't sensitive; concentrate protection on **critical** functions → low cost, high impact.
3. Source → (static analysis) → build → **obfuscate** → diversify (seed) → package/SBOM → **sign** → deploy (all recorded in CI).

Solved Self-Check

Question 1

What is source-to-source obfuscation? Three advantages over manual obfuscation?

Answer: A tool takes C source and produces obfuscated C source. Advantages: preserves behaviour (error-resistant), the readable source stays with you, easy diversification via seed.

Question 2

Tigress's place in the course; which license, why isn't it distributed in class?

Answer: The automatic form of week 9's rules. Non-profit use is free, commercial needs an Arizona license. The source is closed → its binary is not distributed in class; students download it from the official site.

Question 3

Map these transforms to week 9's rules: Flatten, EncodeArithmetic, EncodeLiterals, Virtualize, AddOpaque.

Answer: Flatten→K-04, EncodeArithmetic→K-02, EncodeLiterals→K-07/K-08, Virtualize→K-10, AddOpaque→K-01/K-03.

Question 4

Why is a transform pipeline stronger than a single transform? Why does order matter?

Answer: Layers strengthen each other (week 9's "together"). Order affects the result and performance; some orders needlessly slow things down or give a weak result.

Question 5

Which two things do you always do after every transform pipeline?

Answer: (1) Run the unit tests (was behaviour preserved?), (2) measure size/speed (cost).

Question 6

Why is Virtualize applied only to small and critical functions?

Answer: Its cost is very high (tens of times slower, size). The benefit only justifies the cost for the most valuable, small functions.

Question 7

How does the seed (`--Seed`) provide diversification? Difference in space/time?

Answer: A different seed → different opaque predicate/bogus branch/state. In space: different copies at the same time. In time: each version gets a new seed.

Question 8

Does diversification prevent the strength of obfuscation, or its scalability?

Answer: It does not increase strength (one copy can still be cracked); it prevents **scalability** — one crack doesn't open every door.

Question 9

The main rule for the course from the Banescu et al. study?

Answer: Resilience is a measured quantity. Combining transforms forces symbolic execution exponentially; but cost also increases → **resilience ↔ cost trade-off**.

Question 10

When putting obfuscation into the build pipeline, why does signing come after obfuscation?

Answer: The signature must cover the **final** distributed binary. It's obfuscated first, then signed; the build identity/hash stays consistent.

Question 11

The rationale for S15's "only sensitive functions are obfuscated" rule?

Answer: Obfuscating every function slows down/bloats the program a lot. It makes sense to take on the cost only for valuable functions.

Question 12

Write the common rule of weeks 9, 11, and 14 in one sentence.

Answer: Protection provides not unbreakability but **delay**; its strength comes from **combining layers**, from **diversification**, and from being **measured**.

Closing · Three Weeks

- **9:** obfuscation rules (by hand)
- **11:** whitebox (key)
- **14:** automation + diversification

All one framework: cost + layers + measurement.

Appendix A · A Worked Measurement

Scenario

Let's obfuscate the `erisim_ver` function with two pipelines and measure:

- **Pipeline A:** EncodeLiterals + EncodeArithmetic (light)
- **Pipeline B:** Pipeline A + Flatten + AddOpaque (heavy)

Before · Baseline

```
program_temiz:
  boyut: 100 KB
  işlem süresi: 1.00× (referans)
  erisim_ver temel blok: 5
```

Pipeline A · Measurement (Example)

```
boyut: 103 KB   (+%3)
süre:  1.05×
blok:  7
strings hassas dize: 0  (öncesi 3)
```

Cheap; string obfuscation gives a big return.

Pipeline B · Measurement (Example)

```
boyut: 131 KB   (+%31)
süre:  1.80×
blok:  40
```

Potency increased (blocks 5→40); cost increased too.

Comparison Table

Metric	Clean	Pipeline A	Pipeline B
Size	100 KB	103 KB	131 KB
Time	1.00×	1.05×	1.80×
Blocks	5	7	40
Sensitive strings	3	0	0

Decision

- Asset value **low** → Pipeline A is enough.
- Asset value **high** → Pipeline B (accept the cost) + diversification.

> The decision rests on numbers, not a feeling.

Diversification Measurement (Example)

- Produce Pipeline B with seed 1001 and 2002.
- Byte difference of the protected function: **92%**.
- Comment: a patch written for one copy most likely won't work on the other.

Appendix B · Mini Case

Case · Setup

A synthetic application has a **license check** and a **key derivation** function.

Should we heavily obfuscate both?

Case · Decision

- **License check:** medium value → Pipeline B (Flatten + AddOpaque).
- **Key derivation:** high value → Pipeline B + **Virtualize** (acceptable cost because it's small).
- The rest: not obfuscated (unnecessary cost).

Case · Diversification + Pipeline

- Each version, a different seed.
- Build pipeline: obfuscate → sign → build identity.
- Unit tests + measurement automatic in CI.

Case · S9/S15 Output

Function	Pipeline	Size/Time	Rationale
lisans_dogrula	B	+31% / 1.8×	medium value
anahtar_turet	B+Virtualize	+60% / 3×	high value, small fn.

Case · Residual Risk

- A sufficiently determined attacker can still solve a single copy.
- Mitigation: diversification + a short-lived key + server-side checking.
- **Written down explicitly.**

Appendix C · Alternatives and Limits

O-LLVM (Compiler-Based)

- **Obfuscator-LLVM:** applies transforms **at compile time**.
- Not source-to-source; on top of the LLVM intermediate representation.
- Week 9 K-11. An alternative to Tigress.

Tigress or O-LLVM?

	Tigress	O-LLVM
Where	Source (C→C)	Compiler (IR)
Visibility	You can see the obfuscated source	A build step
Transform richness	Many (including Virtualize)	More limited

Shared Limit

- Both are well known → their patterns can become recognisable.
- Both require **diversification** and **layered defence**.
- Neither one grants unbreakability.

Glossary

Term	Meaning
Source-to-source	Obfuscation that takes C in and gives obfuscated C out
Transform	A single obfuscation operation
Pipeline	The ordered whole of the transforms
Seed	A number that governs randomness (diversification)
Symbolic execution	Path-solving automated analysis (KLEE)

Sources

- The **Tigress** official site/worksheets (`tigress.wtf`) — transforms, syntax, v4, license
- Collberg & Nagra, *Surreptitious Software* — taxonomy, measurement
- Banescu et al. — Tigress + KLEE resilience measurement
- Obfuscator-LLVM — a compiler-based alternative

Summary: This Week in One Sentence

> Tigress applies week 9's obfuscation rules **automatically** and **diversified**; but the rule is the same: obfuscation
> is **not unbreakability but delay**; combine, diversify, **measure**.

Next Week

Week 15 — Final Project Demonstrations (RAP2)

The semester's content is complete. The final report is expected to include this week's pipeline (S15) and measurements (S9). Week 16: Quiz-2.

Appendix D · Transform by Transform

Why One by One?

When building a pipeline, you need to know **which transform, and why**.

For every transform: what it does · when · cost.

Flatten

- **What it does:** moves control flow into a single switch (K-04).
- **When:** to hide the algorithm's structure; in most pipelines.
- **Cost:** moderate.

AddOpaque / InitOpaque

- **What it does:** adds opaque predicates and bogus branches (K-01, K-03).
- **When:** to strengthen Flatten.
- **Cost:** moderate.

EncodeArithmetic

- **What it does:** replaces arithmetic with an equivalent complex expression (K-02).
- **When:** in functions with calculations/comparisons.
- **Cost:** low.

EncodeLiterals

- **What it does:** encodes constants and strings (K-07, K-08).
- **When:** almost always (cheap, high return).
- **Cost:** very low.

EncodeData

- **What it does:** encodes the representation of variables (K-09).
- **When:** to hide sensitive variables.
- **Cost:** low–moderate.

Split / Merge

- **What it does:** splits/merges functions; blurs structure (K-06).
- **When:** to hide function boundaries.
- **Cost:** low–moderate.

Virtualize

- **What it does:** turns the function into custom VM bytecode (K-10).
- **When:** only the most critical, **small** functions.
- **Cost: high** (tens of times slower).

Jit

- **What it does:** generates code at run time (dynamic, K-12).
- **When:** very selective; advanced use.
- **Cost:** high; take care with OS protections.

AntiBranchAnalysis / AntiAliasAnalysis / AntiTaintAnalysis

- **What it does:** makes the corresponding static analysis technique harder (preventive family).
- **When:** against a specific analysis threat.
- **Cost:** variable.

RandomFuns / RndArgs

- **What it does:** adds random bogus functions/parameters.
- **When:** to strengthen diversification (with a seed).
- **Cost:** low–moderate.

Which Transform First?

Typical order (conceptual):

1. EncodeLiterals (cheap cleanup)
2. EncodeArithmetic
3. Flatten
4. AddOpaque
5. (if needed) Virtualize
6. RandomFuns/RndArgs (diversification)

Appendix E · Obfuscation Decision Flow

"Should I Obfuscate This Function?" — 1

Question 1: Is this function sensitive? (license, key, integrity, check)

- **No** → don't obfuscate; don't waste the cost.
- **Yes** → continue.

"Should I Obfuscate This Function?" — 2

Question 2: Is its value high?

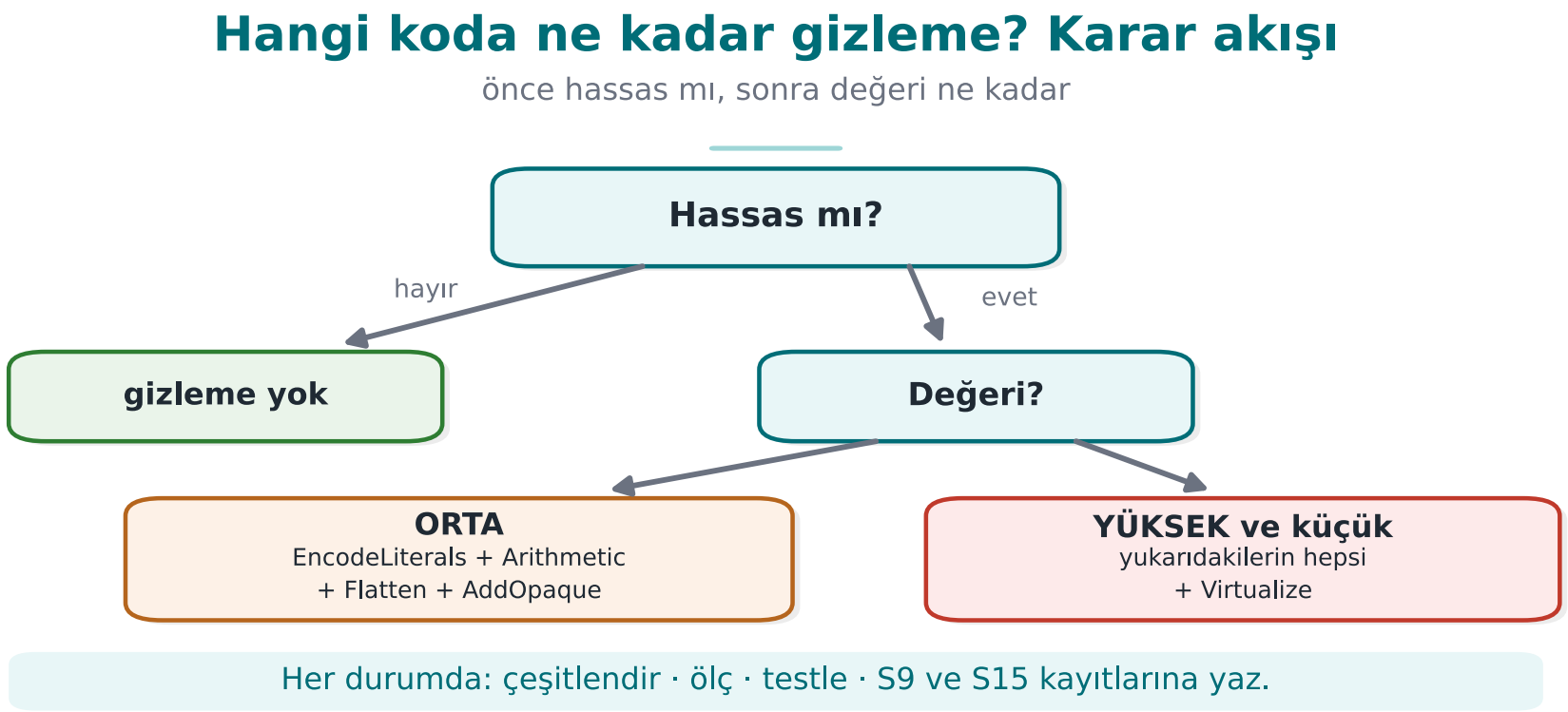
- **Medium** → EncodeLiterals + EncodeArithmetic + Flatten + AddOpaque.
- **High and small** → the above + Virtualize.

"Should I Obfuscate This Function?" — 3

Question 3: In every case:

- Diversify (seed)
- Measure (size, time, blocks)
- Verify behaviour with unit tests
- Write it into S9/S15

Decision Flow · At a Glance



Her durumda: çeşitlendir · ölç · testle · S9 ve S15 kayıtlarına yaz.

Appendix F · Common Mistakes

Mistake · Obfuscating Everything

- Obfuscating all functions slows down/bloats the program a lot.
- Only sensitive and valuable functions.

Mistake · Obfuscating Without Tests

- If unit tests aren't run after obfuscation, broken behaviour goes unnoticed.
- Testing after every pipeline is **mandatory**.

Mistake · Saying "Strong" Without Measuring

- Without measurement, the evaluator (week 12) treats it as a **claim**.
- Write down size/time/block numbers.

Mistake · Skipping Diversification

- Uniform obfuscation → one crack opens every copy.
- A different seed for every version/deployment.

Mistake · Signing Before Obfuscating

- The signature must cover the **final** distributed binary.
- Obfuscate first, then sign.

Mistake · Using an Old/Distributed Tigress

- The binary isn't distributed in class; download the current version from the official site.
- Verify the license terms.

Appendix · Closing Note

These appendices gave you a concrete template for building **your own pipeline**:

- transform-by-transform selection
- decision flow
- mistakes to avoid

> The right tool + the right measurement + diversification = a defensible S9/S15.

Appendix G · A Second Worked Pipeline

New Example · Integrity Check

```
int dosya_saglam(const uint8_t *veri, size_t n,
                 const uint8_t *beklenen) {
    uint8_t ozet[32];
    ozet_hesapla(veri, n, ozet);        /* SHA-256 benzeri */
    return sabit_zamanli_esit(ozet, beklenen, 32);
}
```

A synthetic function that checks version integrity.

Why Obfuscate This?

- An attacker wants to find this check and **bypass** it (to get a tampered file to pass).
- The check's return point and comparison are the target.
- Week 9: opaque boolean + random exit are critical here.

Pipeline Choice

```
tigress \
  --Transform=EncodeLiterals \
  --Transform=EncodeArithmetic \
  --Transform=Flatten \
  --Transform=AddOpaque \
  --Functions=dosya_saglam \
  --out=gizli.c temiz.c
```

Step by Step · What Happens?

- EncodeLiterals: `32` and any strings are no longer plain.
- EncodeArithmetic: the comparison becomes complex.
- Flatten: the single return point is not visible.
- AddOpaque: bogus branches; the "success" branch isn't in one place.

Behaviour Verification (Mandatory)

- A file with the correct hash → **passes**.
- A file with a corrupted hash → **rejected**.
- The obfuscated version gives the same result → pipeline correct.

CFG Before/After (Concept)

Where the check passes/fails can no longer be read from the flow.

⚠️ Constant-Time Behaviour Must Be Preserved

- Even obfuscated, `sabit_zamanli_esit` must **stay constant time**.
- If an early exit is added for the sake of obfuscation, a side channel opens (week 3).
- **Test** that the transforms don't break this.

Measurement (Example)

Metric	Before	After
Size	100 KB	128 KB
Time	1.0×	1.7×
Blocks (dosya_saglam)	4	33

Diversification

- Produce with two seeds → two different binaries.
- A "skip the integrity check" patch that works on one copy won't work on the other, even if it succeeds once.

Second Example · Takeaway

- Obfuscation is valuable on functions that are a **bypass target**, like integrity checks.
- But the real strength: obfuscation + RASP (week 6) + server-side checking.
- Obfuscation alone **delays** bypass, it doesn't prevent it.

Appendix H · Quick Reference

Transform Selection Guide

You Want	Transform
Hide a string/constant	EncodeLiterals
Hide a calculation	EncodeArithmetic
Hide the structure	Flatten
Add bogus/opaque	AddOpaque
Hide machine code	Virtualize (expensive)
Diversify	--Seed, RandomFuns

Checklist

- [ ] Only sensitive functions (`--Functions`)
- [ ] Pipeline order makes sense (cheap to expensive)
- [ ] Unit tests pass (behaviour)
- [ ] Size/time/blocks measured
- [ ] Diversified with two seeds
- [ ] Signing after obfuscation
- [ ] Written into S9/S15

Final Word (Week 14)

> The tool is powerful, but the **decision is yours**: what, why, at what cost are you obfuscating?

Measure, diversify, layer. Obfuscation is not unbreakability, it is **time bought**.