Hafta 14 — Tigress ve Çeşitlendirme

Tarih	18.12.2026
Öğrenme çıktıları	ÖÇ.3
Süre	3 saat
Ön bilgi	Hafta 9'dan gizleme kuralları ve ölçüm çerçevesi; C'de derleme; Linux/WSL terminali (Tigress yalnız Linux'ta çalışır)
Uygulamalar	`code/week-14` — 1 demo; WSL/Linux'ta `sh tigress-hatti.sh`; Tigress kurulu değilse betik temiz bir türevle aynı akışı gösterir

Bu haftanın çalışan demosu

`code/week-14/01-kaynaktan-kaynaga` — Tigress donusum hatti (kuruluysa) ya da yerel ornek + indirme yonergesi; fonksiyon maliyetini olcer.

Çalıştırma: `code/week-14/01-kaynaktan-kaynaga` klasöründe `sh tigress-hatti.sh` (Linux/WSL; Windows'ta WSL). Tigress kurulu değilse betik temiz bir türevle aynı akışı gösterir. Adım adım aşağıdaki kutuda. Tümüyle sentetik ve güvenlidir; öğrenci bilgisayarına zarar vermez.

Demoyu kendiniz çalıştırın — adım adım (kopyala-yapıştır)

Bu demo **Tigress** ister (kaynaktan kaynağa C gizleyici). `code` klasöründen:

```
# WSL / Linux (Windows'ta WSL kullanın)
cd week-14/01-kaynaktan-kaynaga
sh tigress-hatti.sh
```

Beklenen çıktı: Tigress kuruluysa `erisim_ver` fonksiyonu kaynaktan kaynağa gizlenir: **dönüşüm hattı** → **davranış doğrulama** (çıktı değişmedi) → `objdump` ile **maliyet ölçümü** → **iki tohumla** farklı ikili (çeşitlendirme). Tigress yoksa betik **temiz bir türevle** aynı akışı gösterir. Kurulum: https://tigress.wtf (akademik kullanım ücretsiz).

Bu haftanın sonunda şunları yapabileceksiniz

1. **Kaynaktan kaynağa** (source-to-source) gizlemenin ne olduğunu ve 9. haftadaki el ile kuralların **otomatik** karşılığını (Tigress) açıklamak.
2. Tigress'in dönüşüm ailelerini (kontrol, veri, fonksiyon, bütünlük, analiz-önleme) 9. haftanın kurallarıyla eşlemek ve bir **dönüşüm hattı** (birden çok dönüşümün sırayla uygulanması) kurmayı tarif etmek.
3. **Çeşitlendirmeyi** araçla üretmek: tohum (`--Seed`) ve rastgeleleştirilmiş dönüşümlerle aynı kaynaktan farklı ama eş-davranışlı ikili dosyalar üretmenin mantığını anlatmak.
4. Gizlemenin ve çeşitlendirmenin **etkinliğini ölçmek**: maliyet (boyut, hız) ve dayanıklılık (sembolik yürütmeye direnç); 9. haftadaki güç/dayanıklılık/maliyet çerçevesini araç üstünde uygulamak.
5. Gizlemeyi bir **derleme ve dağıtım hattına** (S15) yerleştirmek: hangi fonksiyonlara, neden, hangi maliyetle; imzalama ve sürüm kimliğiyle birlikte.

Ders akışı (öğretim üyesi için zaman planı)

Zaman	Bölüm	Ne yapılıyor
0:00–0:20	1	Kaynaktan kaynağa gizleme; Tigress nedir, nereye oturur; kurulum ve lisans
0:20–0:50	2	Dönüşüm aileleri ve 9. hafta kurallarıyla eşleme; dönüşüm hattı kurma
0:50–0:55	Ara	
0:55–1:35	2	Örnek hat: bir denetimi düzleştir + veri kodla + fonksiyonları karıştır (kavram, sentetik)
1:35–1:40	Ara	
1:40–2:10	3	Çeşitlendirme: tohum ve rastgele dönüşümler; uzayda/zamanda çeşitlendirme
2:10–2:40	3–4	Ölçme: boyut/hız maliyeti; sembolik yürütmeye dayanıklılık (Tigress + KLEE çalışması)
2:40–3:00	4+	Derleme hattına yerleştirme (S15); sınırlar ve etik; proje; kendini sınama

Bu haftanın kaynakları ve Tigress lisansı hakkında

Bu hafta 9. haftadaki kuralları **araçla** uygular. Ana kaynak Tigress'in kendi çalışma sayfaları/derslerdir. Tigress bir **kaynaktan kaynağa C gizleyici/çeşitlendiricidir**; güncel sürümü **v4**'tür (Linux, macOS, Windows, Android; Intel/ARM/WebAssembly). Lisans: **kâr amacı gütmeyen** kullanım için ücretsizdir; ticari kullanım için Arizona Üniversitesi'nden lisans gerekir. **Bu yüzden derste bir Tigress ikilisi ya da eski bir sürüm dağıtılmaz;** öğrenciler güncel sürümü resmî siteden (`tigress.wtf`) kendileri indirir ve lisans koşullarını orada doğrular. Bütün örnekler **sentetiktir** ve yalnız kendi kodunuza uygulanır.

0. Temel kavramlar (sıfırdan)

Bu bölüm **hiçbir ön bilgi varsaymaz**. Haftanın geri kalanında kullanacağımız terimleri sıfırdan tanımlıyoruz. Bir terimi bilmiyorsanız önce burayı okuyun; sonraki bölümler bunların üzerine kurulur.

Neden bu bölüm?

Bugün "kaynaktan kaynağa", "dönüşüm", "tohum", "derleme hattı" gibi terimler geçecek.

Önce hepsini **tek tek** tanımlayalım.

Hatırlatma · kaynak, derleyici, ikili

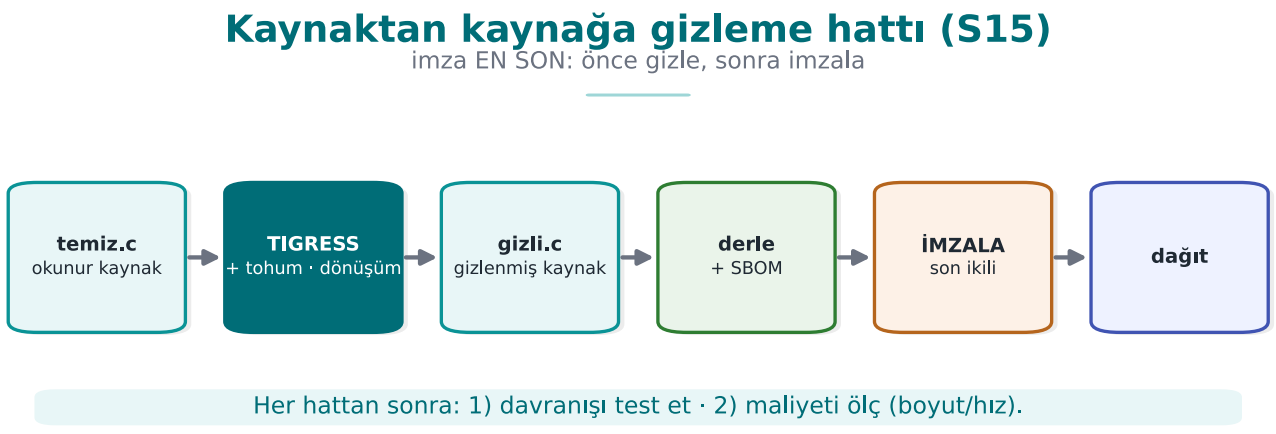
- **Kaynak kod:** insanın yazdığı program metni (C dosyası).
- **Derleyici:** kaynağı makine koduna çeviren program (gcc/clang).
- **İkili dosya:** çalıştırılan sonuç.

Hatırlatma · gizleme

- **Kod gizleme (obfuscation):** davranışı değiştirmeden kodu **anlaşılması zor** hale getirmek.
- Amaç: saldırgan için maliyeti artırmak (9. hafta).

Kaynaktan kaynağa (source-to-source)

- Girdi: C kaynağı. Çıktı: **yine C kaynağı** — ama gizlenmiş.
- Sonra normal derleyicinizle derlenir.



Dönüşüm (transform) nedir?

- **Dönüşüm:** kaynağa uygulanan tek bir gizleme işlemi (ör. düzleştirme).
- Araç birçok dönüşüm sunar; siz hangisini, nereye seçersiniz.

Dönüşüm hattı (pipeline)

- **Hat:** birden çok dönüşümün **sırayla** uygulanması.
- Her dönüşüm bir öncekinin çıktısına uygulanır.
- Sıra önemlidir.

Tohum (seed) nedir?

- **Tohum:** rastgeleliği yöneten bir başlangıç sayısı.
- Aynı dönüşüm + farklı tohum = **farklı** gizlenmiş çıktı.
- Çeşitlendirmenin anahtarı budur.

Çeşitlendirme (diversification)

- **Çeşitlendirme:** aynı kaynaktan **davranışça eş, yapıca farklı** ikili dosyalar üretmek.
- Bir kopyaya yazılan saldırı diğerinde çalışmaz (9. hafta Kural 2).

CLI (komut satırı) nedir?

- **CLI (Command-Line Interface):** komutları yazarak çalıştırdığınız arayüz.
- Tigress bir CLI aracıdır: `tigress --Transform=... dosya.c`.

Birim testi (unit test)

- **Birim testi:** bir fonksiyonun doğru çalıştığını otomatik denetleyen küçük test.
- Gizlemeden **sonra** aynı testler geçmeli (davranış korunmalı).

CFG hatırlatma

- **CFG (Control Flow Graph):** temel blokları düğüm, geçişleri kenar yapan şema.
- Gizlemenin **gücünü** düğüm/kenar sayısıyla ölçeriz (9. hafta).

Sembolik yürütme (kısaca)

- **Sembolik yürütme:** program yollarını matematiksel kısıt olarak çözen otomatik analiz (ör. KLEE).
- Gizlemeye karşı bir **deobfuscation** yöntemidir; dayanıklılığı bununla sınarız.

Şimdi hazırız

Terimler:

kaynaktan kaynağa · dönüşüm · hat · tohum · çeşitlendirme · CLI · birim testi · CFG · sembolik yürütme

Şimdi: Tigress nedir ve nasıl çalışır?

`objdump` ve komut/dal sayımı nedir?

Bu haftanın demo betiği (`tigress-hatti.sh`) bir ikili dosyadaki **tek bir fonksiyonun** ne kadar büyüdüğünü ölçmek için `objdump` adlı bir araç kullanır. Sıfırdan tanımlayalım:

- **`objdump`:** bir ikili dosyayı (`.exe` , ELF, ...) açıp içindeki **assembly komutlarını** insan tarafından okunabilir biçimde döken bir komut satırı aracı (GNU Binutils paketinin parçası; Linux/WSL'de hazır gelir).
- **`objdump -d dosya`:** "disassemble" (tersine çöz) demektir; ikili dosyanın **kod bölümünü** makine kodundan assembly'ye çevirip yazdırır.
- **Komut (instruction) sayısı:** bir fonksiyonun assembly çıktısında kaç **satır** komut olduğu — her satır `mov` , `cmp` , `add` gibi tek bir işlemci komutudur.
- **Dal/çağrı (branch/call) sayısı:** bu komutlardan kaçının **atlama** (`j` ile başlayan `jmp` , `je` , `jne` , ...) ya da **çağrı** (`call`) olduğu — yani programın "başka bir yere gitme" kararı aldığı kaç nokta olduğu.

Demo betiğindeki ölçüm fonksiyonu tam olarak bunu yapar (`tigress-hatti.sh` , `olc()` fonksiyonu):

Ölçüm mantığı (tigress-hatti.sh içinden, kısaltılmış)

```
objdump -d "$f" | awk '/<erisim_ver>:/{a=1;next} /^$/{a=0} a{n++; if($0 ~ /\t(j|call)/)b++}
END{printf "%d komut, %d dal/cagri", n, b}'
```

Satır satır ne yapıyor: `objdump -d` bütün ikiliyi assembly'ye çevirir; `awk` betiği yalnız `<erisim_ver>:` etiketiyle başlayan bölümü (`a=1` bayrağı) okur, boş satıra (fonksiyonun sonu) gelince durur (`a=0`); okuduğu her satırı `n` sayacına ekler, satırda sekme sonrası `j...` ya da `call` görürse `b` sayacını da artırır. Sonuç: **o fonksiyona ait kaç komut, kaçının dal/çağrı olduğu.**

Bu sayı, 9. haftada elle ölçtüğümüz "temel blok" ve "CFG düğüm/kenar" sayımının **otomatik ve daha ince taneli** karşılığıdır: 9. hafta CFG'yi kaynak koddan elle çizip düğüm/kenar sayardı (bkz. 9. hafta bölüm 0, `denetim` fonksiyonu örneği: 3 düğüm, 2 kenar); burada aynı fikri **derlenmiş ikili üzerinde**, bir araçla otomatik sayıyoruz. İkisi de aynı soruyu sorar: *"bu fonksiyonu okumak/analiz etmek ne kadar iş?"*

> **Neden komut sayısı tek başına yetmez, dal sayısı da gerekir?**
>
> Bir fonksiyona yalnız **düz, dallanmayan** komutlar eklemek (ör. gereksiz birkaç `mov`) komut sayısını artırır ama analistin işini pek zorlaştırmaz — düz bir listeyi okumak kolaydır. Asıl zorluk **dal sayısının** artmasıdır: her yeni dal, analistin izlemesi gereken **yeni bir yol** demektir (bölüm 7'deki çevrimsel karmaşıklık `dal_sayısı + 1` formülünü hatırlayın). Bu yüzden ölçüm her zaman **ikisini birden** raporlar; yalnız komut sayısı vermek eksik bir ölçümdür.

CI (sürekli entegrasyon) ve derleme hattı nedir?

- **CI (Continuous Integration, sürekli entegrasyon):** her kod değişikliğinde **otomatik olarak** derleme, test ve (bu ders bağlamında) gizleme adımlarını çalıştıran bir sistem (ör. GitHub Actions, GitLab CI, Jenkins).
- **Derleme hattı (build pipeline):** kaynak koddan dağıtılabilir ürüne giden **sıralı adımlar** dizisi: derle → test et → (varsa) gizle → imzala → paketle.

Bu terim bölüm 9'da ("Gizlemeyi derleme ve dağıtım hattına yerleştirmek") tekrar karşımıza çıkacak: gizleme, elle çalıştırılan ayrı bir adım değil, **CI'nin bir parçası** olmalıdır — böylece her sürüm otomatik olarak gizlenir, test edilir ve ölçülür; hiçbiri unutulmaz.

Sürüm kimliği ve özet (hash) değeri (kısaca)

- **Sürüm kimliği (version identity):** bir yazılımın **tam olarak hangi hâlinin** değerlendirildiğini/dağıtıldığını gösteren etiket (ör. `v3.2.1`).
- **Özet (hash) değeri:** bir dosyanın içeriğinden hesaplanan, o içeriğe **özgü** sabit uzunlukta bir sayı (ör. SHA-256). İçerik bir bit bile değişse özet tamamen değişir.

Bu iki kavram 12. haftadaki **TOE (Target of Evaluation — değerlendirme hedefi)** kimliğinin yapı taşlarıdır: 12. hafta bir ürünün TOE kimliğini "sürüm + ikili dosya + kaynak kod + özet değeri" dörtlüsüyle tanımlar ve yalnız "sürüm 3.2.0" demenin **yetmediğini** vurgular — çünkü aynı sürüm numarasıyla, farklı bir derleyici bayrağıyla derlenmiş iki ikili dosya **farklı** davranabilir. Bu hafta öğreneceğimiz gizleme + çeşitlendirme, tam olarak bu uyarıyı **daha da güçlendirir**: aynı kaynaktan, aynı sürüm etiketiyle, farklı tohumlarla üretilmiş iki ikili dosya **kasıtlı olarak farklıdır** — bu yüzden bölüm 9'da her sürümün tohumu ve özet değeri **ayrı ayrı** kaydedilecek.

Tigress ortamı (`--Environment`) nedir?

- **`--Environment`:** Tigress'e hedef platformu söyleyen bayrak; işlemci mimarisi, işletim sistemi ve derleyici sürümünü belirtir (demo betiğinde: `--Environment=x86_64:Linux:Gcc:11` → 64-bit Intel/AMD mimarisi, Linux, GCC sürüm 11).

- Neden gerekir? Bazı dönüşümler (özellikle sanallaştırma ve düşük seviyeli opak yüklem türleri) **derleyiciye özgü** ayrıntılar bilmek zorundadır; yanlış ortam bilgisiyle üretilen kaynak, hedef makinede **derlenemeyebilir** ya da yanlış davranabilir.

Dönüşüm bağımlılığı nedir (kısaca)?

- **Dönüşüm bağımlılığı:** bazı dönüşümlerin çalışabilmesi için **önce başka bir dönüşümün** çalışmış olması gerekmesi (ör. `AddOpaque`'in `InitOpaque`'e ihtiyaç duyması — bölüm 3'te ayrıntısıyla göreceğiz).
- Bu, bölüm 4'teki "sıra önemlidir" kuralının yalnız *sonucu iyileştirme* değil, bazen *çalışmanın önkoşulu* olduğu anlamına gelir.

Hızlı sözlük: bu haftanın yeni İngilizce terimleri

Dokuzuncu haftadaki sözlüğe (Obfuscation, Reverse engineering, Decompiler, ...) bu hafta şunlar eklenir; Tigress'in resmî belgelerinde ve komut satırında bu İngilizce adlarla karşılaşacaksınız:

Türkçe	İngilizce
Kaynaktan kaynağa (gizleme)	Source-to-source (obfuscation)
Dönüşüm	Transform
Dönüşüm hattı	Transform pipeline
Tohum	Seed
Ortam (hedef platform)	Environment
Opak yüklem (hazırlık / ekleme / güncelleme)	(Init / Add / Update) Opaque
Sanallaştırma tabanlı gizleme	Virtualization-based obfuscation
Sürekli entegrasyon	Continuous Integration (CI)
Değerlendirme hedefi	Target of Evaluation (TOE)
Fonksiyon bölme / birleştirme	Split / Merge
Çalışma anında kod üretimi	Just-In-Time (Jit)
Rastgele fonksiyon/parametre	Random functions / arguments

`diff` ve `cmp` arasındaki fark nedir?

Bölüm 6'da her ikisini de kullanacağız; farkı burada netleştirelim:

- **`cmp`:** iki dosyayı **bayt bayt** karşılaştırır, yalnız "aynı mı, farklı mı" (ve isteğe bağlı olarak ilk farkın konumunu) söyler. Bütün dosya için **tek** bir sonuç verir.

- `diff`: iki metin/kaynak dosyasını **satır satır** karşılaştırır, hangi satırların eklendiğini/çıkarıldığını/ değiştiğini ayrıntılı listeler. Assembly çıktısı gibi satırlara ayrılmış metinlerde `cmp`'den daha **bilgilendiricidir**.

Bölüm 6'daki "önce `cmp` ile kaba kontrol, sonra `objdump` + `diff` ile hedefe odaklı doğrulama" akışı, bu iki aracın **birbirini tamamladığı** fikrine dayanır: `cmp` hızlı ama kaba, `diff` yavaş ama ayrıntılıdır.

Ek terimler: pekiştirme

Yukarıdaki tanımlarla birlikte, bu bölümde toplam terim listemiz şöyle genişledi:

`objdump` · komut sayısı · dal/çağrı sayısı · CI (sürekli entegrasyon) · derleme hattı · sürüm kimliği · özet (hash) değeri · `--Environment` · dönüşüm bağımlılığı

Bu terimleri bölüm 3, 7 (ölçüm) ve 9'da (S15 hattı) tekrar kullanacağız; takıldığınızda buraya dönün.

1. Kaynaktan kaynağa gizleme nedir?

Dokuzuncu haftada gizleme kurallarını **el ile** uyguladık: opak yüklem, düzleştirme, dize kodlama, sahte dal. El ile gizleme öğreticidir ama üç sorunu vardır: (1) **hataya açıktır** — el ile yazılan gizleme kodu davranışı bozabilir; (2) **bakımı zordur** — kaynak okunamaz hale gelir; (3) **çeşitlendirilemez** — her kopyayı el ile farklılaştıramazsınız. Çözüm, gizlemeyi bir **araca** yaptırmaktır.

Kısa tarihçe: kaynaktan kaynağa gizleme ve çeşitlendirme

- **1993** — Cohen, "program evolution" ile **çeşitlendirme** fikrini ortaya atar: aynı işlev, farklı ikili.
- **1997** — Collberg vd. gizleme taksonomisi (9. hafta'nın temeli).
- **2013** — **Obfuscator-LLVM (O-LLVM)**: derleyici tabanlı gizleme.
- **2010'lar** — **Tigress** (Christian Collberg): C için **kaynaktan kaynağa** gizleyici + sanallaştırma + çeşitlendirme; araştırmada dayanıklılık ölçümünün fiili standardı.
- **2016–2017** — Banescu vd. Tigress + KLEE ile dönüşüm **dayanıklılığını ölçer** → "**dayanıklılık ↔ maliyet**" kuralı.

Tarihçeyi biraz daha açalım

- **1993 — Cohen, "program evolution":** Fred Cohen (bilgisayar virüsü kavramını da akademik olarak tanımlayan araştırmacı), bir programın **işlevini değiştirmeden yapısını otomatik olarak değiştirebileceğini** gösterdi; bu, "her kopya farklı görünsün" fikrinin (bu haftaki çeşitlendirmenin) akademik kökenidir.
- **1997 — Collberg, Thomborson, Low:** Bugün 9. haftada kullandığımız beş ailelik taksonomiyi (düzen, veri, kontrol akışı, önleyici, sanallaştırma) ve güç/dayanıklılık/gizlilik/maliyet çerçevesini yayımladılar; bu makale hâlâ alanın referans noktasıdır ve bu haftaki eşleme tablosunun (bölüm 3) temelidir.
- **2013 — Obfuscator-LLVM (O-LLVM):** LLVM derleyici altyapısına eklenen açık kaynaklı bir gizleme geçidi; düzleştirme ve opak yüklem gibi teknikleri **derleme sırasında**, ayrı bir kaynaktan-kaynağa adımı olmadan uygular (9. hafta K-11). Tigress'ten farkı, kaynak kod yerine derleyicinin ara temsiline (IR) müdahale etmesidir — bu yüzden O-LLVM'in çıktısını "yine C kaynağı" olarak göremezsiniz, doğrudan ikili üretir.
- **2010'lar — Tigress:** Christian Collberg ve ekibi, 1997'deki taksonomiyi **çalışan bir kaynaktan-kaynağa araca** dönüştürdü; akademik dayanıklılık araştırmalarının (aşağıdaki Banescu çalışması dahil) fiili test platformu hâline geldi — bu yüzden bu hafta Tigress'i öğrenmek, yalnız bir aracı değil, alanın **ortak ölçüm dilini** öğrenmek demektir.

- **2016–2017 — Banescu vd.:** Tigress'in ürettiği dönüşümleri KLEE ile sistematik olarak **kırıp**, hangi dönüşüm kombinasyonunun ne kadar sürede çözüldüğünü ölçtüler; bu hafta bölüm 7'de tekrar değineceğimiz "dayanıklılık ↔ maliyet" ilişkisinin **deneysel kanıtı** budur.

Bu beş nokta, aslında tek bir çizginin parçalarıdır: **fikir (1993) → sınıflandırma (1997) → iki farklı uygulama yolu: derleyici tabanlı (2013) ve kaynaktan-kaynağa (2010'lar) → o uygulamanın bilimsel olarak sınanması (2016–2017).** Bu hafta bu çizginin **son iki adımını** (Tigress'i kullanmak ve onu Banescu'nun yöntemiyle değerlendirmeyi anlamak — bölüm 7) uyguluyoruz.

Kaynaktan kaynağa (source-to-source) gizleme, bir aracın **C kaynağını girdi alıp yine C kaynağı** üretmesidir; üretilen kaynak, davranışça özdeş ama okunması çok daha zordur ve normal derleyicinizle derlenir. Bu yaklaşımın en bilinen aracı **Tigress**'tir.

Önemli fark: siz **okunur** kaynağı korur ve bakımını yaparsınız; gizleme, **derleme hattında** otomatik bir adım olarak çalışır. Bu, 9. haftadaki "her gizleme bakım maliyeti getirir" sorununu büyük ölçüde çözer: bakım maliyeti okunur kaynakta kalır, dağıtılan kaynak gizlidir.

Tigress nedir, nereye oturur?

Tigress, Arizona Üniversitesi'nden Christian Collberg ve ekibinin geliştirdiği bir **kaynaktan kaynağa C gizleyici ve çeşitlendiricidir**. Bu dersteki yeri:

- a. haftadaki **el ile kuralların otomatik karşılığıdır**: düzleştirme, opak yüklem, veri kodlama, sanallaştırma — hepsi birer dönüşüm (transform) olarak.
- Çok platformludur (Linux, macOS, Windows, Android; Intel/ARM/WebAssembly) ve GCC/Clang/MSVC ile çalışır.
- Güncel sürümü **v4**'tür.

Lisans ve derste kullanım

Tigress, **kâr amacı gütmeyen** (akademik/araştırma) kullanım için ücretsizdir; **ticari** kullanım için Arizona Üniversitesi'nden lisans gerekir. Kaynak kodu açık değildir; araştırmacılar şifreli kaynağa erişim isteyebilir. Bu yüzden **derste bir Tigress ikilisi ya da eski bir sürüm dağıtılmaz.** Öğrenciler güncel sürümü resmî siteden (`tigress.wtf`) kendileri indirir ve **lisans koşullarını orada doğrular.** Ödevlerde Tigress yalnız öğrencinin **kendi** koduna, kendi makinesinde uygulanır.

Neden bir araç, el ile değil?

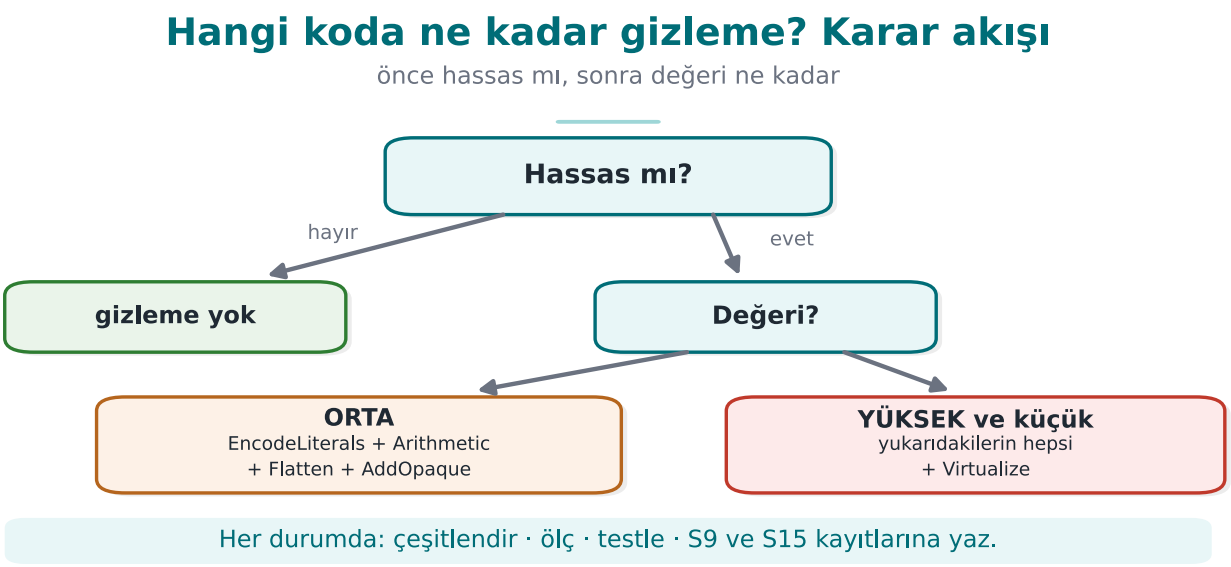
El ile gizleme (9. hafta)	Araçla gizleme (Tigress)
Öğreticidir, kavramı gösterir	Ölçeklenir, üretimde kullanılır
Hataya açık (davranışı bozabilir)	Dönüşümler davranışı korur (test edilmiş)
Kaynak okunamaz hale gelir	Okunur kaynak sizde kalır
Çeşitlendirme el ile olmaz	Tohumla otomatik çeşitlendirme
Tek tip, tanınabilir	Dönüşüm ve tohum kombinasyonuyla değişken

Ama araç sihir değildir

1. haftanın ana kuralı burada da geçerlidir: Tigress de kırılamazlık vermez, **maliyet** yükseltir. Üstelik iyi bilinen bir aracın ürettiği kalıplar zamanla tanınabilir; bu yüzden **çeşitlendirme** (bölüm 6) ve **katmanlı savunma** (RASP, sunucu denetimi) yine şarttır. Tigress'in kendisi de dönüşümlerin sembolik yürütmeye ne kadar dayandığını ölçen araştırmaların (bölüm 7) baş öznesidir.

Hangi koda ne kadar gizleme? (karar akışı)

Her fonksiyonu gizlemek hem gereksiz hem pahalıdır: boyut ve çalışma süresi artar, hata ayıklama zorlaşır. Karar iki soruyla verilir. Önce **hassas mı** — lisans denetimi, anahtar işleme, ödeme mantığı gibi saldırganın ilgilendiği bir kod mu? Değilse gizlemeye gerek yoktur. Hassassa **değeri ne kadar** — orta değerli kod için sabit gizleme, aritmetik dönüşüm, düzleştirme ve opak yüklem yeterlidir; yüksek değerli ve **küçük** bir çekirdek için bunlara sanallaştırma da eklenir (sanallaştırma yavaştır, bu yüzden yalnız küçük bölümlere uygulanır). Her durumda sonuç **çeşitlendirilir**, **ölçülür**, **testten geçirilir** ve kararın gerekçesi S9/S15 kayıtlarına yazılır.



İşlenmiş örnek: aynı fonksiyonu el ile ve araçla karşılaştıralım

Dokuzuncu haftada K-07 kuralını (dize kodlama) **el ile** uyguladığımızda, kaynak kodun kendisi değişiyordu: sabit dizge yerine XOR'lu baytlar ve çözme döngüsü yazıyorduk. Aynı fikri şimdi **kaynaktan kaynağa** bir araçla yapalım — girdi ve çıktı hâlâ ikisi de C'dir, ama çıktıyı **siz yazmazsınız**, araç üretir.

ÖNCE — temiz.c (K-07 uygulanmadan, elle yazılmış hâliyle aynı başlangıç noktası)

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

SONRA — kavramsal gösterim: EncodeLiterals uygulanmış gibi (gerçek çıktı sürüme göre değişir)

```c
int erisim_ver(const char *jeton) {
    /* IZIN ve RED artık düz sabit değil; çalışma anında bir kod-çözme adımından geçer.
       Fikir 9. haftadaki K-07 ile birebir aynıdır (bkz. 0x41 XOR 0x5A örneği); farkı
       çözme kodunu SİZİN değil, ARACIN üretmiş olmasıdır. */
    if (jeton_gecerli(jeton)) return _cozul_sabit_0x9F2A();  /* == IZIN, ama kaynakta IZIN yazmıyor
*/
    return _cozul_sabit_0x117C();                            /* == RED */
}
```

İkinci blok **gerçek bir Tigress çıktısı değildir** — sürüme, derleyiciye ve `--Seed` değerine göre üretilen kod değişir; bu yalnız EncodeLiterals'in **fikrini** göstermek içindir (tıpkı bu dosyadaki diğer "kavramsal akış" bloklarının komut sözdizimini gösterip gerçek çıktıyı iddia etmemesi gibi). Önemli olan şu **fark**: 9. haftada `_cozul_sabit_0x9F2A()` gibi bir fonksiyonu siz elle yazıp, elle test eder, bir sonraki fonksiyona geçtiğinizde **yeniden** yazardınız. Burada bu adımı `--Functions=erisim_ver` ile işaretlediğiniz her fonksiyona araç otomatik uygular.

Sık yapılan hata: Tigress'in ürettiği `gizli.c` dosyasını elle düzenlemek

Bir gizlenmiş kaynak dosyasını (`gizli.c`) açıp "bir satırını düzeltmek" ya da "biraz daha okunur hâle getirmek" isteyen bir geliştirici, iki şeyi bozar: (1) elle yaptığı değişiklik bir sonraki gizleme çalıştırmasında **kaybolur** (araç `temiz.c` 'den yeniden üretir), (2) dönüşümün varsaydığı iç yapıyı (ör. düzleştirilmiş `switch` durum değerlerinin birbirine bağımlılığı) bozup **davranışı kırabilir**. **Kural:** yalnız `temiz.c` 'yi (orijinal, okunur kaynağı) düzenleyin; `gizli.c` her zaman **yeniden üretilen, elle dokunulmayan** bir çıktıdır — tıpkı bir derleyicinin ürettiği `.o` dosyasına elle dokunmadığınız gibi.

Ne kadar zaman kazandırır? Kaba bir karşılaştırma (varsayımsal)

Neden-bir-araç sorusuna bir de zaman açısından bakalım. Aşağıdaki sayılar **varsayımsaldır**, yalnızca büyüklük mertebesini göstermek içindir:

	El ile (9. hafta yöntemiyle)	Araçla (Tigress)
1 fonksiyona K-07 uygulamak	~15 dakika (kodlama + çözme fonksiyonu yaz, test et)	~1 dakika (`--Transform=EncodeLiterals --Functions=f` ekle)
10 fonksiyona aynı kuralı uygulamak	~150 dakika (her biri ayrı, hataya açık)	~1 dakika (aynı bayrak, `--Functions` listesine 10 ad)
Yeni bir tohumla ikinci bir kopya üretmek	Mümkün değil (elle her seferinde farklı yazmak gerekir)	~saniyeler (`--Seed` değerini değiştirip yeniden çalıştırmak)

Buradaki asıl kazanç **doğrusal değil, katlanarak** büyüyen bir farktır: fonksiyon sayısı arttıkça el ile yöntemin maliyeti orantılı artar, araçla yöntemin maliyeti hemen hemen **sabit** kalır. Bu, bölüm 9'daki "S15'te CI'ye gizleme adımı ekleyin" tavsiyesinin de altında yatan pratik nedendir.

İşlenmiş örnek: bir yıllık bakım senaryosu (varsayımsal)

"Ne kadar zaman kazandırır?" tablosunu bir yıla yayalım. Diyelim bir ürünün **12 ayda 12 sürümü** çıkıyor ve her sürümde ortalama **3 fonksiyon** yeni hassas hâle geliyor (ör. yeni bir ödeme akışı, yeni bir lisans kontrolü):

	El ile (9. hafta yöntemiyle)	Araçla (Tigress, CI içinde)
Aylık ek fonksiyon başına iş	~15 dakika/fonksiyon × 3 = 45 dakika	`--Functions` listesine 3 ad eklemek, ~2 dakika
Yıllık toplam (12 ay)	45 × 12 = **540 dakika (9 saat)**	2 × 12 = **24 dakika**
Çeşitlendirme (her sürüm farklı tohum)	Pratikte yapılmaz (bölüm 6'daki "sık yapılan hata")	CI'nin bir parçası, **ek emek gerektirmez** (bölüm 9)
Unutma riski (bir fonksiyonun gizlenmeyi atlaması)	Yüksek (elle takip)	Düşük (`--Functions` listesi kod incelemesinde görülür)

Bu tablo **varsayımsaldır**, ama iki gerçek eğilimi doğru yansıtır: (1) el ile yöntemin maliyeti fonksiyon ve sürüm sayısıyla **doğrusal** büyür, araçla yöntemin maliyeti **hemen hemen sabit** kalır (yukarıdaki ilk tablonun büyütülmüş hâli); (2) el ile yöntemde "unutma riski" gerçek bir güvenlik açığıdır — yeni eklenen hassas bir fonksiyonun gizlenmeyi atlaması, kod incelemesinde fark edilmezse **hiç korunmadan** yayınlanabilir; araçla yöntemde bu risk `--Functions` listesinin kendisi bir **denetlenebilir belge** olduğu için azalır.

> ✓ **Kural: `--Functions` listesini kod incelemesinin bir parçası yapın**
>
> Yeni bir hassas fonksiyon eklendiğinde, kod incelemesi (code review) sürecinizde "bu fonksiyon `--Functions` listesine eklendi mi?" sorusu **açıkça sorulmalıdır** — tıpkı "bu fonksiyon için birim testi eklendi mi?" sorusu gibi. Bu, 12. haftadaki "her değişiklik bir etki analizi gerektirir" kuralının gizleme özelindeki karşılığıdır.

2. Temel akış: bir programı adım adım gizlemek

Tigress komut satırından çalışır: hangi dönüşümlerin, hangi fonksiyonlara, hangi sırayla uygulanacağını söylersiniz; çıktı gizlenmiş bir C dosyasıdır. Kavramsal akış (sözdizimi sürümle değişebilir, resmî belgeden doğrulayın):

Kavramsal akış (tek dönüşüm)

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

Buradaki üç fikir, bütün Tigress kullanımının temelidir:

1. `--Transform=...` hangi dönüşümün uygulanacağını söyler (Flatten = kontrol akışı düzleştirme, 9. hafta K-04).
2. `--Functions=...` dönüşümün **hangi fonksiyonlara** uygulanacağını söyler — çünkü gizlemeyi yalnız hassas fonksiyonlara uygularız (maliyet kuralı).
3. Çıktı yine C'dir; **kendi derleyicinizle** derlersiniz. Gizleme, derleme hattına eklenen bir adımdır.

Neden `--Functions` bu kadar önemli? Seçici ile toptan gizlemeyi karşılaştıralım

`--Functions=erisim_ver` gibi bir bayrak, "yalnız bu fonksiyona uygula" demektir. Bunun tersini — bir dosyadaki **bütün** fonksiyonları gizlemeyi — düşünelim ve iki yaklaşımı karşılaştıralım:

	Seçici (`--Functions=erisim_ver`)	**Toptan (bütün dosyaya)**
Maliyet	Yalnız hassas fonksiyona	Her fonksiyona (çoğu hassas değil)
Hata ayıklama	Kolay: gizlenmemiş kod okunabilir kalır	Zor: bütün program gizlenmiş, günlükler bile anlaşılmaz
Değer/maliyet oranı	Yüksek (bölüm 0'daki karar akışının hedefi)	Düşük — gereksiz yere büyük bir yüzeyi korursunuz
S9/S15'te gerekçe yazmak	Kolay: "şu fonksiyon şu nedenle"	Zor: "hepsini gizledim" bir gerekçe değildir

Sık yapılan hata: '`--Functions` belirtmezsem daha güvenli olur' sanmak

`--Functions` bayrağı **atlanırsa** (ya da bilinçsizce dosyadaki her fonksiyonun adı listeye eklenirse), dönüşüm `main` gibi hassas olmayan yardımcı fonksiyonlara da uygulanır. Bu, bölüm 0'daki karar akışının (hassas mı, değeri ne kadar) atlanması demektir; sonuç **daha güvenli değil, yalnızca daha büyük ve daha yavaş** bir programdır. **Kural:** `--Functions` listesini her zaman **bilinçli ve gerekçeli** doldurun; "olur da bir işe yarar" mantığıyla genişletmeyin.

Uçtan uca: `tigress-hatti.sh` betiğini satır satır okuyalım

Kavramsal akışı somutlaştırmak için, bu haftanın gerçek demo betiğini (`code/week-14/01-kaynaktan-kaynaga/tigress-hatti.sh`) baştan sona, adım adım okuyalım. Betik dört adımdan oluşur; hiçbirini atlamadan geçelim.

Adım 0 — Hazırlık. Betik önce çalıştığı klasöre geçer (`cd "$(dirname "$0")"`) ve bir derleyici seçer (`CC=cc` , yoksa `gcc` 'ye düşer). Bu, hangi makinede çalıştırılırsa çalıştırılsın betiğin **kendi kendine yeteceğini** garanti eder — 1. haftadan beri gördüğümüz "ortam farkını betikte çöz" alışkanlığının bir örneğidir.

ADIM 1 — Temiz sürümü derle ve çalıştır.

```
"$CC" -O2 -o ornek_temiz ornek.c && ./ornek_temiz "CEN429-OK" && ./ornek_temiz "yanlis"
```

Bu satır `ornek.c` 'yi (bölüm 0'da tanımladığımız `erisim_ver` fonksiyonunun bulunduğu gerçek dosya) **hiçbir gizleme uygulanmadan** derler ve iki kez çalıştırır: bir kez doğru jetonla (`CEN429-OK`), bir kez yanlış jetonla (`yanlis`). Beklenen çıktı:

Beklenen çıktı — ADIM 1

```
erisim_ver("CEN429-OK") = IZIN
erisim_ver("yanlis") = RED
```

Bu, gizlemeden **önceki referans davranıştır** — bölüm 3 ve sonrasında yapılacak her gizlemenin doğruluğu bu iki satırla karşılaştırılarak sınanır (davranış korundu mu?).

ADIM 2 — Tigress kontrolü ve dönüşüm hattı. Betik önce `command -v tigress` ile Tigress'in kurulu olup olmadığına bakar. **İki yol vardır:**

Betiğin çalıştırdığı gerçek komut (tigress-hatti.sh içinden)

```
tigress --Environment=x86_64:Linux:Gcc:11 \
        --Transform=EncodeLiterals --Functions=erisim_ver \
        --Transform=EncodeArithmetic --Functions=erisim_ver \
        --Transform=Flatten --Functions=erisim_ver \
        --Transform=InitOpaque --Functions=main \
        --Transform=AddOpaque --Functions=erisim_ver --AddOpaqueKinds=call \
        --out=gizli.c ornek.c
```

Bu tek komutta **beş** dönüşüm sırayla uygulanır (bölüm 4'ün konusu): önce dizeler/sabitler kodlanır (`EncodeLiterals`), sonra aritmetik kodlanır (`EncodeArithmetic`), sonra kontrol akışı düzleştirilir (`Flatten`), sonra `main` fonksiyonuna opak yüklemler için gerekli altyapı hazırlanır (`InitOpaque` — Tigress'te opak yüklem eklemeden önce bir kez çağrılması gereken hazırlık dönüşümü), son olarak `erisim_ver` 'e çağrı tabanlı opak yüklemler eklenir (`AddOpaque --AddOpaqueKinds=call`). `--Environment` bayrağı hedef platformu (burada 64-bit Linux, GCC 11) belirtir; Tigress bazı dönüşümler için derleyiciye özgü ayrıntılar bilmek zorundadır (bölüm 0'daki "Tigress ortamı" tanımı).

Ardından gizlenmiş kaynak normal derlenir (`"$CC" -O2 -o ornek_gizli gizli.c`) ve **ADIM 3**'te davranışı karşılaştırılır.

Betik gerçek bir hata vermez; bunun yerine kurulum/lisans yönergesini basar ve öğrenciyi 9. haftanın **el ile** çalışan demosuna yönlendirir:

Beklenen çıktı — Tigress yoksa (tigress-hatti.sh'nin gerçek metni)

```
ADIM 2 - Tigress KURULU DEGIL.
  Tigress'i resmi siteden indirin: https://tigress.wtf
  Lisans: kar amaci gutmeyen (akademik) kullanim UCRETSIZ; ticari icin Arizona Univ. lisansi.
  Kurulumdan sonra 'tigress' PATH'te olunca bu betik hatti otomatik uygular ve olcer.

  Tigress olmadan bile: ayni kurallarin EL ILE surumu ve olcumu week-09'da CALISIR:
    cd ../../week-09/01-manuel-gizleme && sh demo.sh
```

Bu, bu dersin genel ilkesinin somut uygulamasıdır: **araç eksikse ders durmaz**, aynı kavramın el ile çalışan hâline geçilir.

ADIM 3 — Davranışı doğrula. Tigress bulunduysa betik şunu yapar:

```
[ "$(./ornek_temiz CEN429-OK)" = "$(./ornek_gizli CEN429-OK)" ] && echo "  ✓ ayni cikti" || echo "
X FARK"
```

Temiz ve gizli sürümün **aynı girdiyle aynı çıktıyı** verip vermediğini karşılaştırır. `✓ ayni cikti` çıkmazsa, gizleme hattı davranışı bozmuş demektir — bölüm 4'teki "sıra ve test kuralı"nın betikteki karşılığı budur.

ADIM 4 — Çeşitlendirme. İki farklı tohumla iki ayrı ikili üretilir (`--Seed=1001` , `--Seed=2002`) ve `cmp -s A B` ile bayt bayt karşılaştırılır; betik ikisi farklıysa `✓ iki tohum farkli ikili uretti` yazar. Bu, bölüm 6'nın konusudur.

Kural: her adımı çalıştırdıktan sonra çıktıyı doğrulayın, sonucu görmeden ilerlemeyin

Betiğin dört adımı bilinçli olarak birbirine **bağımlıdır**: ADIM 2 çalışmazsa ADIM 3'ün karşılaştıracağı bir `ornek_gizli` yoktur; ADIM 3 "✗ FARK" verirse ADIM 4'ün çeşitlendirdiği ikililer zaten **hatalı** demektir. Aynı disiplin sizin kendi projenizde de geçerli: bir dönüşüm hattını çalıştırdıktan sonra **önce davranışı** doğrulayın, sonra maliyeti ölçün, sonra çeşitlendirin — sırayı atlamak hatayı geç fark etmenize yol açar.

`--out` ve birden çok fonksiyonu aynı anda gizlemek

`--out=gizli.c` bayrağı, üretilecek gizlenmiş kaynağın **hangi dosyaya** yazılacağını belirtir; belirtilmezse Tigress varsayılan bir ad kullanır (resmî belgeden doğrulanmalı). Bu hafta boyunca gördüğümüz örneklerin hepsi **tek** bir fonksiyona (`erisim_ver`) odaklandı; ama `--Functions` bir **liste** alabilir. Diyelim projenizde `erisim_ver`'in yanında bir de `anahtar_turet` fonksiyonu var ve ikisi de hassas:

Kavramsal: aynı dönüşümü iki fonksiyona birden uygulamak

```
tigress --Transform=Flatten --Functions=erisim_ver,anahtar_turet \
        --Transform=AddOpaque --Functions=erisim_ver,anahtar_turet \
        --out=gizli.c proje.c
```

`--Functions=erisim_ver,anahtar_turet` (virgülle ayrılmış liste), her iki fonksiyona da **aynı** dönüşümü uygular. Bu, bölüm 1'deki "10 fonksiyona aynı kuralı uygulamak" karşılaştırmasının somut söz dizimidir: el ile yöntemde her fonksiyon için ayrı bir kodlama yazmanız gerekirdi, burada tek bir bayrağa bir isim daha eklemeniz yeterlidir.

Farklı fonksiyonlara farklı dönüşümler gerekebilir

`erisim_ver` ve `anahtar_turet` aynı **değerde** olmayabilir — belki `anahtar_turet` daha kritik ve Virtualize de gerektiriyor, `erisim_ver` ise Flatten+AddOpaque ile yetiniyor (bölüm 1'deki karar akışı). Bu durumda tek bir komutta **hepsine aynısını** uygulamak yerine, **her fonksiyon için ayrı bir `--Transform` / `--Functions` çifti** (ya da ayrı bir Tigress çağrısı) kullanılır; "hepsine aynı hat" varsayımı bölüm 2'deki "`--Functions` bu kadar önemli" tartışmasının bir uzantısıdır — seçicilik yalnız *hangi* fonksiyonlar için değil, *ne kadar* gizleme için de geçerlidir.

---

3. Dönüşüm aileleri ve 9. hafta kurallarıyla eşleme

Tigress dönüşümleri, 9. haftada gördüğümüz gizleme ailelerine karşılık gelir. Aşağıdaki tablo, en çok kullanılan dönüşümleri ve desteki karşılıklarını verir (dönüşüm adları resmî belgeden doğrulanmalıdır; sürümle küçük farklar olabilir):

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.

Tigress dönüşümü	Ne yapar?	9. hafta karşılığı
Flatten	Kontrol akışını düzleştirir (switch dağıtıcı)	K-04 kontrol akışı düzleştirme
InitOpaque / AddOpaque / UpdateOpaque	Opak yüklemler ekler; sahte dallar	K-01 opak yüklem, K-03 ölü dal
EncodeArithmetic	Aritmetik işlemleri denk karmaşık ifadelerle değiştirir (MBA)	K-02 aritmetik kodlama
EncodeLiterals	Sabitleri ve dizgeleri kodlar	K-07 dize kodlama, K-08 sabit dönüşümü
EncodeData	Değişkenlerin gösterimini kodlar	K-09 değişken bölme/kodlama
Split / Merge	Fonksiyonları böler / birleştirir (yapıyı bulanıklaştırır)	K-06 çağrı/yapı gizleme
Virtualize	Fonksiyonu özel bir VM bayt koduna çevirir	K-10 sanallaştırma
Jit	Kodu çalışma anında üretir (dinamik)	K-12 dinamik (kavram)
AntiBranchAnalysis / AntiAliasAnalysis / AntiTaintAnalysis	Statik analiz tekniklerini zorlaştırır	Önleyici aile
RandomFuns / RndArgs	Rastgele fonksiyonlar / sahte parametreler ekler	Çeşitlendirme, K-06 sahte parametre

Aileler ve maliyet

Bu dönüşümleri 9. haftadaki maliyet sırasıyla düşünün: EncodeArithmetic/EncodeLiterals **ucuzdur**; Flatten ve opak yüklemler **orta**; Virtualize **pahalıdır** (onlarca kat yavaşlama). O yüzden Virtualize'ı yalnız en kritik, küçük fonksiyonlara uygularsınız — Tigress'te `--Functions` ile hedef seçmek tam da bunun içindir.

İşlenmiş örnek: K-01'i Tigress dönüşümüne eşleyelim, adım adım

Eşleme tablosunu soyut bırakmayalım; K-01'i (9. hafta, opak yüklem/döngü) uçtan uca Tigress'e bağlayalım.

1. **9. haftadaki tanım:** K-01, doğruluğu **sabit** ama saldırgana **belirsiz görünen** bir koşulla (opak yüklem) sahte bir dallanma eklemektir — örnek: `((x*(x+1)) & 1) == 0` ifadesi *her zaman* doğrudur (çünkü `x*(x+1)` ardışık iki tam sayının çarpımıdır, bunlardan biri her zaman çifttir), ama bunu görmek insan için anında değildir.
2. **Tigress'teki karşılığı iki adımdır, tek değil:** demo betiğinin gerçek komutunda görüldüğü gibi önce `--Transform=InitOpaque --Functions=main` çalışır, **sonra** `--Transform=AddOpaque --Functions=erisim_ver --AddOpaqueKinds=call`. `InitOpaque`, opak yüklemlerin dayanacağı **gizli durum değişkenlerini** hazırlar (bölüm 2'de gördüğümüz gibi genelde `main` içinde ilklendirilir); `AddOpaque` bu durumu kullanarak hedef fonksiyona (`erisim_ver`) gerçek opak koşulları ekler. `--AddOpaqueKinds=call` ise eklenecek opak yüklemin **türünü** seçer (burada "call" tabanlı bir çeşit).
3. **Sonuç:** K-01'in "sabit doğru ama belirsiz görünen koşul" tanımı, `AddOpaque`'in ürettiği koşullarla birebir örtüşür; `InitOpaque` ise K-01'in kendi başına yetmediği, bir **hazırlık** adımı gerektirdiği gerçeğini gösterir.

Sık yapılan hata: `InitOpaque` adımını atlayıp doğrudan `AddOpaque` çalıştırmak

Dönüşüm hattını kısaltmaya çalışan bir öğrenci, yalnız `--Transform=AddOpaque --Functions=erisim_ver` yazıp `InitOpaque` adımını atlayabilir. Bu, ya bir hata verir ya da opak yüklemlerin dayanacağı durum yapısı eksik kaldığı için **beklenmeyen** bir sonuç üretir. **Kural:** dönüşümler arasında **bağımlılık** olabilir (bu demoda `InitOpaque` → `AddOpaque`); resmî belgeyi okumadan bir dönüşümü "tek başına yeter" varsaymayın. Bu, bölüm 4'teki "sıra önemlidir" kuralının yalnız *hangi sırada* değil, *hangi dönüşümün hangi dönüşümü gerektirdiği* boyutuna da uzandığını gösterir (bölüm 0'daki "dönüşüm bağımlılığı" tanımı).

İşlenmiş örnek: K-07'yi `EncodeLiterals`'e eşleyelim

Aynı adımları K-07 (dize kodlama) için de tekrarlayalım, çünkü bu kural demo betiğinde de kullanılıyor:

1. **9. haftadaki tanım:** K-07, `"CEN429-OK"` gibi bir dizgeyi ikili dosyada **düz metin olarak bırakmamak**; XOR gibi tersine çevrilebilir bir kodlamayla saklamak ve yalnız kullanım anında çözmektir (bölüm 0'daki `0x41 ^ 0x5A` örneğini hatırlayın — bkz. 9. hafta bölüm 0).
2. **Tigress'teki karşılığı:** `--Transform=EncodeLiterals --Functions=erisim_ver`. Bu dönüşüm, hedef fonksiyondaki sabit dizgeleri ve sayısal sabitleri kodlayıp, çalışma anında çözen kod üretir — K-07'nin "elle yazılan" hâlinin otomatikleştirilmiş karşılığı (bölüm 2'deki "ÖNCE/SONRA" örneği).
3. **Doğrulama:** demo betiğindeki gibi bir `strings` denetimi (9. haftanın demosunda olduğu gibi) gizlenmiş ikili dosyada `CEN429-OK` metninin **artık görünmediğini** doğrular; bu, K-07'nin "başarı ölçütü"dür.

Bu iki örnek (K-01 → `InitOpaque` + `AddOpaque`, K-07 → `EncodeLiterals`) tablonun geri kalanını da nasıl okuyacağınızı gösterir: her satırda **önce 9. haftadaki tanımı hatırlayın, sonra hangi bayrağın/bayrakların bunu ürettiğini bulun, sonra bir doğrulama yolu belirleyin.**

İşlenmiş örnek: `Flatten` in K-04'ü nasıl uyguladığını kod üzerinde görelim

K-01 ve K-07'den sonra, tablodaki en güçlü kontrol akışı dönüşümünü — `Flatten`i — kod düzeyinde somutlaştıralım. Bölüm 2'deki `erisim_ver`'i temel alalım:

ÖNCE — erisim_ver (düzleştirilmemiş)

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

SONRA — kavramsal gösterim: Flatten uygulanmış gibi (gerçek çıktı sürüme/derleyiciye göre değişir)

```
int erisim_ver(const char *jeton) {
    int _durum = 7341;          /* dağınık, öngörülemez başlangıç durumu */
    int _sonuc;
    while (1) {
        switch (_durum) {
            case 7341:
                _durum = jeton_gecerli(jeton) ? 2098 : 5560;   /* karar burada gizlendi */
                break;
            case 2098:
                _sonuc = IZIN;
                _durum = 9999;
                break;
            case 5560:
                _sonuc = RED;
                _durum = 9999;
                break;
            case 9999:
                return _sonuc;
        }
    }
}
```

Bu ikinci blok da **gerçek bir Tigress çıktısı değildir**; yalnız `Flatten`'in **fikrini** göstermek içindir. Üç noktaya dikkat edin:

1. **Orijinal `if` / `return` yapısı kayboldu**; yerine tek bir `while(1)` döngüsü içinde bir `switch` **dağıtıcısı** geldi. Bir tersine derleyicide bu döngü, `if (jeton_gecerli(...))` kadar **anında okunmaz**.
2. **Durum sabitleri** (`7341`, `2098`, `5560`, `9999`) **ardışık değil, dağınıktır** — yukarıdaki "değerleri K-02 ile ya da rastgele, dağınık sabitlerle üretilmeli" notunun somut karşılığı budur; ardışık sabitler (`1, 2, 3, 4`) bir tersine derleyicide anında "bu bir düzleştirilmiş dağıtıcı" imzası olarak tanınır.
3. **Dal sayısı arttı:** orijinalde tek bir `if` (1 dal noktası) vardı; düzleştirilmiş hâlde `switch` içinde 4 `case` ve bunlar arasındaki geçişler var — bölüm 5 ve 7'deki "dal sayısı yalnız Flatten'de sıçrıyor" gözleminin kaynağı tam olarak budur.

Sık yapılan hata: düzleştirilmiş kodu görüp 'artık mantık tamamen kayboldu' sanmak

Yukarıdaki örnekte dikkatli bakan biri hâlâ **iki** çıkış durumu (`2098`→IZIN, `5560`→RED) olduğunu ve aralarındaki kararın `case 7341`'de verildiğini görebilir — düzleştirme mantığı **tamamen yok etmez**, yalnız **doğrudan görünürlüğünü kaldırır**. Bu yüzden 9. haftanın K-04 kuralı Flatten'i **tek başına yeterli** görmez; durum sabitlerinin kendisi de (K-02 ile) kodlanmalı ve opak yüklemlerle (K-01) karar noktaları gizlenmelidir — tam olarak bu haftanın dönüşüm hattının (`EncodeLiterals → EncodeArithmetic → Flatten → AddOpaque`) dört dönüşümü birlikte kullanmasının nedeni budur.

Dönüşümleri tek tek açalım: her satırın arkasındaki fikir

Tablodaki on satırı sırayla, 9. haftadaki tanımlarına referans vererek açalım. Her birinde üç soruyu yanıtlıyoruz: **ne yapar, hangi K kuralının otomatik hâlidir, ne zaman tercih edilir.**

1. `Flatten` **— kontrol akışı düzleştirme.** Bir fonksiyonun `if` / `else` /döngü yapısını, tek bir `while` döngüsü içindeki bir `switch` **dağıtıcısına** çevirir; her orijinal blok bir `case` olur, bloklar arası geçiş `case` numarasını değiştirmekle sağlanır. 9. hafta K-04'ün (bölüm 5, "KURAL K-04 — Kontrol akışı düzleştirme") birebir otomatik karşılığıdır. Bölüm 5'teki `erisim_ver` örneğinde gördüğümüz gibi, "tek dal, tek dönüş" gibi kolay hedefler için **orta** maliyetli, **orta-yüksek** güçlü bir ilk adımdır.

2. `InitOpaque` / `AddOpaque` / `UpdateOpaque` **— opak yüklem ailesi.** K-01'in (opak yüklem/döngü) otomatik hâlidir. Üç ayrı komuta bölünmüş olması bilinçlidir: `InitOpaque` **hazırlık** (durum değişkenlerini kurar, genelde `main`'de), `AddOpaque` **ekleme** (hedef fonksiyona gerçek koşulları yerleştirir), `UpdateOpaque` ise zaten eklenmiş opak yapıları **güncellemek/çeşitlendirmek** için kullanılır (birden çok gizleme turunda). Yukarıda ayrıntılandırdık.

3. `EncodeArithmetic` **— aritmetik kodlama (MBA).** K-02'nin (9. hafta bölüm 6: "aritmetik komutların kodlanması") otomatik hâlidir; `a + b` gibi basit bir işlemi, aynı sonucu veren ama okunması zor **mixed boolean-arithmetic** (karma mantıksal-aritmetik) ifadelere çevirir (9. haftanın `(a^b) + 2·(a&b) = a+b` örneğini hatırlayın). **Düşük** maliyetlidir; ama 9. haftanın "MBA sadeleştiricilerle açılabilir" uyarısı burada da geçerlidir (bölüm 7).

4. `EncodeLiterals` **— sabit ve dizge kodlama.** K-07'nin (dize kodlama) otomatik hâlidir; yukarıda ayrıntısıyla işledik. **Çok düşük** maliyetli, ama tek başına **zayıf** (çalışırken bellekte açığa çıkar — 9. haftanın "gizleme anahtar saklamaz" uyarısı).

5. `EncodeData` **— değişken gösterimini kodlama.** K-09'un (9. hafta bölüm 6: "değişken bölme, birleştirme ve dizi yeniden yapılandırma") otomatik hâline en yakın karşılıktır; bir değişkenin bellekteki **temsilini** (ör. tek bir 32-bit tamsayı yerine parçalanmış/kodlanmış bir yapı) değiştirir. Amaç, bir bellek dökümünde değerin **doğrudan** görünmemesidir.

6. `Split` / `Merge` **— fonksiyon bölme/birleştirme.** K-06'nın (9. hafta bölüm 5: "fonksiyon çağrılarının ve dış kütüphane bağımlılığının gizlenmesi") bir parçasıdır; `Split` bir fonksiyonu birden çok küçük parçaya böler, `Merge` birden çok fonksiyonu **tek** bir büyük fonksiyonda birleştirir. İkisi de fonksiyonun **sınırlarını** bulanıklaştırır — bir tersine derleyicide "bu, tek bir mantıksal işlem mi yoksa birkaçının birleşimi mi?" sorusunu zorlaştırır.

7. `Virtualize` **— sanallaştırma tabanlı gizleme.** K-10'un (9. hafta bölüm 7) otomatik hâlidir; fonksiyonu özel bir sanal makinenin bayt koduna çevirir. Tablodaki en **pahalı** satırdır (yukarıdaki "Aileler ve maliyet" kutusundaki "onlarca kat yavaşlama" uyarısı); yalnız **küçük, kritik** fonksiyonlara uygulanır (bölüm 1'deki karar akışı, bölüm 5'teki "Adım 5").

8. `Jit` **— çalışma anında kod üretimi.** K-12'nin (9. hafta bölüm 7: "kendini değiştiren kod ve dinamik şifreleme") kavramsal karşılığıdır; kodun bir kısmı **derleme anında değil, çalışma anında** üretilir. 9. haftanın K-12 için verdiği uyarı (işletim sisteminin bellek korumalarıyla — DEP/NX, W^X — çatışabilmesi) burada da geçerlidir; bu yüzden tabloda " (kavram)" notuyla işaretlidir — derste **uygulanmaz**, yalnız tanınır.

9. `AntiBranchAnalysis` / `AntiAliasAnalysis` / `AntiTaintAnalysis` **— önleyici (anti-analysis) ailesi.** 9. haftanın "önleyici" ailesinin (bölüm 2'deki beş aileden biri) doğrudan otomatik karşılığıdır; bu üçü sırasıyla **dal analizini**, **takma ad (alias) analizini** ve **kirlenme (taint) analizini** yapan araçları (bir tersine derleyicinin veya sembolik yürütücünün kullandığı teknikleri) hedef alıp zorlaştırır. K-06'daki "analiz araçlarının kendisini hedef almak daha zahmetlidir" notunun somut örnekleridir.

10. `RandomFuns` / `RndArgs` **— rastgele fonksiyon/parametre ekleme.** Bölüm 6'daki "Çeşitlendirme" notunda göreceğimiz gibi, bunlar tek başlarına bir K kuralına değil, **çeşitlendirmeye** ve K-06'daki "sahte parametre" fikrine hizmet eder: `RandomFuns` rastgele, işlevsiz (ama gerçek gibi görünen) fonksiyonlar ekler; `RndArgs` mevcut fonksiyonlara sahte parametreler ekler. İkisi de **tohumla birlikte** kullanıldığında her kopyayı farklılaştırır.

Kural: her satırı okurken üç soruyu sırayla sorun

Yeni bir Tigress dönüşümüyle (resmî belgede) karşılaştığınızda, bu bölümdeki disiplini tekrarlayın: **(1) Bu hangi 9. hafta ailesine ait? (2) En yakın K kuralı hangisi? (3) Maliyeti düşük mü, orta mı, yüksek mi?** Bu üç soru, yeni bir dönüşümü hiç görmeseniz bile onu doğru **kategoriye** ve doğru **maliyet bütçesine** yerleştirmenizi sağlar.

İşlenmiş örnek: `EncodeData` bir değişkeni nasıl parçalar (kavramsal)

K-09'un (9. hafta: değişken bölme/birleştirme) otomatik karşılığı olan `EncodeData`'yı somutlaştıralım. Diyelim `erisim_ver` içinde bir "deneme sayacı" değişkeni var:

ÖNCE — tek parça, doğrudan okunabilir bir değişken

```
int deneme_sayaci = 0;
...
deneme_sayaci++;
if (deneme_sayaci > 3) return KILITLI;
```

SONRA — kavramsal gösterim: EncodeData uygulanmış gibi (gerçek çıktı sürüme göre değişir)

```
struct { unsigned char alt; unsigned char ust; } _sayac_parcalari = {0, 0};
...
/* deneme_sayaci++ karşılığı: alt baytı artır, taşarsa üst baytı güncelle */
_sayac_parcalari.alt++;
if (_sayac_parcalari.alt == 0) _sayac_parcalari.ust++;
/* deneme_sayaci > 3 karşılığı: iki parçayı birleştirip karşılaştır */
if (((_sayac_parcalari.ust << 8) | _sayac_parcalari.alt) > 3) return KILITLI;
```

Bir bellek dökümünde (ör. bir hata ayıklayıcıyla programı durdurup belleği okumak), `deneme_sayaci`'nin **tek, bütün bir 32-bit tam sayı** olarak görünmesi yerine, birbirinden bağımsız iki bayt olarak, üstelik bir yapı (`struct`) içinde dağılmış olarak durur — bölüm 0'daki "bellek dökümünde değerin doğrudan görünmemesi" hedefine birebir uyar.

Sık yapılan hata: parçaları kullanımdan hemen sonra tekrar birleştirip saklamak

Bir geliştirici performans kaygısıyla `_sayac_parcalari`'ni her karşılaştırmadan önce **tek bir değişkende** (`int birlesik = (...) << 8 | (...)`) önbelleğe alıp o değişkeni **fonksiyon boyunca** saklarsa, artık ortada yine tek, bütün ve tanınabilir bir 32-bit blok vardır — K-09'un kazandırdığı avantaj kaybolur (9. haftanın aynı konudaki uyarısıyla birebir örtüşür). **Kural:** birleştirmeyi yalnız **tam ihtiyaç duyulan anda** yapın, kullanımdan hemen sonra atın; kalıcı bir "birleşik" kopya tutmayın.

İşlenmiş örnek: `Split` bir fonksiyonu nasıl böler (kavramsal)

`Split` dönüşümünü de somutlaştıralım. Diyelim `erisim_ver` iki mantıksal adımı art arda yapıyor: uzunluk kontrolü ve içerik karşılaştırması (tam olarak `ornek.c`'deki gerçek `erisim_ver`'in yaptığı gibi — bölüm 0'daki kod).

ÖNCE — tek fonksiyon, iki adım art arda

```
int erisim_ver(const char *jeton) {
    if (strlen(jeton) != strlen(GECERLI)) return 0;
    unsigned fark = 0;
    for (unsigned i = 0; i < sizeof GECERLI - 1; i++)
        fark |= (unsigned)((unsigned char)jeton[i] ^ (unsigned char)GECERLI[i]);
    return fark == 0;
}
```

SONRA — kavramsal gösterim: Split uygulanmış gibi (gerçek çıktı sürüme göre değişir)

```
static int _adim_a(const char *jeton) {          /* yalnız uzunluk kontrolü */
    return strlen(jeton) == strlen(GECERLI);
}
static unsigned _adim_b(const char *jeton) {      /* yalnız fark hesabı */
    unsigned fark = 0;
    for (unsigned i = 0; i < sizeof GECERLI - 1; i++)
        fark |= (unsigned)((unsigned char)jeton[i] ^ (unsigned char)GECERLI[i]);
    return fark;
}
int erisim_ver(const char *jeton) {
    if (!_adim_a(jeton)) return 0;
    return _adim_b(jeton) == 0;
}
```

Sonuç iki yeni, adı jenerik iki yardımcı fonksiyondur (`_adim_a` , `_adim_b`); bir tersine derleyicide artık **üç** ayrı sembol vardır (bölüm 0'daki "sembol tablosu" tanımını hatırlayın), ve `erisim_ver` 'in **kendisi** artık yalnız iki çağrıdan ibarettir — asıl mantık başka fonksiyonlara **dağılmıştır**. `Merge` bunun tam tersini yapar: birden çok küçük fonksiyonu tek, büyük bir fonksiyonda birleştirir; sonuç, "bu tek bir işlem mi yoksa birkaçının toplamı mı?" sorusunu tersinden zorlaştırır.

Split ile fonksiyon sınırları neden önemlidir?

Bir tersine mühendis genelde önce fonksiyon sınırlarına (isim, parametre sayısı, giriş/çıkış) bakarak bir "harita" çıkarır (9. hafta bölüm 0'daki tersine derleyici örneğini hatırlayın). `Split` , bu haritanın **kendisini** yanıltıcı hâle getirir: `_adim_a` ve `_adim_b` gibi jenerik adlı, küçük fonksiyonlar, bir mühendisin dikkatini dağıtacak sayıda ve büyüklükte olabilir. K-06'nın (fonksiyon gizleme) bu satırdaki karşılığı budur.

4. Dönüşüm hattı: birden çok dönüşümü birleştirmek

Dokuzuncu haftanın en önemli kuralı "tek teknik değil, birlikte" idi. Tigress'te bu, dönüşümleri **sırayla** (bir hat olarak) uygulamak demektir. Her `--Transform` bir öncekinin çıktısına uygulanır; sıra önemlidir.

İmza **en son** gelir (önce gizle, sonra imzala); her hattan sonra **davranışı test et** ve **maliyeti ölç**.

Kavramsal hat (birden çok dönüşüm sırayla)

```
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
```

Bu hat, tek bir denetimi (`erisim_ver`) önce sabit/dize kodlar, sonra aritmetiğini kodlar, sonra düzleştirir, sonra opak yüklemlerle güçlendirir. Sonuç, 9. haftada K-04'ün "güçlendirilmiş düzleştirme" dediği şeyin otomatik halidir.

> ⚠ **Sıra ve test kuralı**
>
> Dönüşüm sırası sonucu etkiler ve bazı sıralar başarımı gereksiz bozar. **Kural:** her dönüşüm hattından sonra programın **birim testlerini** çalıştırın (davranış korunmuş mu?) ve boyut/hız ölçün. Gizleme, davranışı asla değiştirmemelidir; değiştiriyorsa hat yanlıştır. Bu, 12. haftadaki "S16 test sonuçları"na doğrudan bağlanır.

İki farklı sırayı somut karşılaştıralım

"Sıra önemlidir" cümlesini soyut bırakmayalım. Aynı dört dönüşümü (`EncodeLiterals` , `EncodeArithmetic` , `Flatten` , `AddOpaque`) **iki farklı sırayla** düşünelim ve her sırada **neyin neyi kapsadığını** izleyelim. Kural basit ama sonuçları farklıdır: **bir dönüşüm, yalnızca çalıştığı anda kaynakta zaten var olan şeyi dönüştürebilir; kendisinden sonra gelen bir dönüşümün ürettiği yeni kodu göremez.**

Sıra 1 (bu bölümdeki hat, demo betiğinin kullandığı sıra):

```
EncodeLiterals → EncodeArithmetic → Flatten → AddOpaque
```

`EncodeLiterals` ilk çalıştığı için yalnızca **orijinal kaynaktaki** (`ornek.c` 'deki) sabitleri/dizgeleri kodlar. `Flatten` üçüncü sırada çalıştığında, düzleştirilmiş `switch` dağıtıcısının **kendi ürettiği** yeni durum sabitlerini (`case 1` , `case 2` , ...) `EncodeLiterals` artık **göremez** — çünkü o adım çoktan geçti. Bunun **sakıncası yoktur**, çünkü Flatten kendi durum değerlerini zaten dağınık/öngörülemez üretir (bölüm 5'teki not: "değerleri K-02 ile ya da rastgele, dağınık sabitlerle üretilmeli" — bu iş burada Flatten'in kendisine düşer). Bu sırada kazanç şudur: `Flatten` çalıştığında dağıtıcının her

`case`'inin **içindeki** aritmetik zaten (`EncodeArithmetic` sayesinde) karmaşıklaştırılmış durumdadır — yani düzleştirilmiş kod hem **yapı** hem **içerik** olarak zor okunur.

Sıra 2 (yapısal dönüşümler önce, kodlama dönüşümleri sonra):

```
Flatten → AddOpaque → EncodeArithmetic → EncodeLiterals
```

Burada `EncodeLiterals` **son adımdır**; artık yalnız orijinal kaynaktaki sabitleri değil, `Flatten`'in ürettiği durum sabitlerini ve `AddOpaque`'in eklediği opak koşul sabitlerini de görebilir ve kodlayabilir — **daha geniş bir kapsama alanı**. Ama bir bedeli vardır: `EncodeArithmetic` de dördüncü sırada çalıştığı için artık **tek bir küçük fonksiyonun** aritmetiğini değil, `Flatten`'in dağıtıcı mantığına ve `AddOpaque`'in eklediği koşullara **yayılmış** aritmetiği kodlamak zorundadır — dönüşümün işi büyür, üretilen kod ve maliyet de büyür.

	Sıra 1 (kodla → yapılandır)	**Sıra 2 (yapılandır → kodla)**
`EncodeLiterals` neyi kapsar?	Yalnız orijinal kaynaktaki sabitler	Orijinal + Flatten/AddOpaque'in ürettiği yeni sabitler
`Flatten`'in gördüğü aritmetik	Zaten kodlanmış (daha zor okunur `case` içerikleri)	Ham, sade (daha kolay okunur `case` içerikleri)
Beklenen maliyet	Orta	Muhtemelen daha yüksek (son adımlar daha büyük bir yüzeye uygulanır)

Çıkarım: "doğru" tek bir sıra yoktur; sıra bir **kapsam/maliyet ödünleşimidir**. Bu hafta demo betiğinin kullandığı Sıra 1'i temel alıyoruz çünkü basit ve öngörülebilir; ama kendi projenizde farklı bir sıra denerseniz, bölüm 7'deki ölçüm adımlarını **her sıra için ayrı ayrı** çalıştırıp karşılaştırmadan "bu sıra daha iyi" demeyin — bu tam olarak yukarıdaki "sıra ve test kuralı"nın pratikte ne anlama geldiğidir.

Sık yapılan hata: hattı yalnız 'daha çok dönüşüm = daha güvenli' mantığıyla uzatmak

Dört dönüşümü art arda koymak dururken, bazı öğrenciler "10 dönüşüm daha güvenlidir" diyerek aynı aileden (ör. birden fazla farklı opak yüklem dönüşümünü peş peşe) tekrar tekrar ekler. Bu hem maliyeti gereksiz büyütür hem de bazı dönüşüm çiftleri **birbirini etkisizleştirebilir** ya da hiçbir ek fayda sağlamadan yalnızca boyutu şişirebilir. **Kural:** hattı **her aileden bir temsilci** içerecek şekilde kurun (kontrol akışı + veri + önleyici), aynı aileden birden fazla dönüşümü yalnız ölçümle (bölüm 7) gerekçelendirilmişse ekleyin.

İşlenmiş örnek: imzalama sırası bozulursa ne olur?

"İmza en son gelir" kuralını bir somut senaryoyla test edelim. Diyelim bir geliştirici yanlışlıkla sırayı tersine çevirdi:

1. `temiz.c` derlenir → `program` (imzasız).
2. `program` **imzalanır** → `program.imzali` (bir dijital imza, dosyanın **o anki** bayt içeriğine bağlıdır).
3. Sonra "unutulan" gizleme adımı fark edilir; `temiz.c`'den `gizli.c` üretilip yeniden derlenip, çıkan yeni ikili eski imzalı dosyanın **üzerine** kopyalanır.

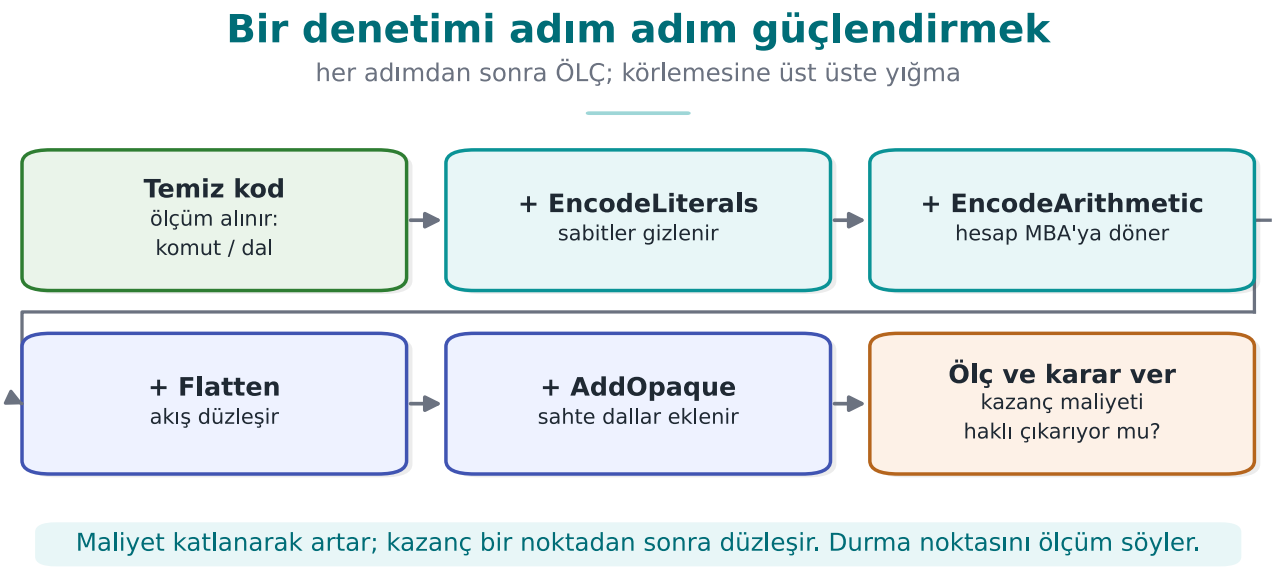
Sonuç: adım 3'te üretilen yeni ikili dosya, adım 2'de imzalanan bayt içeriğinden **farklıdır** (gizleme dosyanın her baytını değiştirebilir — bölüm 5/7'deki komut sayısı artışını hatırlayın). Doğrulama yapan taraf (ör. bir yükleyici, bir mağaza) imzayı **geçersiz** bulur ve programı **reddeder** — üretim ortamında bu, "sürüm yayınlanamadı" anlamına gelir.

Sık yapılan hata: 'gizlemeyi sonradan ekledim, imzayı yeniden hesaplamam yeter' diye düşünmek

Bazı ekipler bu hatayı fark ettiğinde yalnız imzayı yeniden hesaplayıp sorunu çözdüklerini düşünür — teknik olarak doğrudur (yeni içerik yeni imzayla eşleşir) ama **süreç** hâlâ yanlıştır: bu, "imzalama CI'nin **son** adımı olmalı" kuralının el ile, defalarca hatırlanması gereken bir istisna hâline gelmesi demektir. **Kural:** CI iskeletinde (bölüm 9) `imzala` aşaması **yapısal olarak** `gizle` ve `olc` aşamalarından **sonra** tanımlanır; bu sırayı korumak insan hafızasına değil, **CI'nin kendi tanımına** bırakılmalıdır.

5. Adım adım örnek: bir denetimi güçlendirmek (sentetik)

Tek bir sentetik erişim denetimini alıp katman katman güçlendirelim. Amaç, her adımın **ne eklediğini** ve **ne kadara mal olduğunu** görmektir.



Başlangıç: temiz.c (okunur, korumasız)

```
int erisim_ver(const char *jeton) {
    if (jeton_gecerli(jeton)) return IZIN;   /* tek dal, tek dönüş: kolay hedef */
    return RED;
}
```

Adım	Uygulanan dönüşüm	Ne değişir?	Maliyet
0	(yok)	`strings`, tek dal, tek bayt yamasıyla atlanabilir	—
1	EncodeLiterals	`IZIN` / `RED` ve dizgeler düz görünmez	Çok düşük
2	EncodeArithmetic	Karşılaştırma/hesap karmaşık ifadeye döner	Düşük
3	Flatten	Tek dal görünmez; switch dağıtıcı	Orta
4	AddOpaque	Sahte dallar, opak koşullar eklenir	Orta
5	(Virtualize, yalnız gerekirse)	Fonksiyon VM bayt koduna döner	Yüksek

Çıkarım: Her adım saldırıyı biraz daha pahalı yapar; ama her adım bir maliyet ekler. Nerede duracağınıza, korunan varlığın değerine ve ölçtüğünüz maliyete göre karar verirsiniz (bölüm 4). Adım 5 (Virtualize) çoğu denetim için gereksizdir; yalnız en kritik ve küçük fonksiyonlara uygulanır.

Sayısal bir örnekle adım adım maliyet artışını izleyelim (varsayımsal)

Yukarıdaki tabloyu sayılarla dolduralım. Aşağıdaki komut/dal sayıları **varsayımsaldır** (Tigress kurulu değilken gerçek ölçüm alınamaz); yalnız her adımın **büyüklük mertebesini** ve **aritmetiğin nasıl doğrulandığını** göstermek içindir. Gerçek sayılar için `tigress-hatti.sh`'yi Tigress kuruluyken çalıştırıp betiğin bastığı `objdump` çıktısını okuyun (bölüm 0'daki "komut/dal sayımı" tanımını hatırlayın).

Adım	Komut	Dal	Bir önceki adıma göre artış	Adım 0'a göre kümülatif artış
0 — temiz	22	2	—	—
1 — + EncodeLiterals	30	2	`(30-22)/22×100 ≈ %36,4`	%36,4
2 — + EncodeArithmetic	46	2	`(46-30)/30×100 ≈ %53,3`	`(46-22)/22×100 ≈ %109,1`
3 — + Flatten	78	5	`(78-46)/46×100 ≈ %69,6`	`(78-22)/22×100 ≈ %254,5`
4 — + AddOpaque	101	7	`(101-78)/78×100 ≈ %29,5`	`(101-22)/22×100 ≈ %359,1`

Üç gözlem:

1. **Dal sayısı yalnız Flatten'de sıçrıyor** (2 → 5): bu beklenir, çünkü düzleştirme mantığı gereği tek bir `if` yerine çok yollu bir `switch` dağıtıcısı kurar (bölüm 7'nin "güç" ölçütünün doğrudan kaynağı).
2. **Kümülatif artış, adım artışlarının toplamı değildir** — her yüzde bir öncekinin üzerine **çarpımsal** eklenir (`%36,4` sonra `%53,3` ekleyince toplam `%109,1` olur, `%36,4+%53,3=%89,7` değil). Bu, yüzde hesaplarken sık yapılan bir aritmetik hatadır; her satırı **kendi paydasıyla** (bir önceki adımın sayısıyla) hesaplayın.

3. **%359 gibi büyük bir kümülatif sayı**, dört dönüşümü üst üste yığmanın **beklenen** sonucudur; tek bir dönüşümün (9. haftadaki elle Flatten+AddOpaque örneğinde ölçülen gerçek **%82**) maliyetiyle karıştırılmamalıdır — burada dört dönüşüm birlikte sayılıyor.

Sık yapılan hata: Adım 5'i (Virtualize) 'madem buradayım, hepsini ekleyeyim' diye uygulamak

Yukarıdaki tabloda %359 gibi bir kümülatif maliyet gördükten sonra "zaten çok arttı, bir de Virtualize ekleyeyim" diye düşünmek yaygın bir hatadır. Virtualize, diğer dört dönüşümün toplamından **kat kat** daha pahalıdır (bölüm 1'in "Aileler ve maliyet" kutusu: "onlarca kat yavaşlama") — yüzdelerle değil, **kat** ile ölçülür. Bir denetim fonksiyonu için %359'luk bir büyüme çoğu zaman kabul edilebilirken, üstüne bir de 10-50 kat yavaşlama eklemek çoğu ürün için anlamsızdır. **Kural:** Virtualize kararını ayrı verin; "zaten gizledim" mantığıyla otomatik eklemeyin — bölüm 1'deki karar akışına (hassas mı, değeri ne kadar) geri dönün.

İşlenmiş örnek: tek dönüşüm mü, dört dönüşüm birlikte mi? Sayısal karşılaştırma

Bölüm 1'deki "birlikte kullanmanın" gerekçesini yukarıdaki sayılarla sınayalım. Aynı `erisim_ver` fonksiyonuna **yalnız tek bir dönüşüm** uygulasaydık ne olurdu, dört dönüşümü **birlikte** uyguladığımızda ne oldu — ikisini yan yana koyalım (sayılar yukarıdaki tablodan alınmış/varsayımsaldır):

Yaklaşım	Komut	Dal	Maliyet artışı (adım 0'a göre)	Kaç farklı analiz tekniği zorlanır?
Yalnız `EncodeLiterals`	30	2	%36,4	Yalnız statik dize taraması (`strings`)
Yalnız `Flatten` (varsayımsal, tek başına uygulansaydı)	~54	5	~%145,5	Yalnız kalıp tanıma/CFG okuma
Dört dönüşüm birlikte (bu haftanın hattı)	101	7	%359,1	Dize taraması **+** kalıp tanıma **+** MBA sadeleştirme **+** opak yüklem çözme

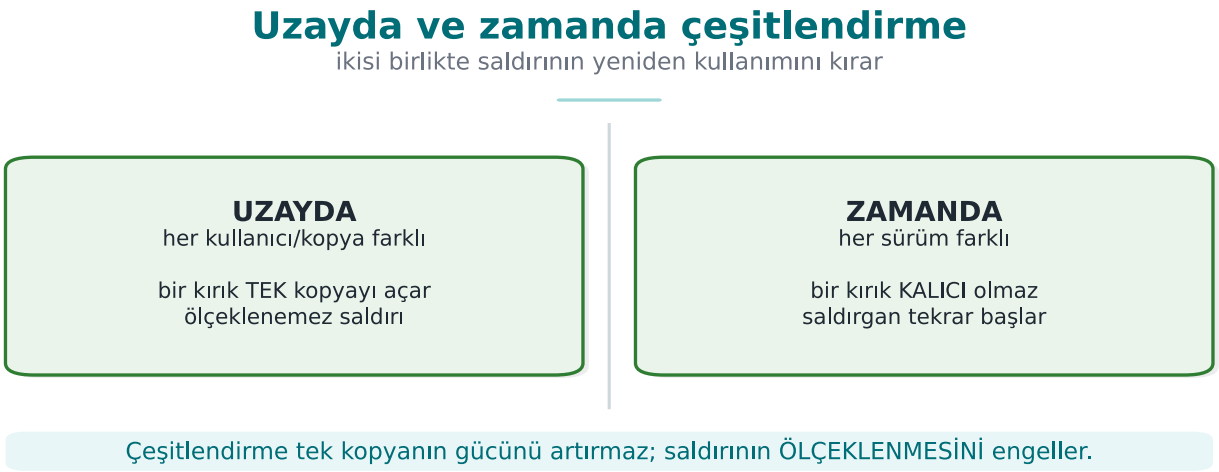
Son sütun asıl noktadır: yalnız `EncodeLiterals` uygulamak, saldırganın yalnız **bir** aracını (`strings`) etkisiz kılar; saldırgan hâlâ kontrol akışını doğrudan okuyabilir. Dört dönüşümü birlikte uygulamak, saldırganın **dört farklı** analiz tekniğini **aynı anda** ve **birbirine bağımlı** biçimde yenmesini gerektirir (9. haftanın "iki zayıf katman, birlikte tek katmandan daha güçlüdür" kuralının sayısal gerekçesi).

Kural: maliyet artışını değil, 'kaç farklı tekniği zorladığını' raporlayın

S9/S15'inizde yalnız "%359 büyüdü" demek eksiktir. Yukarıdaki tablonun son sütunu gibi, hangi dönüşümün **hangi analiz tekniğine karşı** koyduğunu da yazın (bölüm 7'deki "hangi araca karşı ölçüldü" tablosuyla birlikte okuyun). Bu, bir değerlendiriciye yalnız bir sayı değil, **savunmanın genişliğini** gösterir.

6. Çeşitlendirme: aynı kaynaktan farklı ikili dosyalar

Dokuzuncu haftanın "Kural 2 — otomasyonu kır" ilkesini hatırlayın: bir saldırgan bir kopyayı kırıp saldırısını bütün kopyalara dağıtabiliyorsa, bir kırık her yeri açar. **Çeşitlendirme** (diversification), aynı kaynaktan **davranışça özdeş ama yapıca farklı** ikili dosyalar üretmektir. Tigress bunu bir **tohum** (seed) ile yapar: aynı dönüşümleri farklı tohumlarla çalıştırırsanız, opak yüklemler, sahte dallar ve düzleştirme durumları kopyadan kopyaya değişir.



Kavramsal: iki tohum, iki farklı ikili

```
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.c ve gizli_b.c aynı işi yapar; makine kodları farklıdır
```

İki tür çeşitlendirme (9. haftadan):

- **Uzayda çeşitlendirme:** farklı kullanıcılara/cihazlara farklı tohumla üretilmiş kopyalar dağıtmak. Bir kopyaya yazılan otomatik saldırı, başka kopyada çalışmaz.
- **Zamanda çeşitlendirme:** her sürümü yeni bir tohumla üretmek. Eski sürüme karşı bulunan saldırı, yeni sürümde bozulur. Bu, 10. haftadaki anahtar/sürüm yenileme politikanızla birlikte çalışır.

İlgili dönüşümler

Tigress'te `RandomFuns` rastgele sahte fonksiyonlar, `RndArgs` sahte parametreler ekler; tohumla birleşince her yapı farklı görünür. Çeşitlendirme gizlemenin **gücünü** artırmaz (bir kopyayı kırmak yine mümkündür) ama **ölçeklenmesini** engeller — bu ayrım 9. haftanın çekirdek fikriydi.

İşlenmiş örnek: iki tohumla üretilen ikiliyi adım adım karşılaştıralım

Demo betiğinin ADIM 4'ü tam olarak bunu yapar: `--Seed=1001` ve `--Seed=2002` ile aynı `ornek.c`'den iki ayrı `erisim_ver` üretir, derler ve `cmp -s A B` ile bayt bayt karşılaştırır. Adımları tek tek izleyelim:

1. **Aynı kaynak, iki komut:**

```
tigress --Seed=1001 --Transform=Flatten --Transform=AddOpaque --Functions=erisim_ver --out=a.c
ornek.c
tigress --Seed=2002 --Transform=Flatten --Transform=AddOpaque --Functions=erisim_ver --out=b.c
ornek.c
```

Dikkat: `--Transform` listesi ikisinde de **birebir aynı** (`Flatten`, `AddOpaque`); tek fark `--Seed` değeri. 2. **Derle:** `a.c` ve `b.c`, aynı derleyici ve bayraklarla (`-O2`) iki ayrı ikiliye (`A`, `B`) derlenir. 3. **Davranışı karşılaştır (önce bu!):** her iki ikili de `CEN429-OK` girdisiyle çalıştırılıp **aynı** çıktıyı (`IZIN`) verdiği doğrulanmalıdır — çeşitlendirme davranışı **asla** değiştirmemelidir (bölüm 4'teki kuralın burada da geçerli olması). 4. **Yapıyı karşılaştır:** `cmp -s A B`, iki dosyayı bayt bayt karşılaştırır; `$?` (çıkış kodu) `0` ise dosyalar **özdeştir** (kötü — çeşitlendirme işe yaramamış demektir), `1` ise **farklıdır** (beklenen sonuç).

Fark oranını sayısallaştıralım (varsayımsal, 9. haftanın gerçek %66,4'lük örneğine benzer biçimde). Diyelim `erisim_ver`'in düzleştirilmiş+opak-yüklemli hâli her iki ikilide de **160 bayt** yer kaplıyor; bayt bayt karşılaştırdığımızda **96 baytın** farklı çıktığını ölçtük:

```
fark_orani = degisen_bayt / toplam_bayt × 100
           = 96 / 160 × 100
           = %60
```

Yorum: iki kopyanın **%60'ı farklı**, **%40'ı** ortak (muhtemelen fonksiyon girişi/çıkışı gibi derleyicinin sabit tuttuğu parçalar — 9. haftadaki 512 baytlık örnekte kalan %33,6'nın aynı nedeni). 9. haftadaki elle çeşitlendirmede ölçülen %66,4 ile bu hafta Tigress'in `--Seed` ile ürettiği %60 **aynı büyüklük mertebesindedir** — bu beklenir, çünkü ikisi de aynı fikri (rastgele seçimlerle yapıyı değiştirmek) uygular; tam sayı elle yazılan ile araçla üretilen kod arasında elbette farklı olacaktır.

Sık yapılan hata: `cmp` çıktısını okumadan 'farklı, demek ki çeşitlendirme oldu' demek

`cmp -s A B` yalnız dosyaların **özdeş olup olmadığını** söyler; **neyin** farklı olduğunu göstermez. İki ikili farklı olabilir ama fark yalnız **önemsiz** bir yerde (ör. derleme zaman damgası, sıkıştırılmamış hata ayıklama bilgisi) olabilir; korunan fonksiyonun (`erisim_ver`) kendisi hâlâ aynı kalmış olabilir. **Kural:** çeşitlendirme iddiasını yalnız `cmp` ile değil, **hedef fonksiyonun** kendisini (bölüm 0'daki `objdump` yöntemiyle, yalnız `<erisim_ver>:` etiketinin altını alıp) ayrı ayrı karşılaştırarak kanıtlayın.

Kural: tohumu her zaman kaydedin

Bir ikili dosyayı üretirken kullandığınız `--Seed` değerini **kaydetmezseniz**, o ikiliyi bir daha **asla** aynı şekilde üretemezsiniz (bir hata ayıklama oturumunda o sürümü yeniden derlemeniz gerekirse sorun olur). Bölüm 9'da S15 hattının "her sürüm bir tohumla çeşitlendirilir; tohum ve sürüm kimliği kaydedilir" kuralının nedeni tam olarak budur.

İşlenmiş örnek: `cmp` yerine hedef fonksiyonu doğrudan karşılaştıralım

Yukarıdaki "sık yapılan hata" kutusunun önerdiği doğru yöntemi uygulayalım. `cmp -s A B` yalnız "farklı/aynı" der; hedefin (`erisim_ver`) kendisinin farklı olup olmadığını görmek için bölüm 0'daki `objdump` yöntemini **iki ikiliye birden** uygulayıp yalnız o fonksiyonun bölümünü karşılaştırırız:

Kavramsal: yalnız hedef fonksiyonun bölümünü ayıklayıp karşılaştırma

```
# Her iki ikiliden yalnız <erisim_ver>: etiketinin altını çıkar
objdump -d A | awk '/<erisim_ver>:/{a=1} a{print} /^$/{if(a)exit}' > erisim_A.asm
objdump -d B | awk '/<erisim_ver>:/{a=1} a{print} /^$/{if(a)exit}' > erisim_B.asm
diff erisim_A.asm erisim_B.asm | wc -l    # 0 ise hedef fonksiyon AYNI kalmış demektir
```

Eğer `diff` çıktısı **boşsa** (satır sayısı 0), bu iki tohumun `erisim_ver` üzerinde **hiçbir fark yaratmadığı** anlamına gelir — yani bölüm 6'daki iddia (çeşitlendirme oldu) **yanlış** olur, `cmp`'nin bütün dosyada bulduğu fark yalnız önemsiz bir yerdeydi (ör. derleme zaman damgası). `diff` çıktısı **doluysa**, hedef fonksiyonun kendisi gerçekten değişmiş demektir — ancak o zaman "çeşitlendirme kanıtlandı" denebilir. Bu iki adımlı doğrulama (önce `cmp` ile "bir şeyler farklı mı?", sonra `objdump` + `diff` ile "hedefin kendisi mi farklı?") bölüm 6'nın başındaki demo betiği anlatımını **tamamlar**.

> ✓ **Kural: iddia + doğrulama komutu birlikte yazılır**
>
> S9/S15'inizde "iki tohumla çeşitlendirdim" cümlesi tek başına yeterli değildir; yukarıdaki gibi **hangi komutla doğruladığınızı** da yazın. Bu, 12. haftadaki "plan değil sonuç" kuralının bir başka somut uygulamasıdır: iddia, çalıştırılabilir bir komutla desteklenmelidir.

Kaç kullanıcı, kaç tohum? Uzayda çeşitlendirmeyi ölçekte düşünelim (varsayımsal)

Bölüm 6'nın başındaki "uzayda çeşitlendirme" tanımını bir dağıtım senaryosuyla somutlaştıralım. Diyelim bir mobil uygulamanın **10.000** kullanıcısı var ve her kurulumda **farklı** bir tohum kullanılıyor:

Senaryo	Tohum sayısı	Bir kırığın etkilediği kullanıcı oranı
Çeşitlendirme yok (tek ikili, herkese aynı)	1	%100 (10.000/10.000)
Zayıf çeşitlendirme (yalnız 2 sabit varyant, ör. `-O2` / `-O3`)	2	~%50 (bir varyantı kıran, o varyantı kullanan **herkesi** etkiler — bölüm 6'daki "sık yapılan hata" kutusunun sayısal karşılığı)
Gerçek çeşitlendirme (her kuruluma benzersiz `--Seed`)	10.000	`1/10.000 = %0,01` (yalnız kırılan **o tek** kopya)

Üçüncü satırdaki hesap basittir: bir saldırgan tek bir kopyayı kırıp o kopyaya özel bir bayt yaması üretse bile, bu yama yalnız **aynı tohumla üretilmiş** ikililerde işe yarar; her kullanıcı farklı bir tohuma sahipse, yama başka hiçbir kopyada çalışmaz. **Bu sayılar varsayımsaldır** (gerçek etki, saldırganın yamayı ne kadar genelleştirebildiğine de bağlıdır — ör. bir yama yalnız belirli bir bayt ofsetini hedefliyorsa etkisi daha da dar olur, davranışsal bir desene dayanıyorsa biraz daha geniş olabilir); amaç yalnız "tohum sayısı arttıkça bir kırığın **yayılma yüzeyi** küçülür" ilişkisini göstermektir.

İşlenmiş örnek: zamanda çeşitlendirmeyi bir zaman çizelgesinde izleyelim (varsayımsal)

Yukarıdaki "uzayda çeşitlendirme" örneğinin ikizi olan **zamanda çeşitlendirmeyi** de bir sürüm takvimiyle somutlaştıralım. Diyelim bir ürün her ay yeni bir sürüm yayınlıyor ve her sürüm CI'de (bölüm 9'un kuralı) **yeni bir tohumla** gizleniyor:

Ay	Sürüm	Tohum	Durum
Ocak	v1.0	1001	Yayınlandı
Şubat	v1.1	4832	Yayınlandı; bir saldırgan v1.0'ı Şubat ortasında kırdı (bayt yaması yayıldı)
Mart	v1.2	9214	Yayınlandı; Şubat'taki yama **artık işe yaramıyor** (farklı tohum, farklı yapı)

Şubat'taki kırığın "ömrü" yalnız **bir ay** (v1.1'in yayın tarihinden v1.2'nin yayın tarihine kadar) sürdü — bölüm 1'in "Kural 3 — diğer katmanlara zaman kazandır" ilkesinin sayısal karşılığı budur. Eğer ürün tohumu **hiç değiştirmeden** (zamanda çeşitlendirme olmadan) her ay yeni özellik eklese bile aynı gizleme yapısını korusaydı, Şubat'taki kırık **süresiz** olarak geçerli kalırdı.

Bu döngünün **maliyeti neredeyse sıfırdır**, çünkü bölüm 9'daki CI iskeleti her sürümde gizleme + ölçüm + imzalamayı **otomatik** çalıştırır (bölüm 2'deki "araçla mı, el ile mi" karşılaştırmasını hatırlayın); el ile çeşitlendirme yapılsaydı, her ay yeniden elle gizleme yazmak gerekirdi — bu da pratikte hiç sürdürülemezdi.

Zamanda çeşitlendirme tek başına yeterli mi?

Hayır — bölüm 1'deki "Kural 3" gizlemenin **diğer katmanlara zaman kazandırdığını** söyler, kalıcı bir çözüm olduğunu değil. Ay sonunda yeni sürüm çıkana kadar geçen sürede saldırgan hâlâ Şubat sürümünü kullanan kullanıcılara zarar verebilir. Bu yüzden zamanda çeşitlendirme; RASP (6. hafta, çalışma anında sızma girişimini algılama) ve sunucu tarafı denetimlerle (ör. eski/kırık sürümleri reddetme) **birlikte** kullanılır.

7. Gizlemeyi ve çeşitlendirmeyi ölçmek

Dokuzuncu haftada gizlemeyi dört boyutta (güç, dayanıklılık, gizlilik, maliyet) ölçmeyi öğrendik. Tigress'in en büyük öğretim değeri, bu ölçümleri **somut** yapabilmenizdir: aynı programı gizleyip önce/sonra ölçersiniz.

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

Maliyet ölçümü (kolay ve zorunlu)

Her dönüşüm hattı için şunları ölçün ve S9/S15'e yazın:

Ölçüt	Nasıl	Beklenti
İkili boyut	`ls -l` / `size`	Gizleme boyutu büyütür
Çalışma süresi	Bir işlemi çok kez çalıştırıp ortalama	Flatten/Virtualize yavaşlatır
Kaynak/CFG karmaşıklığı	Tersine derleyicide (ör. Ghidra) temel blok sayısı	Önce/sonra artış

Basit maliyet ölçümü (kavram)

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

Dayanıklılık ölçümü (ileri, kavramsal)

Dokuzuncu haftada dayanıklılığın "bir iddia değil, ölçülen bir nicelik" olduğunu söylemiştik. Tigress bu konuda akademik çalışmanın merkezindedir: Banescu ve arkadaşlarının çalışması, Tigress dönüşümlerinin **sembolik yürütmeye** (KLEE gibi) ne kadar dayandığını ölçer. Bulgu, ders için önemli bir kuraldır:

- Tek bir dönüşüm (yalnız Flatten) sembolik yürütmeyle çoğu zaman geri açılabilir.
- Dönüşümleri **birleştirmek** ve durum uzayını büyütmek (opak yüklemleri çözmesi pahalı yapılara bağlamak) çözücünün işini üstel olarak zorlaştırabilir.
- Ama bu, **maliyeti** de artırır. Yani dayanıklılık ve maliyet, tam da 9. haftada dediğimiz gibi ödünleşir.

Ölçüm kuralı (S9/S15)

Projenizde bir gizleme kararını "güçlü" diye değil, **ölçüyle** savunun: "Flatten + AddOpaque hattı ikili boyutu %X büyüttü, işlemi %Y yavaşlattı; buna karşılık hedef fonksiyonun temel blok sayısı Z katına çıktı." Ölçülmemiş gizleme, 12. haftadaki değerlendiricinin gözünde bir iddiadır, kanıt değil.

Boyut ve süre ölçümünü tamamlayalım: bölüm 5'teki sayılara ekleyelim

Bölüm 5'te tek bir fonksiyonun (`erisim_ver`) komut ve dal sayısındaki artışı adım adım izledik (varsayımsal %359,1 kümülatif komut artışı, adım 0'daki 2 daldan adım 4'teki 7 dala çıkış). Şimdi bu sayıları Collberg'in dört ölçütüne (yukarıdaki tablo) dağıtalım ve **boyut** ile **süre**yi de ekleyelim.

Güç (potency), 9. haftanın çevrimsel karmaşıklık formülüyle (`dal_sayısı + 1`):

```
Adım 0 (temiz)  : 2 dal + 1 = 3
Adım 4 (tam hat): 7 dal + 1 = 8
Artış           : (8 - 3) / 3 × 100 ≈ %166,7
```

Maliyet (cost), bölüm 5'teki komut sayısı artışı: zaten hesapladık, **%359,1** (kümülatif, dört dönüşüm birlikte).

Boyut: Bu noktada önemli bir uyarı gerekir.

Sık yapılan hata: bütün `.exe`/ikili dosyanın boyutunu (`ls -l`) ölçüp 'neredeyse değişmedi' sonucuna varmak

Bu haftanın demosu gibi **küçük, sentetik** bir programda, `ls -l ornek_temiz ornek_gizli` ile ölçülen **toplam dosya boyutu** çoğu zaman neredeyse aynı çıkar — çünkü dosyanın büyük kısmı (birkaç KB) ELF/PE başlıkları, dinamik bağlama tabloları ve çalışma zamanı kütüphanesine referanslardan oluşur; tek bir küçük fonksiyonun 79 komut büyümesi bu sabit yükün yanında **istatistiksel gürültüde kaybolur**. Bunu görüp "gizleme boyutu büyütmüyor" sonucuna varmak yanlıştır. **Kural:** küçük/sentetik demolarda boyut ölçütü için **bütün dosyayı değil, hedef fonksiyonun kendi kod boyutunu** (bölüm 0'daki `objdump` komut sayımını bayt cinsine çevirerek) ölçün. Gerçek, büyük bir üründe (yüzlerce fonksiyon gizlenmiş) bu etki toplam dosya boyutunda da görünür hâle gelir; küçük demoda görünmemesi **beklenen** bir durumdur, hatanın kanıtı değildir.

Süre: Benzer bir uyarı süre için de geçerlidir — bu kadar küçük bir fonksiyonun tek çağrısı, işletim sisteminin zamanlayıcı hassasiyetinin (genelde milisaniye altı) çok altında sürer; anlamlı bir süre farkı görmek için fonksiyonu **çok sayıda** (ör. bir milyon) kez döngüde çağırıp toplam süreyi ölçmek gerekir:

Kavramsal: süre ölçümü (çok sayıda çağrıyla anlamlı hâle getirme)

```
# Sözde kod: gerçek ölçüm aracı platforma göre değişir (ör. `time`, bir kıyaslama döngüsü)
for i in $(seq 1 1000000); do ./ornek_temiz CEN429-OK >/dev/null; done   # zaman ölçülür
for i in $(seq 1 1000000); do ./ornek_gizli CEN429-OK >/dev/null; done  # zaman ölçülür,
karşılaştırılır
```

Not: yukarıdaki döngü aslında her seferinde **yeni bir süreç başlatma** maliyetini de ölçer (`fork` / `exec`), bu da gerçek fonksiyon süresini gölgeleyebilir; gerçek bir ölçümde fonksiyon bir döngü **içinde**, aynı süreçte milyonlarca kez çağrılıp yalnız o döngünün süresi ölçülür. Bu ayrım — "sürecin başlatma maliyeti" ile "fonksiyonun kendi süresi" — küçük demolarda sık karıştırılan bir noktadır.

Dört ölçütü tek tabloda toplayalım

Ölçüt	Bu haftaki demo için değer	Kaynak
Güç (potency)	~%166,7 karmaşıklık artışı (3→8)	Yukarıdaki hesap, dal sayısı + 1
Maliyet (cost)	~%359,1 komut artışı (22→101, dört dönüşüm birlikte)	Bölüm 5
Dayanıklılık (resilience)	Ölçülmedi bu demoda; aşağıdaki alt bölümün konusu	Aşağıda
Gizlilik (stealth)	Ölçülmedi bu demoda; istatistiksel ayırt edilebilirlik gerekir	9. hafta bölüm 9

Kural: dördünü birden raporlayın, yalnız kolay ölçüleni değil

1. haftadaki "sık yapılan hata: yalnızca maliyeti ölçüp 'güçlü' demek" uyarısı burada da geçerlidir. Bu demo yalnız güç ve maliyeti **doğrudan sayıyla** verir; dayanıklılık ve gizlilik için ayrı bir araç (KLEE gibi bir sembolik yürütücü, ya da istatistiksel bir ayırt edicilik testi) gerekir. S9/S15'inizde dördünü de yazın; hangi ikisinin **ölçülmediğini** de açıkça belirtin — bu, "ölçmedim" demekten daha dürüst bir "henüz ölçülmedi, şu araçla ölçülebilir" cümlesidir.

Dayanıklılığı biraz daha somutlaştıralım: `AddOpaque`'in ürettiği koşulu sembolik yürütme nasıl kırar?

Yukarıdaki bölüm "dayanıklılık ölçümü (ileri, kavramsal)" dedi ve Banescu vd. çalışmasına atıfta bulundu; 9. hafta bu mekanizmayı klasik bir opak yüklem üzerinde (`((x*(x+1)) & 1) == 0`) adım adım göstermişti. Aynı akıl yürütmeyi, bu haftanın konusu olan **araçla üretilmiş** bir opak yükleme (Tigress'in `AddOpaque` ile eklediği koşullardan biri, kavramsal olarak) uygulayalım:

1. Bir sembolik yürütme aracı (KLEE gibi), programı **somut** değerlerle değil, girdiyi bilinmeyen bir **sembol** olarak tutup çalıştırır.
2. `AddOpaque`'in eklediği bir dala geldiğinde, aracın sorması gereken soru aynıdır: *"bu koşul, girdinin **her** değeri için aynı sonucu mu verir, yoksa onu bozan bir girdi var mı?"*
3. Bu soruyu bir **kısıt çözücüye** (SMT solver) devreder. Eğer `AddOpaque`'in ürettiği koşul, 9. haftadaki `x*(x+1)` örneği gibi **matematiksel olarak basit ve bilinen bir kalıba** dayanıyorsa, çözücü bunu saniyeler içinde çözer ve o dalın "ölü" (hiç çalışmayan) olduğunu kanıtlar.
4. Sonuç: araçla üretilmiş olması, opak yüklemi **otomatik olarak** daha dayanıklı yapmaz — üretilen koşulun **matematiksel yapısı** hâlâ belirleyicidir. Bu, 9. haftanın "her tekniğin her araca eşit dayanmadığı" uyarısının Tigress'in kendi çıktısı için de geçerli olduğunu gösterir.

> **Peki Tigress hiçbir şey katmıyor mu?**
>
> Katıyor — ama katkısı **tek bir opak yüklemi kırılmaz yapmak** değil, (1) çok sayıda opak yüklemi **otomatik ve tutarlı** biçimde eklemek (elle yüzlerce yüklem yazmak pratik değildir), (2) bunları **tohumla çeşitlendirmek** (bölüm 6), (3) diğer dönüşümlerle (Flatten, EncodeArithmetic) **birleştirmek** — tek başına zayıf katmanları üst üste koyarak çözücünün işini zorlaştırmaktır (9. haftanın "iki zayıf katman, birlikte tek katmandan daha güçlüdür" kuralı). Yani kazanç **tek bir yüklemin matematiksel gücünden değil, ölçekten ve birleşimden** gelir.

12. haftanın ölçüm/kanıt disipliniyle bağ

On ikinci hafta, bir güvenlik iddiasının yalnız bir **test sonucuyla** (S16) desteklenmesi gerektiğini, "plan değil **sonuç**" beklendiğini öğretmişti; aynı disiplin burada da geçerlidir. Bu haftaki dört ölçüt tablosu (güç, dayanıklılık, gizlilik, maliyet), S9/S15 belgenizde tek başına bir **iddia** değil, her biri kendi **kanıtına** (ölçüm tablosu, `cmp` çıktısı, sembolik yürütme denemesi) bağlı bir **sonuç** olarak yer almalıdır. 12. haftadaki TOE kimliğine (bölüm 0) bu ölçüm tablosunun **hangi tohum ve hangi dönüşüm hattıyla** üretilen ikiliye ait olduğunu da eklemeyi unutmayın — aksi hâlde bir değerlendirici, hangi ölçümün hangi ikiliyi anlattığını ayırt edemez.

İşlenmiş örnek: gizlilik (stealth) ölçütünü kavramsallaştıralım

1. haftadaki entropi örneğini (bölüm 0: `[0x41,0x41,0x41,0x41]` düşük entropili, `[0x3F,0xA1,0x08,0xC7]` yüksek entropili) hatırlayın. **Gizlilik**, gizlenmiş kodun **normal koddan istatistiksel olarak ayırt edilebilir olup olmadığını** sorar — ve bu ayırt edilebilirlik çoğu zaman tam olarak entropi/yoğunluk farkından kaynaklanır.

Varsayımsal bir örnek: bir ikili dosyayı tarayan bir araç, her 256 baytlık pencerede **dal komutu yoğunluğunu** ölçüyor olsun:

Bölge	256 baytta dal komutu sayısı	Yorum
Sıradan bir yardımcı fonksiyon	~8	Normal bir C fonksiyonu için beklenen aralık
`erisim_ver` (gizlemeden önce)	~6	Normal aralıkta
`erisim_ver` (Flatten+AddOpaque sonrası)	~34	Normal aralığın **kat kat üzerinde** — dikkat çeker

Yorum: gizlenmiş `erisim_ver`'in dal yoğunluğu, dosyadaki **diğer sıradan fonksiyonlardan istatistiksel olarak belirgin biçimde farklıdır**. Bir saldırgan, bütün ikiliyi taramak zorunda kalmadan, yalnız "anormal derecede yoğun dallı bölgeleri" arayarak **doğrudan** `erisim_ver`'e yönelebilir — bu, "gizlilik düşüyor" demektir: güç ve maliyet artarken (bölüm 7'nin ilk tablosu), gizlilik **azalmıştır**.

Sayısal bir örnek: farklı dönüşüm bileşimleri için varsayımsal çözücü süresi

Banescu vd.'nin yönteminin **biçimini** (gerçek sayılarını değil) bir tabloyla gösterelim; aşağıdaki süreler **tamamen varsayımsaldır**, yalnız "daha çok katman = çözücü için daha uzun süre" ilişkisinin biçimini görmek içindir:

Dönüşüm bileşimi	Varsayımsal çözücü süresi (KLEE benzeri bir araçla)
Yok (temiz kod)	Anında (dallanma zaten açık)
Yalnız `AddOpaque` (klasik yüklem)	Saniyeler (9. haftanın `x*(x+1)` örneğindeki gibi)
`Flatten` + `AddOpaque`	Dakikalar (dağıtıcı + opak koşul birlikte çözülmeli)
`EncodeArithmetic` + `Flatten` + `AddOpaque`	Saatler (MBA sadeleştirme + CFG kurtarma + opak yüklem birlikte)
Yukarıdakiler + çeşitlendirme (her kopya farklı tohum)	Her kopya için **yeniden** başlanmalı — toplam saldırgan emeği kopya sayısıyla çarpılır

Son satır, bölüm 6'nın "çeşitlendirme gücü artırmaz, ölçeklenmeyi engeller" kuralının dayanıklılık ölçütü üzerindeki yansımasıdır: bir kopya için harcanan "saatler" mertebesindeki çözücü emeği, **bir sonraki kopya için sıfırdan** harcanmalıdır — çünkü farklı tohum farklı opak yüklem sabitleri üretir.

> ⚠ **Bu tablo gerçek bir ölçüm değildir**
>
> Yukarıdaki süreler yalnız **mertebe** göstermek içindir; gerçek bir sembolik yürütme aracının belirli bir fonksiyona karşı ne kadar süreceği donanıma, aracın kendisine ve dönüşümün tam yapılandırmasına göre büyük ölçüde değişir. S9/S15'inizde bu tür bir tablo kullanacaksanız, sayıları **gerçekten çalıştırıp** ölçün; aksi hâlde "varsayımsal" ibaresini açıkça koyun (bu hafta boyunca yaptığımız gibi).

Sık yapılan hata: 'ne kadar çok gizlersem o kadar iyi' sanmak

Bölüm 5 ve 7'deki büyük yüzdeler (%359 komut artışı, %166,7 karmaşıklık artışı) etkileyici görünebilir, ama bu bölüm gösteriyor ki **aşırı yoğun bir gizleme kendi varlığını ele verebilir**. Gerçek bir üründe yalnızca `erisim_ver`'i değil, çevresindeki bazı **önemsiz** fonksiyonları da hafifçe gizlemek (ya da `RandomFuns` ile sahte fonksiyonlar eklemek — bölüm 3) dal yoğunluğu haritasını **düzleştirip** hedefi gizler. **Kural:** güç ve gizliliği birlikte düşünün; yalnız tek bir fonksiyonu aşırı gizlemek, o fonksiyonu bir harita üzerinde işaretlemek gibidir.

8. Sınıf içi akış: gizle, karşılaştır, ölç, çeşitlendir

Bu haftanın uygulaması, öğrencinin **kendi** küçük programı üzerinde, kendi makinesinde yapılır (Tigress'i resmî siteden kendisi indirir). Akış tümüyle savunma amaçlıdır: kendi kodunuzu koruyup korumanın maliyetini ölçersiniz.

1. **Hazırlık:** Küçük, sentetik bir program yazın (ör. bir `erisim_ver` denetimi). Birim testleriyle doğru çalıştığını gösterin.
2. **Gizle:** Bir dönüşüm hattı uygulayın (EncodeLiterals → EncodeArithmetic → Flatten → AddOpaque).
3. **Davranışı doğrula:** Aynı birim testlerini gizlenmiş sürümde çalıştırın — sonuç **birebir aynı** olmalı. Değilse hat yanlıştır.
4. **Karşılaştır:** Tersine derleyicide (ör. Ghidra/objdump) temiz ve gizli sürümün kontrol akışını karşılaştırın; temel blok sayısındaki artışı not edin.
5. **Ölç:** İkili boyut ve çalışma süresini önce/sonra ölçün.
6. **Çeşitlendir:** Aynı hattı iki farklı tohumla çalıştırın; iki ikilinin farklı olduğunu gösterin.
7. **Raporla:** Bulguları bir tabloya dökün (dönüşüm, boyut, süre, blok sayısı, tohum farkı) — bu tablo doğrudan S9/S15'e girer.

> **Etik ve güvenli kullanım**
>
> Tigress yalnız **kendi** kodunuza uygulanır. Amaç, başkasının programını analiz etmek değil, **kendi** kodunuzu korumayı ve bunun maliyetini ölçmeyi öğrenmektir. Örnek program öğrencinin bilgisayarına zarar vermez; sistem ayarı değiştirmez, ağa bir şey yapmaz. Bütün değerler sentetiktir.

Sık yapılan hata: adım 3'ü (davranışı doğrula) atlayıp doğrudan adım 5'e (ölç) geçmek

Zaman baskısı altında bazı öğrenciler dönüşüm hattını çalıştırır, ikili boyutunu hemen ölçer ve "gizleme işe yaradı" sonucuna atlar — davranış doğrulamasını (adım 3) hiç yapmadan. Bu tehlikelidir çünkü **büyümüş ama bozuk** bir fonksiyon da boyut ölçütünde "başarılı" görünür; asıl soru büyüklük değil, **doğruluğun korunup korunmadığıdır**.

> **Sık yapılan hata: ölçümü davranış doğrulamasından önce yapmak**
>
> Bir dönüşüm hattı davranışı bozmuşsa (ör. `Flatten` yanlış yapılandırılmış, bir `case` eksik), ortaya çıkan ikili genelde **daha büyük ve daha yavaş** olur (bozuk kod genelde gereksiz komut içerir) — yani boyut/hız ölçütüne bakan biri bunu fark etmeyebilir, hatta "iyi gizlenmiş" sanabilir. **Kural:** adım sırası **asla** değişmez: önce **davranış** (adım 3), sonra **ölçüm** (adım 5). Bozuk bir korumayı ölçmenin hiçbir anlamı yoktur.

> **Yedi adımı tek satırda özetleyen kural**
>
> Hazırla → gizle → **doğrula** → karşılaştır → ölç → çeşitlendir → raporla. Bu sıralamadaki her ok, bir öncekinin **çıktısını girdi olarak alır**; bir adımı atlamak zincirin geri kalanını anlamsızlaştırır — tıpkı bölüm 4'teki dönüşüm hattında olduğu gibi (burada "hat" kavramı, teknik dönüşümlerin ötesinde **sınıf içi sürecin kendisine** de uygulanıyor).

İşlenmiş örnek: yedi adımı gerçek bir öğrenci senaryosunda izleyelim (varsayımsal)

Bir öğrencinin kendi küçük "PIN doğrulama" fonksiyonu üzerinde bu yedi adımı nasıl uyguladığını, kısa bir kayıt tablosuyla özetleyelim (sayılar varsayımsaldır, yalnızca sürecin **biçimini** göstermek içindir):

Adım	Öğrencinin yaptığı	Kaydettiği kanıt
1. Hazırlık	`pin_dogrula.c` yazıldı, 4 birim testi eklendi	Test dosyası + "4/4 geçti" çıktısı
2. Gizle	`EncodeLiterals → Flatten → AddOpaque` hattı çalıştırıldı	Kullanılan komut satırı
3. Davranışı doğrula	Aynı 4 test gizli sürüme karşı çalıştırıldı	"4/4 geçti" çıktısı (değişmedi)
4. Karşılaştır	`objdump` ile temiz/gizli `pin_dogrula` karşılaştırıldı	3 dal → 6 dal
5. Ölç	Komut sayısı ve (yaklaşık) süre ölçüldü	Komut: 18 → 34 (%88,9 artış)
6. Çeşitlendir	İki tohumla iki ikili üretildi, `cmp` ile karşılaştırıldı	"farklı" sonucu
7. Raporla	Yukarıdaki altı satır S9/S15'e kopyalandı	Bu tablonun kendisi

Bu tablo biçimi, dönem projenizin S9/S15 bölümüne **doğrudan kopyalanabilir** bir şablondur; yalnız sayıları kendi ölçümlerinizle değiştirmeniz yeterlidir.

İşlenmiş örnek: bir adımın atlanması zinciri nasıl bozar (varsayımsal)

Yedi adımlık akışın "bir adımı atlamak zincirin geri kalanını anlamsızlaştırır" uyarısını bir tabloyla somutlaştıralım:

Atlanan adım	Ne fark edilmeden kalır?	Sonuç
1. Hazırlık (test yok)	Davranışın **hiç** doğrulanabilir bir referansı yok	Adım 3 anlamsızlaşır: neye göre "aynı" denecek?
3. Davranışı doğrula	Bozuk bir gizleme fark edilmez	Adım 5'teki ölçüm, **bozuk** bir programı "başarılı" gösterir (yukarıdaki "sık yapılan hata")
5. Ölç	Maliyet bilinmez	S9/S15'te "kırılamaz" gibi ölçülmemiş iddialar ortaya çıkar (9. hafta bölüm 1)
6. Çeşitlendir	Tek bir ikili herkese dağıtılır	Bölüm 6'daki "uzayda çeşitlendirme" örneğindeki %100 etki senaryosu gerçekleşir
7. Raporla	Yapılanlar belgelenmez	Bir değerlendirici (12. hafta) hiçbir şeyi doğrulayamaz; iddia kanıtsız kalır

Bu tablo, yedi adımın **hiçbirinin isteğe bağlı olmadığını** gösterir: her satır, bir öncekinin atlanmasının **hangi sonraki adımı** anlamsızlaştırdığını işaret eder — tıpkı bölüm 4'teki dönüşüm hattında olduğu gibi, burada da "hat" fikri sürecin kendisine uygulanır.

9. Gizlemeyi derleme ve dağıtım hattına yerleştirmek (S15)

Gizleme, "en sonda elle yapılan bir iş" değil, **derleme hattının** bir adımı olmalıdır. Vize sonrası projenizin S15 bölümü tam olarak bunu belgeler:

- **Yalnız hassas fonksiyonlar** gizlenir (`--Functions`); gerekçesi yazılır.
- Her sürüm bir **tohumla** çeşitlendirilir; tohum ve sürüm kimliği kaydedilir.
- Gizlemeden **sonra** imzalama yapılır; sürüm kimliği ve özet değerleri (12. haftadaki TOE kimliği) tutarlı olur.
- Hat, sürekli entegrasyonda (CI) çalışır; her sürümde birim testleri + boyut/hız ölçümü otomatik koşar.

9, 11 ve 14. haftalar birlikte

Bu üç hafta bir bütündür: 9. hafta gizleme **kurallarını**, 11. hafta anahtar için **whitebox** sınırını, 14. hafta bunların **otomatik ve çeşitlendirilmiş** uygulamasını verir. Üçünün ortak kuralı aynıdır: gizleme kırılamazlık vermez, maliyet yükseltir; gücü katmanlı savunmadan (RASP, anahtar yenileme, sunucu denetimi) ve ölçülmüş olmaktan gelir.

S15 hattını adım adım okuyalım

Görseldeki (h14-07) dört kutuyu (gizleme, imzalama, sürüm kimliği, CI) sıradaki bir listeye açalım — her adım bir öncekinin **girdisidir**, tıpkı bölüm 4'teki dönüşüm hattında olduğu gibi:

1. **Derle (temiz):** `temiz.c` (okunur kaynak) normal derlenir; birim testleri bu sürüme karşı çalıştırılır.
2. **Gizle:** yalnız `--Functions` ile işaretlenmiş **hassas** fonksiyonlara, kararlaştırılmış dönüşüm hattı (bölüm 4) uygulanır; **yeni bir** `--Seed` üretilir (bölüm 6'nın "tohumu her zaman kaydedin" kuralı burada devreye girer — CI, kullandığı tohumu bir günlüğe/artefakta yazmalıdır).
3. **Yeniden derle (gizli):** `gizli.c` normal derlenir; **aynı** birim testleri bu sürüme karşı da çalıştırılır (bölüm 2'deki ADIM 3'ün CI'deki karşılığı — davranış değişmediyse testler geçer, değiştiyse CI **başarısız** olmalıdır).
4. **Ölç:** boyut/komut/dal (bölüm 5, 7) otomatik ölçülür ve bir önceki sürümle karşılaştırılır; ani ve açıklanamayan bir sapma (ör. beklenmedik biçimde küçülme) bir hataya işaret edebilir.
5. **İmzala:** yalnız test + ölçüm adımlarını **geçen** ikili imzalanır — bölüm 4'teki "imza en son gelir" kuralının CI'deki karşılığı.
6. **Sürüm kimliğini kaydet:** kullanılan tohum, dönüşüm hattı, özet (hash) değeri ve imza birlikte bir kayda (S15 belgesine) yazılır — bu, 12. haftadaki **TOE kimliği** (sürüm + ikili + kaynak + özet) tanımının doğrudan uygulamasıdır; farkı, burada dört bileşene **beşinci** bir alan (kullanılan tohum) eklenmesidir, çünkü aynı kaynak+sürüm etiketi farklı tohumlarla **kasıtlı olarak farklı** ikililer üretebilir (bölüm 0'daki "Sürüm kimliği ve özet değeri" tanımını hatırlayın).

Bu altı adımı bir CI betiğinin **kavramsal** iskeleti olarak yazalım (belirli bir CI ürününün sözdizimi değil, adımların sırasını gösteren sözde kod):

Kavramsal CI iskeleti (belirli bir araca özgü sözdizimi değil)

```
asama derle_temiz:
    cc -O2 -o cikti/temiz temiz.c
    calistir_birim_testleri cikti/temiz

asama gizle:
    tohum = yeni_rastgele_tohum()
    tigress --Seed=$tohum --Transform=... --Functions=... --out=gizli.c temiz.c
    kaydet tohum -> gunluk/surum-$SURUM.txt

asama derle_gizli:
    cc -O2 -o cikti/gizli gizli.c
    calistir_birim_testleri cikti/gizli   # basarisizsa CI DURUR

asama olc:
    objdump -d cikti/gizli | olc_komut_dal erisim_ver >> gunluk/surum-$SURUM.txt

asama imzala:
    imzala cikti/gizli -> cikti/gizli.imzali

asama kaydet_surum:
    ozet = sha256(cikti/gizli.imzali)
    kaydet SURUM, tohum, ozet, donusum_hatti -> TOE-kayit-$SURUM.txt
```

Bu iskeletin her satırı, bu haftanın bir bölümüne karşılık gelir: `derle_temiz` → bölüm 2 ADIM 1; `gizle` → bölüm 4 (hat) + bölüm 6 (tohum); `derle_gizli`nin test adımı → bölüm 2 ADIM 3; `olc` → bölüm 0/5/7; `imzala` → bu bölümün "imza en son" kuralı; `kaydet_surum` → 12. haftadaki TOE kimliği.

Sık yapılan hata: tohumu yalnızca geliştiricinin kendi makinesinde, CI dışında tutmak

Bir geliştirici gizlemeyi kendi bilgisayarında elle çalıştırıp `--Seed=42` gibi sabit bir değeri **alışkanlıkla** her seferinde kullanırsa, iki şey bozulur: (1) bu artık **çeşitlendirme değildir** (bölüm 6'daki "sık yapılan hata" kutusuyla aynı sorun — sabit bir değer, tek bir öngörülebilir varyant üretir), (2) CI'nin ürettiği resmî sürümle geliştiricinin masaüstünde ürettiği sürüm **farklı ikili dosyalar** olabilir; hangisinin test edildiği belirsizleşir. **Kural:** tohum **yalnızca CI içinde**, her çalıştırmada yeni ve **CI'nin kendi günlüğüne kaydedilerek** üretilir; hiçbir gizlenmiş ikili bir geliştiricinin kişisel makinesinden elle dağıtılmaz.

İşlenmiş örnek: üç sürümlük bir TOE kaydı

Bölüm 0 ve bu bölümdeki "sürüm kimliği + tohum" fikrini, üç ardışık sürüm için doldurulmuş bir kayıt tablosuyla somutlaştıralım (varsayımsal):

Sürüm	Tohum	Dönüşüm hattı	Özet (SHA-256, kısaltılmış)	İmza duru
v1.0	1001	EncodeLiterals→EncodeArithmetic→Flatten→AddOpaque	`a1b2c3...`	Geçerli
v1.1	4832	Aynı hat, yeni tohum	`f9e8d7...`	Geçerli
v1.2	9214	Aynı hat + `RandomFuns` eklendi	`4c5b6a...`	Geçerli

Bu tablo, bölüm 6'daki "zamanda çeşitlendirme" örneğinin **S15 belgesindeki karşılığıdır**: her satır, o sürümün **tam olarak hangi ikili** olduğunu (tohum + hat + özet ile) ve o ikilinin **imzalı ve test edilmiş** olduğunu kanıtlar. v1.2'deki "hat değişikliği" (`RandomFuns` eklenmesi) da kayda geçmiştir — bu, 12. haftadaki "delta değerlendirme" (yalnız değişen kısmın yeniden değerlendirilmesi) için gereken bilgidir: bir değerlendirici v1.1'den v1.2'ye ne değiştiğini bu tablodan **doğrudan** okuyabilir, kaynak kodu satır satır karşılaştırmak zorunda kalmaz.

10. Dönem projesi: bu hafta (S9 ileri + S15 hat)

1. **Bir dönüşüm hattı** seçin ve S9'a yazın: hangi fonksiyonlar, hangi dönüşümler, hangi sırayla, **neden**.
2. **Ölçüm tablosu:** boyut, süre ve (varsa) blok sayısı — önce/sonra.
3. **Çeşitlendirme:** iki tohumla üretim ve fark ölçütü; uygulamayacaksanız gerekçesi.
4. **S15 hattı:** gizleme + imzalama + sürüm kimliği adımlarını bir diyagramla belgeleyin.
5. **Davranış kanıtı:** gizlenmiş sürümün birim testlerini geçtiğini gösterin (S16 ile bağ).

Kontrol listesi: bu haftanın teslimi neyi içermeli?

- [ ] Seçilen dönüşüm hattının **her bir adımı** ve **neden o sırada** olduğu yazılı mı? (bölüm 4)
- [ ] Her adımdan sonra **davranış doğrulaması** (birim testi sonucu) belgelendi mi? (bölüm 2, ADIM 3)
- [ ] Ölçüm tablosu **en az** komut/dal sayısı içeriyor mu; mümkünse boyut ve süre de eklendi mi? (bölüm 7)
- [ ] Çeşitlendirme uygulandıysa iki tohumla üretilen ikilinin **farklı olduğu kanıtı** (ör. `cmp` çıktısı) var mı? Uygulanmadıysa **gerekçesi** yazıldı mı? (bölüm 6)
- [ ] S15 diyagramında gizleme, imzalama ve sürüm kimliği adımlarının **sırası** doğru mu (bölüm 9)?
- [ ] "Kırılamaz" gibi bir ifade **kullanılmadı** mı; bütün iddialar ölçülebilir cümlelerle mi yazıldı? (9. hafta bölüm 1)

Doldurulmuş bir örnek: S9/S15 için "KURAL" şablonu

Dokuzuncu haftanın bölüm 4'ü, her koruma kararını "Neyi korur / Nasıl / Maliyet / Sınır" şablonuyla yazmayı öğretmişti. Bu haftanın Tigress hattı için aynı şablonu dolduralım:

KURAL K-04+K-01+K-07-Tigress-uygulama: erisim_ver fonksiyonuna otomatik dönüşüm hattı

```
Neyi korur?       : Sentetik erişim denetiminin mantığını (hangi jetonun geçerli sayıldığı)
                    tersine mühendisliğe karşı geciktirir.
Nasıl?            : Tigress --Transform=EncodeLiterals,EncodeArithmetic,Flatten,InitOpaque,AddOpaque
                    (bölüm 2 ADIM 2'deki tam komut), --Seed her sürümde yeni üretilir (bölüm 6).
Maliyet           : ~%359,1 komut artışı, ~%166,7 karmaşıklık (güç) artışı (bölüm 7, varsayımsal
                    demo sayılarıyla; gerçek sayılar `tigress-hatti.sh` ile ölçülür).
Sınır             : Dayanıklılık ve gizlilik bu demoda ölçülmedi (bölüm 7); tek başına Virtualize
                    olmadan whitebox düzeyinde bir koruma iddia edilmez (11. hafta ile
karşılaştırın).
```

Bu doldurulmuş örneği kendi projenizde **kendi fonksiyonunuz, kendi ölçtüğünüz sayılarla** tekrarlayın; şablonun dört satırının hiçbiri boş bırakılmamalıdır.

Örnek doldurulmuş bir mini-rapor (S9/S15 için başlangıç şablonu)

Bu haftanın bütün parçalarını (dönüşüm hattı, ölçüm, çeşitlendirme, S15 hattı) tek bir örnek raporda birleştirelim. Aşağıdaki metin, sayıları kendi ölçümlerinizle değiştirerek **doğrudan** S9/S15 bölümünüze taşıyabileceğiniz bir başlangıç noktasıdır (bütün sayılar bu haftanın varsayımsal örnekleriyle tutarlıdır):

Örnek S9/S15 girişi (varsayımsal sayılarla; kendi ölçümünüzle değiştirin)

```
Fonksiyon         : erisim_ver
Neden hassas?     : Sentetik jeton denetimi (bölüm 1'deki karar akışı: hassas + orta değerli)
Dönüşüm hattı     : EncodeLiterals -> EncodeArithmetic -> Flatten -> InitOpaque(main) -> AddOpaque
Tohum (surum 1.0) : 1001
Davranış kanıtı   : Temiz ve gizli sürüm CEN429-OK / yanlis girdileriyle aynı çıktıyı verdi (ADIM 3:
ayni cikti)
Maliyet           : Komut 22->101 (%359,1), dal 2->7, karmaşıklık (güç) 3->8 (%166,7)
Çeşitlendirme     : Seed=1001 ve Seed=2002 ile üretilen ikililer %60 bayt farkı gösterdi (cmp:
farklı)
Dayanıklılık      : Ölçülmedi bu turda; bir sonraki iterasyonda KLEE ile sınanacak (bölüm 7)
Gizlilik          : Ölçülmedi bu turda; dal yoğunluğu haritası çıkarılacak (bölüm 7)
S15 durumu        : CI'de gizle->test->ölç->imzala->kaydet adımları tanımlı; sürüm kaydı TOE-kayit-
1.0.txt
Sınır             : Tek başına whitebox düzeyinde koruma iddia edilmiyor (11. hafta ile
karşılaştırın)
```

Bu şablonun her satırı, bu haftanın bir bölümüne karşılık gelir (yukarıdaki "Doldurulmuş bir örnek: KURAL şablonu" bölümüyle birlikte okuyun): **Fonksiyon/Neden hassas** → bölüm 1; **Dönüşüm hattı/Tohum** → bölüm 4, 6, 9; **Davranış kanıtı** → bölüm 2 ADIM 3; **Maliyet** → bölüm 5, 7; **Çeşitlendirme** → bölüm 6; **Dayanıklılık/Gizlilik** → bölüm 7; **S15 durumu** → bölüm 9; **Sınır** → bölüm 1, 9, 11.

Sık yapılan hata: 'Dayanıklılık: yüksek' gibi ölçülmemiş bir değer yazmak

Yukarıdaki örnekte "Dayanıklılık" ve "Gizlilik" satırlarının **'ölçülmedi'** yazdığına dikkat edin — bu bilinçli bir tercihtir. Bir öğrenci bu satırlara "yüksek" ya da "güçlü" gibi öznel bir değer yazma eğiliminde olabilir; bu, bölüm 7'deki "dördünü birden raporlayın" kuralının **ihlalidir**. **Kural:** ölçülmemiş bir ölçütü boş bırakmak ya da "ölçülmedi, şu yöntemle ölçülebilir" yazmak, uydurma bir değer yazmaktan her zaman daha doğrudur — hem akademik olarak hem de bir değerlendirici karşısında.

Haftanın büyük resmi: bölümler nasıl birbirine bağlanıyor?

Çok fazla yeni terim ve örnek gördükten sonra, hepsini **tek bir zincir** olarak görmek zorlaşabilir. Aşağıdaki tablo bu haftanın on bir bölümünü sırayla özetler; her satır bir öncekine **neyi eklediğini** anlatır:

Bölüm	Bir öncekine eklediği
0	Terimler (kaynaktan kaynağa, dönüşüm, tohum, `objdump`, CI, TOE)
1	Terimlerle "neden bir araç, el ile değil" sorusunu yanıtlar
2	Aracın (Tigress) gerçek komut satırını, adım adım
3	Komut satırındaki her bayrağı 9. haftanın K kurallarına bağlar
4	Birden çok bayrağı bir **hatta** (sıralı dönüşüm dizisi) birleştirir
5	Hattı tek bir fonksiyon üzerinde uçtan uca, sayılarla izler
6	Aynı hattı farklı tohumlarla çalıştırıp **çeşitlendirir**
7	Bölüm 5 ve 6'nın sayılarını dört ölçüte (güç/dayanıklılık/gizlilik/maliyet) dağıtıp **ölçer**
8	Bölüm 2–7'yi bir sınıf içi **yedi adımlık süreç** olarak sıralar
9	Bölüm 8'in sürecini bir **CI hattına** (otomasyon) taşır
10	Her şeyi bir **dönem projesi teslimine** (S9/S15) dönüştürür

Bu tablo, sınav ya da proje hazırlığında "hangi bölümü unuttum?" sorusuna hızlı bir cevap verir: bir bölümün bir öncekine **ne eklediğini** anlatamıyorsanız, o bölümü yeniden okuyun; zincirin bir halkası eksikse geri kalanı da anlamını yitirir (tıpkı bölüm 4'teki dönüşüm hattında olduğu gibi).

11. Kendini sınama

1. Kaynaktan kaynağa gizleme nedir? El ile gizlemeye göre üç avantajı nedir?

Kaynak koda dönüşüm uygulanıp **yeni kaynak kod (yine C)** üretilir, sonra normal derlenir. Avantajları: **tekrarlanabilir/otomatik** (her derlemede), **tutarlı ve ölçekli** (çok fonksiyona), **orijinal kaynak temiz kalır** (bakım kolay) — ayrıca tohumla çeşitlendirme.

2. Tigress'in dersteki yeri nedir; hangi lisansla, neden derste ikilisi dağıtılmaz?

Tigress kaynaktan kaynağa çalışan bir C gizleyicidir; akademik/ücretsiz olsa da kendi lisans/kullanım koşullarıyla dağıtılır, bu yüzden **herkes kendisi indirir**, derste ikilisi paylaşılmaz. Demolar Tigress yoksa temiz türev/fallback ile de çalışacak biçimde yazılmıştır.

3. Şu dönüşümleri 9. hafta kurallarıyla eşleyin: Flatten, EncodeArithmetic, EncodeLiterals, Virtualize, AddOpaque.

Flatten = kontrol akışı düzleştirme; **EncodeArithmetic** = aritmetik gizleme; **EncodeLiterals** = sabit/dize gizleme; **Virtualize** = sanallaştırma (en güçlü, en pahalı); **AddOpaque** = opak yüklem/ölü dal ekleme.

4. Bir dönüşüm hattı neden tek bir dönüşümden güçlüdür? Sıra neden önemlidir?

Farklı dönüşümler farklı analiz tekniklerine karşı korur; birlikte **katmanlı direnç** oluştururlar. Sıra önemlidir: örn. önce düzleştirip sonra opak yüklem eklemek gerekir; yanlış sıra bazı dönüşümleri etkisizleştirebilir veya maliyeti gereksiz yere artırabilir.

5. Her dönüşüm hattından sonra hangi iki şeyi mutlaka yaparsınız? (İpucu: davranış + maliyet)

(1) **Davranışın değişmediğini test edersiniz** (birim testleri geçmeli — gizleme işlevi bozmamalı), (2) **maliyeti ölçersiniz** (boyut/hız/komut sayısı; önce ve sonra karşılaştırılır).

6. Virtualize neden yalnız küçük ve kritik fonksiyonlara uygulanır?

Sanallaştırma büyük **performans** ve **boyut** cezası getirir; her yere uygulanamaz. Bu yüzden yalnız değeri yüksek, küçük ve kritik fonksiyonlarda — maliyetin haklı çıktığı yerlerde — kullanılır.

7. Tohum (`--Seed`) çeşitlendirmeyi nasıl sağlar? Uzayda ve zamanda çeşitlendirme farkı nedir?

Aynı kaynak farklı **tohumla** gizlendiğinde araç farklı rastgele seçimler yapar → farklı ikili (aynı davranış). **Uzayda çeşitlendirme:** farklı kopyalar/kullanıcılar farklıdır (bir kırık herkesi kırmaz). **Zamanda çeşitlendirme:** sürümler arası değişir (bir kırık kalıcı olmaz).

8. Çeşitlendirme gizlemenin gücünü mü, ölçeklenmesini mi engeller?

Ölçeklenmesini engeller: tek bir kopyanın gücünü artırmaz ama bir saldırının tüm kopyalara/kullanıcılara yayılmasını (yeniden kullanımını) engeller.

9. Banescu vd. çalışmasının ders için ana kuralı nedir? (Dayanıklılık ↔ maliyet)

Daha güçlü dönüşüm daha çok **dayanıklılık** verir ama **performans/boyut maliyeti** de artar. Koruma, varlığın değeriyle orantılı seçilir; her şey en güçlü dönüşümle gizlenmez.

10. Gizlemeyi derleme hattına koyarken imzalama neden gizlemeden sonra gelir?

İmza **son ikiliyi** korur. Önce imzalanıp sonra gizlenirse ikili değişir ve imza geçersizleşir. Doğru sıra: **derle → gizle → (paketle) → imzala**.

11. S15'te 'yalnız hassas fonksiyonlar gizlenir' kuralının gerekçesi nedir?

Bütünü gizlemek performans/boyutu patlatır ve kodun çoğu zaten hassas değildir. Korumayı **kritik** fonksiyonlara yoğunlaştırmak maliyeti düşürür ve etkiyi artırır.

12. 9, 11 ve 14. haftaların ortak kuralını bir cümleyle yazın.

Koruma kırılamazlık değil **gecikme** sağlar; gücü **birliktelikten**, **çeşitlendirmeden** ve **ölçülmüş** olmaktan gelir.

13. objdump ile ölçülen 'komut sayısı' ve 'dal sayısı' neyi temsil eder; 9. haftadaki CFG düğüm/kenar sayımıyla ilişkisi nedir?

`objdump -d`, bir ikili dosyayı assembly'ye çevirir; hedef fonksiyonun etiketinden (`<erisim_ver>:`) sonraki her satır **komut sayısını**, bunlardan `j...` / `call` ile başlayanlar **dal/çağrı sayısını** verir. Bu, 9. haftada kaynak koddan elle çizilen CFG düğüm/kenar sayımının **derlenmiş ikili üzerindeki otomatik, daha ince taneli** karşılığıdır; ikisi de "bu fonksiyonu analiz etmek ne kadar iş?" sorusunu ölçer.

14. `InitOpaque` ve `AddOpaque` neden ayrı iki dönüşümdür; hangisi önce çalıştırılmalıdır?

`InitOpaque`, opak yüklemlerin dayanacağı gizli durum değişkenlerini **hazırlar** (genelde `main` içinde); `AddOpaque` bu durumu kullanarak hedef fonksiyona gerçek opak koşulları **ekler**. `InitOpaque` atlanırsa `AddOpaque`'in dayanacağı altyapı eksik kalır; bu yüzden sıra **önce `InitOpaque`, sonra `AddOpaque`**'dır (demo betiğindeki gerçek sıra budur).

15. Sıra 1 (EncodeLiterals→EncodeArithmetic→Flatten→AddOpaque) ile Sıra 2 (Flatten→AddOpaque→EncodeArithmetic→EncodeLiterals) arasındaki temel kapsam farkı nedir?

Sıra 1'de `EncodeLiterals` ilk çalıştığı için yalnız **orijinal kaynaktaki** sabitleri kodlar; `Flatten`'in sonradan ürettiği durum sabitlerini kapsamaz. Sıra 2'de `EncodeLiterals` **son** çalıştığı için `Flatten` ve `AddOpaque`'in ürettiği yeni sabitleri de kodlayabilir, ama `EncodeArithmetic` artık dağıtıcıya yayılmış aritmetiği işlemek zorunda kalır ve maliyeti büyütebilir. Kesin "daha iyi" bir sıra yoktur; karar ölçümle (bölüm 7) verilir.

16. Bölüm 5'teki tabloda adım artış yüzdelerini (%36,4 + %53,3 gibi) toplayarak kümülatif artışı hesaplamak neden yanlıştır?

Yüzdeler farklı paydalara (farklı bir önceki adımın mutlak sayısına) dayandığı için doğrudan toplanamaz. Doğru yöntem, kümülatif artışı **her zaman adım 0'ın mutlak sayısı üzerinden** yeniden hesaplamaktır: `(46-22)/22×100 ≈ %109,1` doğrudur; `%36,4+%53,3=%89,7` toplamı yanlıştır.

17. Bölüm 7'deki güç (potency) ölçütü hangi formülle hesaplanır; bu haftaki demo sayılarıyla sonucu nedir?

Çevrimsel karmaşıklık ≈ `dal_sayısı + 1`. Adım 0 (temiz): `2+1=3`. Adım 4 (tam hat): `7+1=8`. Artış: `(8-3)/3×100 ≈ %166,7`.

18. Küçük bir demoda bütün ikili dosyanın boyutunun (`ls -l`) neredeyse değişmemesi neden bir hata değildir; doğru ölçüm nasıl yapılır?

Küçük programlarda dosyanın büyük kısmı sabit başlıklar, dinamik bağlama tabloları ve kütüphane referanslarından oluşur; tek bir fonksiyonun birkaç on komut büyümesi bu sabit yükün yanında görünmez kalır. Anlamlı ölçüm için **bütün dosya değil, hedef fonksiyonun kendi kod boyutu** (`objdump` komut sayımı üzerinden) ölçülmelidir.

19. S15 hattında TOE kimliğine (12. hafta: sürüm+ikili+kaynak+özet) neden bir de 'tohum' alanı eklenir?

Çünkü bu hafta öğrendiğimiz çeşitlendirme, aynı kaynak ve aynı sürüm etiketiyle **kasıtlı olarak farklı** ikililer üretebilir. Hangi ikilinin hangi tohumla üretildiğini ayırt etmek için tohum da TOE kaydına eklenmelidir; aksi hâlde aynı sürüm etiketi altında birbirinden farklı ikililer karışabilir.

20. Çeşitlendirme örneğinde 96/160 bayt farklıysa fark oranı nedir; bu oran tek başına neyi kanıtlamaz?

`96/160×100=%60`. Bu oran yalnız iki ikilinin ne kadar farklı olduğunu gösterir; farkın **hedef fonksiyonun içinde mi** yoksa zaman damgası gibi **önemsiz** bir yerde mi olduğunu göstermez — bunun için hedef fonksiyon `objdump` ile ayrıca karşılaştırılmalıdır (`cmp` tek başına yetmez).

21. CI içine gizleme adımı koymanın, elle çalıştırmaya göre iki temel avantajı nedir?

(1) Her sürümde tohum otomatik ve yeni üretilir; unutulmaz, sabitlenmez (gerçek çeşitlendirme sağlanır). (2) CI'nin ürettiği resmî ikili ile test edilen ikili aynı olur; bir geliştiricinin masaüstünde elle ürettiği farklı bir ikiliyle karışıklık oluşmaz.

22. Demo betiğinde ADIM 3 'FARK' derse, ADIM 4'e (çeşitlendirme) geçmek neden yanlıştır?

ADIM 3, davranışın korunduğunu doğrular. 'FARK' çıkması, dönüşüm hattının davranışı bozduğu anlamına gelir; bozuk bir ikiliyi çeşitlendirmek yalnızca bozuk davranışın **farklı kopyalarını** üretir, sorunu çözmez. Önce hat düzeltilmeli, davranış yeniden doğrulanmalı, ancak sonra çeşitlendirilmelidir.

23. Virtualize'ı 'madem gizliyorum, hepsini ekleyeyim' mantığıyla otomatik eklemek neden bir hatadır?

Virtualize, diğer dönüşümlerin toplamından **kat kat** (onlarca kat) daha pahalıdır; yüzdeyle değil **kat** ile ölçülür. Bir fonksiyon için %359 gibi bir büyüme kabul edilebilirken üstüne 10-50 kat yavaşlama eklemek çoğu ürün için anlamsızdır; karar bölüm 1'deki "hassas mı, değeri ne kadar" akışıyla **ayrı** verilmelidir.

24. Bu haftanın demo betiği Tigress kurulu değilken neden hata vermez, bunun yerine ne yapar?

Betik önce `command -v tigress` ile kontrol eder; kurulu değilse kurulum/lisans yönergesini (`tigress.wtf`, akademik kullanım ücretsiz) basar ve öğrenciyi 9. haftanın el ile çalışan demosuna (`week-09/01-manuel-gizleme`) yönlendirir — ders akışı durmaz, aynı kavramın el ile çalışan yoluna geçilir.

25. Uzayda çeşitlendirme örneğinde (10.000 kullanıcı), tek bir tohumdan her kullanıcıya benzersiz tohuma geçmek bir kırığın etki oranını nasıl değiştirir?

Tek tohumda bir kırık kullanıcıların **%100'ünü** etkiler. Her kullanıcıya benzersiz tohum verildiğinde bir kırık yalnız **o tek kopyayı** etkiler: `1/10.000×100=%0,01`. Sayılar varsayımsaldır ama ilişki (tohum sayısı arttıkça yayılma yüzeyi küçülür) geneldir.

26. Zamanda çeşitlendirme neden tek başına yeterli bir savunma değildir?

Bölüm 1'in "Kural 3"ü gizlemenin diğer katmanlara yalnız **zaman kazandırdığını** söyler, kalıcı çözüm olduğunu değil. Bir sonraki sürüm çıkana kadar geçen sürede saldırgan hâlâ eski sürümü kullanan kullanıcılara zarar verebilir; bu yüzden zamanda çeşitlendirme RASP ve sunucu tarafı denetimlerle (eski/kırık sürümleri reddetme) **birlikte** kullanılmalıdır.

27. Gizlilik (stealth) ölçütü neden 'ne kadar çok gizlersem o kadar iyi' mantığıyla çelişir?

Aşırı yoğun bir gizleme (ör. anormal derecede yüksek dal yoğunluğu) korunan fonksiyonu dosyadaki diğer fonksiyonlardan **istatistiksel olarak ayırt edilebilir** kılar; bu da bir saldırganın doğrudan o fonksiyona yönelmesini kolaylaştırır. Güç ve maliyet artarken gizlilik **azalabilir** — dördü birlikte dengelenmelidir.

28. `Flatten` uygulanmış bir fonksiyonda durum sabitlerinin (7341, 2098, ... gibi) neden ardışık değil, dağınık seçildiğini açıklayın.

Ardışık sabitler (`1, 2, 3, 4` gibi) bir tersine derleyicide anında "bu bir düzleştirilmiş dağıtıcı" imzası olarak tanınır (kalıp tanıma — bölüm 4/9. hafta bölüm 10). Dağınık/öngörülemez değerler bu tanımayı zorlaştırır; ayrıca bu değerlerin kendisi K-02 (aritmetik kodlama) ile de kodlanabilir, böylece iki katman birleşir.

29. S9/S15'teki 'KURAL' şablonunun dört satırı nelerdir? Bu hafta bunları `erisim_ver` için nasıl doldurduk?

Şablon dört satırdır: **Neyi korur / Nasıl / Maliyet / Sınır**. Bölüm 10'daki doldurulmuş örnekte: neyi korur → erişim denetiminin mantığı; nasıl → beş dönüşümlük Tigress hattı + tohum; maliyet → varsayımsal %359,1 komut / %166,7 karmaşıklık artışı; sınır → dayanıklılık/gizlilik ölçülmedi, tek başına whitebox düzeyinde koruma iddia edilmez.

30. Bu haftanın 'imza en son gelir' kuralı ile 9. haftanın 'gizleme anahtar saklamaz' kuralı çelişir mi?

Hayır, farklı katmanları anlatırlar. 'İmza en son gelir' derleme hattındaki **sıra** kuralıdır: gizlenmiş ikili değişmeden imzalanmalıdır (bölüm 4, bölüm 9). 'Anahtar saklamaz' ise gizlemenin **matematiksel bir gizlilik iddiası** taşımadığını anlatır (K-07 bir kodlamadır, şifreleme değil — 9. hafta bölüm 0). İkisi de "gizleme kırılamazlık değil disiplin ve ölçülmüş maliyet sağlar" temasının farklı yüzleridir.

31. `EncodeData` bir değişkeni nasıl korur; bu korumayı boşa çıkaran sık yapılan hata nedir?

Bir değişkenin bellekteki temsilini parçalar/kodlar (ör. tek bir 32-bit tam sayı yerine iki bayt parçası, bir yapı içinde); bir bellek dökümünde değer **doğrudan** görünmez. Sık hata: parçaları kullanımdan hemen sonra tekrar birleştirip fonksiyon boyunca tek bir değişkende saklamak — bu, yeniden tek, tanınabilir bir blok yaratıp korumayı boşa çıkarır.

32. Yalnız `EncodeLiterals` uygulamakla dört dönüşümü birlikte uygulamak arasındaki fark yalnızca maliyet mi?

Hayır. Yalnız `EncodeLiterals`, saldırganın yalnız **bir** tekniğini (`strings` taraması) etkisiz kılar; kontrol akışı hâlâ doğrudan okunabilir. Dört dönüşümü birlikte uygulamak, saldırganın **aynı anda birden fazla** farklı analiz tekniğini (dize taraması + kalıp tanıma + MBA sadeleştirme + opak yüklem çözme) yenmesini gerektirir — asıl kazanç yalnız maliyetten değil, **kaç farklı tekniğin zorlandığından** gelir.

33. TOE kaydı tablosunda v1.1'den v1.2'ye dönüşüm hattı sütununda bir değişiklik (`RandomFuns` eklenmesi) neden ayrıca kaydedilir?

1. haftadaki delta değerlendirme, yalnız değişen kısmın yeniden değerlendirilmesini ister; bir değerlendirici bu sütun sayesinde iki sürüm arasında **neyin** değiştiğini kaynak kodu satır satır karşılaştırmadan görebilir.

34. İmzalamayı gizlemeden önce yapıp sonra 'yalnız imzayı yeniden hesaplarım' diye düzeltmek neden yeterli bir çözüm değildir?

Teknik olarak yeni içerik yeni imzayla eşleşir, ama **süreç** hâlâ hatalıdır: doğru sıra insan hafızasına bırakılmış olur ve hata tekrarlanabilir. Kural, imzalamanın CI'de **yapısal olarak** gizleme ve ölçüm adımlarından sonra tanımlanmasıdır, böylece sıra hatası mümkün olmaz.

35. Dayanıklılık tablosundaki (varsayımsal) çözücü sürelerinin son satırı ('+ çeşitlendirme') neden özellikle önemlidir?

Gösterir ki çeşitlendirme, tek bir kopya için harcanan saldırgan emeğini (saatler mertebesinde) yok etmez, ama her yeni kopya için o emeği **sıfırdan** gerektirir — bölüm 6'nın "gücü artırmaz, ölçeklenmeyi engeller" kuralının dayanıklılık ölçütündeki karşılığıdır.

36. `Split` dönüşümü bir tersine mühendisin 'haritasını' nasıl bozar?

Bir tersine mühendis genelde fonksiyon sınırlarına (isim, parametre sayısı, giriş/çıkış) bakarak bir harita çıkarır; `Split` tek bir mantıksal işlemi birkaç küçük, jenerik adlı fonksiyona (ör. `_adim_a`, `_adim_b`) dağıtarak bu haritayı yanıltıcı hâle getirir — kaç fonksiyonun aslında **tek** bir işlemin parçası olduğunu anlamak ek çaba gerektirir.

37. `cmp -s A B` ile `objdump` tabanlı hedef fonksiyon karşılaştırması arasındaki fark nedir; hangisi çeşitlendirmeyi gerçekten kanıtlar?

`cmp` yalnız iki dosyanın **bütünüyle** aynı/farklı olduğunu söyler, neyin farklı olduğunu göstermez. `objdump` ile yalnız hedef fonksiyonun (`erisim_ver`) bölümünü ayıklayıp `diff` almak, farkın gerçekten **korunan fonksiyonun içinde** olup olmadığını gösterir; yalnız bu ikinci yöntem çeşitlendirmeyi gerçekten kanıtlar.

38. Bir yıllık bakım senaryosunda 'unutma riski' neden el ile yöntemde daha yüksektir?

El ile yöntemde her yeni hassas fonksiyon için ayrı, elle yazılan bir gizleme adımı gerekir; bu adım kod incelemesinde açıkça görünmeyebilir ve unutulabilir. Araçla yöntemde `--Functions` listesi denetlenebilir bir belgedir, kod incelemesinde doğrudan görülür ve eksiklik fark edilmesi kolaylaşır.

39. `--Functions=erisim_ver,anahtar_turet` gibi bir liste kullanmak her zaman doğru mudur? Ne zaman yanlış olabilir?

Yalnız iki fonksiyon **aynı değerde/hassaslıkta** ve aynı dönüşüm hattını hak ediyorsa doğrudur. Farklı değerdeki fonksiyonlara (ör. biri Virtualize gerektiren kritik, diğeri hafif) aynı listeyle aynı hattı uygulamak, bölüm 1'deki karar akışını (hassas mı, değeri ne kadar) atlamak olur.

40. Örnek mini-raporda 'Dayanıklılık: ölçülmedi' yazmak neden 'Dayanıklılık: yüksek' yazmaktan daha iyi bir mühendislik pratiğidir?

'Yüksek' öznel ve kanıtsız bir iddiadır (9. haftanın 'kırılamaz' uyarısıyla aynı kategoride bir hata). 'Ölçülmedi' dürüst bir durumdur; hangi yöntemle ölçülebileceğini de belirtirse (ör. KLEE ile), bir değerlendiricinin bir sonraki adımı görmesini sağlar — bu, 12. haftanın 'plan değil sonuç' disipliniyle tutarlıdır.

41. `cmp` ile `diff` arasındaki temel fark nedir; bölüm 6'daki iki aşamalı doğrulama akışında hangisi hangi rolü oynar?

`cmp` iki dosyayı bayt bayt karşılaştırıp yalnız "aynı mı, farklı mı" der (kaba ama hızlı); `diff` metin/satır düzeyinde ayrıntılı karşılaştırma yapar (yavaş ama bilgilendirici). Bölüm 6'daki akışta önce `cmp` ile kaba bir kontrol yapılır, fark bulunursa `objdump` çıktısı üzerinde `diff` ile hedef fonksiyonun **gerçekten** değişip değişmediği doğrulanır.

42. Bölüm 10'daki 'haftanın büyük resmi' tablosuna göre, bölüm 9 bölüm 8'e tam olarak neyi ekler?

Bölüm 8, sınıf içinde elle izlenen yedi adımlık bir süreç tanımlar (hazırla → gizle → doğrula → karşılaştır → ölç → çeşitlendir → raporla). Bölüm 9 bu aynı süreci **otomatikleştirip** bir CI hattına taşır: her sürümde tohum otomatik üretilir, testler otomatik çalışır, ölçüm ve imzalama otomatik yapılır — insan hafızasına bağımlılık ortadan kalkar.

43. Bu haftanın demo betiğindeki `erisim_ver` (`ornek.c`) ile bölüm 5'teki basitleştirilmiş `temiz.c` örneği aynı fonksiyon mudur? Bu iki gösterimi neden bir arada kullandık?

Tam olarak aynı değildir: `ornek.c`'deki gerçek `erisim_ver` bir uzunluk kontrolü ve sabit-zamanlı bir XOR karşılaştırması yapar (bölüm 0'daki gerçek kod); bölüm 4-5'teki `temiz.c` ise `jeton_gecerli(...)` çağrısıyla basitleştirilmiş, öğretici bir gösterimdir. İkisini bir arada kullandık çünkü basitleştirilmiş gösterim (`if` / `return` yapısı) dönüşümlerin **fikrini** göstermede daha nettir; gerçek kod ise betiği **gerçekten çalıştırdığınızda** ne göreceğinizi tanımlar. Bölüm 3'teki `Split` örneği gerçek kodu, diğer çoğu örnek basitleştirilmiş gösterimi kullanır — hangisinin kullanıldığı her zaman başlıkta belirtilmiştir.

44. Bu haftaki bütün 'SONRA' kod bloklarının başlığında neden hep 'kavramsal gösterim' ibaresi var; bu neden önemlidir?

Çünkü Tigress'in gerçek çıktısı sürüme, derleyiciye ve `--Seed` değerine göre değişir; bu haftanın notunda gösterilen "SONRA" blokları (`EncodeLiterals`, `Flatten`, `EncodeData`, `Split` örnekleri) gerçek bir Tigress komutunun **birebir** çıktısı değil, dönüşümün **fikrini** aktaran öğretici gösterimlerdir. Bu ayrımı açıkça belirtmemek, öğrenciyi gerçek çıktının böyle görüneceğini sanmaya iterdi — bu da uydurma bilgi vermek olurdu. **Kural:** gerçek çıktıyı görmek için `tigress-hatti.sh`'yi Tigress kuruluyken çalıştırıp `gizli.c` dosyasını okuyun.

45. Bu haftanın notunda geçen bütün sayısal maliyet/güç/çeşitlendirme örnekleri (%359,1; %166,7; %60 gibi) gerçek ölçümler midir?

Hayır, açıkça belirtildiği gibi **varsayımsaldır** — Tigress bu notu hazırlarken kurulu olmadığından gerçek `objdump` / `cmp` çıktısı alınamamıştır. Bu sayılar yalnız aritmetiğin nasıl kurulduğunu ve büyüklük mertebelerini göstermek içindir. **Kural:** kendi projenizde bu tablo biçimlerini kullanın ama sayıları `tigress-hatti.sh`'yi Tigress kuruluyken çalıştırarak **kendiniz ölçün**; varsayımsal bir sayıyı ölçülmüş gibi sunmak, 9. haftanın "kırılamaz" uyarısıyla aynı kategoride bir dürüstlük hatasıdır.

12. Kaynaklar ve ileri okuma

- **Tigress** resmî sitesi ve çalışma sayfaları (`tigress.wtf`) — dönüşümler, sözdizimi, güncel sürüm (v4) ve lisans. Öğrenciler güncel sürümü ve koşulları buradan doğrular.
- C. Collberg, J. Nagra, *Surreptitious Software* — gizleme taksonomisi ve ölçme çerçevesi (9. haftayla ortak).
- S. Banescu, C. Collberg vd. — Tigress dönüşümlerinin sembolik yürütmeye (KLEE) dayanıklılığının ölçülmesi.
- Obfuscator-LLVM (O-LLVM) — derleyici tabanlı gizlemeye alternatif (9. hafta K-11).

Bir sonraki hafta

15. hafta — Final proje gösterimleri (RAP2). Dönem içeriği tamamlandı; final raporunda bu haftanın hattı (S15) ve ölçümleri (S9) beklenir. 16. haftada Quiz-2 (9–14. haftalar).