

# **SON İŞLEM ALGORİTMALARININ İNCELENMESİ VE YAZILIM SİTİ GELİŞTİRİLMESİ**

## **YÜKSEK LİSANS TEZİ**

**DİDEM YOSUNLU**

**MERSİN ÜNİVERSİTESİ  
FEN BİLİMLERİ ENSTİTÜSÜ**

**BİLGİSAYAR MÜHENDİSLİĞİ  
ANABİLİM DALI**

**MERSİN  
OCAK- 2021**

# **SON İŞLEM ALGORİTMALARININ İNCELENMESİ VE YAZILIM SİTİ GELİŞTİRİLMESİ**

## **YÜKSEK LİSANS TEZİ**

**DİDEM YOSUNLU**

ORCID ID: 0000 – 0001 – 6917 – 4912

**MERSİN ÜNİVERSİTESİ  
FEN BİLİMLERİ ENSTİTÜSÜ**

**BİLGİSAYAR MÜHENDİSLİĞİ  
ANABİLİM DALI**

**Danışman**

Doç. Dr. Erdiç AVAROĞLU  
ORCID ID: 0000-0003-1976-2526

**MERSİN  
OCAK – 2021**

## ETİK BEYAN

Mersin Üniversitesi Lisansüstü Eğitim-Öğretim Yönetmeliğinde belirtilen kurallara uygun olarak hazırladığım bu tez çalışmada,

- Tez içindeki bütün bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- Görsel, işitsel ve yazılı tüm bilgi ve sonuçları bilimsel ahlâk kurallarına uygun olarak sunduğumu,
- Başkalarının eserlerinden yararlanılması durumunda ilgili eserlere bilimsel normlara uygun olarak atıfta bulunduğumu,
- Atıfta bulunduğum eserlerin tümünü kaynak olarak kullandığımı,
- Kullanılan verilerde herhangi bir tahrifat yapmadığımı,
- Bu tezin herhangi bir bölümünü Mersin Üniversitesi veya başka bir üniversitede başka bir tez çalışması olarak sunmadığımı,
- Tezin tüm telif haklarını Mersin Üniversitesi'ne devrettiğimi

beyan ederim.


## ETHICAL DECLARATION

This thesis is prepared in accordance with the rules specified in Mersin University Graduate Education Regulation and I declare to comply with the following conditions:

- I have obtained all the information and the documents of the thesis in accordance with the academic rules.
- I presented all the visual, auditory and written information and results in accordance with scientific ethics.
- I refer in accordance with the norms of scientific works about the case of exploitation of others' works.
- I used all of the referred works as the references.
- I did not do any tampering in the used data.
- I did not present any part of this thesis as an another thesis at Mersin University or another university.
- I transfer all copyrights of this thesis to the Mersin University.

19/01/2021

İmza / Signature



Didem YOSUNLU

## ÖZET

### SON İŞLEM ALGORİTMALARININ İNCELENMESİ VE YAZILIM SÜİTİ GELİŞTİRİLMESİ

Rasgele sayılar, geçmişten günümüze kadar birçok alanda kullanılmıştır. Simülasyon, oyun programlama, eğlence, nümerik analiz gibi alanların yanında kriptolojik uygulamaların büyük çoğunluğunda da ihtiyaç duyulan temel bir araç haline gelmiştir. Özellikle anahtar üretimi ve dağıtımında, başlangıç vektörünün oluşturulmasında ve kimlik doğrulama protokolleri gibi kritik işlemlerde rasgele sayılara ihtiyaç duyulmaktadır. Üretilen sayıların rasgeleliği ise kriptolojik uygulamanın güvenliğini direkt olarak etkilemektedir. Bu sebeple tasarlanan sistemlerin genel amacı üretilen rasgele sayıların tahmin edilemez ve tekrar üretilemez olması ve ayrıca da iyi istatistiksel özellikler göstermesidir.

Bu özelliklerde rasgele sayıların üretilmesi için çeşitli rasgele sayı üreteçleri tasarlanmıştır. Bu üreteçler 3 grupta incelenmektedir; gerçek rasgele sayı üreteçleri (GRSÜ), sözde rasgele sayı üreteçleri (SRSÜ) ve hibrit rasgele sayı üreteçleri (HRSÜ). Ne yazık ki bu üreteçler kullanılarak elde edilen bit dizilerinin hepsi istenilen gereksinimleri sağlamamaktadır. Zayıf istatistiksel özellikler gösteren bu bit dizilerini daha güçlü hale getirmek amacıyla son işlem algoritmaları kullanılmaktadır.

Bu tezin amacı tasarlanmış olan son işlem algoritmalarından seçilenlerin incelenerek, hepsinin bir arada olduğu bir yazılım süiti geliştirmek ve bunu kullanıma açmaktır. Bu amaçla seçilen algoritmalar XOR, Von Neumann, H, H2 ve H4 Fonksiyonları, SBOX, Mixing Bits in Steps and XORing, Iterating Von Neumann'dur. Tasarlanan arayüzde kullanıcının bu algoritmalarından birini seçmesi ve daha sonra da sisteme, seçilen algoritmaya tabii tutulmak istenen bit dizisini .txt dosya formatında yüklemesi beklenmektedir. Sistem yüklenen bit dizisine seçilen son işlem algoritmasını uygulayarak çıktıyı yine .txt dosya formatında olarak oluşturur ve kullanıcının kullanımına sunar. Literatürde benzer bir örneği bulunmayan bu uygulama gerçek rasgele sayılar üzerine çalışanlar için ulaşılması ve kullanımı oldukça kolay olan önemli bir kaynak haline gelecektir.

Geliştirilen web tabanlı yazılıma son kullanıcılar tarafından "postprocess.mersin.edu.tr" adresinden ulaşarak, gerçek rasgele sayı üreteçlerinden elde edilen saf bit dizilerinin son işlem sonuçları alınabilecektir.

**Anahtar Kelimeler:** Rasgele Sayı Üreteçleri, Son İşlem, Son İşlem Algoritmaları, Uygulama Geliştirme.

**Danışman:** Doçent Doktor, Erdiç AVAROĞLU, Mersin Üniversitesi, Bilgisayar Mühendisliği Anabilim Dalı, Mersin.

## ABSTRACT

### EXAMINATION OF POST PROCESSING ALGORITHMS AND DEVELOPING OF SOFTWARE

Random numbers have been used in many fields from past to present. In addition to simulation, game programming, entertainment, numerical analysis, it has become a basic tool needed in the vast majority of cryptologic applications. In particular, random numbers are required for critical operations such as key generation and distribution, initialization vector generation, authentication protocols. The randomness of the generated numbers directly affects the security of the cryptologic application. For this reason, the general purpose of the designed systems is that the generated random numbers are unpredictable and non-reproducible and also show good statistical properties.

Various random number generators have been designed to generate random numbers with these features. These generators are examined in 3 groups: true random number generators (TRNG), pseudo random number generators (PRNG) and hybrid random number generators (HRNG). Unfortunately, not all bit strings obtained using these generators meet the desired requirements. Post processing algorithms are used to make these bit sequences stronger, which show weak statistical properties.

The aim of this thesis is to examine selected post-processing algorithms, to develop a software suite that is all in one and make it available. The algorithms chosen for this purpose are XOR, Von Neumann, H, H2 and H4 Functions, SBOX, Mixing Bits in Steps and XORing and Iterating Von Neumann. In the designed interface, the user is expected to choose one of these algorithms and then upload the bitstream to the system in .txt file format to be subjected to the selected algorithm. The system creates the output in .txt file format by applying the selected post-processing algorithm to the loaded bitstream and makes it available to the user. This application, which has no similar example in the literature, will become an important resource that is very easy to access and use for those who study on true random numbers.

The developed web-based software can be accessed by the end users at "postprocess.mersin.edu.tr" address, and the post processing results of raw bit strings obtained from true random number generators can be obtained.

**Keywords:** Random Number Generators, Post Processing, Post Processing Algorithms, Developing Software.

**Advisor:** Assoc. Prof. Dr., Erdinç AVAROĞLU, Department of Computer Engineering, University of Mersin, Mersin.

## TEŞEKKÜR

Tez konumun belirlenmesinde ve planlanmasında süreç boyunca yardımlarını esirgemeyen, deneyimlerini, fikirlerini büyük bir özveri ile aktaran çok değerli danışmanım Doçent Doktor Erdiñ AVAROĞLU hocama sonsuz teşekkürlerimi sunarım.

Tez çalışmalarım boyunca maddi ve manevi olarak her zaman yanımda olan ve bana olan inançları ile beni yüreklendiren aileme ve eşime teşekkürlerimi sunarım.



## İÇİNDEKİLER

	Sayfa
İÇ KAPAK	i
ONAY	ii
ETİK BEYAN	iii
ÖZET	iv
ABSTRACT	v
TEŞEKKÜR	vi
İÇİNDEKİLER	vii
TABLolar DİZİNİ	ix
ŞEKİLLER DİZİNİ	x
KISALTMALAR ve SİMGELER	xi
<b>1. GİRİŞ</b>	<b>1</b>
1.1. Tezin Amacı	2
1.2. Tezin İçeriği	3
<b>2. RASGELE SAYILAR VE RASGELE SAYI ÜRETEÇLERİ</b>	<b>3</b>
2.1. Rasgele Sayılar	3
2.2. Rasgele Sayı Üreteçleri	3
2.3. Rasgele Sayı Üreteçlerinin Özellikleri	3
2.4. Entropi Kavramı	4
2.5. Rasgele Sayı Üreteçlerinin Sınıflandırılması	4
2.5.1. Sözde Rasgele Sayı Üreteçleri (SRSÜ)	5
2.5.1.1. Saf SRSÜ	5
2.5.1.2. Hibrit SRSÜ	6
2.5.2. Gerçek Rasgele Sayı Üreteçleri	6
2.5.2.1. Fiziksel GRSÜ	7
2.5.2.2. Fiziksel Olmayan GRSÜ	7
2.5.3. Hibrit Rasgele Sayı Üreteçleri	8
2.6. Rasgele Sayı Üreteçlerinin Test Edilmesi	9
2.6.1. FIPS-140-1 Testi	9
2.6.1.1. Monobit Testi (Monobit Test)	9
2.6.1.2. Poker Testi (Poker Test)	10
2.6.1.3. Koşu Testi (Run Test)	10
2.6.1.4. Uzun Koşu Testi (Long Run Test)	10
2.6.2. NIST-800-22 Testi	11
2.6.2.1. Frekans Testi	12
2.6.2.2. Blok Frekans Testi	12
2.6.2.3. Akış Testi	12
2.6.2.4. Blok İçindeki Birlerin En Uzun Akış Testi	13
2.6.2.5. İkili Matris Rankı Testi	13
2.6.2.6. Ayrık Fourier Dönüşüm Testi	13
2.6.2.7. Örtüşmeyen Şablon Eşleştirme Testi	14
2.6.2.8. Örtüşen Şablon Eşleştirme Testi	14
2.6.2.9. Maurer “Evrensel İstatistik” Testi	14
2.6.2.10. Lempel-Ziv Sıkıştırma Testi	14
2.6.2.11. Doğrusal Karmaşıklık Testi	15
2.6.2.12. Seri Testi	15
2.6.2.13. Yaklaşık Entropi Testi	15
2.6.2.14. Kümülatif Toplamlar Testi	15
2.6.2.15. Rasgele Gezinimler Testi	16
2.6.2.16. Rasgele Gezinimler Değişken Testi	16
<b>3. SON İŞLEM ALGORİTMALARI</b>	<b>17</b>
3.1. XOR Algoritması	17

	Sayfa
3.2. Von Neumann Algoritması	18
3.3. Doğrusal Beslemeli Kayan Yazmaç	19
3.4. Lojistik Harita	20
3.5. H Fonksiyonu	21
3.6. SHA-256	22
3.7. Mixing Bits in Steps and XORing	22
3.8. N Bits Von Neumann	24
3.9. Iterating Von Neumann (IVN)	24
3.10. Hig-Throughput Von Neumann	24
3.11. Keccak Algoritması	25
3.12. SBOX	25
3.13. Resilient Fonksiyon	26
<b>4. GELİŞTİRİLEN YAZILIM</b>	28
4.1. Kullanılan Teknolojiler	28
4.1.1. ASP.NET	28
4.1.2. MVC YAPISI	29
4.1.2.1. Model	30
4.1.2.2. View	30
4.1.2.3. Controller	30
4.1.3. Bootstrap	31
4.2. Projenin Tasarımı	31
4.2.1. XOR Algoritmasının Entegrasyonu	34
4.2.2. Von Neumann Algoritmasının Entegrasyonu	35
4.2.3. H Fonksiyonun Entegrasyonu	36
4.2.4. H2 Fonksiyonun Entegrasyonu	37
4.2.5. H4 Fonksiyonun Entegrasyonu	38
4.2.6. SBOX Entegrasyonu	39
4.2.7. Mixing Bits in Steps and XORing	40
4.2.8. Iterating Von Neumann	41
4.3. Arayüz Kullanımı	42
4.4. NIST Test Sonuçları	45
<b>5. SONUÇ</b>	48
KAYNAKLAR	49
ÖZGEÇMİŞ	51



## TABLolar DİZİNİ

	Sayfa
Tablo 2.1. Koşu Testi için kriterler	10
Tablo 2.2. Blok içindeki Birlerin En Uzun Akış Testi için test parametreleri	13
Tablo 3.1. XOR Son İşlem Çıkışı	17
Tablo 3.2. Von Neumann Son İşlem Çıkışı	18
Tablo 4.1. Makale [18] kullanılarak üretilen bit dizisine uygulanılan test sonuçları	45
Tablo 4.2. Makale [31] kullanılarak üretilen bit dizisine uygulanılan test sonuçları	46



## ŞEKİLLER DİZİNİ

	Sayfa
Şekil 2.1. Rasgele Sayı Üreteçlerinin Sınıflandırılması	4
Şekil 2.2. Saf Sözde Rasgele Sayı Üreteçlerinin genel tasarımı	5
Şekil 2.3. Hibrit Sözde Rasgele Sayı Üreteçlerinin genel tasarımı [5]	6
Şekil 2.4. Fiziksel GRSÜ'nün genel tasarımı	7
Şekil 2.5. Fiziksel Olmayan GRSÜ'nün genel tasarımı	8
Şekil 2.6. Seri bağlanan (a) ve paralel bağlanan (b) Hibrit RSÜ'nün genel tasarımı	8
Şekil 3.1. XOR Son İşlem Algoritması Örneği	18
Şekil 3.2. Von Neumann Son İşlem Algoritması Örneği	19
Şekil 3.3. Doğrusal Beslemeli Kayan Yazmaç Genel Tasarımı [6]	19
Şekil 3.4. DBKY ile kullanılan RSÜ'nün genel yapısı [16]	19
Şekil 3.5. [17]'de kullanılan DBKY ile RSÜ'nün genel yapısı	20
Şekil 3.6. H son işlem fonksiyon [6]	21
Şekil 3.7. Mixing Bits in Steps and XORing of Adjacent Bits yöntemi gösterimi	23
Şekil 3.8. IVN tasarımı	24
Şekil 4.1. .NET framework'ü genel mimarisi	29
Şekil 4.2. MVC mimarisi hayat döngüsü	30
Şekil 4.3. Geliştirilen projenin klasör yapısı	31
Şekil 4.4. Uygulamanın arayüz html kodu örneği	32
Şekil 4.5. HomeController.cs fonksiyon yapısı	33
Şekil 4.6. "Create" fonksiyonunun içeriği	34
Şekil 4.7. XOR son işlem yönteminin algoritması	35
Şekil 4.8. Von Neumann son işlem yönteminin algoritması	36
Şekil 4.9. H Fonksiyonu son işlem yönteminin algoritması	36
Şekil 4.10. H2 Fonksiyonu son işlem yönteminin algoritması	37
Şekil 4.11. H4 Fonksiyonu son işlem yönteminin algoritması	38
Şekil 4.12. Kullanılan S-kutularının tanımlanması	39
Şekil 4.13. SBOXs son işlem yönteminin algoritması	40
Şekil 4.14. Mixing Bits in Steps and XORing algoritması	41
Şekil 4.15. Iterating Von Neumann algoritması	42
Şekil 4.16. Uygulama Arayüzü	43
Şekil 4.17. Dosya seçme ekranı	43
Şekil 4.18. Kullanılacak son işlem algoritmasını seçme ve işlemi başlatma	44
Şekil 4.19. Hatalı şekilde sonlanan işlem	44
Şekil 4.20. Başarılı şekilde sonlanan işlem	45

## KISALTMALAR ve SİMGELER

Kısaltma/Simge	Tanım
RSÜ	Rasgele Sayı Üretici
SRSÜ	Sözde Rasgele Sayı Üreteçleri
SSRSÜ	Saf Sözde Rasgele Sayı Üretici
HSRSÜ	Hibrit Sözde Rasgele Sayı Üretici
GRSÜ	Gerçek Rasgele Sayı Üretici
FGRSÜ	Fiziksel Gerçek Rasgele Sayı Üretici
FOGRSÜ	Fiziksel Olmayan Gerçek Rasgele Sayı Üretici
XOR	Exclusive Or
DAS	Sayısallaştırılmış Analog Sinyal
NIST	Uluslararası Standartlar ve Teknoloji Enstitüsü
FIPS	Federal Bilgi İşleme Standartları
DBKY	Doğrusal Beslemeli Kayan Yazmaç
VN_N	N bit Von Neumann
IVN	Iterating Von Neumann

## 1. GİRİŞ

Geçmişten bu yana bilgi oldukça kıymetliydi. İnsanlar gizli kalmasını istedikleri bilgiyi şifrelemeyi aslında çok eski zamanlardan beri kullanmaktadırlar. Bunun en bilindik örneklerinden biri de MÖ yaşayan Julius Caesar'ın devlet ile ilgili haberleşmelerde kullandığı "Sezar Şifreleme" olarak da bilinen yöntemdir. Bu yöntemde iletilecek metnin içindeki her harf alfabede kendinden sonraki 3. harf ile değiştirilerek yeni şifreli bir metin oluşturularak haberleşme sırasında mesajı ele geçiren kişinin bu mesajdan bir bilgi elde edememesi amaçlanmıştır.

Günümüzde de gelişen teknoloji ile birlikte bilgisayar ağlarının ve internetin yaygın olarak kullanılması bilgi güvenliğini oldukça önemli hale getirmiştir. Gelişen teknolojinin birçok avantajı olmakla birlikte bize getirdiği en önemli dezavantaj da bilginin güvenli bir şekilde saklanması ve paylaşılmasının zorlaşmasıdır. Verilere yetkisiz bir kişinin erişimi ile verinin ele geçirilmesi, bozulması ya da değiştirilmesi önemli bir problem haline gelmiştir. Bu noktada bilgiyi koruma amaçlı kriptografi kullanılmaktadır. Kriptografi basitçe şifreleme bilimidir. Temelde yapılan işlem açık metnin gizli bir anahtar yardımı ile şifreli hale getirilmesidir. Şifrenin çözülmesi kısmında ise şifreli metin yine aynı anahtar ile açık metne dönüştürülmesi işlemidir. Buradan da anlaşılacağı üzere sistemin güvenliği kullanılan anahtarların gizliliğine bağlıdır. Bu nedenle bu anahtarların tahmin edilemez ya da tekrar üretilemez olması gerekmektedir.

Rasgele sayılar ile kriptografiyi bağlayan nokta tam da burasıdır. Rasgele sayılar belirli bir aralık için tanımlanmış, oluşma olasılıkları birbirine eşit ve bu sayılar arasında belirli bir ilişki olmayan sayılar olarak tanımlanabilir. Yani geçmiş değerler ya da şimdiki değerler temel alınarak gelecekteki değerleri tahmin etmek mümkün olmamalı. Kriptografide de kullanılan anahtarın rasgeleliği sistemin güvenliğini doğrudan etkilemektedir. Bu sebeple de kullanılan rasgele sayılar tahmin edilememe, tekrar üretilmeme ve iyi istatistiksel özellikler gibi sıkı gereksinimleri sağlamaları gerekmektedir [1]. Bu sebeple de rasgele sayı üreteçlerine ihtiyaç doğmuştur.

Rasgele sayı üreteçleri (RSÜ) üç ana sınıfa ayrılmaktadır; Sözde Rasgele Sayı Üretici (SRSÜ), Gerçek Rasgele Sayı Üretici (GRSÜ) ve Hibrit Rasgele Sayı Üretici (HRSÜ).

SRSÜ ögeleri arasında kolay kolay ilişki kurulamayacak bir sayı dizisi üreten algoritma türleridir. Tohum olarak adlandırılan başlangıç vektörü ile deterministik olarak hesaplanır. Bu sebepten dolayı bu üreteçlerin ürettiği sayı dizisinde periyodiklik görülebilmektedir [2].

GRSÜ'ler ise entropi kaynağı olarak deterministik olmayan fiziksel olaylar kullanır. Faz gürültüsü, termal gürültü, titreşim, rastgele telgraf gürültüsü, fotoelektrik etki ve doğada istatistiksel olarak rasgele olan diğer fenomenler gibi fiziksel süreçlere dayanan yüksek hızlı ve

gerçek rastgele diziler oluştururlar [3]. Bu kaynaklardan gelen sinyaller sayısal olarak dönüştürülerek örnekleme yapılır. Bu süreçten sonra üretilen sayılar zayıf istatistikî özellikler gösterebilmektedir. Bu sebeple üretilen bu diziler son işlemden geçirilerek zayıflıklarının giderilmesi amaçlanır. Son işlem algoritmaları çıkış bit oranında azalmaya sebep olabilmesine rağmen üreticinin güvenliğini arttırdığından dolayı GRSÜ'ler kriptolojide birçok uygulamada kullanılmaktadır.

HRSÜ ise GRSÜ'den elde edilen rasgele sayının SRSÜ'de tohum değeri olarak kullanılmasıyla her iki sistemin birlikte çalıştığı rasgele sayı üreticidir [1].

### 1.1. Tezin Amacı

Son yıllarda kriptolojide ve diğer birçok alanda rasgele sayılara ihtiyaç artmaktadır. Bu amaçla oluşturulan birçok yazılımsal ve donanımsal rasgele sayı üreticileri vardır. Fakat bu üreticilerden elde edilen rasgele sayı dizileri kriptografik uygulamalar için zorunlu olan kestirilememe, tekrar üretilmeme ve iyi istatistikî özellikleri sağlamadığından dolayı güvenilir olmamaktadırlar. Bu sayıların zayıflıklarının giderilmesi için son işlem algoritmasına ihtiyaç duyulmaktadır. Geçmişten günümüze birçok çalışmada birçok son işlem algoritması önerilmiştir fakat bu algoritmalar çoğunlukla ya yazılı olarak açıklanmış ya da sözde kod olarak sunulmuştur.

Bu çalışmada öncelikle son işlem algoritmasının ne olduğu açıklanmıştır. Daha sonra da literatür taraması yapılarak günümüze kadar önerilmiş olan bazı son işlem yöntemleri açıklanmıştır. Bu yöntemler; XOR algoritması, Von Neumann algoritması, Doğrusal Beslemeli Kayan Yazmaç, Lojistik Harita, H fonksiyonu, H2 fonksiyonu, H4 fonksiyonu, SHA-256, Mixing Bits in Steps and XORing, High-Throughput Von Neumann, Keccak algoritması, SBOX ve Resilient fonksiyon.

Bu tezde, yukarıda bahsedilen son işlem yöntemlerinden XOR algoritması, Von Neumann algoritması, H, H2 ve H4 fonksiyonları, SBOX, Mixing Bits in Steps and XORing ve Iterating Von Neumann algoritmalarının bir arada bulunduğu bir uygulama geliştirilerek bu uygulamanın herkesin ulaşip kullanabileceği şekilde sunulması amaçlanmaktadır. Bu uygulamanın literatürde benzer bir örneğinin olmamasından dolayı rasgele sayılar üzerine çalışanlar için önemli bir kaynak olacaktır.

Son işlem yazılımının sonuçlarını görebilmek amacıyla örnek saf bit dizileri yazılımdan geçirilerek üretilen son işlem sonuçları NIST testlerine tabi tutulmuştur ve tasarlanan yazılımın başarılı bir şekilde çalıştığı doğrulanmıştır.

## **1.2. Tezin İçeriği**

Yukarıda belirtilen amaç doğrultusunda tezin içeriği 4 bölümden oluşmaktadır. Bölüm 2’de rasgele sayı üreteçleri, bunların özellikleri ve uygulanan testlerden bahsedilmiştir. Bölüm 3’te ise son işlem yöntemlerinin önemi ve şimdiye kadar önerilmiş olan yöntemlerden bazıları açıklamalı olarak anlatılmıştır. Bölüm 4’te geliştirilen yazılım ve kullanılan teknolojilerden bahsedilirken son bölümde ise çalışmayla ilgili sonuçlara ve bu sonuçlarla ilgili yorumlara yer verilmiştir.

## **2. RASGELE SAYILAR VE RASGELE SAYI ÜRETEÇLERİ**

### **2.1. Rasgele Sayılar**

Rasgele kelimesi en basit anlamıyla kesin olarak bilinemeyen ve tahmin edilemeyen anlamına gelmektedir. Bu sayılar birçok uygulamanın en önemli parçasını oluşturmaktadır. Bilgisayar bilimleri dışında zar atma, sayısal loto, nümerik analiz, modelleme, simülasyon gibi daha birçok alanda kullanılan rasgele sayıların iyi istatistiksel özellikler göstermesi ve tahmin edilemez olması gerekmektedir [4].

### **2.2. Rasgele Sayı Üreteçleri**

Rasgele sayı üreteçleri rasgele sayıları üretmede kullanılan araçlardır. Günümüze kadar birçok rasgele sayı üretici geliştirilmiştir. Bu araçların içinde fiziksel araçların yanında, yazılımsal olan algoritmaya dayalı, olasılık dağılımına dayalı fonksiyonlar ve insanlar tarafından oluşturulan veriler ile rasgele sayı üreten araçlar bulunmaktadır.

### **2.3. Rasgele Sayı Üreteçlerinin Özellikleri**

Kriptoloji alanında da sık sık kullanılan rasgele sayılar sistemin güvenliğini doğrudan etkilemektedir. Rasgele sayıların birbirinden bağımsız olması, tekrar etmemesi, tahmin edilemez ve yeniden üretilemez olması bu gibi uygulamalar için oldukça kritik öneme sahiptir. Bu sebeple RSÜ’lerin taşıması gereken birçok özellik bulunmaktadır. Bunlar;

- Üretilen rasgele sayıların iyi istatistiksel özellikler göstermesi gerekmektedir.
- Üretilen çıkış değerinin tahmin edilemez olması yani, şimdiki değerden geçmiş ya da gelecekteki çıkışlar üretilmemelidir.
- Çevresel koşullardan etkilenmemelidir.
- Dışarıdan gelecek olan saldırılara karşı dayanıklı olmalıdır.

- Kullanılacak uygulamanın gereksinimine uygun yüksek hızda veri çıkışı sağlamalıdır.
- Düzgün dağılıma sahip olmalıdır.
- Üretcin özellikleri kullanım ömrü boyunca değişmemelidir.
- Yüksek entropi ile çıkış bitleri üretebiliyor olmalıdır.

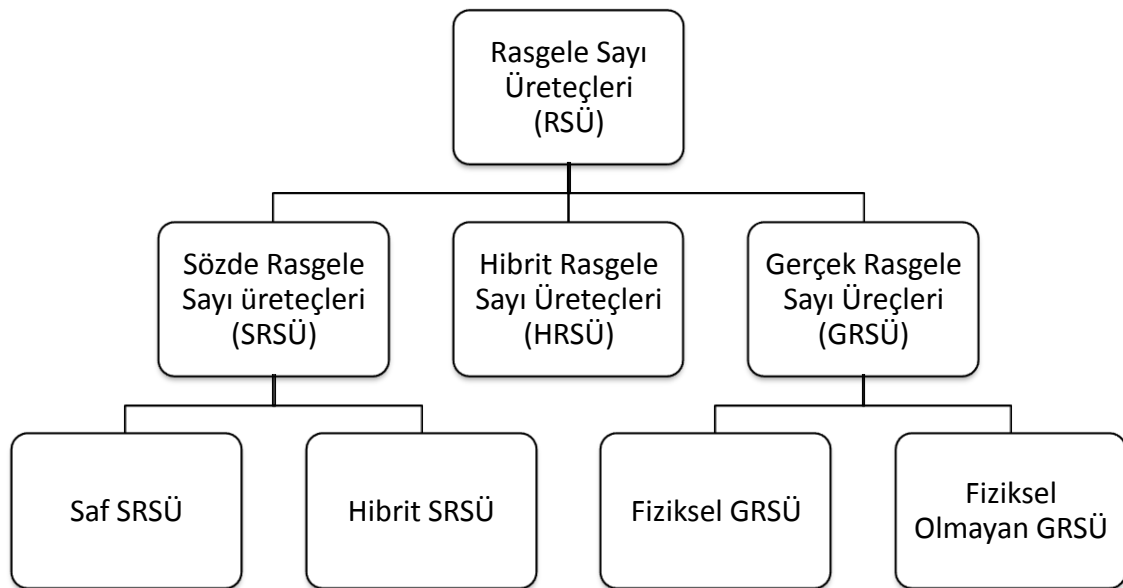
Kullanılan rasgele sayılar yukarıdaki özellikleri sağlayan bir üretici ile üretilmediği takdirde, sistemin güvenliğini zayıflatabilir. Özellikle kriptografik uygulamalar için bu özellikler oldukça kritiktir.

## 2.4. Entropi Kavramı

En basit haliyle entropi bir sistemin düzensizlik ölçüsü anlamına gelmekte iken RSÜ'de entropi bir rasgele ayının belirsizlik ölçüsü olarak tanımlanmaktadır. Çoğunlukla Shannon entropi olarak bilinmektedir. RSÜ'de minimum entropi her zaman sonuçlardan birinin ortaya çıkması ile meydana gelirken, maksimum entropi mümkün olan bütün olasılıkların eşit olduğu olasılıklarda elde edilebilir. Entropi yükseldikçe üretilen sayı dizilerinin rasgeleliği artacaktır.

## 2.5. Rasgele Sayı Üreteçlerinin Sınıflandırılması

Şekil 2.1'de gösterildiği gibi RSÜ'ler üç ana sınıfa ayrılmaktadır; Sözde Rasgele Sayı Üretici (SRSÜ), Gerçek Rasgele Sayı Üretici (GRSÜ) ve Hibrit Rasgele Sayı Üretici (HRSÜ). SRSÜ'ler kendi içerisinde Saf SRSÜ ve Hibrit SRSÜ olarak ikiye ayrılırken, GRSÜ de kendi içerisinde Fiziksel GRSÜ ve Fiziksel Olmayan GRSÜ olarak ikiye ayrılmaktadır.

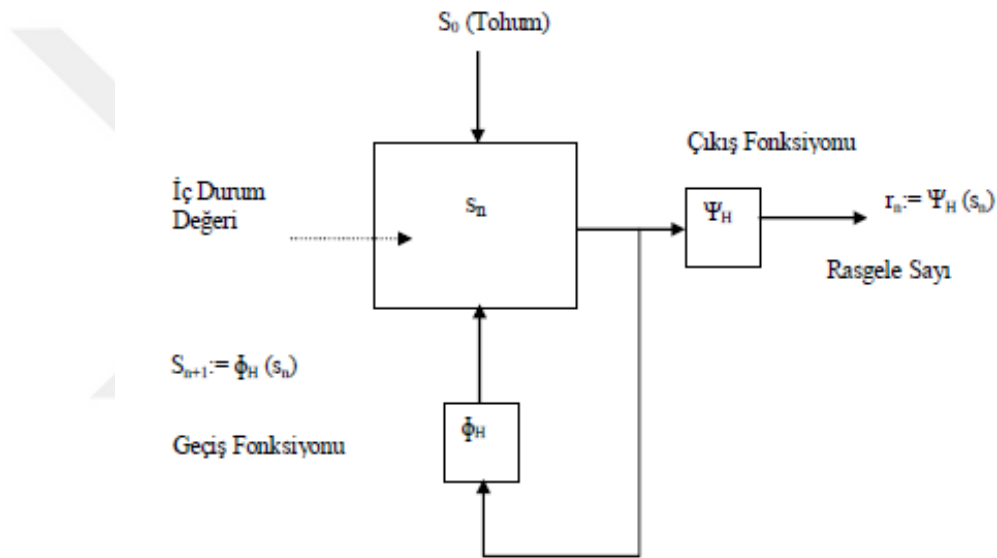


Şekil 2.1. Rasgele Sayı Üreteçlerinin Sınıflandırılması.

### 2.5.1. Sözde Rasgele Sayı Üreteçleri

SRSÜ'leri (Pseudo Random Number Generator) deterministik yollarla yani matematiksel denklem, algoritma ya da daha önceden tanımlanmış olan kurallar kullanarak rasgele sayı dizileri oluşturan üreteçlerdir. Bir başlangıç anahtarına ihtiyaç duyarlar. Fakat başlangıç anahtarı ve kullanılan algoritmalar ya da denklemler bilindiğinde sayıların tekrar üretilmesi mümkün olmaktadır. Bu sebeple kriptografik uygulamalar için uygun değildir. Fakat diğer yöntemlere göre daha ucuz olması, donanım ihtiyacı olmaması ve hızlı olması birçok uygulama için tercih sebebi olmaktadır. SRSÜ'ler saf ve hibrit olmak üzere iki gruba ayrılmaktadır.

#### 2.5.1.1. Saf SRSÜ



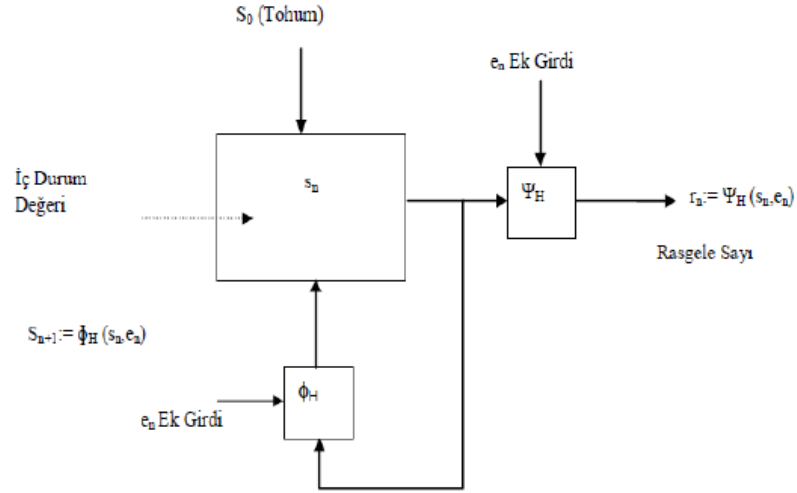
Şekil 2.2. Saf Sözde Rasgele Sayı Üreteçlerinin genel tasarımı [5].

Şekil 2.2’de saf SRSÜ’nin genel tasarımı gösterilmiştir.  $r_1, r_2, r_3 \dots r_n \in R$  üretilen rasgele sayıları gösterirken  $S_n \in S$  SRSÜ’nin iç durumlarını göstermektedir.  $\psi: S \rightarrow R$  çıkış fonksiyonu geçerli iç durum değeri olan  $S_n$ ’den bir sonraki rasgele sayı olan  $r_n$ ’i hesaplamaktadır. Sonrasında ise Şekil 2.2’de de görülebileceği gibi  $S_n$  durumu  $\Phi_H$  geçiş fonksiyonu kullanılarak  $S_{n+1}$  olarak güncellenir. İlk durum değeri olan  $S_1$  ilk girişte kullanılan tohum değeri  $S_0$ ’dan  $S_1 = \Phi(S_0)$  şeklinde üretilir [5].

Tasarımdan da anlaşılacağı üzere sistemin güvenliği tohum değerinin ve geçerli iç durum değerinin bilinmemesine bağlıdır. Bu sebeple geçerli iç durum değeri sistemin aktif olmaması durumunda bile korunmalıdır. Sistemin entropisi tohum değerinin entropisine bağlıdır. Bu sebeple tohum değeri tahmin edilememe özelliğinin sağlayabilmek için bir GRSÜ tarafından üretilir [5].



### 2.5.1.2. Hibrit SRSÜ



**Şekil 2.3.** Hibrit Sözde Rasgele Sayı Üreteçlerinin genel tasarımı [5].

Şekil 2.3'te görülebileceği üzere Saf SRSÜ'den farklı olarak sisteme  $e_n$  ek girdisi dahil edilmiştir. Bu ek girdi sayesinde sistemin güvenliği artmaktadır. Çünkü sisteme dahil edilen bu ek girdi sisteme rasgelelik ve tahmin edilememe özelliklerini kazandırmaktadır. Ek girdinin biliniyor olması sistemin güvenliğinin azaltmaz.

### 2.5.2. Gerçek Rasgele Sayı Üreteçleri

Daha önce de bahsedildiği gibi rasgele sayılar özellikle kriptografik uygulamalar için oldukça önemlidir. Gerçek rasgele sayı üreteçleri de bu uygulamalar için kilit noktadadır. GRSÜ'ler kriptografide başlangıç vektörü ve anahtar üretimi gibi kısımlarda kullanılırken aynı zamanda SRSÜ'ler için de tohum üretici olarak kullanılmaktadır. Uzun vadede gizliliği korunmak zorunda olan rasgele sayıların üretimi için GRSÜ'nün kullanılması daha uygundur [6].

GRSÜ'ler 3 temel bileşenden oluşmaktadır. Bunlar;

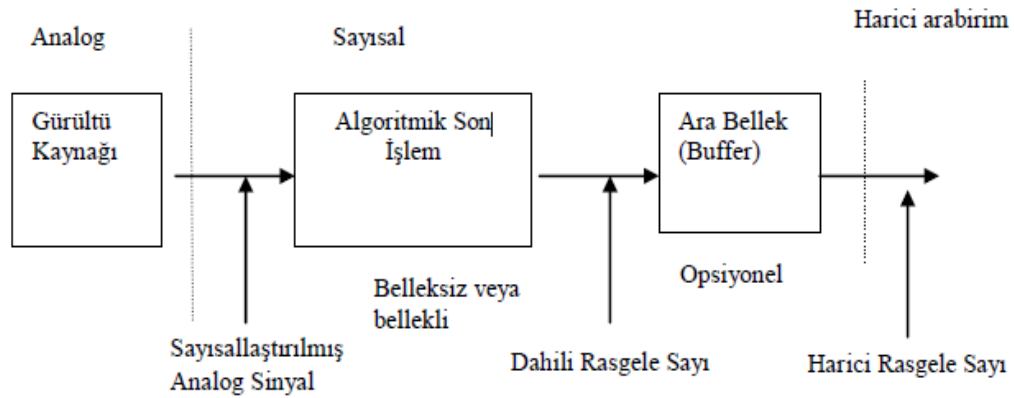
- *Entropi (Gürültü) Kaynağı:* Gürültü kaynağı GRSÜ için en önemli bloktur çünkü entropi kaynağı gerekli rasgeleliği üretmektedir. Örnek entropi kaynakları; termal gürültü, insan kaynaklı etkileşimler (fare ve klavye hareketleri), faz gürültüsü.
- *Örnekleyici:* Sayısallaştırıcı olarak da bilinen örnekleyici üretilen analog sinyalin sayısallaştırılmasından sorumludur. Genellikle D-tipi flip flop örnekleyici olarak kullanılmaktadır.
- *Son İşlem:* Örneklemme işleminden sonra üretilen sayıları istatistiksel olarak güçlendirmek amaçlı son işlem uygulanır. Ancak son işlem uygulamaları

zayıflıkları giderip sistemi güçlendirirken bit oranında azalmaya sebep olabilmektedirler. Son işlem Bölüm 3'te ayrıntılı olarak açıklanacaktır.

Gerçek Rasgele Sayı Üreteçleri, Fiziksel Gerçek Rasgele Sayı Üreteci (FGRSÜ) ve Fiziksel Olmayan Gerçek Rasgele Sayı Üreteci (FOGRSÜ) olmak üzere iki gruba ayrılmaktadır. Bu bölümde bu üreteçlerin genel tasarımı ve özellikleri açıklanmıştır.

### 2.5.2.1. Fiziksel Gerçek Rasgele Sayı Üreteci (FGRSÜ)

Genel tasarımı Şekil 2.4'te gösterilen FGRSÜ'ler deterministik olmayan, tam entropi sağlayabilen rasgele sayı üreteçleridir [3]. Entropi (gürültü) kaynağı olarak tahmin ve kontrol edilemeyen doğal fiziksel olaylar (termik gürültü, atmosferik gürültü veya nükleer bozulma gibi) kullanılır. Sistemin rasgeleliği tamamen kullanılan entropi kaynağına bağlıdır çünkü bu kaynaklar üretilen dizideki herhangi bir bitin tahmin edilmesini ve tekrardan üretilebilmesini engellemektedir.

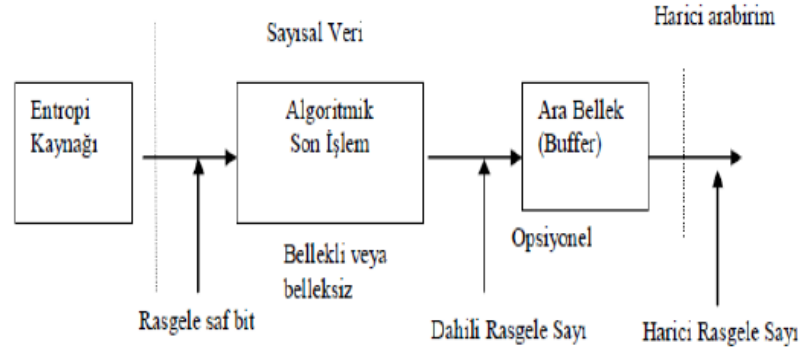


Şekil 2.4. Fiziksel GRSÜ'nün genel tasarımı.

Sistemin genel yapısına baktığımızda, entropi (gürültü) kaynağından elde edilen analog gürültü aynı anda sayısallaştırılarak analog sinyale (binary) dönüştürülmektedir. Bu sinyaller kısaca DAS (Digitized analog signal) olarak adlandırılmaktadırlar. Üretilen bu analog sinyallerin içerebileceği potansiyel zayıflıkları giderebilmek amacıyla bu sayılar çeşitli son işlem uygulamalarından geçirilmektedir. Fakat bu işlemlerin basit bir dönüşümden ibaret olmadığı unutulmamalıdır. Sistemin güvenliği artarken çıkış bit oranı azalmakta ve çıkış hızı düşmektedir. Bu sebeple güçlü entropi kaynakları kullanılarak algoritmik son işlem kullanmaksızın rasgele sayılar üreten yapılar kurulmaya çaba gösterilmektedir.

### 2.5.2.2. Fiziksel Olmayan Gerçek Rasgele Sayı Üreteci (FOGRSÜ)

FGRSÜ tasarımı ile oldukça benzer olan FOGRSÜ genel tasarımı Şekil 2.5'te gösterilmiştir.



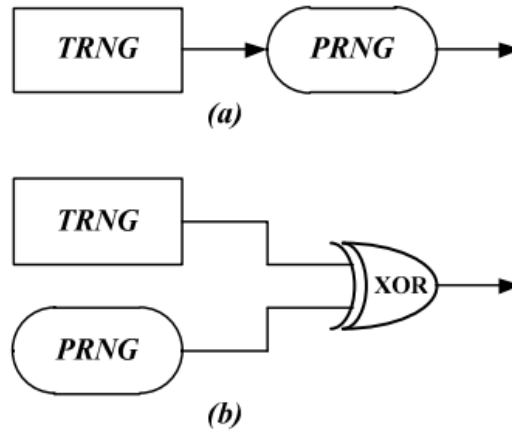
**Şekil 2.5.** Fiziksel Olmayan GRSÜ'nün genel tasarımı.

FOGRSÜ'nin FGRSÜ'den farkı gürültü kaynağı olarak zorunlu donanım ihtiyacı olmamasıdır. FOGRSÜ genel olarak PC zamanı, RAM verileri gibi sistem verilerini ya da fare hareketi gibi kullanıcı etkileşimi gürültü kaynağı olarak kullanır. Entropisi düşük kaynaklar kullanılmasından ötürü güçlü son işlem algoritmalarına ihtiyaç duyulmaktadır. Bir diğer fark da entropi kaynağının kullanıcının kontrolü altında olmasıdır.

Bu tür üreteçler genellikle saf SRSÜ için tohum değeri üretmede ya da Hibrit SRSÜ için ek girdi üretmek için kullanılmaktadırlar.

### 2.5.3. Hibrit Rasgele Sayı Üreteçleri

Hibrit Rasgele Sayı Üretici ise her iki sistemin de beraber çalıştığı rasgele sayı üreteçleridir.



**Şekil 2.6.** Seri bağlanan (a) ve paralel bağlanan (b) Hibrit RSÜ'nün genel tasarımı.

Şekil 2.6'da gösterildiği gibi 2 türlü ilişkilendirilebilirler. Genellikle SRSÜ için tohum değerinin üretilmesinde GRSÜ kullanılarak daha güçlü hale getirilebilmektedir. Bir diğer yaklaşım ise SRSÜ ve GRSÜ çıkışlarının XOR'lanarak çıkış bit dizisinin oluşturulmasını amaçlayan paralel bağlantı şeklidir [7].

## 2.6. Rasgele Sayı Üreteçlerinin Test Edilmesi

Rasgele sayı üreteçlerinin üretmiş olduğu ikili bir dizinin rasgeleliği matematiksel olarak kanıtlanamamasına rağmen uluslararası seviyede kabul edilmiş bazı istatistiksel testler ile üretilen sayıların rasgeleliği ve üreticinin kalitesi hakkında yorum yapmak mümkün olabilmektedir.

Rasgele sayı üreteçlerini test etmenin en yaygın yöntemi, çıktılarının istatistiksel analizine dayanır. İstatistiksel test paketleri, rastgele sayı üreteçleri tarafından üretilen dizilerin rasgelelik kalitesinin değerlendirilmesinde çok önemli bir rol oynar. Ve bu diziler, rasgele sayı dizilerinin öngörülemezlik, tekrar üretilemezlik, tekdüze dağıtım veya diğer belirli özellikleri gerektiren uygulamalar için temel bir girdi oluştururken, istenen rasgelelik kalitesi, bir uygulama alanından diğerine farklılık gösterebilir ve değişir. Bu nedenle de birkaç istatistiksel test grubu mevcuttur [8].

Bilinen testlerden başlıca iki tanesi, “Kriptografik modüller için güvenlik araçları” isimli çalışma olarak bilinen Federal Bilgi İşleme Standartları 140-1 (Federal Information Processing Standards (FIPS 140-1)) testi ve Ulusal Standartlar ve Teknoloji Enstitüsü (National Institute of Standards and Technology (NIST)) tarafından yayınlanan “Kriptografik uygulamalar için rasgele ve sözde rasgele sayı üreteçleri için bir istatistiksel test aracı” (A statistical test suite for random and pseudo random number generators for cryptographic applications) ismiyle yayınlanan NIST-800-22 testidir. FIPS-140-1 testi, blok uzunluğu 20 Kbit gibi küçük boyutlu bit dizilerini test etmek amacıyla kullanılmaktadır. NIST- 800-22 testi, FIPS-140-1 testine göre 100 Kbit, 500 Kbit ve 1 Mbit gibi daha büyük boyutlu bit dizilerini test etmek için kullanılmaktadır. NIST-800-22 testleri, FIPS- 140-1 testlerine göre daha zor ölçütler kullanılarak yapılan testler olarak görülmektedir [9].

Testlerin açıklanmasında referans [10] ve [12] kullanılmıştır.

### 2.6.1. FIPS-140-1 Testi

FIPS-140-1 test süiti dört test içermektedir. RSÜ çıkışından alınan 20 Kbit’lik bir bit dizisi Monobit, Poker, Koşu ve Uzun Koşu testlerine tabi tutulmaktadır ve dizinin rasgele kabul edilebilmesi için dört testten de geçmesi gerekmektedir.

#### 2.6.1.1. Monobit Testi (Monobit Test)

Test, RSÜ tarafından üretilen bit dizisindeki ‘0’ ve ‘1’ dağılım oranını inceler. Test edilen 20k bitlik verinin testi geçebilmesi için dizideki ‘1’ sayısının  $9654 < n < 10346$  aralığında olması gerekmektedir.

### 2.6.1.2. Poker Testi (Poker Test)

Bu testte, 'k' bit içeren bir dizi,  $5 \geq 5 * 2m$  olacak şekilde, üst üste çakışmayan  $m$  bitlik parçalara ayrılmaktadır.  $i$ . parça  $n_i$  olarak adlandırılır. Rasgele bir bit dizisinden beklenen,  $k$  uzunluklu bir bit dizisinde tüm  $m$  bitlik blok parçalarının aynı sayıda birbirini tekrar etmesidir. Testin başarılı olabilmesi için aşağıda verilen denklemde hesaplanan  $X$  değerinin  $1,03 < X \leq 57,4$  aralığında olması gerekmektedir.

$$X = \left( \frac{16}{5000} \right) \left( \sum_{i=0}^{15} [f(i)]^2 \right) - 5000 \quad \text{Denklem (1)}$$

### 2.6.1.3. Koşu Testi (Run Test)

Bu testte ise üretilen bit dizisinin testi başarıyla geçebilmesi için dizide ardı ardına gelen '1' ve '0'lerden oluşan çeşitli uzunluktaki blokların sayısının Tablo 2.1'de belirtildiği gibi olması beklenmektedir. 6 bitten daha uzun olan bloklar 6 bit kabul edilmekte ve blok sayısı arttırılmaktadır.

**Tablo 2.1.** Koşu Testi için kriterler

Blok Uzunluğu	Blok sayısı aralığı
1	2267-2733
2	1079-1421
3	502-748
4	223-402
5	90-223
6 ve 6+	90-223

### 2.6.1.4. Uzun Koşu Testi (Long Run Test)

Uzun koşu testinin başarılı olma koşulu ise üretilen 20 Kbit'lik veri içerisindeki '0' veya '1'lerden oluşan tüm blokların sayısının 34'ten küçük olmasıdır.

### 2.6.2. NIST 800-22 Testi

Uluslararası düzeyde kabul görmüş istatistiksel testlerden biri de NIST-800-22 test süitidir. FIPS-140-1 testlerine göre daha güçlü oldukları için FIPS-140-1 testini geçen bir rasgele bit dizisi NIST-800-22 testinden kalabilir. Bu nedenle NIST-800-22 test süiti daha güvenlidir ve kritik uygulamalar için bu testlerin kullanılması önerilmektedir.

NIST-800-22 test süiti rasgele sayı dizilerini daha detaylı incelediğinden dolayı 16 tane test içermektedir. Bit dizisinin gerçek rasgele olduğunu söyleyebilmek için tüm testleri başarıyla geçmesi beklenmektedir. Testlerden birinden başarısız olması üretilen rasgele bit dizisinin gerçek rasgele olmadığını gösterir. Bu testler aşağıdaki gibidir:

1. Frekans (monobit) testi
2. Bir blok içinde frekans testi
3. Akış (Runs) testi
4. Bir blok içinde en-uzun-birlerin akış (longest-run-of-ones) testi
5. İkili matris rankı testi
6. Ayrık Fourier dönüşümü (spektral) testi
7. Örtüşmeyen şablon eşleştirme (Non-overlapping template matching) testi
8. Örtüşen şablon eşleştirme (Overlapping template matching) testi
9. Maurer'in "Evrensel İstatistik" testi
10. Lempel-Ziv sıkıştırma testi
11. Doğrusal karmaşıklık (linear complexity) testi
12. Seri (serial) test
13. Yaklaşık entropi (approximate entropy) testi
14. Kümülatif toplamlar (cumulative sums – cusums) testi
15. Rasgele gezinimler (random excursions) testi
16. Rasgele gezinimler değişken (random excursions variant) testi

Test süitinde bulunan 16 testin uygulanma sırası tamamen isteğe bağlı olmakla beraber, frekans testinin, bir dizide rasgele olmayan bölgelerin varlığı ile ilgili temel ipuçları verdiğinden ilk sırada uygulanması önerilmektedir. Eğer bu test başarısız sonuçlanırsa diğer testlerin de başarısız sonuçlanma ihtimali yüksektir.

Test süitindeki bazı testler hesaplama aşamasında referans dağılımı olarak standart normal dağılım ve Ki-kare dağılımlarını kullanmaktadırlar.

Eğer test altındaki dizi, gerçekte rasgele değilse, hesaplanan test istatistiği, referans dağılımın ekstrem bölgelerinde başarısız olur. Standart normal dağılım, RSÜ'den elde edilen test istatistik değeri ile rasgelelik varsayımı altındaki istatistiğin beklenen değerinin

karşılaştırılmasında kullanılır. Standard normal dağılımın test istatistiği  $z = (x - \mu) / \sigma$  biçimindedir. Burada  $x$ , örnek istatistik test değeri,  $\sigma^2$  ve  $\mu$  ise varyans ve beklenen değer olarak verilir.

$\chi^2$  dağılımı ise örnek bir ölçümün gözlemlenen frekansları ile buna denk düşen farz edilen dağılımın beklenen frekansları arasındaki uyumun-iyiliği (goodness-of-fit) kriterinin karşılaştırılmasında kullanılır. Test istatistiği  $\chi^2 = \sum((o_i - e_i)^2 / e_i)$  biçimindedir. Burada  $o_i$  ve  $e_i$ , meydana gelen ölçümlerin gözlenen ve beklenen frekanslarıdır.

NIST-800-22 testinde test için gerekli bazı parametreler dışarıdan belirlenir. En önemli parametrelerden birisi ise P-değeridir. P-değeri teste tabi tutulan ikili dizilerin rasgeleliğinin bir ölçütü olarak kabul edilmektedir. P-değeri 1'e yakın ise ikili sayı dizisinin rasgele özelliği artarken, 0'a yakın olma durumunda rasgele özelliği azalır [11].

#### 2.6.2.1. Frekans (Monobit) Testi

Bu test rasgele bit dizisindeki "1" ve "0" dengesini inceler yani gerçek bir RSÜ'de olması gerektiği gibi "0" ve "1" sayılarının yaklaşık olarak aynı olup olmadığını ölçer. Bu sebeple bu testi geçemeyen RSÜ'nün diğer testlerde başarılı olma ihtimali çok azdır. Bu testin sonucu dizinin rasgele olup olmadığı ile ilgili birçok bilgi sağlar.

Test için herhangi bir parametre gerekmemektedir fakat dizinin uzunluğu en az 100 bit olmalıdır. Referans [6]'da belirtilen denklemler ile P değeri hesaplanır.  $P < 0.01$  olması durumunda dizinin rasgele olmadığı kabul edilir.

#### 2.6.2.2. Blok frekans testi

Bu testte tek parametre olarak M blok uzunluğu kullanılır. Tüm bit dizisini inceleyen Frekans testinin aksine burada rasgele bit dizisini M bitlik bloklara ayırarak bloklar içerisindeki "1" oranı incelenir. M değerinin 1 olarak seçilmesi frekans testi yapıldığı anlamına gelir. Her bir blok içerisindeki "1" değerlerinin beklenen oranı  $M/2$ 'dir. Teste tabi tutulacak dizinin blok uzunluğu  $n=100$  ve bit uzunluğu en az  $M=20$  olarak alınması gerekmektedir. Test istatistiklerini ve referans dağılımını hesaplamak için ki-kare dağılımı kullanılır.

#### 2.6.2.3. Akış testi

Yinelemeler testi olarak da bilinen akış testi bir rasgele bit dizisi içerisindeki ardışık 0 ve 1 bloklarına bakılır. Gerçek rasgele dizilerde istenmeyen durum olan "1" ve "0" ların birbirini yinelemesi gibi bir durumun olup olmadığı kontrol edilir. Akış testinin amacı ikili düzen

içerisindeki “0” ve “1” ler arasındaki değişimin hızlı veya yavaş olduğuna karar verecek hesaplamayı yapmaktır.

#### 2.6.2.4. Blok İçindeki Birlerin En Uzun Akış Testi (Longest run of ones)

Testin amacı uzunluğu  $M$  bit olan grupta en uzun birlerin uzunluğunun rasgele bir dizide beklenen en uzun çalışma süresinin uzunluğu ile tutarlı olup olmadığını test etmektir. Alınan dizide  $M$  bloğu  $n$  tane bloğa bölünür. Bu bloklar içerisindeki en uzun birler grubuna bakılır. Beklenen değerlerle bu değerler kıyaslanır sapma miktarı ölçülür. Bu test için kullanılan dağılım ki-kare dağılımıdır.

Dizideki en uzun bir yinelemesindeki düzensizlikler en uzun sıfır yinelemesinde de görülebileceğinden dolayı sadece birlere test yapılması yeterlidir. Test parametreleri Tablo 2.2’de gösterildiği gibi olmalıdır.

**Tablo 2.2.** Blok içindeki Birlerin En Uzun Akış Testi için test parametreleri

Minimum n	M
128	8
6272	128
750000	$10^4$

#### 2.6.2.5. İkili matris rankı testi

Bu test, bütün dizinin ayrışık alt matrislerinin ranklarına odaklanan bir testtir. Testin amacı, orijinal dizinin alt dizileri arasında doğrusal bir bağıntı olup olmadığını kontrol etmektir. Bu test ayrıca DIEHARD test suite’inde de bulunmaktadır.

Test içerisinde sıra  $M \times M$ -bitlik alt matrisler halinde parçalanır ve her bir alt matrisin rankı hesaplanır. Sıra ile oluşturulan matrislerin ranklarının frekansları hesaplanır, beklenen frekansla karşılaştırılır ve ciddi bir sapma olup olmadığı kontrol edilir.

#### 2.6.2.6. Ayrık fourier dönüşüm testi

Spektral test olarak da adlandırılan Ayrık Fourier Dönüşüm testi dizinin ayrık fourier dönüşümünün tepe yüksekliklerine (peak heights) odaklanan bir testtir. Sıra ile oluşturulan



matrislerin ranklarının frekansları hesaplanır, beklenen frekansla karşılaştırılır ve ciddi bir sapma olup olmadığı kontrol edilir. Fourier testi bit serilerindeki periyodik özellikleri belirler ki bunlar da rasgeleliğin varsayımından bir sapma belirlerler.

#### **2.6.2.7. Örtüşmeyen şablon eşleştirme testi**

Bu testin amacı test öncesi belirlenen  $n$  bitlik veri uzunluğuna sahip rasgele sayı dizisindeki seçilen  $m$  bitlik blokları kontrol ederek periyodik olmayan örnekleri tespit etmektir. Bu örnek sayısına göre dizinin periyodikliği test edilir. Örnek dizi bloklarının tekrarı durumunda bir sonraki bloğun ilk bitinden devam edilir. Eğer örnek dizi bloklarına rastlanmazsa pencere bir bit ötelenerek arama devam ettirilir.

#### **2.6.2.8. Örtüşen şablon eşleştirme testi**

Örtüşmeyen şablon eşleştirme testi ile aynıdır. Farkı, eğer bir örüntü (dizi tekrarı) bulunursa pencere bulunan örüntüden sonraki ilk bit yerine sadece o an bulunan pozisyondan bir sonraki bite yeniden konumlanır ve taramaya devam edilir.

#### **2.6.2.9. Maurer “evrensel istatistik” testi**

Bu test 1992 yılında Princeton Üniversitesi Bilgisayar Bilimi Bölümü’ndeki Ueli Murer tarafından geliştirilmiştir. Testin amacı, dizinin bilgi kaybı olmadan anlamlı bir şekilde sıkıştırılıp sıkıştırılamayacağını tespit edilmesidir. Anlamlı bir biçimde sıkıştırılabilen bir dizi rasgele bir dizi olarak kabul edilmez.

#### **2.6.2.10. Lempel-Ziv sıkıştırma testi**

Bu test, bir dizide bulunan kümülatif olarak ayrık örneklerin (kelimelerin) sayısına odaklanır. Amaç, test edilen dizinin ne kadar sıkıştırılabildiğinin belirlenmesidir. Eğer dizi anlamlı bir şekilde sıkıştırılamazsa dizi rasgele olarak kabul edilir.

Testte rasgele dizi Lempel-Ziv algoritması kullanılarak sıkıştırılır. Eğer sıkıştırma beklenen sonuca göre fazlaysa dizinin rasgele olmadığı kabul edilir. Bir üretici test etmek için çok sayıda dizi bu şekilde test edilir.

Lempel-Ziv testinin frekans ve koşu testlerini, diğer sıkıştırma testlerini ve muhtemelen spektral testini kapsadığı düşünülür, ama rasgele ikili matris rankı testiyle kesişebilir. Bu test entropi testiyle özdeştir, hatta Maurer’in evrensel istatistik testiyle daha da özdeştir. Bununla birlikte, Lempel-Ziv testi doğrudan doğruya modern bilişim teorisini tanımlayan sıkıştırma olgusunu içerir.

#### **2.6.2.11. Doğrusal karmaşıklık testi**

Bu test genel olarak, Doğrusal Geri beslemeli Kayan Yazmaç (DGBKY) uzunluğuna odaklıdır. Testin amacı, rasgele dizinin yeterince karmaşık olup olmadığının incelenmesidir. Rasgele diziler daha uzun DGBKY ile karakterize edilir. Çok kısa DGBKY'ler büyük olasılıkla rasgele olmayan dizileri gösterir.

Testte dizi M bitlik bloklara ayrılır ve bloktaki bitlerin lineer karmaşıklıkları Berlekamp-Massey algoritması kullanılarak hesaplanır. Hesaplanan lineer karmaşıklıkların beklenen dağılıma uygun olup olmadıklarına bakılır. Testte kullanılan referans dağılım ki-kare dağılımıdır. Testin geçerli olabilmesi için dizinin boyu en az 1.000.000; blok uzunluğu da 500 ve 5000 arasında olmalıdır.

#### **2.6.2.12. Seri testi**

Bu test, istatistiksel dağılımların tek düzeliğini test etmektedir. Tüm dizideki m-bitlik örtüşen olası örüntülerin frekansına odaklanır. Testin amacı,  $2^m$  adet m-bitlik örtüşen örüntülerin sayısının, rasgele bir dizide beklenen sayıya ne kadar yakın olduğunun araştırılmasıdır.

Rasgele dizilerde tek-biçimlilik (uniformity) önemli bir özelliktir. Bu tek-biçimlilikte her m-bitlik örüntünün diğer m-bitlik örüntüler gibi ortaya çıkma olasılığının aynı olması beklenir. m=1 için Seri Test Frekans Testine denk düşer. Bu test sonucunda, P-değeri1 ve P-değeri2 olmak üzere iki değer elde edilir.

#### **2.6.2.13. Yaklaşık entropi testi**

Bu testte amaç, seri testinde olduğu gibi tüm muhtemel örtüşen m bitlik örnek dizinin frekansının incelenmesidir. Seri testinden farklı dizi için beklenen frekansın iki ardışık veya bitişik uzunluktaki (m ve m+1) örtüşen blokların frekanslarını karşılaştırmasıdır.

Testte kullanılan referans dağılımı ki-kare dağılımıdır. Diziden kesişen n tane m-bitlik blok üretilir. Bu blokların frekansları ve entropisi hesaplanır. m ve m+ 1 bit için hesaplanan değerlerin farkına bağlı test istatistiği hesaplanır. Bu farkın düşük olması rastgelelikten uzaklığı gösterir.

#### **2.6.2.14. Kümülatif toplamlar testi**

Bu testte amaç, rasgele bir dizi için kümülâtif toplamın beklenen davranışı için kısmi alt blokların kümülâtif toplamının çok büyük veya çok küçük olup olmadığının belirlenmesidir. Test edilecek dizide 0 değerleri için -1, 1 değerleri için +1 verilerek çeşitli şekillerde bitler

toplamı elde edilir ve rastgeleliğin sağlanıyor olması adına sonucun 0'a yakın olması beklenir. Test sonucu yüksek değerler verirse bu dizide çok fazla "0" veya çok fazla "1" olduğunu belirtirken, sonucun düşük çıkması "1" ler ve "0" ların dizide yaklaşık olarak aynı oranda bulunduğunu ifade eder.

Dizi uzunluğunun en az 100 bit olması başlangıç için yeterlidir.

#### **2.6.2.15. Rasgele gezinimler testi**

Bu test, kümülatif toplam rasgele yürüyüşündeki tam olarak K ziyaretlik döngünün sayısına odaklanır. Kümülatif toplam rasgele yürüyüşü, (0,1)'lerden oluşan dizinin (-1, +1) olarak düzenledikten sonra kısmi toplamlarından elde edilir. Bir rasgele yürüyüş döngüsü, rasgele kabul edilen bir yerden başlayıp tam bir döngü olana kadar belli bir uzunluktaki adım dizisinden oluşur.

Bu testin amacı, bu döngü sırasında rasgele dizide beklenen sapmadan kaynaklanan belirli bir durumun ziyaretlerinin sayısının belirlenmesidir. Bu test aslında 8 testten ve çıkarımlarından oluşan bir seridir.

#### **2.6.2.16. Rasgele gezinimler değişken testi**

Bu testte, bir kümülatif toplam rasgele yürüyüşünde, özel bir durumun ziyaret edilme sayısı ölçülür. Testin amacı, bir rasgele yürüyüşte özel durumların beklenen ziyaret sayısındaki sapmalarının gözlenmesidir. Bu test aslında 18 testin (ve çıkarımların) serisidir.

### 3. SON İŞLEM ALGORİTMALARI

Kriptografik uygulamalar için oldukça önemli yere sahip olan rasgele sayılar bu uygulamalarda başlangıç vektörü, özel anahtar ve gizli anahtarların oluşumunda ve aynı zamanda da SRSÜ'de tohum değerinin oluşturulmasında kullanılmaktadır. Üretilen bu değerler uygulamalar için güvenlik açısından kritik değere sahiptir. Yani kullanılan rasgele sayılar sistemin güvenliğini doğrudan etkilemektedir. Fakat RSÜ'lerden elde edilen bit dizilerinin bazıları bu sistemler için istenmeyecek zayıf istatistiksel özellikler göstermektedir. Bu nedenle üretilen dizilere bu zayıflığın giderilmesi amacıyla çeşitli son işlem algoritmaları uygulanmaktadır.

Son işlem, üretilen bit dizilerini düzenli (uniform) dağılımlı hale getirmek için kullanılan mantıksal ve/veya aritmetik işlemdir. Bir diğer amacı ise saldırgan kurcalamaları ve çevresel değişikliklere karşı dirençli hale getirmesidir. Son işlem algoritmasına bağlı olarak üreticinin güvenliği artmaktadır. Ancak kullanılan son işlemler GRSÜ'den elde edilen çıkış bit oranını düşürmektedir. Bu yüzden gerçek RSÜ tasarımlarında güçlü gürültü kaynakları kullanılarak algoritmik son işleme ihtiyaç duymadan iç rasgele sayıları doğrudan çıkışa veren mimariler tercih edilmektedir [13]. Son işlemin rasgele sayılar için bu denli önemli olmasından dolayı çok eskilerden beri bu konuda araştırmalar yapılmakta ve yeni yöntemler duyurulmaktadır. Bu bölümde geçmişten günümüze önerilmiş olan bazı son işlem algoritmalarından bahsedilecektir.

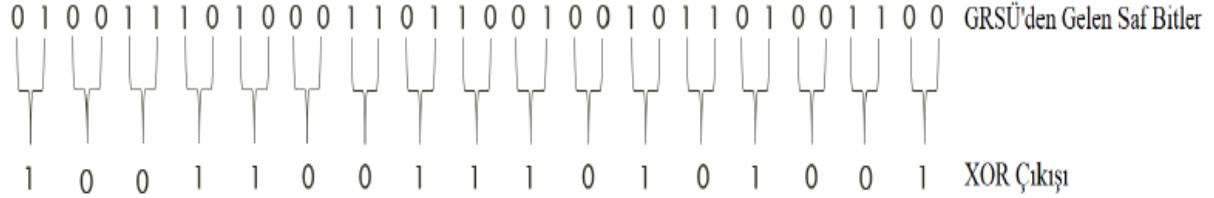
#### 3.1. XOR Algoritması

XOR algoritması uygulanmasının kolay olmasından dolayı sık sık son işlem algoritması olarak kullanılan en basit yöntemdir. Rasgele sayı üretici tarafından üretilen bitlerden kaynaklanan biası azaltmak için yaygın olarak kullanılır [14]. Rasgele sayı üreticinden elde edilen bit dizisi  $n$  bitlik ( $n=2$ ) bloklara ayrılır ve bu bloklara kendi içerisinde XOR işlemi uygulanarak 1 çıkış biti üretilir. Çıkış bit verimini  $1/n$  kez azaltmaktadır. Aşağıdaki tabloda XOR son işlem çıkış durumları verilmiştir.

**Tablo 3.1.** XOR Son İşlem Çıkışı

Bit Dizisi	XOR Çıkışı
00	0
01	1
10	1
11	0

Örnek verecek olursak elimizdeki aşağıdaki gibi bir saf bit dizisi olduğunu düşünelim. Burada yapılacak olan şey diziyi 2 bitlik bloklara ayırarak her bloğu kendi içinde XOR'lamak olacaktır. Aşağıdaki şekilde bu işlem görsel olarak anlatılmıştır.



**Şekil 3.1.** XOR Son İşlem Algoritması Örneği.

Görüldüğü gibi örnek olarak alınan 34 bitlik bir bit dizisi XOR işleminden sonra yarı yarıya yani 17 bit olarak çıkış vermiştir. XOR son işlem algoritması basit ve sabit bit çıkış hızı sağlamasından dolayı tercih edilse de sistemin güvenliğini sağlama konusunda zayıf kalmaktadır.

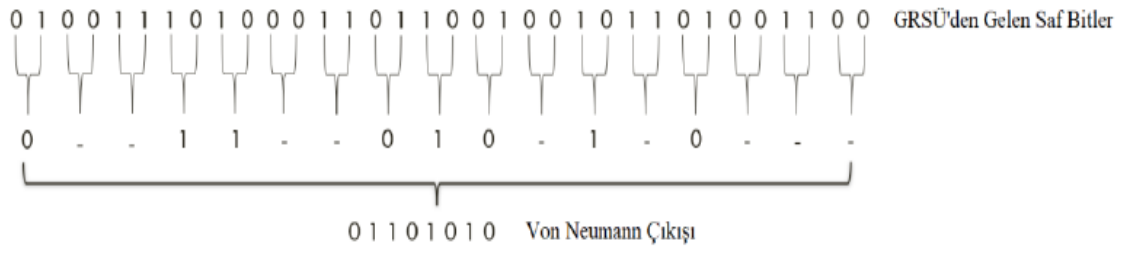
### 3.2. Von Neumann Algoritması

En eski ve en basit son işlem algoritmalarından olan Von Neumann [15] algoritması bit dizisindeki düzensizlikleri gidermeyi amaçlar. XOR işleminde olduğu gibi üreteçten gelen dizi 2 bitlik bloklara ayrılır. Tabloda gösterildiği gibi bit dizisi (1,0) ise çıkış biti 1, eğer bit dizisi (0,1) ise çıkış biti 0 olarak üretilir. (0,0) ve (1,1) bit dizileri dikkate alınmaz. Sistemin çıkış bit hızı bu dikkate alınmayan bit dizilerinden dolayı sabit değildir. Tablo 3.2’de Von Neumann algoritmasının çıkış durumları verilmiştir.

**Tablo 3.2.** Von Neumann Son İşlem Çıkışı

Bit Dizisi	Von Neumann Çıkışı
00	Çıkış Yok
01	0
10	1
11	Çıkış Yok

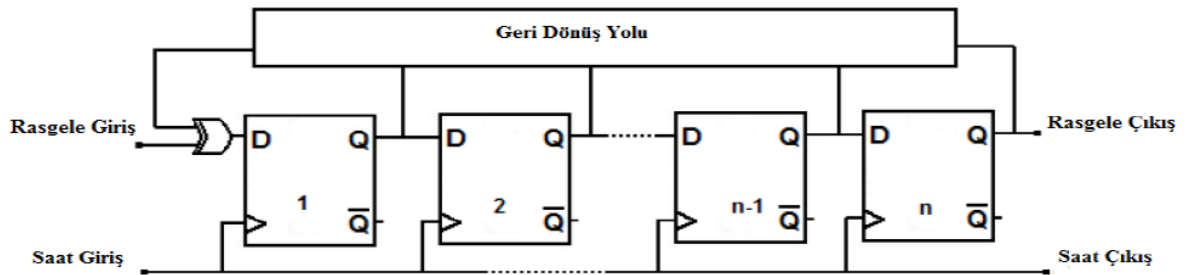
Yine bir örnek üzerinden anlatacak olursak aşağıdaki gibi bir saf bit dizimiz olsun. XOR son işlem yönteminde yapıldığı gibi burada da diziyi n=2 bitlik bloklara ayırarak her bir bloğa Von Neumann işlemi uygulanır ve sonuç aşağıdaki gibi olur.



**Şekil 3.2.** Von Neumann Son İşlem Algoritması Örneği.

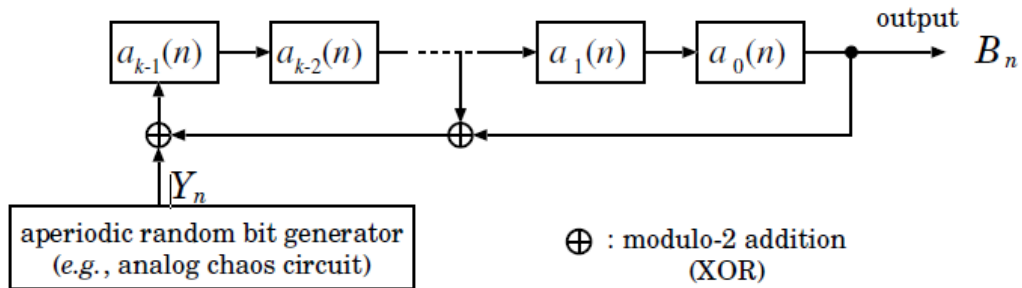
### 3.3. Doğrusal Beslemeli Kayan Yazmaç

Doğrusal Geri Beslemeli Kayan Yazmaç birçok uygulamada son işlem algoritması olarak kullanılmıştır. Bunun sebepleri ise donanım üzerinde uygulamasının kolay ve kullanışlı olması, iyi istatistiksel özelliklere sahip diziler üretmesi ve aynı zamanda geniş tekrarlılama periyodudur. Genel yapısı Şekil 3.3'te gösterildiği gibidir.



**Şekil 3.3.** Doğrusal Beslemeli Kayan Yazmaç Genel Tasarımı [6].

Örneğin [16]'da üretilen rasgele sayıların periyodikliğini gidermek ve iyi istatistiksel özellikler göstermesi sağlamak için aşağıdaki şekilde bir yapı kullanılmıştır;

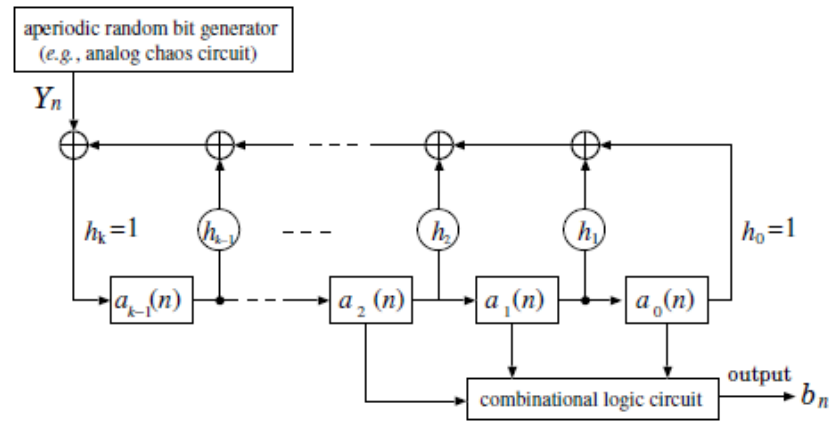


**Şekil 3.4.** DBKY ile kullanılan RSÜ'nün genel yapısı [16].

Şekilde görüleceği üzere sistem bir rasgele sayı üretici, k hafıza hücresi ve XOR dan oluşmaktadır. Her bir zaman biriminde her hafıza hücresi bir sağa kaydırılır, ayrıca  $a_{k-1}(n)$  aşağıdaki denklemdeki gibi güncellenir.

$$a_{k-1}(n+1) = a_0(n) \oplus a_1(n) \oplus Y_n \quad \text{Denklem (2)}$$

Bir diğer çalışmada [17] ise bir rastgele sayı üretici, bir doğrusal geri beslemeli kayan yazmaç (DBKY) ve bir kombinasyonel mantık devresi kullanan bir rastgele sayı üretici önerilmiştir. DBKY ve kombinasyonel mantık devresi, kötü istatistiksel özelliklere sahip periyodik bit dizilerinin sonradan işlenmesinde bir rol oynar. Genel yapısı ise Şekil 3.5'te gösterildiği gibidir.



Şekil 3.5. [17]'de kullanılan DBKY ile RSÜ'nün genel yapısı.

Burada ise referans [16] ile aynı işlemler tekrar edilirken ayrıca  $a_{k-1}(n)$  aşağıdaki denklemdeki gibi güncellenir.

$$a_{k-1}(n+1) = a_0(n) \oplus h_1 a_1(n) \oplus \dots \oplus h_{k-1} a_{k-1}(n) \oplus Y_n \quad \text{Denklem (3)}$$

Yapılan çalışmalar göstermektedir ki önerilen son işlem algoritması k büyük olduğunda çok etkili olmaktadır.

### 3.4. Lojistik Harita

Lojistik harita; donanım üzerinde uygulaması kolay olan, bir başlatma koşulu ve bir kontrol parametresi gerektiren birinci dereceden bir denklemdir [18]. Lojistik haritanın yapısı en basit hali ile aşağıdaki denklemdeki gibidir.

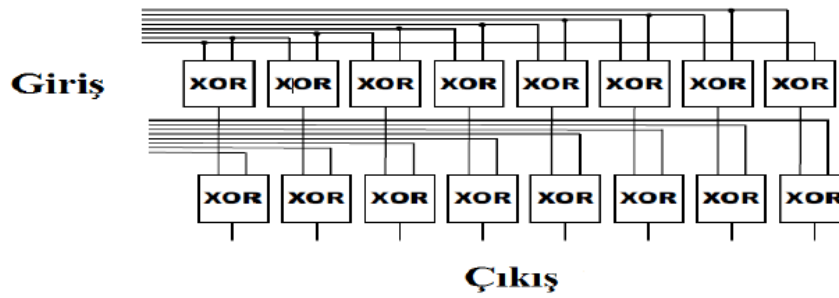
$$x(n+1) = r x(n)(1 - x(n))$$

Denklem (4)

Burada  $r$  sistem parametresi,  $n$  ise tekrarlama sayısıdır. Aynı zamanda  $0 \leq x(n) \leq 1$  koşulunun sağlanması gerekmektedir.  $x(n)$  değeri  $x(0)$  başlangıç değerinin seçilmesi ile hesaplanır. Sistem  $r$  parametresinin değerine göre farklı davranışlar sergileyebilmektedir. Aynı zamanda lojistik haritada başlangıç koşulunun değiştirilmesi ile farklı sayı dizileri üretilebilmektedir. Bu sebeple lojistik harita kullanan sistemler saldırılara karşı daha güvenlidir. Yüksek güç tüketimi sebebi ile dezavantajlı olmasına rağmen birçok son işlem algoritması veri çıkış oranını azaltırken lojistik harita veri çıkış oranını değiştirmemektedir.

### 3.5. H Fonksiyonu

Bir diğer son işlem yöntemi ise Dichtl tarafından önerilen, Quasigroup dizi dönüşümüne dayalı olan H son işlem algoritmasıdır [19]. Şekil 3.6'da görülebileceği gibi 16 bit giriş bitinden 8 bit çıkış elde edilir.



Şekil 3.6. H son işlem fonksiyon [6].

Son işlem için giriş bitleri  $a_0, a_1, a_2, \dots, a_{15}$  olarak tanımlanmıştır. Denklem (5) ile  $b_0, b_1, \dots, b_7$  8 bit tanımlanır.  $c_0, c_1, \dots, c_7$  çıkışı ise Denklem (6) ile tanımlanır.

$$b_i = a_i \oplus a_{(i+1) \bmod 8} \quad \text{Denklem (5)}$$

$$c_i = b_i \oplus a_{(i+8)} \quad \text{Denklem (6)}$$

16 bit giriş dizisi  $a_1$  ve  $a_2$  olarak ikiye ayrılır ve aşağıdaki denklem uygulanır.

$$H(a_1, a_2) = \text{XOR}(\text{XOR}(a_1, \text{rotateleft}(a_1, 1)), a_2) \quad \text{Denklem (7)}$$



Bir örnek ile açıklamak gerekirse (1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1) rasgele sayı üreticinden gelen saf bit dizisi olsun. Denklem (6) kullanılarak b dizisi aşağıdaki gibi elde edilmektedir.

$$\begin{aligned}
 i = 0 \text{ için; } b_0 &= a_0 \oplus a_1 = 1 \oplus 0 = 1 \\
 i = 1 \text{ için; } b_1 &= a_1 \oplus a_2 = 0 \oplus 1 = 1 \\
 i = 2 \text{ için; } b_2 &= a_2 \oplus a_3 = 1 \oplus 1 = 0 \\
 i = 3 \text{ için; } b_3 &= a_3 \oplus a_4 = 1 \oplus 0 = 1 \\
 i = 4 \text{ için; } b_4 &= a_4 \oplus a_5 = 0 \oplus 0 = 0 \\
 i = 5 \text{ için; } b_5 &= a_5 \oplus a_6 = 0 \oplus 0 = 0 \\
 i = 6 \text{ için; } b_6 &= a_6 \oplus a_7 = 0 \oplus 1 = 1 \\
 i = 7 \text{ için; } b_7 &= a_7 \oplus a_0 = 1 \oplus 1 = 0
 \end{aligned}$$

Bir sonraki aşamada ise Denklem (7) kullanılarak çıkış dizisi olan c dizisi elde edilmektedir. Oluşan çıkış bit dizisi (1, 0, 1, 1, 0, 1, 0, 1) şeklinde olmaktadır.

H fonksiyonu daha sonra geliştirilmeye devam edilerek Denklem (8)'da gösterilen H2 ve Denklem (9)'da gösterilen H4 fonksiyonlar tanımlanmıştır.

$$H2(a1, a2) = \text{XOR} ( \text{XOR} ( \text{XOR} (a1, \text{rotateleft} (a1, 1) ), \text{rotateleft} (a1, 2) ), a2 ) \quad \text{Denklem (8)}$$

$$H4(a1, a2) = \text{XOR} ( \text{XOR} ( \text{XOR} ( \text{XOR} (a1, \text{rotateleft} (a1, 1) ), \text{rotateleft} (a1, 2) ), \text{rotateleft} (a1, 4) ), a2 ) \quad \text{Denklem (9)}$$

### 3.6. SHA-256

Referans [20]'de halka osilatörler ile tasarlanmış bir GRSÜ ile birlikte son işlem algoritması olarak SHA-2 hash ailesinden olan SHA-256 kullanımı önerilmiştir. Kurulan tasarıma göre 512 rasgele biti giriş olarak alan SHA-256 algoritmasına yollanmak üzere GRSÜ'den gelen rasgele bitler 8 bitlik bloklara ayrılarak, her blok 64 byte genişliğinde olan FIFO buffer içerisinde saklanır. Hash algoritmasının sonucunda ise 8 tane 32 bitlik kelime (256 bit toplam) üretilir.

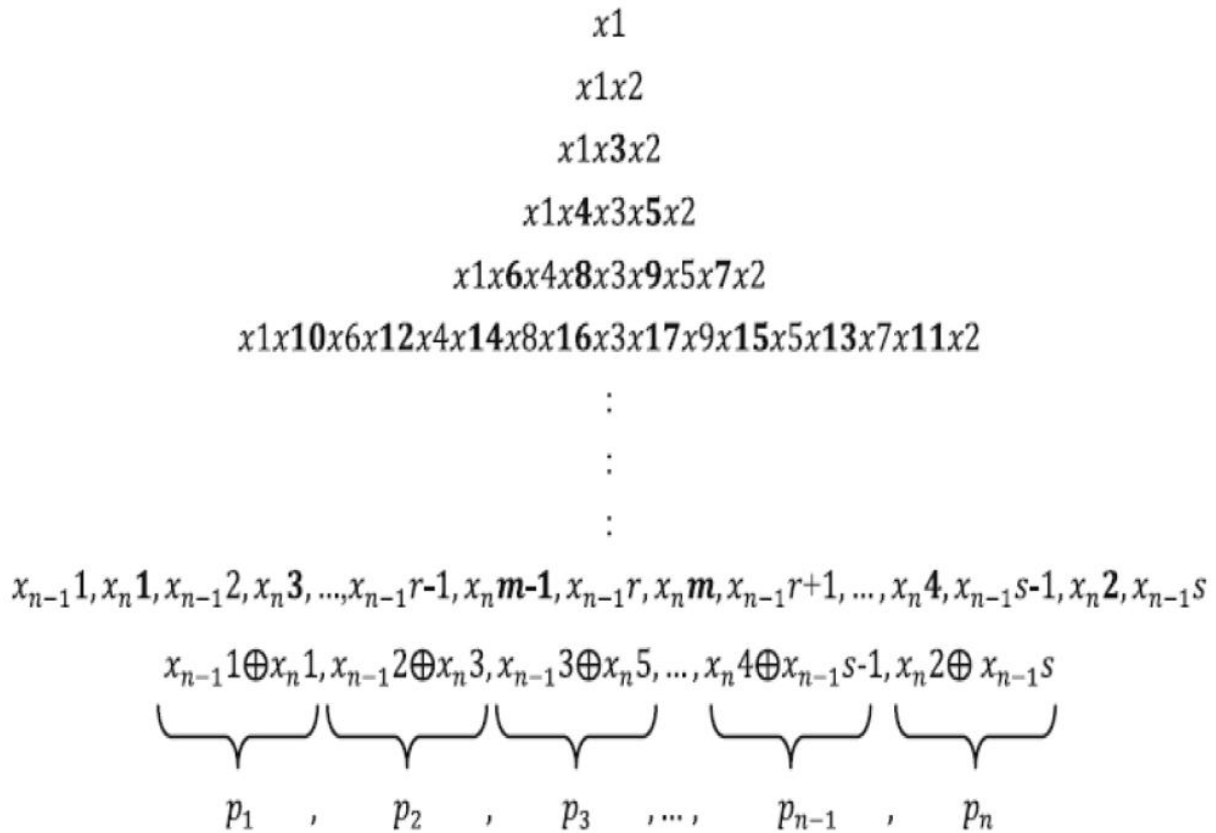
### 3.7. Mixing Bits in Steps and XORing

Önerilen bir diğer son işlem algoritmasında rasgele sayı üretmek için mikrofon aracılığı ile standart bir bilgisayarın, yeni nesil bir dizüstü bilgisayarın, tabletlerin ya da akıllı telefonların ses kartı kullanılarak bir rasgele gürültü sinyali sağlanmıştır [21]. Son işlem

algoritması olarak da otokorelasyonu azaltmanın ve çıktı bit dizisinin toplam entropisini artırmanın yenilikçi bir yolu olduğu söylenen Mixing Bits in Steps and XORing of Adjacent Bits yöntemi kullanılmıştır. Kurulan sistemin 3 aşaması bulunmaktadır. İlk aşama, gürültü kaynağından elde edilen analog sinyalin sayısallaştırılmasıdır. İkinci aşamada bu sinyaller zayıflığın azaltılması amacı ile son işlem algoritmasından geçirilerek dahili rasgele sayılar üretilir. Üçüncü aşamada ise harici rasgele sayılar elde edilir.

Karıştırma algoritması gelen ilk iki biti alır. İkinci aşamada üçüncü bit alınır ve ilk iki bitin arasına yerleştirilir. Üçüncü aşamada ise dördüncü bit 1. ve 3. bitin arasına yerleştirilir, beşinci bit ise 3. ve 2. bit arasına yerleştirilir. Bu aşamada görünüm  $x_1; x_4; x_3; x_5; x_2$  şeklindedir. Dördüncü adımda, altıncı bit 1. ve 4., yedinci 2. ve 5., sekizinci 4. ve 3. ve dokuzuncusu 5. ve 3. arasına yerleştirilir. Bu, dördüncü adımı tamamlar ve ortaya çıkan düzen  $x_1; x_6; x_4; x_8; x_3; x_9; x_5; x_7; x_2$  şeklinde olur. Tüm yerleştirmeler yapıldıktan sonra komşu olan bitler XOR işlemine tabi tutularak rasgele bit dizisi son halini almış olur. Şekil 3.7’de yöntemin adım adım uygulanışı gösterilmektedir.

Bu uygulama yavaş olmasına rağmen istatistiki açıdan oldukça iyi sonuçlar vermiştir. Makalede aynı zamanda yavaşlığı azaltmak için de farklı bir yöntem önerilmektedir.



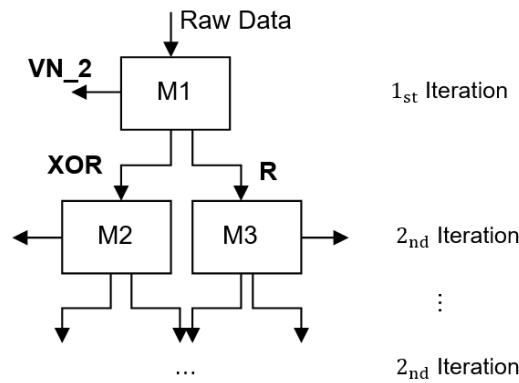
**Şekil 3.7.** Mixing Bits in Steps and XORing of Adjacent Bits yöntemi gösterimi.

### 3.8. N Bits Von Neumann

İlk olarak J.von Neumann tarafından 1951'de önerilen Von Neumann algoritması hala birçok rasgele sayı üreticinde son işlem algoritması olarak kullanılmaktadır. Veri çıkış oranını düşürdüğünden dolayı geliştirilmeye devam edilmektedir. Örneğin 1972'de ise Elias N bits Von Neumann [22] algoritmasını önermiştir. Orijinal Von Neumann (VN\_2) algoritmasında 2 bit ile işlem yapılırken VN\_N algoritmasında N bit ile işlem yapılmaktadır. Burada N'nin değerini artırarak, VN\_N, yalnızca 1 kez işleme ile Shannon entropisine yakın yüksek çıktı oranı elde edebilir. Bununla birlikte, teorik değer ile gerçekçi değer arasında bir çıktı oranı farkı vardır. Teorik çıktı oranı  $N = 4, 8, 16$  olduğunda sırasıyla %49,2, %68,2, %81,0'dır. Gerçekçi çıktı oranı sırasıyla %40,6, %55,2 ve %73,1'dir. Bu arada,  $N = 17$ 'nin ötesinde verimlilik doymuş hale gelir. Bu yüzden uygun N değerlerinin seçilmesi gerekiyor.

### 3.9. Iterating Von Neumann (IVN)

Kaynak [23]'de ise Iterating Von Neumann (IVN) yöntemi sunulmuştur. Bu yöntemde IVN atılan bitleri toplayıp onları bir sonraki aşamaya yollayarak tekrardan kullanmaktadır. Aşağıdaki şekilde de görüleceği gibi her IVN modülü üç blok içerir: VN\_2 (giriş çifti "01" veya "10" ise, ardından "0" veya "1" çıkışı verir, diğerleri çıkış olmaz), XOR ("00" veya "11" çifti oluşursa çıkış '0', '01' veya '10' çiftleri daha sonra '1' çıktısı oluşur), R ('00' ise '0', '11' ise '1' çıktı, diğerleri çıktı olmaz). XOR ve R işleminden sonra çıkış verisi tekrardan işlenmemiş veri olarak bir sonraki yinelemeye verilir.



Şekil 3.8. IVN tasarımı.

### 3.10. High-Throughput Von Neumann

Kaynak [24]'de VN\_N algoritması geliştirilerek teorik değer ile gerçekçi değer birbirine yakınlaştırılması amaçlanmıştır. Burada herhangi bir N ( $N \geq 2$ ) değeri için bir eşleşme

algoritması hazırlanmıştır. Böylece örneğin VN\_2 algoritması 25.0% veri çıkış oranına sahipken VN\_4 40.6% oranında çıkış vermiştir. Makalenin devamında VN\_N+waiting adı verilen bir bekleme stratejisi de önerilerek VN\_4+waiting için çıkış oranı 46.9% 'a kadar çıkarılmıştır. Böylece teorik çıktı oranı olan 49.2% a oldukça yaklaşılmıştır.

### 3.11. Keccak Algoritması

Kriptolojik uygulamalar için güvenliği arttırmak amaçlı kullanılan bir diğer son işlem yöntemi ise özet fonksiyonlar için son standart olan Keccak Algoritması. Selman Yakut, Taner Tuncer ve Ahmet Bedri Özer'in yayınladığı makalede [25] güvenli rasgele sayılar üreten Keccak algoritması ve tahmin edilebilirliği ve yeniden üretimi engelleyen ek girdiler üreten halka osilatörler beraber kullanılmıştır. Çalışmada üretici daha verimli hale getirmek için Keccak algoritması yeniden düzenlenip kullanılmıştır. Önerilen üreteçte, ek girdi olarak 512 bit ham gerçek rasgele sayı alınıp 1024 bitlik gerçek rasgele sayı dizisi üretilmektedir.

Aynı yazarların yayınladığı bir diğer makalede ise yine Keccak algoritması kullanılarak 1600 bit giriş için sıfır veri kaybı ile 1600 bitlik rasgele sayı dizisi üretilmiştir [26].

Keccak algoritmasında ek girdiler kullanılarak rasgele sayıların yeniden üretilmesi ve tahmin edilebilirlik önlenmiştir. Yapılan testler göstermektedir ki kriptografik uygulamalar için gerekli olan güvenlik gereksinimleri karşılanmaktadır. Algoritmanın en önemli avantajı ise veri kaybına sebep olmamasıdır.

### 3.12. SBOX

SBOX ile amaç bir tablo vasıtası ile kimin kiminle yer değiştireceğinin belirlenmesidir. Oluşturulan S-kutusunun tipine veya kategorisine bakılmaksızın, herhangi bir S-kutusunun etkili olabilmesi için belirli özellikler göstermesi gerekir. İstedığınız herhangi bir yer değiştirme şemasını bir araya getirip iyi bir S-kutusu oluşturamazsınız. Bir S-kutusu karışıklık (confusion) ve difüzyon (diffusion) sağlamalıdır. S-kutularında bu girişleri üretmenin bir yolu, n giriş bitlerini m çıkış bitlerine eşleyen doğrusal olmayan bir Boole fonksiyonu (Boolean function) oluşturmaktır.

[27]'de önerilen son işlem yönteminde DES için hazırlanan 8 SBOX kullanılmıştır. DES S-Kutuları 6 bit giriş alır ve 4 bit çıkış verir. Görüldüğü üzere 6 bitin dışta olanları (1. ve 6. bit) satırları, iç tarafta kalan bitler (2.,3.,4.,5.) sütunları oluşturmaktadır. Buna göre seçilen satır ve sütundaki sayı da çıkışı verir.

Örnek ile açıklamak gerekirse, aşağıdaki bit dizisi ile adım adım yapılacak olan işlemleri açıklayalım. Örnek için kullanılan S-kutuları [27]'de verilmiştir.

0 1 0 0 1 0   1 1 0 1 0 0   1 0 1 0 0 0   1 0 1 0 1 0   1 0 1 1 1 1   0 1 0 0 0 1   0 1 1 1 0 1   0 1 0 1 0 0

S1                      S2                      S3                      S4                      S5                      S6                      S7                      S8

1. Adım : İlk 6 bitlik bloğu ele aldığımızda 1. Ve 6. biti alıp satırımızı buluyoruz. (00 -> 0)
2. Adım : Kalan bitler ile sütunu belirliyoruz. (1001 -> 9)
3. Adım : S1 kutusunda 1. ve 2. adımda denk gelen değeri buluyoruz. Tabloda karşılık gelen değer in ikili gösterimi bizim çıkış dizimizi oluşturmaktadır. ( 10 -> 1010 )

S1																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
01	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
10	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
11	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

4. Adım : Tüm blokları işleyene kadar bu işlemi tekrar ediyoruz ve 48 bit giriş dizisi için 32 bit çıkış dizisini üretiyoruz.

0 1 0 0 1 0   1 1 0 1 0 0   1 0 1 0 0 0   1 0 1 0 1 0   1 0 1 1 1 1   0 1 0 0 0 1   0 1 1 1 0 1   0 1 0 1 0 0

S1                      S2                      S3                      S4                      S5                      S6                      S7                      S8

1010                      1100                      1000                      1011                      1101                      0110                      1000                      0011

SBOX son işlem algoritması NIST testlerinde başarılı sonuçlar elde etmesinin yanı sıra otokorelasyon ve karmaşıklık ölçümlerinde de başarılı sonuçlar elde etmiştir. Ayrıca entropi değeri 1 olarak elde edilmektedir. Bu başarılı sonuçlardan dolayı kriptografi gibi birçok alan için kullanıma uygun olduğu görülmektedir.

### 3.13. Resilient Fonksiyonu

(n, m, t)-resilient fonksiyonu  $F(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_m)$  şeklinde,  $Z_2^n \rightarrow Z_2^m$  olan ve  $i_1, \dots, i_t$  koordine eden herhangi bir t için  $Z_2$  'den herhangi bir  $a_1, \dots, a_t$  sabiti için ve ortak etki alanının herhangi bir y ögesi için  $\text{Prob} [\mathbb{Z}(\mathbb{Z}) = \mathbb{Z} | \chi_{i_1} = a_1, \dots, \chi_{i_t} = a_t] = \frac{1}{2^m}$  özelliğini sağlayan bir fonksiyondur.

Daha basit bir ifade ile açıklayacak olursak; fonksiyon girdi olarak verilen herhangi bir değerinin bilinmesi kişinin çıktıda rasgele bir tahminden fazlasını yapmasına izin vermez. Bu yönüyle oldukça güvenlidir ve birçok kriptografik uygulamada kullanılmaktadır.

Sunar, Martin ve Stinson tarafından yapılan bir çalışmada da resilient fonksiyonlar deterministik bitlerin filtrelenmesi için kullanılmıştır [28]. Burada hata düzeltme kodlarından resilient fonksiyon oluşturulmuştur. Bunun için de basit bir teknik verilmiştir:  $G, [n, m, d]$  doğrusal kod için bir üretici matrisi olsun.  $\mathbb{Z}(\mathbb{Z}) = \mathbb{Z}G^T$  kuralına göre bir  $\mathbb{Z} : \{0, 1\}^n \mapsto \{0, 1\}^m$  fonksiyonu tanımlandığında, bu fonksiyon  $f(n, m, d-1)$ -resilient fonksiyonudur. Uygulama için kullanılan kod ise  $H_m$  Hamming kodunun ikilisi olan, tek yönlü  $[2^m - 1, \mathbb{Z}, 2^{m-1}]$  doğrusal kod  $H_m^\perp$  'dir.

Makalede 114 halka osilatörden alınan çıktı resilient fonksiyonuna verilir. Resilient fonksiyon için bilinen bir genişletilmiş BCH kodu olan  $[256, 16, 113]$  kodu kullanılmaktadır. Yani 256 bitlik bloklara ayrılan osilatör çıktıları resilient fonksiyona verilir ve fonksiyondan yalnızca 16 bitlik çıkış alınır. Fonksiyon sistemi saldırılara karşı güçlü hale getirmesine rağmen 15/16 oranında veri kaybına sebep olmaktadır.

#### **4. GELİŞTİRİLEN YAZILIM**

Rasgele sayı üreticileri ve son işlem algoritmaları eski bir konu olmasından dolayı üzerine birçok araştırmacının düşündüğü ve yeni yöntemler üretmeye çalıştığı bir alandır. Son işlem algoritmaları için birçok yeni öneri sunulmasına ve bu önerilerin açık bir şekilde makalelerde ya da tezlerde anlatılmasına rağmen, bunları kullanmak isteyen diğer kullanıcıların yapısını anlayıp bu algoritmaları tasarlaması ve uygulamaya dökmesi zor olabilir. Bu konuda önerilen son işlem algoritmaları yazılı olarak ne kadar açık anlatılmış olurlarsa olsunlar konu üzerinde çalışan herkes kodlama, uygulama geliştirme konusunda yeterli bilgiye sahip olmayabilir.

Yukarıda yazılan sebeplerden dolayı, bu konuda çalışan ya da merak eden kullanıcıların rahatlıkla kullanabilecekleri, sık kullanılan son işlem algoritmalarından oluşan bir yazılım süiti geliştirilmiştir. Kullanımı oldukça basit olan bu uygulama içinde bit dizisi bulunan txt formatında bir dosyayı girdi olarak kabul etmekte ve seçilen son işlem algoritmasına göre çıkış bit dizisini yine bir txt dosyası olarak çıktı vermektedir.

Bu bölümde bu yazılımı geliştirirken kullanılan teknolojilerden, projenin tasarımından ve arayüz kullanımından bahsedilecektir.

##### **4.1. Kullanılan Teknolojiler**

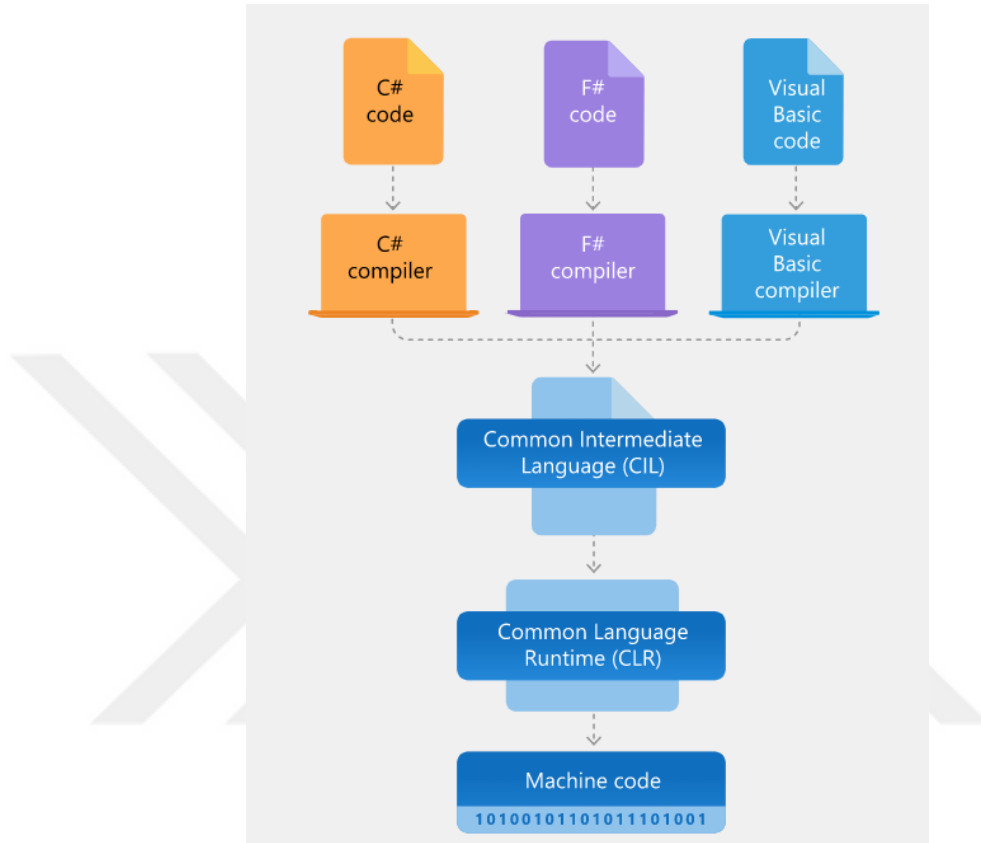
Bu uygulama geliştirilirken güncel, kullanımı kolay ve etkili olmalarından dolayı ASP.NET web geliştirme platformu ile C# uygulama dili, MVC yazılım geliştirme mimarisi ve Bootstrap tasarım aracı kullanılarak bir web uygulaması tasarlanmıştır.

###### **4.1.1. ASP.NET**

ASP.NET, klasik ASP teknolojisinin yerine Microsoft tarafından geliştirilen, birçok farklı türde uygulama oluşturmak için araçlar, programlama dilleri ve kitaplıklardan oluşan bir geliştirici platformudur. ASP'nin açılımı Active Server Page olarak isimlendirilir. Açılımından da anlaşılacağı üzere sunucu taraflı bir web geliştirme teknolojisidir. Güvenlik, tutarlılık ve esneklik gibi birçok kolaylıkla beraber geldiği için uygulama geliştirme aşamalarının hepsini geliştiricinin kendi başına yapmasına göre daha az çaba ve zaman almasını sağlamaktadır. ASP.NET ile sadece statik sayfalar değil, kullanıcı etkileşimli ve sürekli yenilenebilen dinamik sayfalar yapılabilir. Örneğin günümüzde haber portalları, çeşitli kurumsal uygulamalar, forum siteleri, e-ticaret siteleri ASP.NET ile yazılabilir ve yönetilebilir.

.NET uygulamaları C#, F# veya Visual Basic programlama dilinde yazılır. Kod, dilden bağımsız bir Ortak Ara Dil (Common Intermediate Language - CIL) olarak derlenmiştir.

Derlenen kod, .dll veya .exe dosya uzantısına sahip derleme dosyalarında saklanır. Bir uygulama çalıştığında, CLR (Common Language Runtime) derlemeyi alır ve üzerinde çalıştığı bilgisayarın belirli mimarisinde çalıştırılabilen makine koduna dönüştürmek için tam zamanında derleyici (JIT) kullanır. ASP.NET mimarisi Şekil 4.1’de gösterilmiştir.



**Şekil 4.1.** .NET framework’ü genel mimarisi.

.NET Framework, Windows tabanlı uygulamalar ve Web tabanlı uygulamalar gibi çok çeşitli uygulama türlerinde geliştirici deneyimini tutarlı hale getirmek ve .NET Framework tabanlı kodun başka herhangi bir kodla bütünleşmesini sağlamak için tüm iletişimi endüstri standartlarına göre oluşturmak için tasarlanmıştır [29].

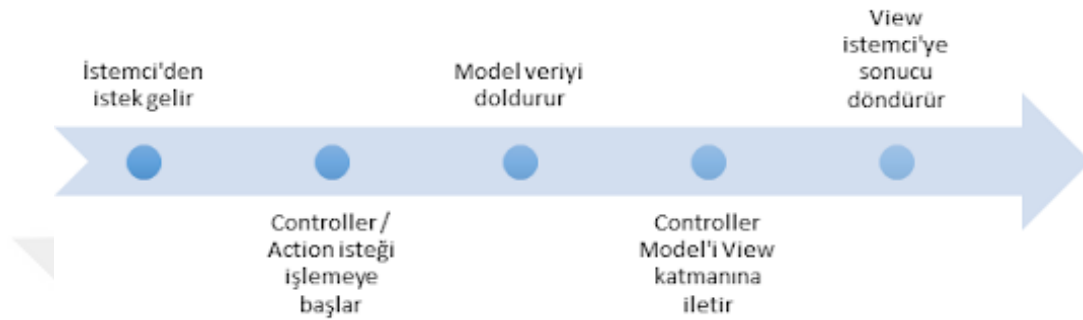
#### 4.1.2. MVC YAPISI

MVC uygulama geliştirmede önemli bir yere sahip mimari desenlerden biridir. Model, View Controller kelimelerinin baş harflerinden oluşur ve her kelime farklı bir katmanı ifade eder.

MVC mimari modeli uygulamayı üç ana gruba ayırır: Model, View, Controller. Web uygulamaları için MVC modelini kullanarak, gelen istekler, eylemleri gerçekleştirmek ve / veya verileri almak için Model ile çalışmaktan sorumlu bir Denetleyiciye (Controller) yönlendirilir.



Denetleyici, görüntülenecek Görünümü (View) seçer ve ona Modeli sağlar. Görünüm, Modeldeki verilere dayalı olarak son sayfayı oluşturur [30]. Kodun farklı amaçlara hizmet eden bu yapılarını birbirinden ayırarak, uygulamayı daha rahat geliştirilebilir, yönetebilir ve test edilebilir hale getirmiş oluruz. Ayrıca geliştirilen projenin detayına göre birçok kişi eş zamanlı olarak projenin üzerinde çalışabilmektedir. Şekil 4.2’de MVC mimarisinin genel yapısı ve hayat döngüsü gösterilmiştir.



**Şekil 4.2.** MVC mimarisi hayat döngüsü.

#### 4.1.2.1. Model

Model, MVC’de uygulama verisinin veya durumunun saklandığı, projenin iş mantığının (business logic) oluşturulduğu bölümdür. İş mantığıyla beraber doğrulama (validation) ve veri erişim (data access) işlemleri de bu bölümde gerçekleştirilmektedir.

Model, veritabanını katmanını uygulamadan ayırır, böylece diğer katmanların bu bilgiyi tutmasına gerek kalmaz. Kısacası veri işlemleri bu katmanda gerçekleşir.

#### 4.1.2.2. View

View katmanı kullanıcının ekranda gördüğü arayüzü içeren katmandır. Bu bölümde projenin kullanıcılara sunulacak olan HTML dosyaları yer almaktadır. Projenin geliştirildiği yazılım dillerine göre dosya uzantıları da değişebilmektedir. Genellikle Model katmanındaki veri kullanılır. View katmanının ayrı olması arayüzde yapılacak değişikliklerin diğer katmanlarda bir değişiklik yapılmasına gerek kalmadan gerçekleştirilebilmesini sağlamaktadır.

View’ın bir görevi de kullanıcılardan alınan istekleri controller’a iletmektir.

#### 4.1.2.3. Controller

Controller, MVC’de projenin iç süreçlerini kontrol eden bölümdür. Bu bölümde View ile Model arasındaki bağlantı kurulur. Kullanıcılardan gelen istekler (request) Controller’larda

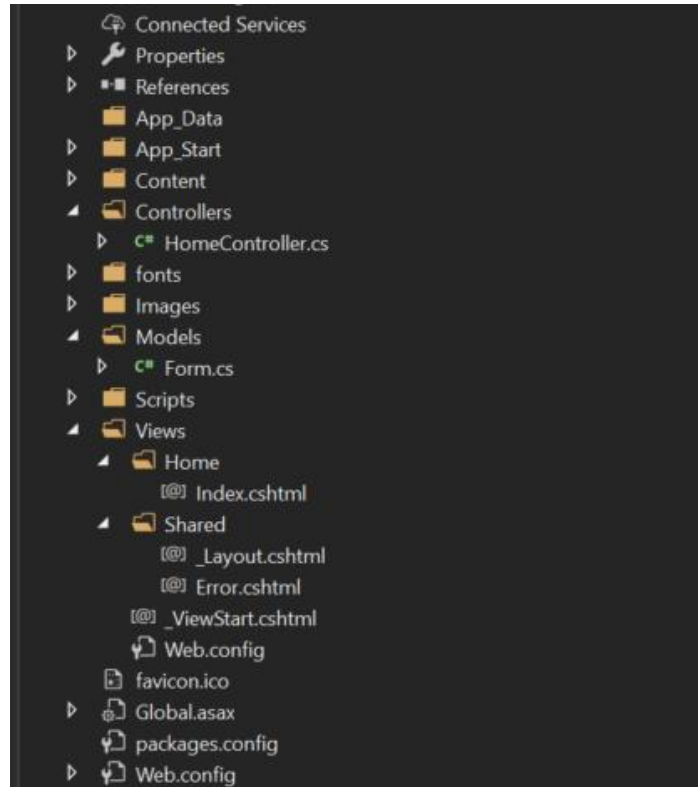
değerlendirilir, isteğin detayına göre hangi işlemlerin yapılacağı ve kullanıcıya hangi View'ın döneceği (response) belirtilir.

#### 4.1.3. Bootstrap

Ücretsiz bir CSS framework'u olan Bootstrap, HTML ve CSS ile desteklenmiş, hazır şablonlar içeren açık kaynak kodlu bir tasarım aracıdır. Günümüzde yapılan tasarımların tüm cihaz boyutlarına uygun şekilde geliştirilmesi oldukça önemli bir hal almıştır. Bu ihtiyaçtan doğan Bootstrap, telefon, tablet ve bilgisayarlar için uygun görünümde ve cihazınızın büyüklüğüyle orantılı şekilde uygulamaların görünmesini sağlar. Bir site için ihtiyaç duyulan tüm elementleri (form öğeleri, tablolar, menüler, butonlar vb.) içerisinde barındıran Bootstrap ile tasarım yaparken bu hazır unsurları kullanarak tüm cihaz boyutlarına uygun tasarımlar yapılmasını kolaylaştırır. Stiller, imajlar ve JavaScript'ler Bootstrap kütüphanesi ile hazır bir şekilde gelmektedir.

#### 4.2. Projenin Tasarımı

Proje Microsoft Visual Studio 2017 IDE (Integrated Development Environment – Entegre Geliştirme Ortamı)'si kullanılarak geliştirilen bir web uygulamasıdır. Visual Studio'da yeni bir web uygulaması oluşturulurken MVC seçerek projenin bu yapıda kurulması sağlanmıştır. Projenin klasör yapısı Şekil 4.3'te gösterildiği gibidir.



Şekil 4.3. Geliştirilen projenin klasör yapısı.

Proje oluşturulduğunda içerisinde varsayılan olarak Bootstrap .css ve .js dosyaları da gelmektedir. Uygulamanın arayüz tasarımı da Bootstrap komponentleri kullanılarak tasarlandı. Genel yapı itibari ile tek sayfadan oluşan proje bir dosya ve bu dosya üzerine uygulanacak son işlem algoritması seçilerek yeni bir dosya oluşturma işlemlerinin gerçekleştirildiği bir uygulamadır.

Şekil 4.3'te görüldüğü gibi uygulamanın veri kısmı "Models" klasörü içerisindeki "Form.cs"dir. Veri olarak, seçilen uygulama ve son işlem algoritması uygulanması istenilen dosya tutulmaktadır. "View" olarak da "View\Home" klasörü içerisinde "Index.cshtml" dosyası kullanılarak sayfa tasarımı burada yapılmıştır. Uygulama tek sayfa olduğundan dolayı sadece bir View dosyası yeterli olmuştur. Tasarım olarak uygulanabilecek son işlem algoritmaları Bootstrap "Tab" yapısı kullanılarak sayfanın en üstüne yerleştirilmiştir. Alt tarafına da kullanıcının sisteme yükleyeceği dosyayı seçebileceği, yine Bootstrap komponenti olan "File Input" yerleştirilmiş ve en alta da son işlem algoritmasının uygulanmasını ve yeni dosya oluşturulmasını sağlayacak bir "Buton" konularak tasarım tamamlanmıştır. Genel tasarım olarak göstermek gerekirse sayfanın html kodu Şekil 4.4'te gösterildiği gibidir.

```
<div class="sonislem_form">
  <h2 style="color:#4c90a5">POST PROCESSING ALGORITHMS</h2>
  <hr />
  <using (Html.BeginForm("Create", "Home", FormMethod.Post, new { enctype = "multipart/form-data" }))
  {
    <div class="btn-group btn-group-toggle" data-toggle="buttons" style="width:100%; padding-bottom:30px">
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="xor" value="xor" checked=""> XOR Algoritim
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="von" value="von"> Von Neumann Algoritim
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="h" value="h"> H Algoritim
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="h2" value="h2"> H2 Algoritim
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="h4" value="h4"> H4 Algoritim
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="sbox" value="sbox"> SBOX Algoritim
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="mixingbits" value="mixingbits"> Mixing Bits in Steps and XORing
      </label>
      <label class="btn btn-outline-info">
        <input type="radio" name="algoritma" id="ivn" value="ivn"> Iterating Von Neumann
      </label>
    </div>
    <div>
      <input type="file" class="form-control-file" id="file" name="file" accept=".txt">
    </div>
    <input type="submit" value="Convert File" class="btn btn-info" />
    <br><br>
    <div style="color:red">
      @ViewBag.ErrorMessage
    </div>
    <div>
      @ViewBag.SuccessMessage
    </div>
  }
</div>
```

Şekil 4.4. Uygulamanın arayüz html kodu örneği.

Controller olarak ise “Controllers” klasörü içerisindeki “HomeController.cs” dosyası kullanılarak, verilerin işlenmesi yani bu proje üzerinden konuşacak olursak sisteme yüklenen, içerisinde ham rasgele bit dizisi bulunan dosya üzerinde, seçilen son işlem algoritmasının uygulandığı ve yeni dosyanın üretildiği yerdir. Sınıf içerisindeki fonksiyon yapısı Şekil 4.5’te gösterildiği gibidir.

```
public class HomeController : Controller
{
    readonly int[, ] sbboxes = new int[, ]{...};

    0 references
    public ActionResult Index()
    {
        return View();
    }

    [System.Web.Mvc.HttpPost]
    0 references
    public ActionResult Create( HttpPostedFileBase file, string algoritma){...}
    2 references
    private string[] XORAlgorithm(bool[] array){...}
    2 references
    private string[] VonNeumannAlgorithm(bool[] array){...}
    3 references
    private bool[] RotateLeft1(bool[] array){...}
    2 references
    private bool[] RotateLeft2(bool[] array){...}
    1 reference
    private bool[] RotateLeft4(bool[] array){...}
    1 reference
    private string[] HFuncAlgorithm(bool[] array){...}
    1 reference
    private string[] H2FuncAlgorithm(bool[] array){...}
    1 reference
    private string[] H4FuncAlgorithm(bool[] array){...}
    2 references
    private byte BoolArrayToInt(bool[] bits){...}
    1 reference
    private string[] SBoxAlgorithm(bool[] array){...}
    1 reference
    private string[] MixingBitsAlgorithm(bool[] array){...}
    1 reference
    private bool[] XORAlgorithmforIVN(bool[] array){...}
    1 reference
    private bool[] RAlgorithmforIVN(bool[] array){...}
    3 references
    private string[] IteratingVonNeumannAlgorithm(bool[] array){...}
}
```

**Şekil 4.5.** HomeController.cs fonksiyon yapısı.

“Index” fonksiyonu sayfanın html kodunu dönen kısımdır. Uygulama ilk açıldığında “\Index” için istemci bir istekte bulunur ve HomeController bu isteği bu fonksiyon ile karşılayarak ilgili View’ı yani bu durumda “Views\Home\Index.cshtml” i döner.

“Create” fonksiyonu ise arayüzdeki “Convert File” butonu tıklandığında form bilgilerini controller’da karşılayan ve işlenmesini sağlayan fonksiyondur. Biz veri olarak arayüzde algoritma ve bu algoritmanın uygulanacağı dosyayı alıyoruz ve bunu controller’da “Create” fonksiyonuna yolluyoruz. Fonksiyonun içeriği aşağıdaki gibidir.

```

public ActionResult Create( HttpPostedFileBase file, string algoritma)
{
    if (file != null && file.ContentLength > 0)
    {
        try
        {
            var fileBytes = new byte[file.ContentLength];
            file.InputStream.Read(fileBytes, 0, fileBytes.Length);
            string result = System.Text.Encoding.UTF8.GetString(fileBytes).Replace("\r\n", string.Empty);

            var bits = result.Select(chr => chr == '1').ToArray();
            string strPath = Environment.GetFolderPath(System.Environment.SpecialFolder.DesktopDirectory);

            string[] res;

            if (algoritma.Equals("xor"))
            {
                if (file.ContentLength < 2)
                {
                    ViewBag.ErrorMessage = "Minimum file length must be 2 bits for selected algorithm!";
                }
                else
                {
                    res = XORAlgorithm(bits);
                    System.IO.File.WriteAllLines(strPath + "\\XOR.txt", res);
                    ViewBag.SuccessMessage = "File created successfully to location : " + strPath + "\\XOR.txt";
                }
            }
            else if (algoritma.Equals("von"))...
            else if (algoritma.Equals("h"))...
            else if (algoritma.Equals("h2"))...
            else if (algoritma.Equals("h4"))...
            else if (algoritma.Equals("sbox"))...
            else if (algoritma.Equals("mixingbits"))...
            else if (algoritma.Equals("ivn"))...
        }
        catch (Exception ex)
        {
            ViewBag.ErrorMessage = "ERROR:" + ex.Message.ToString();
        }
    }
    else
    {
        ViewBag.ErrorMessage = "You have not specified a file.";
    }
    return View("Index");
}

```

Şekil 4.6. "Create" fonksiyonunun içeriği.

Burada öncelikli olarak dosyanın seçilip seçilmediği ve dolu olup olmadığı kontrol ediliyor. Seçilmediyse veya seçilmesine rağmen içi boş ise kullanıcıya hata dönülüyor. Daha sonra ise yüklenen dosya içeriği okunarak içerisindeki bit dizisi bir boolean diziyeye dönüştürülüyor. Seçilen algoritma kontrol edilerek ilgili fonksiyona bu bit dizisi parametre olarak gönderilerek son işlem yöntemi bu diziyeye uygulanıyor ve fonksiyon bir string dizi olarak sonucu geri dönüyor. Dönülen bu sonuç oluşturulan .txt formatındaki bir dosyaya yazılıyor ve bu dosya masaüstüne ilgili algoritma ismi ile kaydediliyor.

Burada her bir son işlem yönteminin ayrı bir fonksiyon olarak yazılmasının sebebi hem kodun okunurluğunu sağlamak hem de herhangi bir hata durumunda hatanın meydana geldiği yeri bulmak ve düzeltmek daha kolay olmaktadır.

#### 4.2.1. XOR Algoritmasının Entegrasyonu

XOR algoritması önceki bölümlerde de açıklandığı gibi ikili bit dizileri ele alınarak ( 0, 1 ) ve ( 1, 0 ) durumunda 1, ( 0, 0 ) ve ( 1, 1 ) durumunda 0 üreten bir algoritmadır.

HomeController.cs içerisindeki "XORAlgorithm" fonksiyonu parametre olarak aldığı boolean türündeki bir dizi içerisinde dolaşarak her iki bit için C# ve birçok diğer dilde de XOR işlemi anlamına gelen "^" operatörünü kullanarak çıkış biti üretir ve bunu yeni bir diziye yazar ve bu oluşan yeni diziye geri döner.

Algoritma çift bit olarak işlem yaptığından dolayı tek sayıda uzunlukta gelen dizilerde son biti görmezden gelir. Algoritma Şekil 4.7'deki gibidir.

```
2 references
private string[] XORAlgorithm(bool[] array)
{
    int x = array.Length % 2;

    array = array.Take(array.Count() - x).ToArray();

    string[] xor = new string[array.Length / 2];
    int index = 0;
    for (int i = 0; i < array.Length; i = i + 2)
    {
        bool sonuc = array[i] ^ array[i + 1];
        xor[index] = sonuc ? "1" : "0";
        index++;
    }

    return xor;
}
```

Şekil 4.7. XOR son işlem yönteminin algoritması.

#### 4.2.2. Von Neumann Algoritmasının Entegrasyonu

Von Neumann algoritması da XOR algoritması gibi ikili bit dizilerini alarak ( 0, 1 ) geldiğinde 0, ( 1, 0 ) geldiğinde 1 diğer durumlarda çıkış vermeyen bir algoritmadır. HomeController.cs içerisindeki "VonNeumannAlgorithm" fonksiyonu parametre olarak aldığı boolean türündeki bir dizi içerisinde dolaşarak her iki biti kontrol eder, duruma göre çıkış biti üretir ve bunu yeni bir diziye yazar ve bu oluşan yeni diziye geri döner.

Algoritma çift bit olarak işlem yaptığından dolayı tek sayıda uzunlukta gelen dizilerde son biti görmezden gelir. Algoritma Şekil 4.8'deki gibidir.

```

2 references
private string[] VonNeumannAlgorithm(bool[] array)
{
    int x = array.Length % 2;

    array = array.Take(array.Count() - x).ToArray();

    string[] von = new string[array.Length / 2];
    int index = 0;
    for (int i = 0; i < array.Length; i = i + 2)
    {
        if (!array[i] && array[i + 1])
        {
            von[index] = "0";
            index++;
        }
        else if (array[i] && !array[i + 1])
        {
            von[index] = "1";
            index++;
        }
    }
    von = von.Where(c => c != null).ToArray();

    return von;
}

```

Şekil 4.8. Von Neumann son işlem yönteminin algoritması.

#### 4.2.3. H Fonksiyonun Entegrasyonu

H fonksiyonu Bölüm 3'te anlatıldığı gibi 8 bit bloklar halinde işlem yapılan ve Denklem (7)'nin kullanıldığı algoritmadır. Şekil 4.9'daki gibi bir yapı oluşturulmuştur.

```

2 references
private bool[] RotateLeft1(bool[] array)...
1 reference
private bool[] RotateLeft2(bool[] array)...
1 reference
private bool[] RotateLeft4(bool[] array)...
1 reference
private string[] HFuncAlgorithm(bool[] array)
{
    bool[] a1 = new bool[8];
    bool[] a2 = new bool[8];
    string[] c = new string[8];

    int x = array.Length % 16;

    array = array.Take(array.Count() - x).ToArray();

    string[] von = new string[array.Length / 2];
    int index = 0;
    for (int i = 0; i <= array.Length - 16; i = i + 16)
    {
        Array.Copy(array, i, a1, 0, 8);
        Array.Copy(array, i + 8, a2, 0, 8);
        bool[] temp = new bool[8];
        Array.Copy(a1, 0, temp, 0, 8);

        bool[] a11 = RotateLeft1(temp);

        for (int j = 0; j < 8; j++)
        {
            bool res = a1[j] ^ a11[j] ^ a2[j];
            c[j] = res ? "1" : "0";
        }

        c.CopyTo(von, index);
        index = index + 8;
    }
    return von;
}

```

Şekil 4.9. H Fonksiyonu son işlem yönteminin algoritması.

#### 4.2.4. H2 Fonksiyonun Entegrasyonu

H2 fonksiyonu ise 3. bölümde gösterin Denklem (8) kullanılarak gerçekleştirilen algoritmadır.  $H2(a1, a2) = \text{XOR}(\text{XOR}(\text{XOR}(a1, \text{rotateleft}(a1, 1)), \text{rotateleft}(a1, 2)), a2)$  şeklinde olan denklem koda geçirilerek HomeController.cs içerisindeki "H2FuncAlgorithm" fonksiyonu oluşturulmuştur. Algoritması Şekil 4.9'da olduğu gibidir.

```

2 references
private bool[] RotateLeft1(bool[] array)...
2 references
private bool[] RotateLeft2(bool[] array)...
1 reference
private bool[] RotateLeft4(bool[] array)...

1 reference
private string[] H2FuncAlgorithm(bool[] array)
{
    bool[] a1 = new bool[8];
    bool[] a2 = new bool[8];
    string[] c = new string[8];

    int x = array.Length % 16;

    array = array.Take(array.Count() - x).ToArray();

    string[] von = new string[array.Length / 2];
    int index = 0;
    for (int i = 0; i <= array.Length - 16; i = i + 16)
    {
        Array.Copy(array, i, a1, 0, 8);
        Array.Copy(array, i + 8, a2, 0, 8);

        bool[] a11 = RotateLeft1(a1);
        bool[] a12 = RotateLeft2(a1);

        for (int j = 0; j < 8; j++)
        {
            bool res = a1[j] ^ a11[j] ^ a12[j] ^ a2[j];
            c[j] = res ? "1" : "0";
        }

        c.CopyTo(von, index);
        index = index + 8;
    }
    return von;
}

```

Şekil 4.10. H2 Fonksiyonu son işlem yönteminin algoritması.

H2 fonksiyonu dizinin 16 bitlik bloklarıyla işlem yapar. Bu blokları a1 ve a2 olarak 8 bitlik 2 parçaya ayırır. Daha sonra a1'i 1 bit sola kaydırarak a11 dizisini ve 2 bit sola kaydırarak a21 dizisini oluşturuyor. Sonrasında ise a1, a11, a12 ve a2 dizilerini XOR işlemine sokarak yeni bir 8 bitlik çıkış dizisi oluşturulur. Bu işlem tüm 16 bitlik bloklara yapılır.

Algoritma 16 bit üzerinde çalıştığı için giriş bit dizisi uzunluğu 16 bit ve katları şeklinde olarak alınır ve fazla bitler görmezden gelinir.



#### 4.2.5. H4 Fonksiyonun Entegrasyonu

H4 fonksiyonu H2 fonksiyonu ile aynı mantıkta çalışır fakat kullanılan denklem farklıdır. H4 fonksiyonu için Denklem (9) kullanılarak 16 bitten 8 bitlik çıkış dizisi üretilir. Bu denkleme göre gelen ham rasgele bit dizisi 16 bitlik bloklara ayrılır. Her bir blok a1 ve a2 olmak üzere iki tane 8 bitlik bloklara ayrılır. Daha sonra a1 1 bit sola kaydırılarak a11 dizisi, 2 bit sola kaydırılarak a12 dizisi ve 4 bit sola kaydırılarak a14 dizisi elde edilir. Son olarak a1, a11, a12, a14 ve a2 dizileri XOR işleminden geçirilerek 8 bitlik çıkış bit dizisi oluşturulur.

```

2 references
private bool[] RotateLeft1(bool[] array)...
2 references
private bool[] RotateLeft2(bool[] array)...
1 reference
private bool[] RotateLeft4(bool[] array)...

1 reference
private string[] H4FuncAlgorithm(bool[] array)
{
    bool[] a1 = new bool[8];
    bool[] a2 = new bool[8];
    string[] c = new string[8];

    int x = array.Length % 16;

    array = array.Take(array.Count() - x).ToArray();

    string[] von = new string[array.Length / 2];
    int index = 0;
    for (int i = 0; i <= array.Length - 16; i = i + 16)
    {
        Array.Copy(array, i, a1, 0, 8);
        Array.Copy(array, i + 8, a2, 0, 8);

        bool[] a11 = RotateLeft1(a1);
        bool[] a12 = RotateLeft2(a1);
        bool[] a14 = RotateLeft4(a1);

        for (int j = 0; j < 8; j++)
        {
            bool res = a1[j] ^ a11[j] ^ a12[j] ^ a14[j] ^ a2[j];
            c[j] = res ? "1" : "0";
        }

        c.CopyTo(von, index);
        index = index + 8;
    }

    return von;
}

```

Şekil 4.11. H4 Fonksiyonu son işlem yönteminin algoritması.

H2 fonksiyonunda olduğu gibi işlem tüm 16 bitlik bloklara yapılır ve giriş bit dizisi uzunluğu 16 bit ve katları şeklinde olarak alındığından dolayı fazla bitler görmezden gelir.

#### 4.2.6. SBOX Entegrasyonu

SBOX algoritması Bölüm 3'te de detaylı şekilde anlatıldığı gibi lookup tablolarına bakılarak işlem yapılan bir yöntemdir. Bu uygulamada da referans [27]'deki DES için hazırlanan 8 SBOX kullanılmıştır. S-kutuları Şekil 4.11'deki gibi tanımlanmıştır.

```
readonly int[,] sboxes = new int[8,16]
{
    {
        { 14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7 },
        { 0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8 },
        { 4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0 },
        { 15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13 }
    },
    {
        { 15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10 },
        { 3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5 },
        { 0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15 },
        { 13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9 }
    },
    {
        { 10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8 },
        { 13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1 },
        { 13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7 },
        { 1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12 }
    },
    {
        { 7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15 },
        { 13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9 },
        { 10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4 },
        { 3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14 }
    },
    {
        { 2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9 },
        { 14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6 },
        { 4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14 },
        { 11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3 }
    },
    {
        { 12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11 },
        { 10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8 },
        { 9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6 },
        { 4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13 }
    },
    {
        { 4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1 },
        { 13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6 },
        { 1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2 },
        { 6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12 }
    },
    {
        { 13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7 },
        { 1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2 },
        { 7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8 },
        { 2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11 }
    }
};
```

Şekil 4.12. Kullanılan S-kutularının tanımlanması.

Algoritma 6 bitlik bloklar üzerinde işlem yapmaktadır. Alınan 6 bitlik bloğun 1. ve 6. biti alır ve satır değerini bulur. Daha sonra ortada kalan 4 biti alarak da sütun değerini bulur. İlgili s-kutusunda satır ve sütun değerine karşılık gelen değer bulunur ve 2'li sisteme çevrilerek 4 bitlik çıkış dizisi oluşturulur. Bu işlem tüm 6 bitlik bloklara uygulanır. Algoritma 6 bitlik bloklar üzerinde çalıştığı için giriş bit dizisi uzunluğu 6 bit ve katları şeklinde olarak alınır ve fazla bitler görmezden gelir.

```

2 references
private byte BoolArrayToInt(bool[] bits)[...]
1 reference
private string[] SBoxAlgorithm(bool[] array)
{
    int x = array.Length % 6;

    array = array.Take(array.Count() - x).ToArray();

    string[] von = new string[array.Length];
    int index = 0;
    int sbbox_number = 0;

    for (int i = 0; i < array.Length; i = i + 6)
    {
        bool[] sub_array = new bool[6];
        Array.Copy(array, i, sub_array, 0, 6);

        bool[] row = new bool[] { sub_array[0], sub_array[5] };
        bool[] column = new bool[] { sub_array[1], sub_array[2], sub_array[3], sub_array[4] };

        Array.Reverse(row);
        Array.Reverse(column);

        int rownum = (int)BoolArrayToInt(row);
        int columnnum = (int)BoolArrayToInt(column);
        int deger = sbboxes[sbbox_number, rownum, columnnum];
        string y = Convert.ToString(deger, 2);
        bool[] arraysonuc = Convert.ToString(deger, 2).Select(s => s.Equals('1')).ToArray();
        if (y.Length == 3) y = "0" + y;
        if (y.Length == 2) y = "00" + y;
        if (y.Length == 1) y = "000" + y;

        von[index] = y;
        index++;

        sbbox_number++;
        sbbox_number = sbbox_number % 8;
    }

    return von;
}

```

Şekil 4.13. SBOXs son işlem yönteminin algoritması.

#### 4.2.7. Mixing Bits in Steps and XORing

Bu algoritma da boolean dizi olarak aldığı ham rasgele bit dizisini önce belirli bir sistem ile karıştırıp sonra da XOR algoritması uygulayarak çıkış bit dizisi üretmektedir. XOR algoritması kullanıldığı için minimum 2 bitlik bir giriş dizisi olması gerekmektedir. Bunun

haricinde bir giriş koşulu olmamakla beraber XOR uygulamasında bahsedildiği gibi tek sayıda gelen bit dizisinde karıştırma işleminden sonra XOR fonksiyonuna gönderildiği için son bit görmezden gelir.

Bölüm 3'te de örnekle anlatıldığı gibi gelen bit dizisinden ilk iki bit olduğu gibi alınır. 3. bit ilk iki bitin arasına yerleştirilir. Daha sonra gelen bitler de sırayla dizinin sonundan ve başından yerleştirilerek giriş bit dizisi karıştırılır. Son aşama olarak da oluşan yeni dizi XOR algoritmasının fonksiyonuna gönderilerek oluşan dizi sonuç olarak döndülür.

```
1 reference
private string[] MixingBitsAlgorithm(bool[] array)
{
    List<bool> stringmixedarray = new List<bool>();
    stringmixedarray.Add(array[0]);
    stringmixedarray.Add(array[1]);

    int number_of_steps = (int)Math.Ceiling(Math.Log(array.Length - 1, 2)) + 1;
    int index_of_array = 2;
    for (int i = 2; i <= number_of_steps; i = i + 1)
    {
        int number_of_inserted_bits = (int)Math.Pow(2, i - 2);

        bool bas = true;
        int index_to_insert = 1;
        for (int t = 0; t < number_of_inserted_bits; t = t + 1)
        {
            if (index_of_array < array.Length)
            {
                if (bas)
                {
                    stringmixedarray.Insert(index_to_insert, array[index_of_array]);
                    bas = false;
                }
                else
                {
                    int indx = stringmixedarray.Count() - index_to_insert;
                    stringmixedarray.Insert(indx, array[index_of_array]);

                    bas = true;
                    index_to_insert = index_to_insert + 2;
                }

                index_of_array++;
            }
            else
            {
                break;
            }
        }
    }
    return XORAlgorithm(stringmixedarray.ToArray());
}
```

Şekil 4.14. Mixing Bits in Steps and XORing algoritması.

#### 4.2.8. Iterating Von Neumann

Iterating Von Neumann algoritması standart Von Neumann algoritmasında görmezden gelinen bit dizilerinin de kullanılmasını amaçlayan bir algoritmadır. Bu amaçla gelen bit dizisi 2'li bloklar halinde alınır. Von Neumann, XOR ve R algoritmasından geçirilen bu bit dizilerinden

Von Neumann çıkışı direk çıkış bit dizisine verilirken XOR ve R algoritması aynı aşamalardan geçmek üzere tekrardan Von Neumann algoritmasına sokulur. Bu durum tek bit kalana kadar devam eder ve ağaç yapısı şeklinde bir görüntü oluşturarak her bir aşamanın Von Neumann çıkışı çıkış bit dizisine verilir ve yeni dizi sonuç olarak döndürülür.

Burada kendini tekrar eden bir yapı bulunduğundan dolayı “recursive” bir algoritma yapısı kurularak gelen bit dizisinde çıkış üretilmiştir. Algoritma Şeki4.14’te gösterildiği şekildedir.

```

1 reference
private bool[] XORAlgorithmforIVN(bool[] array) {...}
1 reference
private bool[] RAlgorithmforIVN(bool[] array) {...}
3 references
private string[] IteratingVonNeumannAlgorithm(bool[] array)
{
    string[] von = new string[array.Length / 2];
    bool[] xor = new bool[array.Length / 2];
    bool[] ralg = new bool[array.Length / 2];
    if (array.Length < 2)
    {
        return von;
    }
    else
    {
        von = VonNeumannAlgorithm(array);
        xor = XORAlgorithmforIVN(array);
        ralg = RAlgorithmforIVN(array);
        string[] result = new string[von.Length + xor.Length + ralg.Length];
        int length = result.Where(c => c != null).Count();
        von.CopyTo(result, length);

        string[] iv_xor = IteratingVonNeumannAlgorithm(xor);
        length = result.Where(c => c != null).Count();
        iv_xor.CopyTo(result, length);

        string[] iv_ralg = IteratingVonNeumannAlgorithm(ralg);
        length = result.Where(c => c != null).Count();
        iv_ralg.CopyTo(result, length);

        result = result.Where(c => c != null).ToArray();
        return result;
    }
}

```

Şekil 4.15. Iterating Von Neumann algoritması.

### 4.3. Arayüz Kullanımı

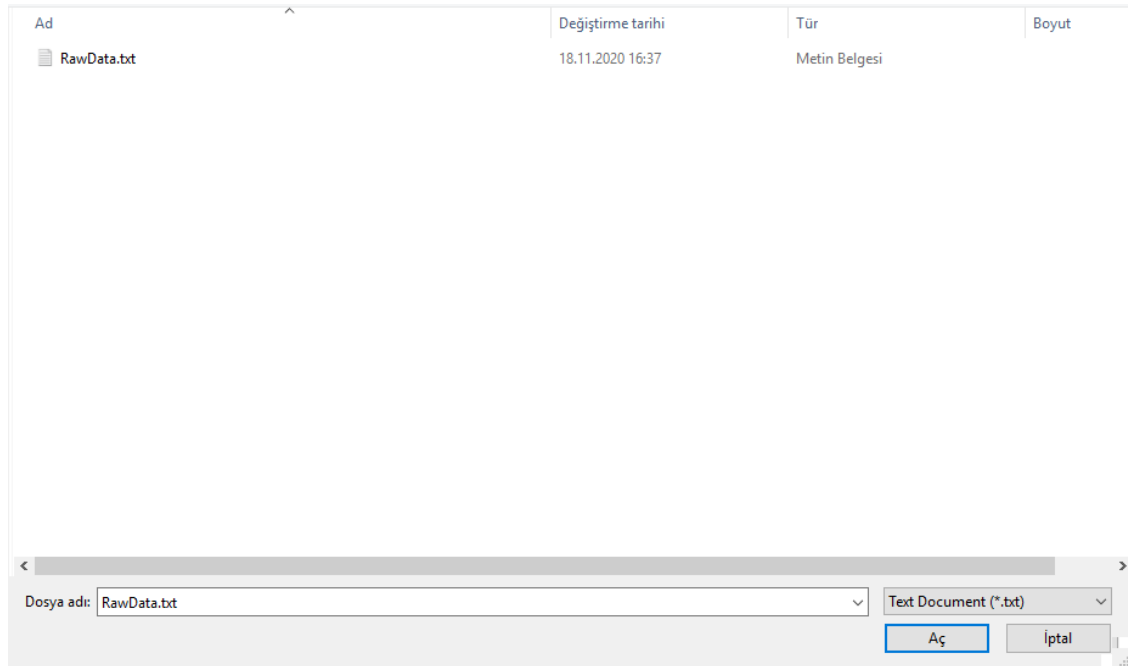
Kolay kullanım ve işlevsellik konusu önemli olduğundan oldukça basit ve sade bir arayüz tasarlanmıştır. Arayüzde gerçekleştirilmesi amaçlanan işlem doğrultusunda, 7 tane tab bulunmakta. Bu tablarda algoritması yazılan son işlem yöntemleri bulunmaktadır. Bunlar; XOR

algoritması, Von Neumann algoritması, H, H2 ve H4 algoritmaları, SBOX, Mixing Bits in Steps and XORing ve Iterating Von Neumann algoritması.



Şekil 4.16. Uygulama Arayüzü.

Arayüzde kullanıcının yapması gereken “Choose File” butonuna tıklayarak açılan pencereden içerisinde ham rasgele sayı dizisinin bulunduğu .txt uzantılı dosyayı seçmek ve “Aç” butonuna basmaktır. Tasarlanan arayüzde “File Input” için kabul edilen tip olarak “.txt” formatı yazıldığı için Şekil 4.16’te görüldüğü gibi sadece .txt uzantılı dosyalar görülecek ve seçilebilecektir.



Şekil 4.17. Dosya seçme ekranı.

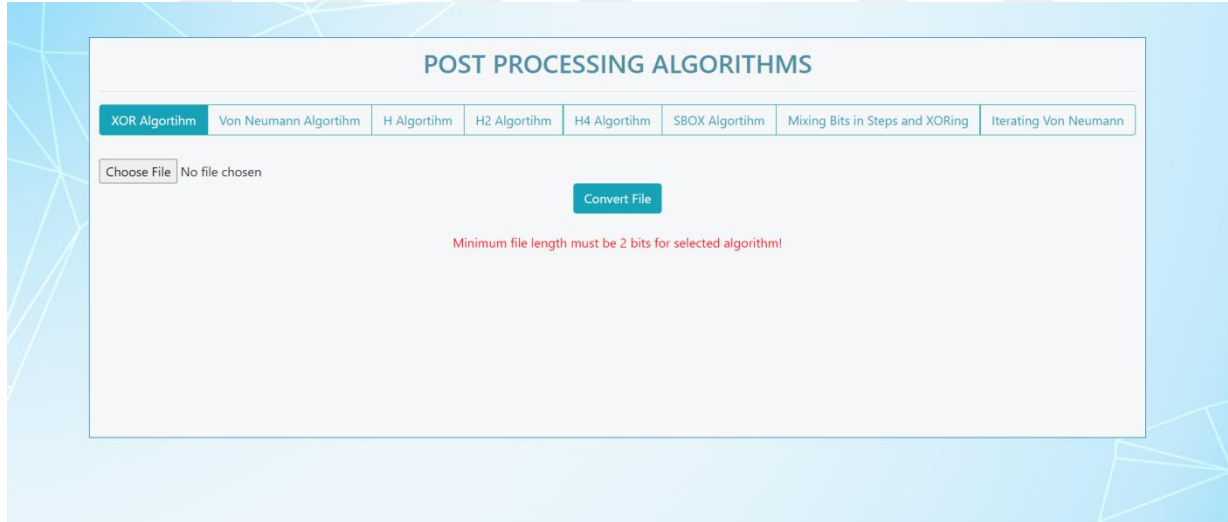


Dosyayı seçtikten sonra kullanılacak algoritmayı da seçerek “Convert File” butonuna basarak son işlem algoritmasını seçilen dosya üzerine uyguluyoruz.



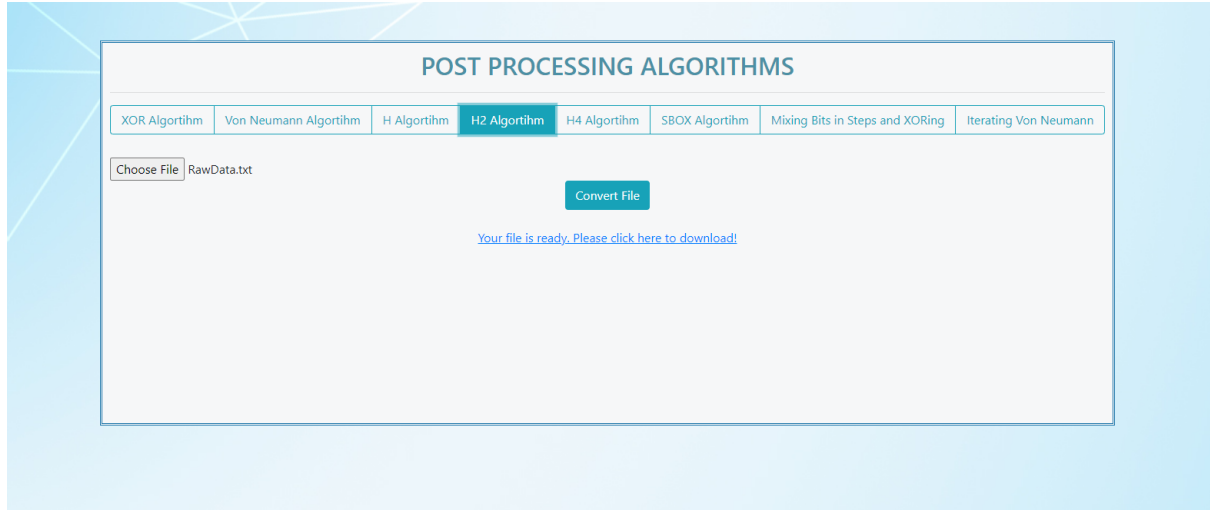
**Şekil 4.18.** Kullanılacak son işlem algoritmasını seçme ve işlemi başlatma.

Herhangi bir hata durumunda (dosya okuyamama ya da dosya uzunluğu hatası) kullanıcı aşağıdaki gibi bir hata mesajı alır.



**Şekil 4.19.** Hatalı şekilde sonlanan işlem.

Düzgün şekilde algoritma çalıştırıldığında ve yeni dosya üretildiğinde Şekil 4.19'deki ekran ile karşılaşılır. Kullanıcının tek yapması gereken indirme linkine basarak hazırlanan dosyayı kaydetmektir.



Şekil 4.20. Başarılı şekilde sonlanan işlem.

#### 4.4. NIST Test Sonuçları

Geliştirilen yazılımın başarılı bir şekilde çalıştığını doğrulayabilmek amacıyla [18]'de halka osilatörün donanımsal gerçekleşmesi sonucu elde edilen saf bit dizisi ile [31]'de kaotik çekerler kullanılarak (2 + 4 çeker) elde edilen saf bit dizileri yazılımdan geçirilmiştir. Son işlem sonucu üretilen rasgele bit dizileri NIST testlerine tabi tutulmuştur.

[18]'de yapılan çalışmada halka osilatörler kullanılarak elde edilen saf bit dizisinin ve bu bit dizisinin son işlemten geçirilmiş sonuçları Tablo 4.1'de verilmiştir.

**Tablo 4.1.** Makale [18] kullanılarak üretilen bit dizisine uygulanan test sonuçları

Halka Osilatörden elde edilen saf bit dizisinin son işlem sonuçları									
	Saf bit	Xor	Von Neumann	H	H2	H4	Sbox	İtervanneu	Mixing
Frekans Testi	-	-	0,360	0,245	0,654	0,650	0,639	0,304	-
Blok Frekans Testi	-	-	0,596	0,922	0,234	0,811	0,108	0,331	0,054
Akış Testi	-	-	0,082	0,292	0,308	0,731	0,180	0,539	0,858
En Uzun Birler Testi	-	-	0,382	0,036	0,259	0,578	0,123	0,315	0,444
İkili Matris Rankı Test	0,444	0,795	0,980	0,270	0,108	0,145	0,564	0,505	0,292
Ayrık Fourier Testi	0,018	0,184	0,098	0,959	0,612	0,181	0,316	0,986	0,199



Örtüşme yen Şablon Eşleştir me Testi	-	0,006	0,013	0,742	0,066	0,327	0,114	0,306	0,251
Örtüşen Şablon Eşleştir me Testi	-	0,272	0,322	0,156	0,509	0,557	0,413	0,220	0,779
Maurer Testi	-	0,207	0,140	0,463	0,154	0,463	0,688	0,218	0,152
Doğrusal Karmaşı klık Testi	0,615	0,567	0,970	0,337	0,772	0,686	0,265	0,989	0,054
Seri Testi	-	0,353	0,755	0,153	0,502	0,376	0,437	0,404	0,175
Yaklaşık Entropi Testi	-	0,012	0,164	0,125	0,112	0,654	0,139	0,878	0,075
Kümülat if Toplaml ar Testi	-	-	0,280	0,192	0,793	0,427	0,319	0,186	-

[31]'de yapılan çalışmada kaotik çekerler kullanılarak elde edilen 2+4 çekerin saf bit dizisinin ve bu bit dizisinin son işleminden geçirilmiş sonuçları Tablo 4.2'de verilmiştir.

**Tablo 4.2.** Makale [31] kullanılarak üretilen bit dizisine uygulanan test sonuçları

Kaotik harita 4 Çekerden elde edilen saf bit dizisinin son işlem sonuçları									
	Saf bit	Xor	Von Neumann	H	H2	H4	Sbox	İtervanneu	Mixing
Frekans Testi	0,545	-	0,413	0,604	0,585	0,493	0,020	0,723	0,045
Blok Frekans Testi	-	-	0,431	0,849	0,067	0,997	0,039	0,453	0,061
Akış Testi	-	-	-	0,038	0,022	0,645	0,052	-	0,111
En Uzun Birler Testi	-	-	-	0,909	0,669	0,522	0,111	-	0,063
İkili Matris Rankı Test	0,584	0,594	0,211	0,601	0,577	0,943	0,822	0,640	0,257
Ayrık Fourier Testi	-	-	0,098	0,025	0,333	0,546	0,158	0,738	0,691
Örtüşmeyen Şablon Eşleştirme Testi	-	-	-	0,543	0,786	0,848	0,148	0,123	0,082
Örtüşen Şablon Eşleştirme Testi	-	-	0,965	0,840	0,969	0,150	0,044	-	0,393

Maurer Testi	-	-	0,233	0,614	0,352	0,434	0,842	0,687	0,648
Doğrusal Karmaşıklık Testi	0,658	0,567	0,214	0,867	0,822	0,816	0,581	0,317	0,579
Seri Testi	-	-	0,116	0,165	0,971	0,920	0,657	0,172	0,340
Yaklaşık Entropi Testi	-	-	-	0,123	0,954	0,806	0,531	0,053	0,426
Kümülatif Topamlar Testi	0,711	-	0,168	0,924	0,605	0,669	0,023	0,791	0,061



## 5. SONUÇ

Rasgele sayı üreteçleri günümüzde oldukça öneme sahip, birçok alanda sıklıkla kullanılan yapılardır. Özellikle kriptoloji alanında kullanıldıklarından ve sistemin güvenliğini direk olarak etkilediklerinden dolayı üretilen sayıların kalitesi ve istenilen özellikleri sağlıyor olması oldukça kritik öneme sahiptir.

Bu tezde üretilen bu rasgele sayıları istatistiksel olarak güçlü hale getirmeyi amaçlayan, şimdiye kadar önerilmiş ve sıklıkla kullanılan son işlem algoritmaları incelenmiştir. En çok kullanılan son işlem yöntemlerinden sekiz tanesi seçilerek, bunların algoritması yazılmış ve araştırmacıların, ihtiyaç duyanların ya da merak edenlerin kullanımına sunulmuştur. Oluşturulan uygulama şimdiye kadar bir örneği olmayan, sade arayüzü ile kullanımı kolay olan ve işlevselliğe önem veren oldukça kullanışlı bir yazılım süitidir.

Fonksiyonel tasarlanan bu uygulama istenildiğinde daha fazla son işlem yöntemi eklenebilecek şekilde tasarlanmıştır. Gelecekte önerilen ya da sık talep edilen başka son işlem algoritmaları da uygulamaya kolaylıkla eklenebilecektir.

Geliştirilen web tabanlı bu yazılıma son kullanıcılar tarafından “postprocess.mersin.edu.tr” adresinden ulaşarak, gerçek rasgele sayı üreteçlerinden elde edilen saf bit dizilerinin son işlem sonuçları alınabilecektir.

## KAYNAKLAR

- [1]. Avaroğlu, E., Türk, M., *Son İşlemin Gerçek Rasgele Sayı Üreteçleri Üzerindeki Etkisinin İncelenmesi*, 6th International Information Security and Cryptology Conference, ISCTURKEY 2013, pp 290–294.
- [2]. Büyüksaraçoğlu, F., Buluş, E., *Sözde Rastsal Sayı Üretiminin Kriptografik Açıdan İncelenmesi*, IV.İletişim Teknolojileri Ulusal Sempozyumu, 15-16 Ekim 2009, Adana, Bildiriler Kitabı, ss. 125-130.
- [3]. Tehranipoor, F., Yan, W., Chandy, J. A., *Robust Hardware True Random Number Generators using DRAM Remanence Effects*, 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 79–84.
- [4]. Hanköylü, S., *Kuantum Rasgele Sayı Üretici Tasarımı ve Uygulaması*, Yüksek Lisans Tezi, Hacettepe Üniversitesi Elektrik ve Elektronik Mühendisliği Anabilim Dalı, Ankara, 2019.
- [5]. Koç, Ç. K., *Cryptographic Engineering*, Springer-Verlag, 2009; p 8.
- [6]. Avaroğlu, E., *Donanım Tabanlı Rasgele Sayı Üreticinin Gerçekleştirilmesi*, Doktora Tezi, Fırat Üniversitesi Elektrik Elektronik Mühendisliği Anabilim Dalı, Elazığ, 2014.
- [7]. Jiteurtragool, N., Masayoshi, T., *Hybrid Random Number Generator Based on Chaotic Oscillator*, The 2016 Management and Innovation Technology International Conference (MITiCON-2016), pp 133-136.
- [8]. Marton, K., Suci, A., Sacarea, C., Cret, O., *Generation and Testing of Random Numbers for Cryptographic Applications*, Proceedings of the Rumanian Academy, Series A, Vol. 13, No. 4, 2012, pp 368– 377.
- [9]. Koyuncu, İ., *Kriptolojik Uygulamalar için FBGA Tabanlı Yeni Kaotik Osilatörlerin ve Gerçek Rasgele Sayı Üreteçlerinin Tasarlanması ve Gerçeklenmesi*, Doktora Tezi, Sakarya Üniversitesi Elektrik Elektronik Mühendisliği Anabilim Dalı, Sakarya, 2014.
- [10]. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Banks, D., *A statistical test suite for random and pseudorandom number generators for statistical applications*, NIST Special Publication in Computer Security, 2001.
- [11]. Coşkun, S., *Kaos Kaynaklı ve ADC Tabanlı Özgün Gerçek Rasgele Sayı Üreteçlerinin Tasarım ve Gerçeklenmesi*, Doktora Tezi, Sakarya Üniversitesi Elektrik Elektronik Mühendisliği Anabilim Dalı, Sakarya, 2017.
- [12]. Büyüksaraçoğlu, F., *Akış şifrelerin tasarım teknikleri ve güç analizi*, Doktora Tezi, Trakya Üniversitesi Fen Bilimleri Enstitüsü, Edirne, 2011.
- [13]. Özkaynak, F., *Kriptolojik Rasgele Sayı Üreteçleri*, Türkiye Bilişim Vakfı Bilgisayar Bilimleri ve Mühendisliği Dergisi, Vol. 8, 2015.
- [14]. Davies, R. B., *Exclusive OR (XOR) and hardware random number generators*, 1-11, <http://www.robertnz.net/pdf/xor2.pdf>
- [15]. Suresh, V. B., Burleson, W. P., *Entropy extraction in metastability-based TRNG*, Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 135–140, 2010.
- [16]. Tsuneda, A., Mitsuishi, S., Inoue, T., *A Study on Generation of Random Bit Sequences with Post-Processing by Linear Feedback Shift Registers*, International Journal of Innovative Computing, Information & Control, 4(10), 2631–2638, 2008.
- [17]. Tsuneda, A., Morikawa, K., *A Study on Random Bit Sequences with Prescribed Auto-Correlations by Post-Processing Using Linear Feedback Shift Registers*, 2013 European Conference on Circuit Theory and Design (ECCTD), 2013.
- [18]. Avaroğlu, E., Tuncer, T., Özer, A.B., Ergen, B., Türk, M., *A novel chaos-based post-processing for TRNG Nonlinear Dynamics*, 81, 189–199, 2015.
- [19]. Dichtl, M., *Bad and Good Ways of Post-processing Biased Physical Random Numbers*, Proceedings of International Workshop on Fast Software Encryption (Luxembourg, Luxembourg, Mar. 26-28, 2007), FSE '07, Lecture Notes in Computer Science, 4593, Springer, Berlin, Germany, 137–152, 2007.

- [20]. Loza, S., Matuszewski, L., *A True Random Number Generator Using Ring Oscillators and SHA-256 as Post-Processings*, International Conference on Signals and Electronic Systems (ICSES) 2014, 1–4.
- [21]. Nikolic, S., Veinovic, M. D., *Advancement of True Random Number Generators Based on Sound Cards Through Utilization of a New Post-processing Method*, Wireless Personal Communications, 91(2), 603–622, 2016.
- [22]. Elias, P., *The efficient construction of an unbiased random sequence*, Ann. Math. Statist, 43(3), 864–870, 1992.
- [23]. Peres, Y., *Iterating Von Neumann's Procedure for Extracting Random Bits*, Annals Statistics, 20(1), 590–597, 1992.
- [24]. Zhang, R., Chen, S., Wan, C. & Shinohara, H., *High-Throughput Von Neumann Post-Processing for Random Number Generator*, 2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 1-4, 2018.
- [25]. Yakut, S., Tuncer, T., Özer, A. B., *Secure and Efficient Hybrid Random Number Generator Based on Sponge Constructions for Cryptographic Applications*, Elektronika Ir Elektrotehnika, 25(4), 40–46, 2019
- [26]. Yakut, S., Tuncer, T., Özer, A. B., *A New Secure and Efficient Approach for TRNG and Its Post-Processing Algorithms*, Journal of Circuits, Systems and Computers, 2020
- [27]. Avaroğlu, E., Tuncer, T., *A novel S-box-based postprocessing method for true random number generation*, Turk. J. Elec. Eng. & Comp. Sci. (2020) 28, 288–301, 2020.
- [28]. Sunar, B., Martin, W. J., Stinson, D. R., *Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks*, IEEE Transactions on Computers 2007, 56 (1), 109–119, 2007.
- [29]. Overview of .NET Framework, <https://docs.microsoft.com/tr-tr/dotnet/framework/get-started/overview> (19.11.2020)
- [30]. Overview of ASP.NET Core MVC, <https://docs.microsoft.com/tr-tr/aspnet/core/mvc/overview?view=aspnetcore-5.0> (19.11.2020)
- [31]. Avaroğlu, E., Tuncer, T., Özer, A.B., Türk, M., *A new method for hybrid pseudo random number generator*, J. Microelectron. Electron. Compon. Mater. 4(4), 303–311, 2014

## ÖZGEÇMİŞ

**Adı ve Soyadı** : Didem YOSUNLU  
**Doğum Tarihi** : 17.01.1993  
**E-mail** : didemoz80@gmail.com

**Öğrenim Durumu** :

Derece	Bölüm/Program	Üniversite	Yıl
Lisans	Bilgisayar Mühendisliği	Dokuz Eylül Üniversitesi	2010 - 2016
Yüksek Lisans	Bilgisayar Mühendisliği	Mersin Üniversitesi	2018 - 2021

## ESERLER (Makaleler ve Bildiriler)

1. *Son İşlem Algoritmalarının İncelenmesi*, Bilgisayar Bilimleri ve Teknolojileri Dergisi, 1(2), 2020