

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

FARKLI MUTASYON STRATEJİLERİ KULLANILARAK XML
TEMELLİ SİSTEMLERDE GÜVENLİK ZAFİYETLERİNİN FUZZ
TESTİ İLE TESPİTİ

DOKTORA TEZİ

Şerafettin ŞENTÜRK

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Bilim Dalı

HAZİRAN 2025

**T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ**

**FARKLI MUTASYON STRATEJİLERİ KULLANILARAK XML
TEMELLİ SİSTEMLERDE GÜVENLİK ZAFİYETLERİNİN FUZZ
TESTİ İLE TESPİTİ**

DOKTORA TEZİ

Şerafettin ŞENTÜRK

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Bilim Dalı

Tez Danışmanı: Prof. Dr. Nejat YUMUŞAK

Ortak Danışman: Dr. Vahid GAROUSHI

HAZİRAN 2025

Şerafettin ŞENTÜRK tarafından hazırlanan “FARKLI MUTASYON STRATEJİLERİ KULLANILARAK XML TEMELLİ SİSTEMLERDE GÜVENLİK ZAFİYETLERİNİN FUZZ TESTİ İLE TESPİTİ” adlı tez çalışması 17.06.2025 tarihinde aşağıdaki jüri tarafından oy birliği ile Sakarya Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı’nda Doktora tezi olarak kabul edilmiştir.

Tez Jürisi

Jüri Başkanı :	Prof. Dr. Pakize ERDOĞMUŞ Düzce Üniversitesi	
Jüri Üyesi :	Prof. Dr. Nejat YUMUŞAK (Danışman) Sakarya Üniversitesi	
Jüri Üyesi :	Prof. Dr. Celal ÇEKEN Sakarya Üniversitesi	
Jüri Üyesi :	Prof. Dr. Ünal ÇAVUŞOĞLU Sakarya Üniversitesi	
Jüri Üyesi :	Prof. Dr. Resul KARA Düzce Üniversitesi	



ETİK İLKE VE KURALLARA UYGUNLUK BEYANNAMESİ

Sakarya Üniversitesi Fen Bilimleri Enstitüsü Lisansüstü Eğitim-Öğretim Yönetmeliğine ve Yükseköğretim Kurumları Bilimsel Araştırma ve Yayın Etiği Yönergesine uygun olarak hazırlamış olduğum “FARKLI MUTASYON STRATEJİLERİ KULLANILARAK XML TEMELLİ SİSTEMLERDE GÜVENLİK ZAFİYETLERİNİN FUZZ TESTİ İLE TESPİTİ” başlıklı tezin bana ait, özgün bir çalışma olduğunu; çalışmamın tüm aşamalarında yukarıda belirtilen yönetmelik ve yönergeye uygun davrandığımı, tezin içerdiği yenilik ve sonuçları başka bir yerden almadığımı, tezde kullandığım eserleri usulüne göre kaynak olarak gösterdiğimi, bu tezi başka bir bilim kuruluna akademik amaç ve unvan almak amacıyla vermediğimi ve 20.04.2016 tarihli Resmi Gazete’de yayımlanan Lisansüstü Eğitim ve Öğretim Yönetmeliğinin 9/2 ve 22/2 maddeleri gereğince Sakarya Üniversitesi’nin abonesi olduğu intihal yazılım programı kullanılarak Enstitü tarafından belirlenmiş ölçütlere uygun rapor alındığını, çalışmamla ilgili yaptığım bu beyana aykırı bir durumun ortaya çıkması halinde doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi beyan ederim.

(17/06/2025).

Şerafettin ŞENTÜRK





Eşime, çocuklarıma ve pek kıymetli büyüklerime



TEŞEKKÜR

Her şeyden önce, lisansüstü çalışmalarım boyunca verdiği destek, cesaretlendirme ve sabır dolu rehberlikleri için danışmanım Prof. Dr. Nejat Yumuşak'a teşekkür etmek istiyorum. Onun gözetimi ve desteği beni araştırmayı yürütmeye, geniş bir bilgi edinmeye, kendimi sürekli geliştirmeme ve bu tezi tamamlamaya olanak sağladı. Doktora çalışmalarım süresince kendisinin rehberliği ve motive edici duruşu olmasaydı araştırmayı yürütmem ve bitirmem zor olurdu. Bu sebeple Prof. Dr. Nejat Yumuşak'ın öğrencilerinden biri olmaktan gurur duyuyorum.

Ayrıca doktora çalışmam boyunca katkısı ve kıymetli tavsiyeleri ile yol gösterici olan Dr. Vahid Garousi'ye teşekkür etmek istiyorum. Tezim hakkındaki değerli yorumları ve geri bildirimleri için kendisine minnettarım.

Diğer tez komite üyelerim Prof. Dr. Celal Çeken ve Prof. Dr. Ünal Çavuşoğlu'na da tezim boyunca süre gelen destekleri ve değerli yorumları için teşekkür etmek istiyorum.

Şerafettin ŞENTÜRK



İÇİNDEKİLER

Sayfa

ETİK İLKE VE KURALLARA UYGUNLUK BEYANNAMESİ	v
TEŞEKKÜR	ix
İÇİNDEKİLER	xi
KISALTMALAR	xiii
SİMGELER	xv
TABLO LİSTESİ	xvii
ŞEKİL LİSTESİ	xix
ÖZET	xxi
SUMMARY	xxv
1. GİRİŞ	1
1.1. Yazılım Güvenlik Testleri	1
1.2. Yazılım Güvenlik Fuzz Testleri	2
1.3. Motivasyon ve Kapsam	3
2. LİTERATÜR TARAMASI	7
2.1. Fuzz Testleri Tarihçesi	8
2.2. Fuzz Testi Uygulama Alanları	8
2.3. Fuzz Testi Sınıflandırması	11
2.3.1. Kara kutu testleri	11
2.3.2. Beyaz kutu testleri	11
2.3.3. Gri kutu testleri	13
2.4. Girdi Üretim Stratejisine Göre Fuzz Testleri	14
2.4.1. Mutasyon tabanlı girdi üretimi	14
2.4.2. Gramer tabanlı girdi üretimi	16
2.5. Benzer Çalışmalar	17
2.5.1. Gramer tabanlı fuzz testleri	17
2.5.2. Öğrenme tabanlı fuzz testleri	21
2.5.3. İyi yapılandırılmış girdi dosyası fuzz testleri	24
3. FUZZ TESTİ VERİMLİLİK ARTIRMA TEKNİKLERİ	29
3.1. Kapsama Yönlendirmeli Fuzz Testleri	30
3.2. Otomat Tabanlı Sistemler	33
3.3. Yönlendirmeli Gri Kutu Sistemleri	35
3.4. Makine Öğrenmesi Tabanlı Sistemler	38
4. ÇALIŞMA BULGULARI	43
4.1. Hedefler ve Araştırma Soruları	48
4.2. Çalışmanın Temel Öğeleri	49
4.3. Kurulum	50
4.3.1. Test senaryosu hazırlığı	50
4.3.2. Fuzz proses akışı	50
4.3.3. AFL (American Fuzzy Lop)	51
4.3.4. Superior	52
4.3.5. Akıllı tohum üretimi	53

4.3.6. Olasılıksal bağlam duyarlı dilbilgisi	53
4.3.7. Fuzz testi çalıştırmaları ve sonuçları	54
5. ÇALIŞMA BULGULARI.....	57
5.1. RQ1- Bayt Seviyesinde Mutasyon Stratejisi İle Ağaç Seviyesinde Mutasyon Stratejisi Test Etkinliği Açısından Nasıl Karşılaştırılır?	57
5.2. RQ2-Öğrenme Tabanlı Oluşturulan Tohumlar, Genel Seçilmiş Tohumlara Göre Hata Bulmada Ne Kadar Etkili?	65
5.3. XML zafiyetlerine yönelik çalışma	72
5.4. Sonuçlar ve gelecek çalışmalar	75
KAYNAKLAR.....	77
ÖZGEÇMİŞ.....	83



KISALTMALAR

AFL	: American Fuzzy Lop
ANTLR	: Another Tool for Language Recognition
AST	: Abstract Syntax Tree
CFG	: Context Free Grammar
CGF	: Coverage Guided Fuzzing
CNN	: Convolutional Neural Network
FSA	: Finite State Automata
GAN	: Generative Adversarial Network
GBF	: Grammar Based Fuzzing
GNF	: Greibach Normal Form
GQM	: Goal Question Metric
PCSG	: Probabilistic Context Sensitive Grammar
PDA	: Push Down Automata
POC	: Proof Of Concept
SCSA	: Simple Client Server Architecture
SUT	: System Under Test



SİMGELER

Σ	: Toplam operatörü
δ	: Tüm geçişlerin kümesi
Γ	: Sembol kümesi
Q	: Sonlu bir durum kümesi





TABLO LİSTESİ

Sayfa

Tablo 2.1. Fuzz testi metotları ve değerlendirme metrikleri.	27
Tablo 5.1. Xerces için hata tespit performanları.	70
Tablo 5.2. Xerces için test senaryo ve hata korelasyonu	71
Tablo 5.3. Test edilen üç sistem için hata tespit sonuçları.	72
Tablo 5.4. Test edilen üç sistem için benzersiz test senaryoları.	72





ŞEKİL LİSTESİ

Sayfa

Şekil 2.1. Fuzz testi uygulama alanları	10
Şekil 3.1. Kapsama geri bildirimli fuzz test süreci	32
Şekil 4.1. Fuzz testleri için ana mimari akış	51
Şekil 4.2. AFL test sonuçları ekranı.....	51
Şekil 5.1. Fuzz testi için oluşturulan benzersiz test senaryoları	64
Şekil 5.2. Üç defa yapılan test için ANOVA sonuçları	65
Şekil 5.3. XML ayrıştırıcıları için üretilen test senaryoları	67
Şekil 5.4. Test senaryosu ile hata arasındaki korelasyon	71



FARKLI MUTASYON STRATEJİLERİ KULLANILARAK XML TEMELLİ SİSTEMLERDE GÜVENLİK ZAFİYETLERİNİN FUZZ TESTİ İLE TESPİTİ

ÖZET

Fuzz testleri, yazılım sistemlerindeki hataları ve güvenlik açıklarını tespit etmek için kullanılan otomatize edilmiş işlemler bütünü olarak bilinir. Genel itibarıyla fuzz testleri üçe ayrılmaktadır; kara kutu testler, beyaz kutu testler ve gri kutu testler. Fuzz testlerini başlatmak için kullanılan girdi verilerinin üretimi açısından ise fuzz testleri ikiye ayrılır; birisi dilbilgisi tabanlı, diğeri ise mutasyon tabanlıdır. Dilbilgisi tabanlı fuzz testi, bir spesifikasyondan girdiler üretir ve karmaşık yapılandırılmış girdiler alırken, mutasyon tabanlı fuzz, iyi biçimlendirilmiş başlangıç dosyalarını ve soyut sözdizimi ağaçlarını rastgele değiştirerek girdiler üretir. Dilbilgisi veya mutasyon tabanlı çalışan fuzz testi yaklaşımları vardır, ancak bunların hata tespit yeteneklerini karşılaştıran yeterli sayıda vaka çalışması yoktur. Bu alandaki deneysel kanıtlara katkıda bulunabilmek için, bu vaka çalışması gri kutu fuzz testini farklı mutasyon stratejileriyle karşılaştırarak üç açıdan etkinliklerini değerlendirir: hata tespit etkinliği, hata tespit performansı ve tespit edilen hata türleri. Ayrıca, farklı başlangıç oluşturma tekniklerinin fuzz testinin etkinliğine olan etkilerini araştırıyoruz. Fuzz testi yürütmeleri için üç farklı test edilen sistem seçilmiştir; XML ayrıştırıcıları, Libxml2, Apache Xerces ve Expat. Fuzz testlerini uygulamak için farklı mutasyon stratejilerine sahip iyi bilinen fuzz araçları olan AFL (American Fuzzy Lop) ve Superion aracı kullanılmıştır. Genele açık olan tohumlar ve PCSG (Olasılıksal Bağlam Duyarlı Dilbilgisi) tabanlı tohumlar kullanılarak tohum üretiminin fuzz testlerine olan etkilerini bulmak için araştırma yürütülmüştür. Hata tespiti etkinliği ve performansı açısından, çökme sayısı ve üretilen test vaka dosya sayısı ölçülmüştür. Kullanılan mutasyon stratejileri açısından, sonuçlar bit/bayt düzeyindeki mutasyon stratejisinin ağaç düzeyindeki mutasyon stratejisinden daha fazla hata tespit ettiğini göstermektedir. Fuzz testleri sonuçlarına göre, PCSG tabanlı tohumlar genele açık olanlardan daha fazla çökme tespit etmeye yardımcı olmaktadır. Test vaka dosyası üretimi açısından, genele açık olarak seçilen tohumlara kıyasla PCSG tabanlı tohumlar için daha az test vakası üretilirken, bit/bayt düzeyindeki mutasyonlar ağaç düzeyindeki mutasyonlara kıyasla daha fazla test vakası ile sonuçlanmaktadır. Ampirik sonuçlar, fuzz testlerinin çökme tespit yeteneklerinin kullandığı mutasyon stratejisine göre önemli ölçüde farklılık gösterdiğini ortaya çıkarmıştır.

Fuzz testleri, yazılımın çalışamaz hale geldiği durumları ve güvenlik açıklarını otomatik şekilde bulmak için yaygın olarak kullanılan etkili bir tekniktir. Fuzz testlerinde, sistem başka bir program tarafından oluşturulan sürekli ve rastgele test durumlarıyla test edilir. Aynı zamanda, sistem bu girdiyi işlerken herhangi bir hatayı ortaya çıkarmak için izlenir. Bu nedenle, her fuzz test metodunun yürütme motorunda çalıştırılacak test durumlarını otomatik olarak oluşturması gerektiğinden girdi oluşturma, test süreci için önemlidir. Fuzz testlerinde girdi ve test senaryoları oluşturmak için iki temel biçim vardır: mutasyon ve dilbilgisi tabanlı girdi oluşturma. Birincisi girdi biçiminin farkındayken, ikincisi değildir. Birincisi bir referans

girdisinde dönüşümler içerirken, ikincisi sistem spesifikasyonunu kullanır. Mutasyon tabanlı fuzz test metotları çoğunlukla kapsama alanından yoksundur ve beklenenden farklı girdilerle daha az etkilidir.

Mutasyon tabanlı yaklaşımlarda, girdideki değişiklik bit/bayt düzeyinde gerçekleştirilir. Bu yöntemler, yeni dosyalar oluşturmak için bazı bitleri rastgele çevirir, siler veya kopyalar. Ancak, rastgele bit çevirmelerinin karmaşık dosya biçimlerini işleyen uygulamalar için geçerli dosyalar üretmesi olası değildir. Bu nedenle, mutasyona dayalı fuzz testi, iyi yapılandırılmış girdileri (örneğin XML, JavaScript) işleyen programlar için pek uygun değildir. Bu tür programlar girdileri üç temel aşamada alır ve işler; sözdizimi ayrıştırma, anlamsal kontrol ve uygulama yürütme. Mutasyona dayalı fuzz testleri tarafından üretilen çoğu kötü biçimli girdi, sözdizimi ayrıştırmada başarısız olur ve bu nedenle işlemenin erken bir aşamasında reddedilir.

Dilbilgisi tabanlı fuzz testleri, girdi modelleri veya veri parçalarının biçim ve bütünlük kısıtlamalarını belirleyen bağlamdan bağımsız dilbilgisi gibi iyi tanımlanmış bir spesifikasyona göre girdiler üretir. Mutasyon tabanlı yaklaşımların aksine, dilbilgisi tabanlı yaklaşımlar fuzz testlerinde sözdizimi ayrıştırma aşamasını daha kolay geçer. Ancak bu yöntemle üretilen test senaryolarının çoğu genellikle anlamsal kontrolden geçmez. Bununla birlikte, anlamsal olarak geçerli test senaryolarının sistematik olarak üretilmesi genellikle çözülmesi oldukça masraflı ve zor bir durumdur. Bu aynı zamanda programın hem sözdizimsel hem de anlamsal özelliklerinin farkında olan akıllı bir fuzz test aracı geliştirmek için yaygın olarak bilinen bir sorundur. Bu nedenle, dilbilgisi tabanlı fuzz testlerinde üretilen girdinin yalnızca küçük bir kısmı, derin hataların gizlendiği uygulama yürütme aşamasına ulaşabilir; bu nedenle, uygulama kodunun çoğuna erişilemez ve sistemde çökmeye neden olan vakaların çoğu açığa çıkarılamaz.

Bu sorunun üstesinden gelmek için, PCSG'ye (Olasılıksal Bağlam Duyarlı Dilbilgisi) dayalı olarak çalışan bazı yaklaşımlar vardır. Bu yaklaşımlar, dil bilgisi ve anlam bilim kurallarının bilgisini otomatik olarak çıkarmak ve ardından çökmeye neden olan durumları tetiklemesi beklenen girdiler üretmek için çok sayıda örnek kullanır. Bu akıllı girdilerin, anlam bilimsel ayrıştırma aşamasını kolayca geçebildiği ve uygulama yürütme aşamasına ulaşabildiği için, test edilen uygulamada gizli olan bilinmeyen hata ve çökmeleri tetikleme şansının daha fazla olduğu iddia edilmektedir. Bu nedenle, öğrenme algoritması açısından üretilen akıllı girdilerin etkinliğini, genele açık girdi dosyalarına kıyasla çökmeleri bulma açısından daha iyi performans gösterebileceklerini ortaya çıkarmaya yönelik artan bir ihtiyaç vardır. Bu tez çalışmasında, iyi yapılandırılmış girdiler alan XML ayrıştırıcıları için fuzz testi yürütmeleri gerçekleştirme açısından bu iki tür girdi dosyası arasında kapsamlı bir karşılaştırma yaparak bu ihtiyaca vurgu yapmaktayız.

Bu tez çalışması boyunca, fuzz testi yürütmeleri için AFL ve Superion olmak üzere iki ana test aracı kullanıyoruz. AFL, CGF (Kapsam Rehberli Fuzz Testi) prensiplerine göre çalışan son teknoloji gri kutu test aracıdır. Superion, ayrıştırılmış girdilerin soyut sözdizimi ağaçlarını (AST'ler) kullanarak ağaç düzeyinde test girdilerini kırpma için dilbilgisi farkında bir kırpma stratejisi kullanır. AFLdeki mutasyonlar dilbilgisi körü olup bit/bayt düzeyinde çalışırken, Superiondaki mutasyonlar dilbilgisi farkındadır ve ağaç düzeyinde işlev görür. Ayrıca, AFLdeki dilbilgisi körü mutasyon stratejilerinin etkinliği ve verimliliği engellediği iddia edilmektedir. Öte yandan, Superion iyi yapılandırılmış girdi dosyaları için oldukça uygundur, yani fuzz testi için geçerli

girdiler üretebilir. Bu anlamda, üretilen ve genele açık girdi dosyaları için nasıl performans gösterdiklerini bulmak için bit/bayt ve ağaç düzeyindeki mutasyonlar arasında karşılaştırma yapılması da gerekir. Bu açıdan, test çalışmamız ayrıca farklı mutasyon stratejilerinin ve bunların çökme algılama yetenekleri üzerindeki etkilerinin karşılaştırılmasını içermektedir.

Giriş oluşturma yöntemlerine ve mutasyon stratejilerine ek olarak, fuzz testlerindeki bir diğer ölçüt kod kapsamıdır. Bu, fuzz testi yürütmesinin test edilen programın farklı bölümlerine ne kadar ulaşabildiğiyle ilgilidir. Fuzz testi araçlarının yeni girdinin kalitesini değerlendirmek ve bu girdinin oluşturulmasını yönlendirmek için bir yardımcı işleve ihtiyacı vardır. Bu yardımcı işlev temelde kod kapsamı olabilir. Bu bağlamda, yardımcı işlev olarak mutasyon testi ile gri kutu fuzz testini yönlendirmek için bazı çalışmalar vardır. Çalışmalara göre, mutasyon testinin etkili gövde giriş dosyaları oluştururken kod kapsamına bir alternatif olduğu gösterilmiştir, bu da bir test gövdesinin yapay olarak enjekte edilen hataları algılama yeteneğine göre değerlendirilebileceği anlamına gelir. Ayrıca çalışma, fuzz testi kalitesini değerlendirmek için bir ölçüt olarak mutasyon puanını kullanır.

Gri kutu fuzz testi yaklaşımları için bu çalışma, vaka çalışması araştırması ilkelerine dayanarak yürütülmüştür. Vaka çalışmamızın doğası hem "keşifsel" hem de "iyileştirici"dir; amacımız (fuzz test yaklaşımını geliştirme sürecinde) ne olduğunu bulmak, yeni içgörüler aramak ve takip iyileştirmeleri (fuzz test yaklaşımında) ve araştırma (keşifsel yön) için fikirler ve hipotezler üretmektir. Ayrıca, "çalışılan olgunun" test etkinliğini ve verimliliğini iyileştirmek istediğimiz için çalışmanın "iyileştirici" yönleri vardır.



DETECTION OF SECURITY VULNERABILITIES IN XML-BASED SYSTEMS WITH FUZZ TESTING USING DIFFERENT MUTATION STRATEGIES

SUMMARY

Fuzz tests are known as a set of automated processes used to detect crashes and security vulnerabilities in software systems. In general, fuzz tests are divided into three; black box tests, white box tests and gray box tests. In terms of the generation of input data used to initiate fuzz tests, fuzz tests are divided into two; one is grammar-based and the other one is mutation-based. While the grammar-based fuzzing generates inputs from a specification and takes complex structured inputs, mutation-based fuzzing generates inputs by modifying well-formed seed files and abstract syntax trees randomly. There are fuzzing approaches working grammar or mutation based, but there are not enough case studies comparing their crash detection capabilities. To add to the body of empirical evidence in this area, this case study compares gray-box fuzzing with different mutation strategies to evaluate their effectiveness in three aspects: fault detection effectiveness, fault detection performance and types of faults detected. Also we investigate the affects of different seed generation techniques to fuzzing effectiveness. We perform the fuzzing executions on three Systems Under Test (SUT); XML parsers, Libxml2, Apache Xerces and Expat. To execute fuzzing, we use the well-known fuzzers AFL (American Fuzzy Lop) and Superior which have different mutation strategies. We carry out the investigations for finding the affects of seed generation to fuzzing by leveraging publicly-available seeds and the PCSG (Probabilistic Context Sensitive Grammar) based seeds. In terms of fault detection effectiveness and performance, we measure the number of crashes, and the number of corpus files generated. With respect to mutation strategies used, our results demonstrate that the bit/byte-level mutation strategy detects more crashes than tree-level mutation strategy. According to the fuzzing results, PCSG-based seeds help detect higher number of crashes than publicly-available ones. In terms of corpus file generation, while there is less corpus generated for PCSG based seeds compared to publicly-selected ones, bit/byte level mutations result in more corpus when compared to tree level mutation. Empirical results show that crash detection capabilities of fuzzing differ significantly based on the mutation strategy it uses.

Fuzzing is an effective and popular technique for finding crashes and vulnerabilities in an automated way. The system is tested with the continuous and random test cases generated by another program. At the same time, the system is monitored to expose any defects while processing this input. Therefore, input generation is important for the fuzzing process since every fuzzer should automatically generate test cases to be run on the execution engine. There are two fundamental forms to create input and test scenarios in fuzz tests: mutation- and grammar-based input generation. While the first is aware of input format, the latter is not. The former includes transformations on a reference input, while the latter employs system specification. Mutation-based fuzzers

mainly lack coverage and are less effective, with inputs that are different from what is expected.

In mutation-based approaches, the modification on the input is performed at the bit/byte level. These methods randomly flips, deletes or copies some bits to generate new files. However, random bitflips are not likely to produce valid files for applications processing complex file formats. Therefore, mutation-based fuzzing is not well suited for programs that handle highly structured inputs (e.g. XML, JavaScript). Those types of programs take and process inputs in three basic stages; syntax parsing, semantic checking, and application execution. Most malformed inputs produced by mutation-based fuzz tests fail at syntax parsing and are therefore rejected at an early stage of processing.

Grammar-based fuzz tests generate entries according to a well-defined specification, such as input models or context-free grammar that determine the format and integrity constraints of data pieces. In contrary to mutation-based approaches, grammar-based approaches pass the syntax parsing stage in fuzz tests more easily. But most of the test cases produced by this method usually do not pass the semantic check. However, systematic generation of semantically valid test cases is often a highly expensive and difficult problem to solve. This is also a widely known issue for developing an intelligent fuzz testing tool which is aware of both syntactic and semantic features of the program. So, only a small fraction of the input produced in grammar-based fuzz tests can reach the application execution stage, where deep errors are hidden; therefore, most of the application code cannot be accessed and most of the crash inducing cases cannot be exposed.

In order to overcome this issue, there are some approaches that work based on the PCSG (Probabilistic Context Sensitive Grammar). Those approaches use a large number of examples to automatically extract knowledge of grammar and semantic rules and then generate seeds that are expected to trigger crash inducing cases. It is claimed that since these smart seeds can pass the semantic parsing stage easily and reach out the application execution phase, it gains more chance to trigger the unknown bugs and crashes hidden in the tested application. For this reason, there is a growing need to show the effectiveness of the smart seeds generated in terms of a learning algorithm that they can outperform in terms of finding crashes compared to publicly-available seed files. In this paper, we address this need by making exhaustive comparison between these two types of seed files in terms of performing fuzzing executions for XML parsers that take highly structured inputs.

Throughout this study, we use two main fuzzers, namely AFL and Superion, for the fuzzing executions. AFL is the state-of-the art gray-box fuzzer which functions based on CGF (Coverage Guided Fuzzing) principles. Superion uses a grammar-aware trimming strategy to trim test inputs at the tree level by using the abstract syntax trees (ASTs) of those parsed inputs. Whereas mutations in AFL are grammar-blind and works in bit/byte level, mutations in Superion are grammar-aware and functioning in tree level. It is also claimed that grammar-blind mutation strategies in AFL hinder the effectiveness and efficiency. On the other hand, Superion is well-suited for complex-structured input files meaning that it can produce valid inputs for fuzzing. In this sense, there is also need to compare between the bit/byte- and tree-level mutations to find out how they perform for generated and publicly available seed files. From this angle, our case study also involves comparing different types of mutation strategies and their effects on their crash detection capabilities.

In addition to input generation methods and mutation strategies, another metric in fuzzing is code coverage. It is related with how far the fuzzing execution can reach out different parts of the tested program. Fuzzing tools need a helping function to evaluate the quality of the new input and guide the generation of this input. This helping function can be basically code coverage. In this context, there are some studies to guide gray-box fuzzing with mutation testing as a helping function. According to their study, mutation testing has shown to be an alternative to code coverage while generating effective corpus input files, meaning that a test corpus can be evaluated on its ability to detect artificially injected faults. Also the study uses the mutation score as a metric for evaluating fuzzing quality.

This study for the gray-box fuzzing approaches is conducted based on the principles of case-study research. The nature of our case study is both “exploratory” and “improving” in that our objective is to find out what happened (in the process of developing the fuzz testing approach), to seek new insights, and to generate ideas and hypotheses for follow-up improvements (on the fuzz test approach) and research (the “exploratory” aspect). Furthermore, the study has “improving” aspects since we wanted to improve test effectiveness and efficiency of the “studied phenomenon”.



1. GİRİŞ

Güvenlik ve mahremiyet konuları yaşamın birçok alanında olduğu gibi dijital sistemlerde de her geçen gün önemli hale gelmiştir. Teknoloji yazılımları, web tabanlı ve mobil uygulamalar, kritik altyapı sistemleri, akıllı şebekeler, akıllı şehirler ve otonom araçlar başta olmak üzere birçok özel sektör ve kamu alanındaki kurumlara siber saldırılar ve tehditler devamlı olarak artmaktadır. Bu bağlamda giderek dijitalleşen ve uzaktan kontrolü daha kolay hale gelen sistemler, işlevselliği geliştirme, maliyetleri düşürme ve verimliliği artırma avantajları sunarken, aynı zamanda ciddi güvenlik ve emniyet zorluklarını da beraberinde getirmektedir. Dolayısıyla bu tip sistemler fiziksel çevreye ve genel kamuoyuna fiziksel zarar ve hasara neden olabilecek ciddi olaylara yol açabilecek çeşitli güvenlik zafiyetlerine eğilimli bulunmaktadır. Bu nedenle, bu sistemlerde mevcut olan karmaşık yazılımlar için güvenlik, emniyet ve sağlamlık kriterlerini karşılamak için daha akıllı ve profesyonel yöntemler önemli hale gelmiştir.

Bu sorunlar göz önünde bulundurularak teknoloji yazılımlarının ve bunların üzerindeki uygulamaların hızlı prototiplenmesi için endüstrinin ihtiyaçlarını karşılayan güçlü ve güvenli bir sistemin sağlanması gerekmektedir. Bahsi edilen uygulamaların siber saldırılara dayanıklı ve güvenli hale gelebilmesi için dış ortamlardan gelebilecek saldırıların sisteme sızabileceği olası giriş noktalarının ve bu noktalardaki zafiyetlerin farkında olunması gerekmektedir. Çünkü sistemlere yapılan bu saldırıların büyük bir çoğunluğu sistemlerin bünyesinde mevcut olan donanımsal ve yazılımsal varlıkların üzerinde bulundurduğu güvenlik zafiyetlerinin istismar edilmesi neticesinde gerçekleştirilmektedir. Bu sebeple sistemlerde var olan güvenlik ve mahremiyet zafiyetlerinin, saldırılar oluşmadan önce bilinmesi ve tespit edilmesi büyük önem arz etmektedir.

1.1. Yazılım Güvenlik Testleri

Yazılım sistemlerinde görülen hatalar ile güvenlik zafiyetleri arasında yakın bir ilişki bulunmaktadır. RFC 2828'de (Shirey 2000) tanımlandığı üzere; güvenlik açığı, sistemin güvenlik politikasını ihlal etmek için kullanılabilecek bir sistemin tasarımı,

uygulanması veya işletimi ve yönetimindeki bir kusur veya zayıflıktır. Yazılım sistemlerinde kritik hatalar güvenlik açıklarına yol açar. Yazılım açıkları, saldırganların veri yapılarını bozmasına, kötü niyetli kod yürütmesine ve hatta yazılımın çalıştığı tüm sistemi kontrol etmesine olanak tanır. Sıfır gün güvenlik açıklarında olduğu gibi güvenlik açıklarına yönelik saldırılar, ciddi hasarlara ve etkilere neden olabilir. Güvenlik açıklarının neden olduğu ciddi zararlarla ilgili olarak, bilgi ve yazılım sistemlerine yönelik, güvenlik açığı bulma amacıyla çok sayıda çalışma yapılmıştır [1]. Yazılımlarda en temelde güvenlik açıklıklarının bulunabilmesi için bir güvenlik analizi çalışması ve sistemin istenilen seviyede güvenli olup olmadığının kontrolü için güvenlik testleri yapılmalıdır.

Sistemlerin güvenlik testleri ve zafiyet taraması için kullanılan yöntemler en genel anlamda statik ve dinamik analiz olarak sınıflandırılmaktadır. Statik Kod Analizi olarak da isimlendirilen güvenlik testlerinde uygulamanın kodları uygulama çalıştırılmadan testlere tabi tutulmaktadır. Bu tip testlerdeki en büyük sorun testler sonucunda ortaya çıkan yüksek yanlış bulgu oranlarıdır. Yani sistemlerde aslında bir güvenlik zafiyeti olmayan noktaların bir güvenlik zafiyeti olarak tespit edilmesi durumuyla bu tip testlerde sıklıkça karşılaşılmaktadır. Dinamik analiz adı verilen ve çalışan sistemler üzerinde yapılan güvenlik testleri arasında sızma testleri ve fuzz testleri gelmektedir. Sızma testleri(pentest), bir ortamdaki ve uygulamadaki zayıflıklardan aktif olarak yararlanmaya, zafiyetleri istismar etmeye çalışır ve çeşitli seviyelerde teknik uzmanlık gerektirir.

1.2. Yazılım Güvenlik Fuzz Testleri

Yazılımlarda uygulanan bir dinamik güvenlik test tipi olan fuzz testleri sistemlerdeki güvenlik zafiyetlerini otonom bir şekilde ortaya çıkarmak için kullanılan, sistemi çok fazla sayıda ve yarı geçerli(semi valid) girdi değerleriyle çalıştırarak bunların sonucunda sistemin verdiği yanıtları irdeleme yoluyla sistemde diğer yöntemlerle bulunması zor ve derin hataların bulunmasını sağlar. Fuzz testleri çalışan sistem üzerinde yürütüldüğü için statik kod analizinde karşılaşılan yanlış bulgu sayısının çokluğu bu tip testlerde ortaya çıkmamaktadır. Diğer yandan güvenlik sızma testlerinde gerekli olan teknik uzmanlık seviyesi fuzz testlerinde gerekli değildir dolayısı ile bu testlere başlangıç yapmak ve ilerlemek kolaydır.

1.3. Motivasyon ve Kapsam

Fuzz testleri esnasında sırasında sistem, başka bir program tarafından oluşturulan sürekli ve rastgele test senaryoları ile test edilir. Aynı zamanda, test senaryoları işlenirken herhangi bir çökme veya hatayı ortaya çıkarmak için sistem izlenir [2]. Bu nedenle, her fuzz test aracının otomatik olarak test senaryosu oluşturması gerektiğinden test girdisi oluşturma işlemi fuzz testleri için önemli bir rol oynar. Fuzz testlerinde test senaryoları oluşturmak için iki temel yaklaşım vardır: dilbilgisi ve mutasyona dayalı oluşturma. Dilbilgisine dayalı olan yaklaşım girdi formatının farkındayken, mutasyona dayalı olan yaklaşım formatın farkında değildir [3]. İlki sistem spesifikasyonunu kullanırken, ikincisi bir referans girdisinde dönüşümler içerir. Mutasyona dayalı fuzz test yaklaşımı çoğunlukla kapsama alanından yoksundur ve beklenenden farklı girdi üretimi neticesinde daha az etkili sonuçlar vermektedir [4].

Mutasyona dayalı test senaryosu üretim yaklaşımlarında girdideki değişiklik genellikle bayt düzeyinde gerçekleştirilir. Bu yöntemler, yeni dosyalar oluşturmak için bazı bitleri rastgele çevirir, siler veya kopyalar. Ancak, rastgele bit çevirmelerinin karmaşık dosya biçimlerini işleyen uygulamalar için geçerli dosyalar üretmesi olası değildir. Bu nedenle, mutasyona dayalı fuzz testleri, iyi yapılandırılmış girdileri (ör. XML, JavaScript) işleyen programlar için pek uygun değildir. Bu tür programlar girdileri üç temel aşamada alır ve işler; sözdizimi ayrıştırma, anlamsal kontrol ve uygulama yürütme. Mutasyona dayalı fuzz testleriyle üretilen çoğu kötü biçimli girdi, sözdizimi ayrıştırmada başarısız olur ve bu nedenle işlemenin erken bir aşamasında reddedilir. Bu zorluğu çözmek için, dilbilgisi tabanlı fuzz (GBF), yeni girdiler oluşturmak için mevcut dilbilgisi özelliklerini kullanır. Bazı GBF'ler, dilbilgisine uygun girdiler üretebilen ve iyi yapılandırılmış girdileri kabul eden programlar için çalışabilen soyut sözdizimi ağaçlarına (AST'ler) dayalı mutasyon gerçekleştirir. Ağaç seviyesinde çalışan mutasyon stratejileri öncelikle iyi biçimlendirilmiş fuzz testi girdilerini soyut sözdizimi ağacı adı verilen bir ağaç yapısına dönüştürür ve ağaç üzerinde gerekli mutasyonları yapar. Bu şekilde, ağaç seviyesinde çalışan GBF'ler mutasyonun rehberliği olarak önceden tanımlanmış dilbilgilerinden yararlanır, böylece üretilen girdilerin ayrıştırıcının sözdizimi ve anlamsal kontrollerinden geçme olasılığı yüksek olur, geçersiz girdilerin sayısı azalır ve fuzz testlerinin etkinliği artar.

Bununla birlikte, dilbilgisi tabanlı fuzz testleri ve farklı mutasyon stratejilerinde önemli ilerleme son yıllarda artarken, farklı mutasyon stratejilerine sahip fuzz

testlerinin etkinliğini ölçmek ve karşılaştırmak hala açık bir araştırma sorusudur [5]. Son zamanlarda yapılan bir ankette, fuzz testi uygulayıcılarının %63'ü çözülmesi gereken en önemli üç zorluktan biri olarak fuzz testi yaklaşımlarının karşılaştırması için ölçümleri öne çıkarmıştır [6]. Bu nedenle, tez çalışmamız kapsamında farklı mutasyon stratejileriyle fuzz testlerinin etkinliğini karşılaştırma ve değerlendirme kriteri olarak uygun metrikleri bulma konusu detaylı incelenmektedir. Fuzz testleri de dahil olmak üzere test teknikleri, özet sonuçları önceki fuzz yaklaşımları değerlendirmelerinde de açıkça görülebileceği üzere, genellikle tespit edilen hatalara göre değerlendirilir. Bu nedenle, bu tez çalışmasında şu ana araştırma sorusunu (RQ) cevaplamayı amaçladık: Bayt düzeyindeki mutasyon stratejisi ve ağaç düzeyindeki mutasyon stratejisi test etkinliği açısından nasıl karşılaştırılır? Bu ana araştırma sorusu altında, bu tez çalışmasının hedefleri fuzz testlerinin üç farklı yönünü kapsamaktadır: hata tespit etkinliği, hata tespit performansı ve tespit edilen hata türleri. Bu hedeflere ulaşmak için, açık kaynaklı fuzz testi araçları olan AFL ve Superion aracını amprik olarak değerlendirmekteyiz.

AFL en yaygın kullanılan mutasyon tabanlı fuzz testi aracıdır ve CGF (kapsam rehberli fuzz) prensiplerine dayalı olarak çalışır. Superion aracı ise dilbilgisi odaklı mutasyona dayalı son teknoloji bir fuzz testi aracıdır ve hem kararlı sözdizimsel doğruluk hem de fuzz testi verimliliği elde edebilir [7]. AFLdeki mutasyonlar dilbilgisi körüdür ve bayt düzeyinde çalışır. Superiondaki mutasyonlar dilbilgisi farkındadır ve ağaç düzeyinde çalışır. Superion, ayrıştırılan girdilerin soyut sözdizimi ağaçlarını (AST'ler) kullanarak ağaç düzeyinde test girdilerini kırpma için dilbilgisi farkında bir kırpma stratejisi kullanır. AFLin dilbilgisi körü mutasyon stratejileri, sözdizimsel ve anlamsal olarak geçerli test girdileri üretme yeteneğinden yoksundur. İyi biçimlendirilmiş girdileri işlemekten uzaktırlar. XML ve JavaScript gibi iyi biçimlendirilmiş girdiler için araştırmacılar, mutasyonun ilerlemesini yönlendirmek için bağlamdan bağımsız dilbilgisi (CFG'ler) gibi önceki bilgilerden yararlanmaya daha fazla odaklanmıştır. Fuzz testi sürecinde, dilbilgisi genellikle geçerli başlangıç tohumları üretmek veya mutasyonu yönlendirmek için kullanılır, bu Superionda görülebilir. Bununla birlikte, AFLdeki dilbilgisi körü mutasyon stratejilerinin fuzz testlerinin etkinliğini engellediği iddia edilmektedir. Öte yandan, Superion iyi biçimlendirilmiş girdi dosyaları için oldukça uygundur, yani fuzz testi için geçerli girdiler üretebilir ve daha fazla sayıda hata tespit etme olasılığı yüksektir.

Mutasyona dayalı yaklaşımların aksine, dilbilgisi tabanlı yaklaşımlar fuzz testlerinde sözdizimi ayrıştırma aşamasını daha kolay geçerler. Ancak bu yöntemle üretilen test senaryolarının çoğu genellikle anlamsal kontrolden geçemez. Bu sorunu aşmak için PCSG'ye (olasılıksal bağlama duyarlı dilbilgisi) dayalı çalışan son teknoloji yaklaşımlar vardır. Bu yaklaşımlar, dilbilgisi ve anlamsal kurallar bilgisini otomatik olarak çıkarmak için çok sayıda örnek kullanır ve ardından çökmeye neden olan durumları tetiklemesi beklenen test girdilerini üretir. Bu akıllı girdilerin anlamsal ayrıştırma aşamasını kolayca geçebildiği ve uygulama yürütme aşamasına ulaşabildiği için test edilen uygulamada gizli çökmeleri tetikleme şanslarının daha fazla olduğu iddia edilmektedir. Bu nedenle, bir öğrenme algoritması tarafından üretilen akıllı girdilerin, genele açık girdilere kıyasla daha fazla çökmeyi tespit edebildiğini gösterme ihtiyacı artmaktadır [8]. Bu temelden yola çıkarak, bu tez çalışmasındaki bir diğer amaç, iyi derece yapılandırılmış test girdi dosyaları alan programlarda çalışan iki farklı mutasyon tabanlı fuzz testi yürütmesinin her biri için akıllı girdilerin fuzz testi etkinliği açısından etkisini araştırmaktır.

Gri kutu mutasyon tabanlı fuzz testi yaklaşımları için bu tez çalışması vaka çalışması araştırmasının ilkelerine dayanarak yürütülmüştür. Vaka çalışmamızın doğası hem "keşifsel" hem de "iyileştirici"dir; amacımız (fuzz testi yaklaşımını geliştirme sürecinde) ne olduğunu bulmak, yeni içgörüler aramak ve takip iyileştirmeleri (fuzz testi yaklaşımı) ve araştırma ("keşifsel" yön) için fikirler ve hipotezler üretmektir. Dahası, "çalışılan olgunun" test etkinliğini ve verimliliğini iyileştirmek istediğimiz için çalışmanın "iyileştirici" yönleri de bulunmaktadır.

Özetle, bu tez çalışmasının bilime katkıları aşağıda listelenmiştir:

- İyi biçimlendirilmiş XML dosyaları alan SUT'lar için iki farklı mutasyon tabanlı fuzz testi yaklaşımında tespit edilen hatalar, hata tespit performansı ve hata türleri açısından fuzz testi etkinliğinin vaka çalışması, ayrıntılı analizi ve karşılaştırılması.
- PCSG tabanlı bir öğrenme algoritması tarafından üretilen akıllı test girdilerinin etkisini bulmak için iki farklı mutasyon tabanlı fuzz testi yaklaşımına yönelik akıllı test girdilerinin etkisine ilişkin vaka çalışması ve ayrıntılı analizi.



2. LİTERATÜR TARAMASI

Etkili bir güvenlik açığı tespiti ve sistem dayanıklılığı testi yöntemi olan fuzz testi sırasında daha fazla sayıda ve daha fazla türde hata bulmak önemlidir. Fuzz testi esnasında testi yapılan sistem çok fazla sayıda ve farklı stratejiler ile üretilen girdi değerleri ile sınanmaktadır ve sistemin çalışmaz hale geldiği durumlar bulunmaktadır [9]. Özellikle rastgele üretim, mutasyona dayalı veya sistem gramerine göre model bazlı girdi üretim metotları ile sisteme girişte geçerli olacak, yani sisteme giriş kapısının daha ilk aşamasında çeşitli doğrulama metotları neticesinde elenmeyecek, fakat sisteme girdikten sonra sistemi çalışmaz hale getirebilecek hataları bulan bir girdi üretimi, fuzz test süreçlerinin ana amaçlarından biridir. Fuzz testlerinde, test edilen sistemde tespit edilen hata ve zafiyetlerin sayısı olduğu kadar, zafiyetlerin kritiklik ve derinlik derecesi de önemlidir. Hem fazla sayıda hem de sistemin derinliklerinde gizlenen ve bulunması zor olan hataların ortaya çıkarılması bir fuzz testinin etkinliğinin ölçümünde en önemli kriterlerdendir. Bununla birlikte yapılan testler esnasında test edilen sistemin ne kadar kısmının analiz edilebildiği, sistemin iş ve kontrol akışlarının ne oranda teste tabi tutulabildiği de fuzz testlerinin etkinliği açısından önemlidir ve buna da kod kapsamı denilmektedir.

Sızma testleri ve benzeri diğer tekniklerle karşılaştırıldığında, fuzz testi çok az test edilen hedef sistem bilgisi gerektirir ve büyük uygulamalara kolayca ölçeklenebilir. Dolayısı ile endüstride en popüler güvenlik açığı bulma yöntemi haline gelmiştir. Bir diğer yandan fuzz testleri gerçek çalışan uygulamalar üzerinde yapıldığından yüksek doğruluk oranlarına sahiptir. Bununla birlikte fuzz testleri hem akademide hem de endüstride oldukça yaygındır. Fuzz testleri kullanılarak Google, Chrome tarayıcısında sekiz yıllık bir süre zarfında 16.000'den fazla hata ve üç yıl boyunca 160'ın üzerinde açık kaynaklı yazılım projesinde 11.000'den fazla hata tespit etmiştir [6]. Microsoft firması SAGE ismini verdiği fuzz test aracını Windows işletim sisteminin geliştirilmesi sırasında aktif olarak kullanmıştır [10].

2.1. Fuzz Testleri Tarihçesi

1990'lı yıllarda ilk kez akademide kullanılmaya başlayan ve kara kutu olarak adlandırılan fuzz testlerinin tarihsel gelişimi en genel anlamda şu şekilde oldu [2];

- 2005 yılından önce sadece temel rastgele mutasyon tabanlı fuzz testleri vardı. Bir süre sonra ise etkinliğini arttırmak için gramer temelli fuzz testleri uygulanmaya başlandı.
- 2006-2010 yılları arasında beyaz kutu fuzz testleri ve dinamik sembolik çalıştırma kullanılmaya başlandı. Ve paralel olarak leke analizi teknikleri etrafında çalışmalar yapıldı.
- 2011-2015 yılları arasında kapsama rehberli fuzz testleri ve zamanlama algoritmaları kullanılmaya başlandı.
- 2016-2017 yıllarında birden fazla fuzz tekniğinin bir arada kullanılması yaygın bir uygulama haline geldi. Etkinliği ve doğruluğu arttırmak için fuzz testi sürecine entegre edilmiş makine öğrenme yaklaşımları yaygınlaştı.

Fuzz testleri, ilk çıktığı zamanlarda test ettiği sistem hakkında bir bilgiye sahip değildi ve sadece test edilen sisteme verilen rastgele ve sayıca çok girdiler sonrasında sistemin verdiği cevapları inceleyerek sistemin dayanıklılığı ve güvenlik zafiyetlerinin ortaya çıkarmaktaydı. Zaman geçtikçe fuzz testleri beyaz kutu adı verilen ve test edilen sistemin iç yapısı ile ilgili bilgi sahibi olunarak yapılan testler ile başarı oranını ve verimliliği artırmaya yönelik çalışmalar yapıldı. Yakın zamanlarda ise beyaz kutu ile kara kutunun arasında yer alan ve sistem ile ilgili kısmi bilgiye sahip ve bu bilgi ile testlerin verimliliğini artırmaya yönelik yeni bir yöntem olarak gri kutu testler ortaya çıktı. Gri kutu testler ile beraber testler esnasında programın hangi bileşenlerinin test edildiği bilgisi, bir diğer deyişle ne kadar bir uygulama kodu kapsamı olduğu bilgisinden de faydalanılarak testlerin başarısını artırmaya yönelik çalışmalar yapıldı.

2.2. Fuzz Testi Uygulama Alanları

Fuzz testlerinin verimliliğini artırmaya yönelik yeni yöntemlerle birlikte testlerin uygulanmış olduğu alanlar da giderek arttı. Testlerin ortaya çıktığı ilk yıllarda işletim sistemleri ve dosya yapıları üzerinde uygulanan fuzz testleri zamanla ağ protokollerine, derleyicilere, kütüphane ve API uygulamalarına, web ve mobil uygulamalara, IOT platformlarına ve otonom araçlara uygulanmaya başladı. Fuzz testleri sağladığı yüksek başarı oranları ile gelecek vaadederken uygulandığı büyük

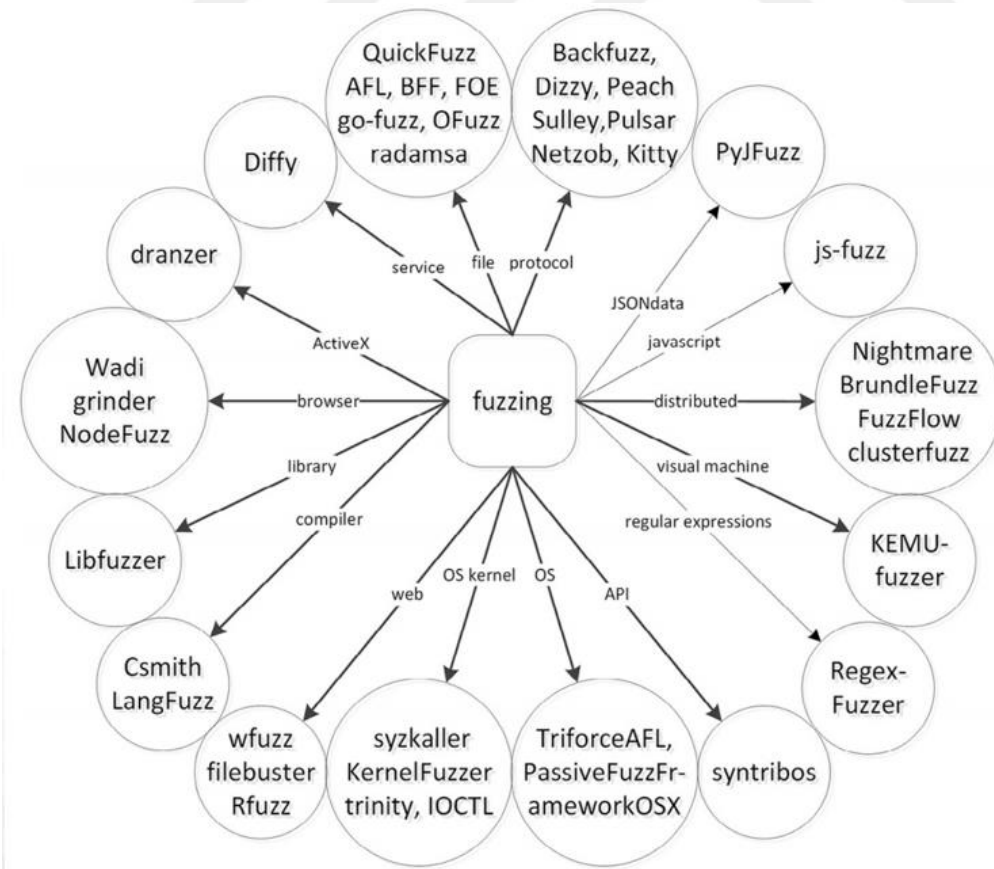
ölçekli uygulamalardaki başarısı ile de öne çıkmaktadır. Bununla birlikte her uygulandığı uygulama alanına özgü yeni zorluklar ve gelişime yönelik konuları da gündeme taşımaktadır.

Fuzz testleri, girdi verisi olarak dosya alan sistemler, işletim sistemi çekirdeği, protokoller, API'ler, web uygulamaları, ActiveX uygulamaları, sanal makineler, web tarayıcıları, JSON verileri, javascript, derleyiciler, dağıtılmış sistemler gibi birçok farklı uygulama alanında yürütülebilir. Komut satırı araçlarının, dosya ayrıştırıcılarının fuzz testlerinin daha da iyileştirilmesi çalışmalarının yanı sıra çevre ile etkileşime giren siber-fiziksel sistemler üzerinde, IOT platformlarında da fuzz testi çalışmaları bulunmaktadır. Ayrıca davranışları eğitim verileriyle belirlenen makine öğrenimi sistemleri üzerinde devam eden fuzz testi çalışmaları da mevcutta aktif olarak devam etmektedir [11]. Bir diğer yandan bir girdi için çeşitli çıktılar üretebilen protokol uygulamalarında durum bilgili yazılımın nasıl test edilebileceği gibi konularda zorluklar ve açık problemler de bulunmaktadır. Olaylar dizisi olarak girdiler alan GUI tabanlı sistemlerin testleri de bir diğer çözüm bekleyen zor problemler arasındadır [6].

Fuzz Testi yapılan başlıca uygulama alanlarından bir tanesi dosya formatı testleridir. Dosya formatı testlerinin önemli bir alt alanı da web tarayıcılarında yapılan testlerdir. Tarayıcılar tarafından işlenen dosya türleri, HTML, CSS, JavaScript dosyaları olarak öne çıkmaktadır. Özellikle, tarayıcıların DOM yapılarının ayrıştırılması ve sayfaların görüntülenmesi şu anda popüler fuzz testi yapılan alanlardır. Web tarayıcılarına yönelik iyi bilinen fuzz testi araçları, Grinder çerçevesi altında COMRaider ve BF3'tür. Diğer yandan ağ sistemlerinde kullanılan protokol testleri de çarpıcı fuzz testi alanlarındandır. Protokol üzerinde yapılan testlerin zor kısmı, servislerin kendi haberleşme protokollerini belirleyebilmesidir. Ayrıca bu tip protokollerin standartlarının belirlenmesi de zordur. Dahası tanımlı dokümanite edilmiş protokoller için bile RFC dokümanı gibi spesifikasyonları takip etmek hala zor bir iştir. Bazı temsili protokol fuzz test araçlarına örnek vermek gerekirse; SPIKE, AutoFuzz, ve SNOOZE bu alanda sık kullanılanlar arasındadır. SPIKE, ağ protokolü stres testlerini hızla oluşturabilir. AutoFuzz, sonlu durum otomati oluşturarak protokol uygulamasını öğrenebilir ve öğrenilen bilgileri test senaryoları oluşturmak için daha fazla kullanabilir. SNOOZE aracı ise durum bilgisine dayalı bir fuzz test yaklaşımı ile protokol kusurlarını tanımlayabilir [1].

Fuzz testlerinde en bilinen araçlardan bir tanesi AFLdir. Bu test aracı test ettiği programın kapsadığı koddan geri bildirim alarak ilerleyen bir araçtır. Test aracı, kod enstrümantasyonu ile yol kapsamı gibi çalışma zamanı bilgilerini toplar. Açık kaynaklı uygulamalar için enstrümantasyon derleme zamanında alınır ve binary dosyalar için enstrümantasyon çalışma zamanında değiştirilmiş bir QEMU aracılığıyla verilir. Yeni yürütme yollarını keşfedebilen test senaryolarının bir sonraki mutasyon turunda seçilme şansı daha yüksektir. AFL aracı; C, C++, Objective C veya yürütülebilir programları destekler ve Linux tabanlı işletim sistemlerinde çalışır. WinAFL ise AFL aracının Windows işletim sistemi üzerinde çalışan bir versiyonudur [12].

SAGE, Microsoft tarafından geliştirilen iyi bilinen bir beyaz kutu test aracıdır. x86 platformunda çalışan Windows uygulamalarını (belge ayrıştırıcıları, medya oynatıcıları, görüntü işlemcileri) alan büyük dosyaları test etmek için kullanılır. SAGE, kod kapsamını en üst düzeye çıkarmak için konkolik yürütmeyi buluşsal bir arama algoritmasıyla birleştirir. SAGE, beyaz kutu test tekniğini kullanan ilk araçtır ve gerçek dünya uygulamalarını test etme yeteneğine sahiptir [12]. Fuzz testi kullanım alanları da aşağıdaki şekilde özetlenmektedir [2].



Şekil 2.1. Fuzz Testi Uygulama Alanları

2.3. Fuzz Testi Sınıflandırması

Fuzz testi teknikleri farklı bakış açıları açısından bazı sınıflara ayrılabilir. Fuzz test teknikleri, hedef sistem hakkında bilgi sahibi olma açısından kara kutu, beyaz kutu, gri kutu olmak üzere üç farklı gruba ayrılabilir. Teknikler, girdi verisi üretme türüne göre ise iki grupta sınıflandırılabilir; bu gruplar mutasyon tabanlı ve nesil tabanlı testlerdir. Ve geri besleme türüne göre, fuzz test teknikleri geri beslemeli ve geri beslemesiz sistemler olarak tanımlanabilir.

2.3.1. Kara kutu testleri

Kara kutu fuzz testleri, test edilen programın dahili iş mantığıyla ilgilenmemektedir. Bu tip kara kutu testlerinde sisteme girdiler verilir ve verilen girdilere karşılık gelen çıktılar toplanarak incelenir. Özellikle mutasyonel kara kutu testlerinde, süreç bir veya daha fazla tohum girdisi adı verilen veri kümeleri ile başlatılır. Tohumlar daha sonra yeni girdiler oluşturmak için değiştirilir yani mutasyona uğrattılır. Girişteki veri üzerinde seçilen rastgele konumlara rastgele mutasyonlar uygulanabilir. Örnek vermek gerekirse giriş verisi bir dosya olan sistemde, dosya girdisindeki mevcut olan rastgele bitler çevrilip yerleri değiştirilerek yeni girdi dosyaları üretilerek sisteme yeniden çalıştırması için verilir. Mutasyon tabanlı kara kutu testlerindeki farklı olarak bir diğer test çeşidi olan nesil tabanlı kara kutu fuzz testlerinde ise girdiler sıfırdan üretilir. Test edilecek sistemin kabul ettiği girdilerin biçimsel olarak yapısal özellikleri yani bir deyişle gramer bilgileri verilirse, girdi dilbilgisi biçimine uygun olacak bir şekilde yeni girdiler oluşturularak fuzz testleri devam ettirilir. Kara kutu testlerde ana fikir, yeni bir fonksiyonel davranışın yürütülmesi veya istisnai bir davranışın tespit edilmesini tetikleyebilecek yüksek değerli girdilerin seçilebilmesidir [13].

Kara kutu fuzz testlerindeki mutasyon kuralları, bit çevirmeleri, bayt kopyaları, bayt kaldırma vb. olabilir. Bu tip testler uygulamada genellikle düşük kod kapsamı sağlar. Son zamanlarda kara kutu fuzz testlerinde, yarı geçerli girdiler oluşturmak için dilbilgisi veya girdiye özgü bilgiyi de kullanılmaya başlanmıştır. Son zamanlarda fuzz testi yapan araçların daha etkili ve daha akıllı hale gelmesi için kod enstrümantasyonu, kusur analizi ve diğer benzer tekniklere odaklanılmıştır.

2.3.2. Beyaz kutu testleri

Beyaz kutu fuzz testleri test edilen programın iç bileşenlerini bilir. Bu tip testlerde, test edilen sistemin kaynak kodu hakkında bilgi, ayrıntılı çalışma zamanı bilgileri ve

programın tasarımı hakkında bazı veriler bilinmektedir. Örnek vermek gerekirse kaynak kod içerisindeki atama işlemleri, matematik işlemleri, döngüler, fonksiyon çağrılarını bilinmektedir ki bu sayede programa verilen girdi verilerinin çalıştırılan program parçaları ile, program içerisinde tanımlanan değişkenlerin farklı durumları ile nasıl bir ilişkisinin olduğu anlaşılmaktadır. Beyaz kutu testleri sembolik yürütme adı verilen bir tekniğe dayanmaktadır. Bu teknik, program yollarını sistematik olarak numaralandırmak için program analizi ve kısıtlama çözümleri kullanmaktadır. Beyaz kutu fuzz testleri sisteme verilen bir girdi verisinin yol koşulunun hesaplamasını yapar yani verilen girdi ile sistem içerisinde kodsalsal olarak nerelere gidileceği, nasıl bir yol izleneceği sembolik olarak bilinebilmektedir [6]. Buradan yola çıkarak sistemde hesaplanan yol koşulları, mutasyon tabanlı yöntemlerle değiştirilerek sisteme daha sonraki test adımlarında ne tip veriler girilmesi açısından yönlendirici olmaktadır. Bununla birlikte beyaz kutu fuzz testlerindeki en büyük sorunlardan bir tanesi yol patlaması(path explosion) problemi olarak adlandırılan sembolik yürütme sorunları ile ilgilidir. Bunun nedeni, hedef programdaki koşullu dalların genellikle çok sayıda olmasıdır, küçük boyutlu bir uygulamada bile çok sayıda yürütme yolu üretilebilir. Dolayısıyla orta ve büyük ölçekli uygulamalarda beyaz kutu testleri çalıştırıldığında sistemde gidilmesi gereken tüm olası yollar zamanla çok fazla artacağı için her birinin çalıştırılması, sistemin performansı açısından neredeyse imkansız hale gelmektedir. Buna bir çözüm olarak gelecekteki çalışmalarda sembolik yürütmeler ile birlikte sistem grameri sunma teknikleri beraber kullanılabilmektedir [10].

Spesifik olarak, somut bir girdi ile yürütmeyi başlatan beyaz kutu testte, yürütme yolu boyunca tüm koşullu ifadelerde sembolik kısıtlamalar toplanır. Bu nedenle, bir yürütmeden sonra beyaz kutu test aracı, bir yol kısıtlaması oluşturmak için AND mantığını kullanarak tüm sembolik kısıtlamaları birleştirir. Daha sonra, beyaz kutu test aracı kısıtlamalardan birini sistematik olarak reddeder ve yeni yol kısıtlamasını çözer. Yeni test durumu daha sonra programı farklı bir yürütme yolu çalıştırmaya yönlendirir [12].

Beyaz kutu testlerde kullanılan sembolik yürütme ile beraber son zamanlarda en sık kullanılan yöntemlerden bir tanesi de dinamik sembolik yürütmedir. Bu tip bir metotta sembolik ve somut yürütme aynı anda çalıştırılır. Bu anlamda dinamik sembolik yürütme konkolik test olarak da adlandırılır [14].

2.3.3. Gri kutu testleri

Gri kutu fuzz testleri, test edilen program hakkında kısmi bilgiler içeren yazılım hatalarını etkili bir şekilde bulabilmek için kara kutu ve beyaz kutu testlerinin ortasında bir yerdedir. Gri kutu fuzz testleri yalnızca derlenmiş bayt kodu içindeki koşullu atlamalar gibi kontrol akışı talimatlarını analiz eder, ancak bir uygulamanın dahili mantığını gerçekten analiz etmez. Gri kutu testlerde yaygın olarak kullanılan yöntem kod enstrümantasyonudur. Bu tür enstrümantasyonla, gri kutu test aracı, çalışma zamanında hedef programın kod kapsamını elde edebilir. Bu bilgiyi, örneğin genetik algoritmalar yardımıyla mutasyon stratejilerini güncellemek için de kullanır. Ve diğer yandan daha fazla yürütmeyi kapsayabilecek test senaryoları oluşturmak için kullanır. Gri kutu fuzz testleri yardımı ile Google, Chrome tarayıcısında sekiz yıllık bir süre zarfında 16.000'den fazla hata ve üç yıl boyunca 160'ın üzerinde açık kaynaklı yazılım projesinde 11.000'den fazla hata tespit etmiştir [6].

Gri kutu ve beyaz kutu fuzz testleri, her iki yöntemin de test senaryosu oluşturmaya rehberlik etmek için hedef program hakkındaki bilgileri kullanması nedeniyle birbirine çok benzemektedir. Gri kutu fuzz testi, programda hangi yolların keşfedildiğini bilmek için hedef programın bazı çalışma zamanı bilgilerini (kod kapsamı, kusur veri akışı vb.) kullanır [14]. Ancak bu bilgilerin kullanılmasının yeni yolları kapsayacak veya yeni hataları bulmak için kesinlikle daha iyi test senaryoları oluşturacağını garanti edemez. Öte yandan beyaz kutu testleri, tüm yürütme yollarını sistematik olarak keşfetmek için hedef programın kaynak kodunu veya binary kodlarını kullanır. Bu şekilde, beyaz kutu test araçları, oluşturulan test senaryolarının hedef programın yeni yürütme yollarını keşfetmesine yardımcı olacağından emin olabilir [12].

Gri kutu testlerde kullanılan diğer bir yöntem de kusur analizidir. Bu tip analiz tekniği, kusurlu veri akışını izleyerek kod enstrümantasyonunu genişletir. Bu nedenle, bir test aracı, hedef programdaki potansiyel saldırı noktalarını etkileyebilecek belirli girdi alanlarını mutasyona uğratmaya odaklanma yeteneği kazanacaktır. Çünkü kusur analizi, giriş verilerinin hangi bölümünün önceden tanımlanmış saldırı noktalarını etkileyebileceği konusunda hesaplama yapar. Somut örnek vermek gerekirse, bu tür saldırı noktaları belirli kütüphane çağrılarını veya güvenlik açığı kalıpları olarak tanımlanabilir [12].

2.4. Girdi Üretim Stratejisine Göre Fuzz Testleri

Fuzz testi yapan araçlar sistemi test etmeden önce ürettikleri girdinin üretim stratejisine göre ikiye ayrılmaktadır; mutasyon temelli ve nesil tabanlı testler. İki yöntem arasındaki en temel fark mutasyon temelli testlerde ilk tohum girdileri üretildikten sonra bunlar üzerinde mutasyon işlemleri yapılarak sonraki girdilerin üretilmesi, nesil tabanlı testlerde test edilecek sistemin kabul edeceği girdi formatı hakkında bir ön bilgi sahibi olunup girdilerin ilk baştan bu formata göre sıfırdan üretilmesidir [2].

2.4.1. Mutasyon tabanlı girdi üretimi

Mutasyona dayalı girdi üretimiyle fuzz testleri başlatılması daha kolay ve daha uygulanabilir ve son teknoloji fuzz test araçları tarafından yaygın olarak kullanılır. Mutasyona dayalı fuzz test araçları için geçerli bir başlangıç girdisi kümesi gerekir. Test durumları, başlangıç girdilerinin mutasyonu ve fuzz testleri işlemi sırasında oluşturulan test durumları aracılığıyla oluşturulur. Ancak, daha fazla program yolunu kapsayabilen ve hataları tetiklemesi daha kolay olan test durumlarının nasıl değiştirilip üretileceği önemli bir zorluktur. Özellikle, mutasyona dayalı fuzz test araçlarının mutasyon ne zaman yapılır sorusuna cevap vermesi gerekir: nerede mutasyona uğranır ve nasıl mutasyona uğranır. Sadece birkaç anahtar konumdaki mutasyon, yürütmenin kontrol akışını etkiler. Bu nedenle, bu anahtar konumların test durumlarında nasıl konumlandırılacağı büyük önem taşımaktadır.

Mutasyona dayalı girdi üretimi, programda geçerli işlevselliği tetikleyen bir dizi tohum girdisi gerektirir. Ve ardından bu tohumları değiştirmek için rastgele mutasyon kullanır. Bir dizi geçerli girdi sağlamak, bir girdi biçimini resmi olarak belirtmekten daha kolaydır. Fuzz test araçları esas olarak test ettikleri programda yeni bir durumu keşfetmek için bir mutasyon stratejisine dayalı girdiler üretir. Test aracı program yapısının farkında olduğu için girdi üretimini programa göre uyarlayabilir. Mutasyona dayalı girdi üretiminde girdi üretimi için ana aşılması gereken zorluklardan bir tanesi; üretilen girdilerin program içerisinde yeni akışları bulabilmesi veya aynı akışları farklı girdilerle (veri akışı) yürütebilmesi arasındaki dengeyi yakalayabilmektir [3].

Örnek girdi üretimi açısından, girdi tohumlarını seçme, mutasyona uğratma ve yeni örnekler üretmek için çeşitli yöntemler kullanılmaktadır. En genel anlamda bu tip testlerde rastgele mutasyon yöntemi ile hazırlanan bir tohumdan yeni girdiler üretilir

ve bu girdiler sistemin çalışmaz hale getirilip getirilmediğini görmek için sisteme verilir. Rastgele mutasyonu kullanan fuzz testleri, uygulama güvenlik testlerinde yoğun olarak kullanılmaktadır. Uygulaması kolaydır ve esas olarak çok fazla hesaplama maliyeti gerektirmez. Ancak test ettiği program hakkında yeterli bilgiye sahip olmadığı için programın derinliğini ortaya çıkaramaz [2].

Mutasyon tabanlı test girdisi oluşturmada cevaplanması gereken en kritik sorun, giriş örneklerinin nerede mutasyona uğradığını ve nasıl mutasyona uğrayacağını belirlemektir. Sadece birkaç kritik konumdaki mutasyonlar, yürütmenin kontrol akışını etkiler. Bu nedenle, bu kritik konumları test durumlarında konumlandırmak özellikle önemlidir. Giriş örneklerinin nasıl mutasyona uğradığı önemli bir konudur. Mutasyon konumlarının seçimi için, mevcut evrimsel algoritmalar genellikle sonuçsuz rastgele mutasyon dizilerinde problemlerle karşılaşır.

Mutasyon tabanlı test girdisi oluşturmada cevaplanması gereken bir diğer sorun da mutasyon operatörünün nasıl seçileceğidir. Fuzz testlerindeki mutasyon operatörleri kavramı genetik algoritmadan türetilmiştir. Fuzz testindeki mutasyon operatörü, bit çevirmeleri, bayt çevirmeleri, aritmetik mutasyonlar, sözlük, baytları silme, baytları klonlama ve baytların üzerine yazma gibi işlemleri içerir. Test senaryolarının kör mutasyonu, test kaynaklarının ciddi şekilde israf edilmesine neden olur ve daha iyi mutasyon stratejisi, fuzz testinin verimliliğini önemli ölçüde artırabilir. Mutasyon operatörlerini seçmek, takviyeli öğrenmenin ele alabileceği soruna benzediğinden, mutasyon operatörü seçimi durumunda, mevcut araştırmalar daha iyi mutasyon operatörlerini seçmek için takviyeli öğrenmeyi kullanmaya odaklanır.

Mutasyona dayalı test senaryosu üretim yaklaşımında girdideki değişiklik genel itibarıyla bayt seviyesinde gerçekleştirilir. Bu yöntemler, mutasyona dayalı olarak yeni girdiler oluşturmak için girdi içerisindeki bazı bitleri rastgele çevirir, siler veya kopyalar. Fakat rastgele bit çevirme işlemlerinin karmaşık dosya biçimlerini işleyen uygulamalar için geçerli dosyalar üretmesi pek mümkün değildir. Bu sebeptendir ki, mutasyona dayalı fuzz testleri, iyi yapılandırılmış girdileri (ör. XML, JavaScript) işleyen programlar için pek uygun değildir. Bu yüzden mutasyona dayalı fuzz testleriyle üretilen kötü biçimli girdiler, sözdizimi ayrıştırmada başarısız olur ve bu nedenle işlemenin erken bir aşamasında reddedilir.

AFL en yaygın kullanılan mutasyon tabanlı fuzz testi aracıdır ve CGF prensiplerine dayalı olarak çalışır. Superion aracı ise dilbilgisi odaklı mutasyona dayalı bir fuzz testi aracıdır ve hem sözdizimsel doğruluk hem de fuzz testi verimliliği elde edebilir. AFLdeki mutasyonlar dilbilgisi körüdür ve bayt seviyesinde çalışır. AFLin dilbilgisi körü mutasyon stratejileri, sözdizimsel ve anlamsal olarak geçerli test girdileri üretme yeteneğinden yoksundur. İyi biçimlendirilmiş girdileri işlemekten uzaktırlar.

2.4.2. Gramer tabanlı girdi üretimi

Rastgele mutasyon tabanlı yaklaşım, test ettiği program hakkında çok az bir bilgiye sahip olduğu için, test senaryolarının çoğu, program yürütme esnasında; ayrıştırma adımı veya girdilerin bütünlük kontrolünün yapıldığı doğrulama adımı reddedilir. Çünkü üretilen girdiler sistemin kabul ettiği girdinin formatı veya grameri hakkında bilgiye sahip değildir. Bu sorundan dolayı, dilbilgisine dayalı fuzz yaklaşımı geliştirilmiştir. Nesil tabanlı girdi üretimi olarak da bilinen bu yöntem, dilbilgisi tabanlı veya model tabanlı teknikler kullanarak girdi veri yapısını tanımlar. Girdi veri yapısı, programı dinamik olarak çalıştırarak veya dilbilgisi dosyalarını veya kodlarını manuel olarak yazarak elde edilebilir. Bazı fuzz test teknikleri, bu yöntemi klasik rastgele tabanlı mutasyon yaklaşımıyla birleştirerek yeni yaklaşımlar getirmiştir. Dilbilgisi gösterimi yaklaşımı, çok karmaşık girdi biçimi kabul eden sistemlerde en etkili test tekniğidir. Ancak hedef sistem hakkında yüksek bilgi gerektirir ve ayrıca belirtilen format manuel olarak yazılır. Bu tip bir testin etkinliğini artırmak ve tohum seçimi ve mutasyon stratejisini optimize etmek için bir takım çizelgeleme algoritmaları kullanılmaktadır. Örnek vermek gerekirse, simüle edilmiş tavlama algoritmaları, Markov algoritmaları ve benzer şekilde çalışan istatistiksel algoritmalar [2].

Nesil tabanlı test araçları, test edecekleri programın grameri hakkında daha fazla bilgiye sahip oldukları için programların doğrulamasını daha kolay geçebilir ve hedef programların daha derin kodunu test etme olasılığı daha yüksektir. Ancak iyi yapılandırılmış grameri anlatan bir belge olmadan dosya biçimini analiz etmek zor bir iştir. Bu nedenle mutasyon tabanlı test araçlarının başlatılması daha kolay ve daha uygundur ve son teknoloji test araçları tarafından yaygın olarak kullanılmaktadır. Bununla birlikte, daha fazla program yolunu kapsayabilen ve hataları ortaya çıkarmayı kolaylaştıran test senaryolarının nasıl mutasyona uğratılacağı ve oluşturulacağı önemli bir zorluktur[5]. Spesifik olarak, mutasyona dayalı fuzz test araçları, mutasyon

yaparken iki soruyu yanıtlamak zorundadır: (1) girdinin neresinde mutasyona yapmalı ve (2) nasıl bir mutasyon işlemi yapmalı [1].

Dilbilgisi tabanlı veya diğer adıyla nesil tabanlı fuzz testleri, model tabanlı testlerle de ilgilidir. Burada bahsi geçen model; test edilecek programın soyut olarak hazırlanmış bir temsildir. Model tabanlı test, programın modele göre uygunluğunu kontrol etmek için modeli analiz ederek testler üretmeyi içerir. Model tabanlı testlerde kullanılan test oluşturma algoritmaları, mümkün olduğunca küçük test takımları oluşturmak için genellikle sonlu durumlu bir makine modelinin her durumunu ve geçişini kapsayan minimum sayıda test oluşturmaya çalışır. Öte yandan, fuzz testi amacıyla girdi örneklerinden girdi dilbilgilerinin otomatik olarak nasıl öğrenileceği, yeni bir araştırma konusudur. Olasılıksal girdi dilbilgilerini öğrenmek için sinir ağlarına dayalı istatistiksel makine öğrenme teknikleri kullanılabilir. Dilbilgisi tabanlı fuzz testleri için makine öğreniminin kullanımı henüz başlangıç aşamasındadır ve henüz yaygın olarak kullanılmamaktadır [10].

Gramer tabanlı girdi üretimine örnek olarak verilebilecek bir araç olan Skyfire, girdi olarak örnek bir veri kümesi ve bir dilbilgisi alır. Bunu kullanarak yüksek düzeyde yapılandırılmış girdileri işleyen fuzz programları için iyi dağıtılmış tohum girdileri oluşturmak için çok sayıda mevcut örnekteki bilgiyi kullanır. Ayrıca, veri kaynağı olarak internet kullanılarak tohum dosyalarının toplanması da yaygındır. Belirli karakterlere (dosya uzantısı, sihirli baytlar, vb.) dayalı olarak, test mühendisleri gerekli tohum girdi dosyaları da indirebilir. Üretilen tohum giriş verilerinin doğrulaması yapılırken, verilerin bütünlüğünü doğrulamak için çeşitli sağlama mekanizmaları ve bazı veri formatı kontrol yöntemleri kullanılır [12].

2.5. Benzer Çalışmalar

Bu bölümde; dilbilgisi tabanlı fuzz testleri, öğrenme tabanlı fuzz testleri ve iyi yapılandırılmış girdi tabanlı test süreci için fuzz testi alanlarındaki ilgili çalışmalara dair kısa bir genel bakış sunuyoruz.

2.5.1. Gramer tabanlı fuzz testleri

Gramer tabanlı fuzz testleri bir spesifikasyondan veya bir dilbilgisinden test girdileri üretir. Bu nedenle, üretilen girdiler test edilen sistem tarafından beklenen standartlara uygundur. Dilbilgisi tabanlı fuzz testleri, HTML belgeleri ve JavaScript kodu dahil

olmak üzere web sayfaları biçiminde girdi alması gereken web tarayıcıları gibi iyi yapılandırılmış girdi biçimlerine sahip fuzz testi uygulamaları için bugün bilinen en etkili test tekniğidir [15].

Peach [16] ve Spike [17] isimleriyle bilinen çeşitli dilbilgisi tabanlı fuzz test araçları vardır. Bunlar, girdi modellerini belirtim olarak kullanır ve mutasyonu test girdileri üretmek için kullanır. Giriş modeli, girdilerin fuzz test sürecinin bütünlük kontrol aşamasını (sözdizimi ayrıştırma) kolayca geçebilmesi için verilerin biçimini ve bütünlük kısıtlamalarını bildirir. Daha spesifik olarak, Peach başlangıçta dosya fuzz testlerinde kullanılmıştır. Sürekli geliştirmelerden sonra, dosya yapıları, COM nesneleri, ağ protokolleri ve API arayüzleri üzerinde test yapmak için kullanıldı. Peach, bir XML dosyası tanımlayarak fuzz test sürecini yönlendirir, çerçevesi esnektir ve yüksek derecede kod yeniden kullanımına izin verir. Peach'te kullanılan test süreci rastgelelik ve otomatik olarak oluşturulan test durumlarını kullanır. Peach'in fuzz testlerindeki otomatik veri modeli, giriş dosya biçiminin analizi ve veri yapısının optimize edilmesiyle geliştirilir. Akıllı fuzz testleri için verimli veri modelleme çerçevesi önerilmiş ve yazılım güvenlik açıklarını tespit etmek için çerçeve uygulanmıştır. Peach veri modeli bir XML dosya yapısından oluşur ve veri modeli, durum modeli, aracı ve test hakkında bilgi içerir. Temel olarak, ayrıştırma XML üzerinde gerçekleştirilir ve her veri modeli için tanıma işlevleri dahil edilir. Ayrıca, bir giriş dosyası kullanılarak otomatik olarak bir veri modeli oluşturulurken, Peach tabanlı bir XML veri modeli oluşturulur. Spike, protokolleri özelliklerini XML dosyalarına ve şablonlarına dönüştürerek test etmek için kullanılan açık kaynaklı otomatik fuzz testi çerçevesidir. Bir fuzz testi aracı yazmak için gereken tüm işlevleri içeren C dilinde yazılmış bir API'dir. Arabellek taşması, tamsayı taşması ve bellek ayırma hataları ve benzeri yazılım hatalarını bulmak için akıllı bir giriş kümesi sağlayan fuzz testi dizeleri içeren büyük bir veritabanına sahiptir. Spike, blok tabanlıdır ve basit betik oluşturma yeteneğine sahiptir ve SPIKE ilkel öğelerini içeren temel metin dosyalarının yorumlanmasına olanak tanır. Blok tabanlı olduğundan, çeşitli protokol uygulamalarındaki güvenlik açıklarını kolayca bulabilir. Spike'ın en büyük avantajlarından biri, hemen hemen her protokole uygulanabildiği için çok genel olmasıdır.

CSmith [18], LangFuzz [19], IFuzzer [20], Radamsa [21], girdileri üretmek için spesifikasyon olarak bağlamdan bağımsız dilbilgisini kullanır. CSmith, hata bulma

yeteneğini yok edebilecek tanımsız davranışlardan kaçınırken C'nin büyük bir alt kümesini kapsayan C programları oluşturan rastgele bir test oluşturma aracıdır. C programlarını üretmek için dilbilgisinden izin verilen üretim kuralını rastgele seçer. CSmith özellikle bir C programı üretir; bir test çalışması ve ardından programı birkaç derleyici kullanarak derler, yürütülebilir dosyaları çalıştırır ve çıktıları karşılaştırır. CSmith'in test oluşturma teknikleri, işaretçiler, diziler ve yapılar başta olmak üzere karmaşık kontrol akışı ve veri yapıları gibi C dili özelliklerini kullanan karmaşık kod içerir. Belirtilen bu C kodu oluşturma yolu, CSmith, GCC, LLVM ve ticari araçlar ve benzeri C derleyicilerindeki hataları keşfetmek için kullanılır. LangFuzz, dilbilgisini kullanarak programları, yani kısmen geçersiz davranışa neden olduğu bilinen programlardan kaynaklanan kod parçalarını rastgele oluşturur. Mozilla JavaScript yorumlayıcısında güvenlik testi için uygulanmıştır ve 3 aylık bir süre zarfında içinde 105 yeni ciddi güvenlik açığı ortaya çıkarmıştır. Langfuzz bağlamdan bağımsız dilbilgisi yoluyla kara kutu fuzz testi gerçekleştirir. Dilbilgisini girdi olarak alır: bir Javascript dilbilgisi verildiğinde, JavaScript programları oluşturur. Langfuzz dilbilgisini kullanarak belirli bir kod tabanından kod parçalarını öğrenebilir. Daha önce başarısız olan bir dizi program verildiğinde Langfuzz test takımının parçalarını kullanır ve yeni programlar oluşturmak için yeniden birleştirir. Langfuzz'un JavaScript motorlarında tespit edilemeyen hataları ne kadar iyi bulabildiğini bulmak için bir değerlendirme yapılmıştır. JavaScript motorlarındaki hataları keşfetmek için de kullanılan ve Mozilla ile Google'ın kalite güvencesinin aktif bir parçası olan ve bunların geliştirilmesinde kullanılan jsfunfuzz ile bir karşılaştırma yapılmıştır. Bu iki fuzz test aracı arasında kullanılan karşılaştırma metriği, tespit edilen hata sayısıdır. IFuzzer, JavaScript yorumlayıcılarındaki hataları bulmak için otomatik bir evrimsel fuzz testi yöntemidir. Genetik Programlama kullanarak kod parçaları üretir ve kullanıcıların yorumlayıcıları bir kara kutu fuzz testi tekniği kullanarak test etmelerine olanak tanır ve temel olarak bellek bozulmaları türünde güvenlik açıklarını arar. IFuzzer, verilen test örneklerinden kod parçalarını çıkarmak için bir dilin bağlamdan bağımsız gramerini kullanır. IFuzzer, hedef dil için ANTLR ayrıştırıcısını kullanır ve ayrıştırıcıyı, dil gramerini girdi olarak kullanarak ve ANTLR ayrıştırıcı oluşturucu çerçevesini kullanarak oluşturur. Bundan sonra yeni test örnekleri oluşturmak için kod parçalarını biyolojik evrimsel bir şekilde yeniden birleştirir. Örneğin, IFuzzer, SpiderMonkey yorumlayıcısının JavaScript gramerini ve test takımını girdi olarak alabilir ve ayrıntılı test için ayrıştırma ağaçları ve uygun kod parçaları oluşturabilir.

Ifuzzer, istismar edilebilir hatalar bulma ve sonuçları LangFuzz'in sonuçlarıyla karşılaştırma amacıyla JavaScript yorumlayıcılarına karşı çalıştırılarak değerlendirilmiştir. Bu test araçları hataları bulmada geçen zaman ve her iki fuzz test aracının bulduğu hatalardaki örtüşmenin derecesi açısından karşılaştırılmıştır. Deney sırasında IFuzzer bir aylık zaman diliminde 40 hata bulurken, Langfuzz üç ayda 51 hata bulmuştur. Çalıştırma sırasında, JavaScript motoru bir erişim ihlali nedeniyle çökmüş ve yığın taşmasına neden olmuştur. Radamsa geçerli girdileri alarak ve bunları değiştirerek çalışan bir kara kutu fuzz test aracıdır [22]. Radamsa verilen eğitim örneklerinin yapısını tanımlayan bir CFG'yi otomatik olarak oluşturur ve CFG'yi sağlamlık testi için benzer veriler üretmek için kullanır. Rastgele değişen bitlere güvenmez, bunun yerine örnek dosyalardaki yapıları tanır ve bunlara farklı fuzz testi teknikleri uygular. Araç ayrıca bir TCP istemcisi veya sunucusu olarak da işlev görebilir, bu nedenle mutasyona uğramış örnekler bir hizmete veya TCP bağlantılarını dinleyen bir istemciye beslemek için kullanılabilir [23]. Radamsa fuzz testi aracı, SampleSpace ve Glade araçlarına karşı kapsam ve geçiş oranı metrikleri açısından değerlendirilmiştir.

Gramer tabanlı fuzz testlerine örnek çalışmalar olarak örnek girdiden algılanan dilbilgisini kullanan etkili bir fuzz testi aracı olan JIMA-Fuzzing adlı bir araç oluşturulmuştur. Bu araç algılanan dilbilgisine dayanarak, JIMA-Fuzzing geçerli kullanıcı girdisinin bir bölümünü seçer ve bu bölümü fuzz testine tabi tutar. Örneğin, araç girdinin boyutunu büyük ölçüde artırabilir, girdiyi kesebilir, sayısal değerleri yeni değerlerle değiştirebilir, kelimeleri sayılarla değiştirebilir. JIMA-Fuzzing, örnek girdi dosyalarından dilbilgisi çıkarabilme yeteneğine sahiptir; bu da etkili fuzz testi dosyaları oluşturmak için kaynak koduna erişim gerektirmediği anlamına gelmektedir [24].

Gramer tabanlı test senaryosu kırpma metodolojisini kullanan çalışmada tekrarlayan döngü yapılarıyla ilgili girdi alanlarını kaldırmak için bir test senaryosu kırpma yaklaşımı tasarlanmıştır. Girdinin dilbilgisi yapısını çıkarmak ve hiyerarşik bir dilbilgisi ağacı oluşturmak için hafif bir leke analizi tekniği önerilmiştir. Daha sonra, delta hata ayıklama yoluyla, kod kapsamını etkilemeyen düğümler belirlenmiş ve karşılık gelen girdi alanları kaldırılmıştır. Bir prototip olarak ScissorFuzz uygulanmış ve deneysel sonuçlar, yaklaşımın kod kapsamında ortalama %8,2 iyileştirme

sağladığını ve en güncel hibrit fuzz test tekniği QSYM ile karşılaştırıldığında 115 daha fazla hataya neden olduğunu göstermiştir [25].

Veri entegrasyonu ayrıştırıcılarının dilbilgisi tabanlı fuzz testleri için, CMS alanındaki dört yaygın yazılım paketi için bir giriş dilbilgisi oluşturulmuştur ve Novel Materials Discovery (NOMAD) ayrıştırıcılarındaki hataları tespit etmek için dilbilgisi tabanlı fuzz testi yetenekleri üzerinde deneysel bir değerlendirme gerçekleştirilmiştir. Bu yaklaşımla, araştırılan NOMAD ayrıştırıcılarında hizmet kullanılabilirliğiyle ilgili üç benzersiz kritik hatanın yanı sıra birkaç ek sözdizimsel, anlamsal, mantıksal ve alt akış hatasını tespit edilebilmiştir [26].

Dilbilgisi tabanlı fuzz testlerinde tohum dosyalarının şekillendirilmesini ele alan bir çalışmada dilbilgisi tabanlı fuzz testlerinin sınırlamaları vurgulayarak en son yöntemler gözden geçirilmiş ve dağılım örneklemesini fuzz testlerine dahil etmek için ilk yaklaşım sağlanmıştır. Ayrıca bu çalışmaya göre dilbilgisi tabanlı fuzz testlerinde kapsamlı girdi alanı keşfine ulaşmak için önemli bir adım olabilir ve fuzz hedeflerinin sağlamlığını ve güvenilirliğini artırmaya yönelik çıkarımlar içerir [27].

Yukarıda belirtilen dilbilgisi tabanlı fuzz testi yöntemleri genellikle sözdizimi ayrıştırmasını geçebilir, ancak test girdilerinin anlamsal geçerliliğini kontrol eden anlamsal denetimi geçemez. Sonuç olarak, bu yaklaşımlardan üretilen girdilerin yalnızca küçük bir kısmı, derin hataların saklandığı uygulama yürütme aşamasına ulaşabilir. Bu tez çalışmasında ise kullanılan öğrenme tabanlı metodolojideki yenilik, PCSG'deki bağlamlar olarak doğrudan anlamsal kuralları hesaba katmasıdır, üretilen girdiler çoğunlukla fuzz testinin anlamsal denetim aşamasını geçebilir.

2.5.2. Öğrenme tabanlı fuzz testleri

Öğrenme tabanlı fuzz testleri iki ana yaklaşımdan oluşur; bunlardan biri dilbilgisi öğrenmesi, diğeri ise makine öğrenimi tabanlı yaklaşımdır. Dilbilgisi öğrenme tabanlı fuzz testi yaklaşımlarında, test edilen programın dilbilgisi, programın sözdizimsel ve anlamsal kurallarını bilmeyi kolaylaştıran bazı öğrenme algoritmaları aracılığıyla öğrenilir. Makine öğrenimi algoritmaları tarafından desteklenen fuzz testi yöntemlerinde, test sürecini iyileştirmek ve girdi tohum dosyalarını değiştirmeyi kolaylaştırma, kod kapsamını artırma ve biçim doğrulamasını etkili bir şekilde atlama açısından sonuçlar elde etmek amaçlanır. Fuzz testi sürecinin birçok aşaması bir sınıflandırma sorunu olarak ele alınabilir, böylece makine öğrenimi algoritmaları fuzz

testi sürecinde kolayca kullanılabilir. Örneğin, tohum dosyalarının ve test durumlarının geçerli olup olmadığına karar vermek, bir çökmenin istismar edilebilirliği, hangi mutasyon operatörünün kullanılacağı, bunların hepsini bir makine öğrenimi sorunu olarak ele almak için uygundur. Öğrenme tabanlı algoritmaların kullandığı test adımları esas olarak aşağıdaki gibidir;

- Tohum Dosyası Oluşturma
- Test Durumu Oluşturma
- Test Durumu Filtreleme
- Mutasyon Operatörü Seçimi
- Kullanılabilirlik Analizi

Learn&Fuzz [15], seq2seq modeli ile PDF formatının dilbilgisi kurallarını öğrenmek için yürütülen ilk araştırma çalışmasıdır. Godefroid ve arkadaşları tarafından gerçekleştirilen bu araştırma [15], dilbilgisi tabanlı fuzz testi araçlarında test senaryosu oluşturmayı iyileştirmek için bir öneri sunmaktadır. Karmaşık dosya formatları hem metinsel hem de metinsel olmayan içerikler içerirken, Learn&Fuzz yaklaşımı öğrenme sürecinde yalnızca metinsel parçaları kullanır. Bu amaçla yazar gözetimsiz öğrenmeyi kullanan iki yinelemeli sinir ağı (RNN) içeren bir diziden diziye (seq2seq) yöntemini kullanır. Buradaki fuzz testi yaklaşımı tüm PDF dosyalarından girdi gövdesi bölümünü alır ve sonucu aynı boyuttaki bloklara böler. Sonuç olarak model, PDF standartlarına uygun test senaryoları oluşturur [28]. Learn&Fuzz yöntemi, istatistiksel bir girdi modelini öğrenmek için char-nn modelinden yararlanır ve öğrenilen modelin olasılık dağılımına göre yeni girdiler oluşturur. Ardından bu PDF dosyalarını oluşturan bazı rastgele fuzz testi algoritmaları uygular. Son olarak, RNN tarafından oluşturulan girdilerin üstündeki rastgele fuzz testinin etkilerini ölçer. Bu yaklaşım esas olarak kapsam, geçme/kalma oranı ve hata sayısı olan üç ana fuzz testi ölçümünü inceler. Özellikle bellek bozulması hatalarını yakalamak için bir izleme aracı olan AppVerifier uygulamasını kullanarak yığın taşması hatalarını tespit edebilir.

Test senaryosu ve tohum üretimi ile ilgili diğer fuzz testi yaklaşımlarından biri, iyi derece yapılandırılmış girdi formatı gerektirmeden ikili(binary) tohum dosyalarını öğrenmek ve üretmek için makine öğrenimi modeline dayalı olarak tasarlanmış SmartSeed'dir [29]. SmartSeed'in temel fikri, verimli bir tohum üretimi ve seçim stratejisi için üretken bir model oluşturmaktır. SmartSeed'in üretken modeli önce bazı

düzenli ikili dosyalar alır ve bunları AFL fuzz testi aracına verir. Daha sonra AFLde çökmelere neden olan tohum dosyalarını alır ve bunları ilk eğitim veri kümesi olarak alır. Bundan sonra, bu yaklaşım bu seçili ham verileri genel matrislere kodlar ve bu matrisleri eğitim verisi olarak alır. Bu eğitim verileriyle, Wasserstein üretken çelişkili ağ (W-GAN) tabanlı bir tohum üretken modeli oluşturur, böylece ilk matrislerde bazı değişiklikler yaparak yeni matrisler üretir ve ardından bunları tekrar uygun girdi dosyalarına dönüştürür. Oluşturulan ikili dosyalar AFL, honggfuzz [30], Vuzzer [31] gibi bazı seçilmiş iyi bilinen fuzz testi araçlarına verilir. Burada test edilen sistem olarak seçilen uygulamalar, mp3, bmp ve flv biçimindeki ikili dosyaları kabul eden toplam 12 uygulamadır. Son olarak farklı seçilmiş tohum seçimi stratejileri ve fuzz testi araçları kombinasyonları arasında benzersiz hata sayısı, kapsam ve güvenlik açıklarının sayısı karşılaştırılmıştır. Bulunan güvenlik açıkları perspektifinden, çoğunlukla arabellek taşması, yığın taşması, bellek sızıntısı gibi bellek bozulması hataları olan toplam 23 güvenlik açığı, 12 uygulamadan dokuzunda tespit edilmiştir. Ye ve diğerleri, sentetik test durumları üreten ve gerçek dünya kütüphanelerini ve GNU, Libxml2 ve Libjson gibi uygulamaları test etmek için veri yapısı özelliklerini yakalamayı geliştiren GAN tabanlı bir fuzz test aracı olan RapidFuzz'u [32] tanıtmıştır. GAN, mutasyon sürecini zenginleştirerek benzer ancak farklı sayısal dağılımlar üretir. Yapısal özelliklerin ve test senaryolarının daha hızlı tanımlanmasını kolaylaştırmada başarılı olur ve AFLin hız, kapsam ve harita boyutu performansını artırır [33]. Makine öğrenimi fuzz test araçları, daha sonra testlerin temeli olan giriş formatının bir modelini çıkarmak için kullanılan bir dizi örneğe ihtiyaç duydukları için dilbilgisi öğrenen fuzz araçlarına benzerdir. Temel fark, dilbilgisi öğrenen fuzz test araçları için modelin bir dilbilgisi olması, makine öğrenimi yaklaşımı için ise modelin bir ML modeli olmasıdır [34].

Derin öğrenme kütüphaneleri için tarih odaklı fuzz testleri çalışmasında Orion adlı yeni bir fuzz test metodu önerilmektedir. Bu yaklaşım, rehberli test girdisi oluşturmayı ve derin öğrenme API'lerinin temel uygulamasında kritik sorunları tetiklediği bilinen ve geçmiş verilerden oluşturulmuş bir fuzz testi sezgisel kuralı kümesine dayalı köşe durum test girdisi oluşturmayı bir araya getirmektedir. Fuzz testi sezgisel kurallarını çıkarmak için, öncelikle en popüler derin öğrenme kütüphanelerinden ikisi olan PyTorch ve TensorFlow'daki 376 güvenlik açığının kök neden analizine ilişkin bir

ampirik çalışma yürütülmüştür. Daha sonra, çıkarılan geçmiş güvenlik açıklarının kök nedenlerine dayalı fuzz testi sezgisel kuralları oluşturulmuştur [35].

Durumsal SDN denetleyicilerini test etmek için öğrenme rehberli fuzz testi çalışmasında durumlu SDN denetleyicilerini test etmek için öğrenme rehberliğinde fuzz testi yöntemi olan SeqFuzzSDN önerilmiştir. SeqFuzzSDN, (1) test edilen SDN denetleyicisinin durum alanını etkili bir şekilde keşfetmeyi, (2) hataları ortaya çıkarmak için etkili ve çeşitli testler üretmeyi ve (3) hatalara yol açan mesaj dizilerini karakterize eden doğru hataya neden olan modeller çıkarmayı amaçlamaktadır. Ek olarak, SeqFuzzSDN, SDN'leri fuzz test etmek için en son teknoloji yöntemlerinin üç uzantısıyla karşılaştırılmıştır. Elde edilen bulgular, genişletilmiş SOTA yöntemleriyle karşılaştırıldığında, SeqFuzzSDN'nin (1) aynı zaman bütçesi içinde hatalara yol açan daha çeşitli mesaj dizileri ürettiğini göstermektedir [36].

5G RRC için uyarlanabilir güçlendirme öğrenme tabanlı fuzz testi güvenlik değerlendirmesi çalışmasında önerilen yaklaşım özellikle mesaj değişim dizilerindeki, kimlik doğrulama prosedürlerindeki ve kaynak tahsis mekanizmalarındaki potansiyel güvenlik açıklarını hedef alır. Birden fazla protokol yığını katmanında fuzz testi stratejilerini optimize etmek için Q-öğrenmeyi kullanır [37].

Üretken çelişkili ağları kullanan derin öğrenme tabanlı MQTT fuzz testi yöntemi çalışmasında önemli sayıda gerçekçi MQTT protokolü üretmek için, SGANFuzz adlı Generative Adversarial Networks (GAN) tabanlı bir protokol fuzz test aracı önerilmiştir. Deneysel sonuçlara göre, SGANFuzz'un 7 MQTT uygulaması arasında 6 güvenlik açığını başarıyla tespit ettiğini göstermektedir. En son teknoloji fuzz testi araçlarıyla karşılaştırıldığında, SGANFuzz'ın güvenlik açığı tespiti açısından en etkili test aracı olduğunu kanıtlamış ve MQTT broker'larından daha benzersiz ağ yanıtları olarak geri bildirim kapsamını genişlettiği görülmüştür [38].

2.5.3. İyi yapılandırılmış girdi dosyası fuzz testleri

İyi derece yapılandırılmış dosyaları alan programlar girdileri sözdizimi ayrıştırma, anlamsal kontrol aşamalarında işler. Dilbilgisi tabanlı fuzz testi teknikleri genellikle sözdizimi ayrıştırma aşamasını geçer çünkü test edilen uygulamanın dilbilgisi kurallarına uyan girdiler üretmeyi başarırlar. Ancak girdilerin çoğu anlamsal kontrol adımını geçemez. Genellikle bu tür programların sözdizimsel özelliklerini tanımlamak için bağlamdan bağımsız dilbilgileri kullanılır. Bu tür fuzz testlerinde kullanılan en

yaygın yöntemlerden biri AST'yi (soyut sözdizimi ağacı) oluşturmak ve AST üzerinde eylemler gerçekleştirmektir.

İyi derece yapılandırılmış girdilerin fuzz testleri açısından, Skyfire [39], iyi derece yapılandırılmış girdileri kabul eden fuzz testi uygulamaları için iyi dağıtılmış başlangıç girdileri üretmek üzere çok sayıda örnekteki bilgiyi değerlendiren bir başlangıç oluşturma yaklaşımı oluşturur. Skyfire girdi olarak bir gövde ve bir dilbilgisi alır ve gövdeye dayalı bir dağıtım modeli oluşturur. Yaklaşım hem sözdizimi özelliklerini hem de anlamsal kuralları belirtmek için bir PCSG öğrenir ve ardından öğrenilen PCSG'yi başlangıç girdileri üretmek için kullanır. Fuzz testlerinin, ayrıştırıcıların sözdizimi ve anlamsal kontrol aşamasını geçmesine yardımcı olmak için dilbilgisi geçerli başlangıçlar üretir. Bu yaklaşımda kullanılan test edilen sistemler XSL, XML ve JavaScript tabanlı ayrıştırıcılardır. Fuzz testinin etkinliğini ölçmek için kullanılan ölçütler, hata sayısı, satır ve kod kapsamı ve hata türüdür. Bu uygulamaların AFL ile test edilmesi sonucunda, sınır dışı okuma, serbest bırakıldıktan sonra kullanım hataları ve hizmet reddi hataları türünde bellek bozulması hataları tespit edilmiştir. Ama bu yöntemde fuzz testi çalıştırma biraz uzun zaman almıştır, aylar mertebesinde vakit almıştır.

NAUTILUS [40], dilbilgisi kullanımını kod kapsamı geri bildirimi kullanımıyla birleştirerek iyi derece yapılandırılmış girdiler gerektiren programlar için fuzz testi gerçekleştirir. Bu yaklaşım bir gövde gerektirmez, yalnızca bir uygulamanın kaynak kodu ve bunun için girdiler üretecek bir dilbilgisi gerektirir. Kural mutasyonu, rastgele yinelemeli ve ekleme vb. gibi bir dizi yeni mutasyon stratejisi önerir. Ayrıca kod kapsamına dayalı zamanlama mekanizmasını bir GBF'ye tanıtır. Yukarıda belirtilen bu stratejiler fuzz testinin verimliliğini önemli ölçüde artırabilir. Ancak NAUTILUS, fuzz testlerinin tüm girdisini tek bir CFG'ye basitleştirir, bu nedenle anlamsal denetimi dikkate almaz. NAUTILUS, dilbilgisi girdilerini JavaScript Nesne Gösterimi (JSON) veya ANother Tool for Language Recognition (ANTLR) için yazılmış dilbilgisi olarak kabul eder. Bu yöntem JavaScript motoru, PHP, mruby ve Lua dahil olmak üzere dört farklı uygulama için fuzz testlerini çalıştırmıştır ve use-after-free, yığın ara bellek taşması, yığın taşması türünde bellek bozulması hatalarını tespit etmiştir. Fuzz testinin etkinliğini ölçmek için kullanılan değerlendirme ölçütleri, hata sayısı ve kapsam bilgisidir. Sonuçlar AFL ve IFuzzer'ın sonuçlarıyla karşılaştırılmıştır. Superior [7], AST tabanlı bir kırpmaya ve mutasyon yöntemi geliştirir ve bu yöntem AFLin bir

uzantısı olarak kullanılır. Yapılandırılmış girdileri işleyen programları hedefleyen, yeni bir dilbilgisi farkında kapsama tabanlı gri kutu fuzz testi yaklaşımı önerir. Yaklaşım, girdi olarak bir hedef program ve genellikle genele açık olan test girdilerinin bir dilbilgisini alır. Dilbilgisine dayanarak, her test girdisini bir AST'ye ayrıştırır. AST'leri kullanarak girdi yapısını geçerli tutarken test girdilerini etkili bir şekilde kırabilen dilbilgisi farkında bir kırpma stratejisi sunar. Yöntem öncelikle belirteçleri dilbilgisi farkında bir şekilde ekleyerek/üzerine yazarak AFLin sözlük tabanlı mutasyon stratejisini geliştirir ve ardından bir test girdisinin AST'sindeki bir alt ağacı, kendisinden veya kuyruktaki başka bir test girdisinden gelen alt ağaçla değiştiren ağaç tabanlı mutasyon stratejisi önerir. Superionun fuzz testi etkinliği XML ve JavaScript motorlarında değerlendirilmiştir. Araç, hata bulma yeteneği ve kod kapsamı açısından AFL ve jsfunfuzz ile karşılaştırılmıştır.

JavaScript motorlarını grafik tabanlı bir yöntemle fuzz testindeki grafik tabanlı ara gösterim, FlowIR, mutasyon hedefi olarak JS kontrol akışını ve veri akışını doğrudan temsil eder. FlowIR, güçlü semantik mutasyonun uygulanması için mutlaka gereklidir. Mutasyon operatörlerini veri akışı ve kontrol akışı düzeyinde destekler, böylece mutasyon operatörlerinin ayrıntı düzeyini genişletir [41].

JavaScript motorları için yama tespiti amaçlı fuzz testleri çalışmasında javaScript motoru yama fuzz testi için uçtan uca sürdürülebilir bir yaklaşım önerilmiştir. Buna PatchFuzz adı verilmiştir. Daha geniş bir tarihsel güvenlik açığı yelpazesinin PoC'lerinin toplanmasını otomatikleştirmekte ve yeni güvenlik açıklarını daha etkili bir şekilde ortaya çıkarmak için hem PoC'leri hem de bunlara karşılık gelen yamaları kullanmaktadır [42].

JavaScript ortamında mutasyon tabanlı derin öğrenme çerçevesi test yöntemi çalışmasında DLJSFuzzer adlı mutasyona dayalı bir JavaScript derin öğrenme çerçevesi test yöntemi önerilmektedir. DLJSFuzzer, test giriş tensörleri üretmek için önbellek yeniden kullanım mekanizmasını hedefleyen 13 tensör mutasyon kuralı tasarlamıştır [43].

Gelişmiş tarayıcı fuzz testi ince taneli semantik hizalama çalışması ile Tacoma adı verilen web tarayıcıları için özel olarak tasarlanmış yeni bir fuzz testi çerçevesi önerilmiştir. Tacoma üç ana modülden oluşur: bir anlamsal ayrıştırıcı, bir anlamsal hizalayıcı ve bir giriş oluşturucudur [44].

Tablo 2.1, fuzz testlerindeki verimliliği ölçmek için kullanılan değerlendirme metrikleri, hedef program formatı, fuzz testi metoduyla karşılaştırılan diğer araçlar gibi fuzz testi yöntemlerinin ayrıntılarını özetlemektedir.

Tablo 2.1. Fuzz testi metotları ve değerlendirme metrikleri.

Fuzz Testi Metodu	Değerlendirme Metriği	Program Formatı	Karşılaştırılan Fuzz Araçları
Csmith	hata sayısı, kapsama	GCC, LLVM	N/A
LangFuzz	hata sayısı	JavaScript motoru	Jsfunfuzz
IFuzzer	hata sayısı	JavaScript motoru	LangFuzz
Radamsa	kapsama, geçme oranı	XML, PDF	SampleSpace, Glade
Learn&Fuzz	hata sayısı, kapsama, geçme oranı	PDF	N/A
SmartSeed	hata sayısı, kapsama	Mp3, bmp, flv	AFL, hongfuzz, Vuzzer
Skyfire	hata sayısı, kapsama	XML, XSL, JavaScript motoru	AFL
Superion	hata sayısı, kapsama, geçme oranı	XML ve JavaScript motorları	AFL, jsfunfuzz
NAUTILUS	hata sayısı, kapsama	JavaScript motoru, php, mruby, Lua	AFL, Ifuzzer
Deng. Et al	Kapsama	XML ve JavaScript motorları	AFL, Superion
Tez Çalışmamız	hata sayısı, kapsama, tespit performansı	XML motoru	AFL, Superion



3. FUZZ TESTİ VERİMLİLİK ARTIRMA TEKNİKLERİ

İlk fuzz testi aracı Miller ve arkadaşları tarafından 1990 yılında geliştirilmiştir ve UNIX araçlarının güvenilirliğini test etmek için tasarlanmıştır. İlk tasarlanan fuzz test sistemleri tipik olarak rastgele mutasyona dayalıydı ve tamamen kara kutu olarak çalışmaktaydı. Zamanla artan ihtiyaçlara karşı verimsiz kalan fuzz testlerini daha verimli çalışır hale getirmek için yeni yöntem ve yaklaşımlar geliştirildi [2]. Bu yöntemler sırasıyla şu şekilde sıralanabilir; dinamik sembolik yürütme, kapsam geri bildirimi, dilbilgisi gösterimi, kusur analizi, statik analiz, zamanlama algoritmaları ve makine öğrenmesi. Bu tip destekleyici ve fuzz testlerinin verimliliğini artırıcı metotların gelişmesindeki en önemli etkenlerden bir tanesi fuzz testlerinin, bir grup program girişinin, yani test senaryolarının üretilmesiyle başlamasıdır. Üretilen test senaryolarının kalitesi test sonuçlarını doğrudan etkilemektedir. Üretilen girdi verileri, giriş formatı için test edilmiş programların gereksinimlerini mümkün olduğunca karşılamalıdır. Yani farklı bir deyişle test edilen sistemin kabul ettiği girdinin sözdizimsel ve mantıksal olarak gerektirdiği kuralları olabildiği kadar karşılamalıdır. Ardından üretilen ve kalite düzeyi belirli bir seviyede olan bu girdilerin program içinde mümkün olduğu kadar fazla ulaşılamayan derin noktalara ulaşması ve test ediyor olabilmesi gerekmektedir. Bunu sağlamanın en önemli yollarından bir tanesi kapsama geri bildirimli fuzz testlerinin kullanılmasıdır. Üretilen her bir yeni girdinin uygulama kodunda ne kadar kapsama sağladığı bilgisine dayalı olarak bir sonraki girdinin üretim stratejisi belirlenmektedir. Fuzz testleri sistemlerdeki hatalı durumları ve bunların yol açabileceği güvenlik zafiyetlerini bulma konusunda çalıştığı için sistemlerde tespit edilen hataların nicelik ve nitelik açısından öne çıkan özellikleri fuzz testlerinin verimliliğini ölçmede kullanılan en temel kriterlerdendir.

Sistemlerde tespit edilen hatalar en temel anlamda iki kategoriye ayrılabilir: sığ hatalar ve gizli hatalar. Yürütmenin erken aşamasında hedef programın çalışmaz hale gelmesine neden olan hatalara sığ hatalar denilir. Program mantığının derinlerinde bulunan ve bulunması daha zor olan hatalar da gizli hatalar olarak adlandırılır. Genel olarak test senaryoları oluşturmak için rastgele mutasyon yöntemini kullanan geleneksel kara kutu test aracı, yüksek kod kapsamına sahip değildir ve netice olarak

sığ hatalar bulur. Bu nedenle hafif, hızlı ve kullanımı kolaydır. Buna karşılık beyaz kutu veya gri kutu fuzz test araçları daha yüksek kod kapsamına ulaşabilir ve normalde daha fazla gizli hata bulabilir. Bu bağlamda sadece temel mutasyon yöntemlerinin kullanıldığı fuzz teknikleri aptal testler olarak adlandırılmaktadır. Bu tür fuzz test yöntemi girdi biçiminin farkında olmadan bir girdi dosyasının bazı baytlarını rastgele değiştirir. Bunun aksine test senaryosu oluşturmaya rehberlik etmek için girdinin spesifikasyonunu kullanan veya çalışma zamanı bilgisini (örneğin, yol kapsamı) benimseyen teknikler genellikle “akıllı” fuzz testleri olarak bilinir [12].

Dolayısı ile bir fuzz test aracının verimliliği, test aracının bulabileceği toplam güvenlik açığı sayısı ile doğrudan ilişkilidir. Diğer yandan, bir test aracının bir sistem için verimliliği, güvenlik açıklarının keşfedilme hızı ile de tanımlanır. Bu şekilde seçilen bir yöntemde sentetik hataları sisteme enjekte ettikten sonra test aracı çalıştırılır ve aracın zafiyetleri tespit etmesi beklenir. Ancak sentetik hatanın gerçek bir hataya ne kadar benzediği ve gerçeği temsil etmesi koşulu oldukça önemlidir. Temsili değilse gerçek güvenlik açıklarından nasıl farklı olduklarını bilmek özellikle önemlidir.

En temel anlamda bir fuzz test aracını verimli bir şekilde geliştirirken dikkat edilen ana kriterler aşağıdaki gibidir.

- Girdi Tohumların ve test durumlarının nasıl oluşturulduğu ve seçildiği
- Hedef programın özelliklerine göre girişlerin nasıl doğrulandığı
- Sistemi çalışmaz hale getiren test durumlarının nasıl yönetildiği
- Çalışma zamanı bilgisinden nasıl yararlanıldığı
- Fuzz testinin ölçeklenebilirliğinin nasıl geliştirildiği

3.1. Kapsama Yönlendirmeli Fuzz Testleri

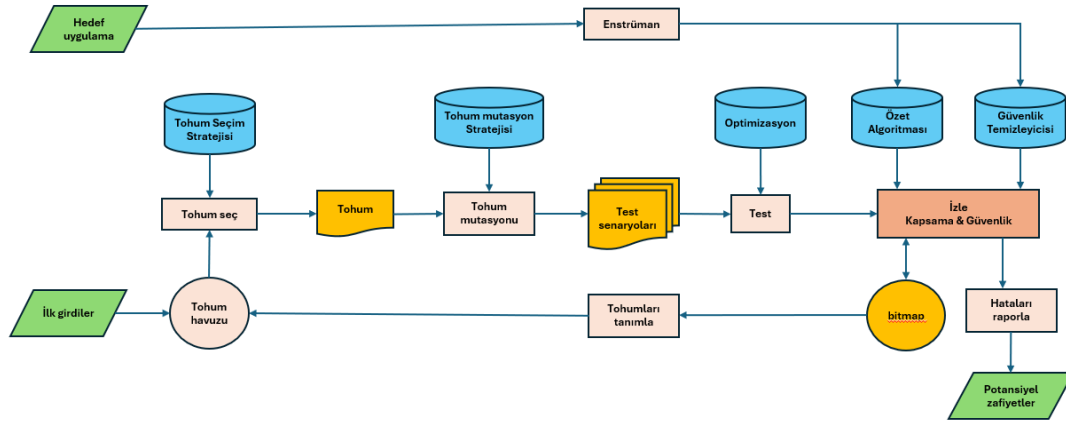
Kapsama yönlendirmeli fuzz testleri, programın yürütülmesinde farklı yolları geçerek girdiler üretir. Kapsama hakkında bilgi edinmek için kod enstrümantasyonundan yararlanır. Fuzz testlerinde geri bildirim, çalışma zamanı bilgilerini dikkate alarak bir sonraki döngüde test senaryosunun oluşturulmasına rehberlik etmek anlamına gelir. Geri bildirim bilgileri genellikle yol kapsamını kullanır. Kapsama kılavuzlu test araçları, esas olarak yol kapsamı bilgilerini enstrümantasyon araçlarından alır. Kapsama kılavuzlu fuzz testi araçlarına birkaç örnek vermek gerekirse; AFL, honggfuzz ve Syzkaller [45].

Derin bir program testinin elde edilmesi için, test araçları mümkün olduğunca çok sayıda program çalıştırma durumundan geçmelidir. Bu bağlamda kod kapsamını ölçmek iyi bir çözüm haline gelmiştir. Program analizinde program temel bloklardan oluşur. Temel bloklar, tek bir giriş ve çıkış noktası olan kod parçacıklarıdır, temel bloklardaki talimatlar sırayla ve bir kez yürütülür. Kod kapsamı ölçümünde, son teknoloji yöntemler temel bloğu en iyi ayrıntı düzeyi olarak ele almaktadır [1].

Kapsama yönlendirmeli fuzz test aracı olan AFL, hafif program enstrümantasyonu ile kapsama bilgisini alır. Kaynak kodunun sağlanıp sağlanmadığına göre AFL, derleme enstrümantasyonu ve harici enstrümantasyon olmak üzere iki enstrümantasyon modu sağlar. AFL kapsam tabanlı fuzz testlerine kenar ölçüm yöntemini ilk uygulayan araçtır. Bununla birlikte bir diğer kapsam yönlendirmeli test aracı olan Driller, daha fazla kapsama alanı keşfetmenin önündeki en büyük engel olan yol kısıtlamalarını çözmek için sembolik yürütmeyi kullanmıştır [1].

Diğer yandan kapsama tabanlı gri kutu fuzz testleri, güvenlik açığı tespiti için popüler ve etkili bir yaklaşım olarak literatürde yerini almıştır. Uygulama hakkında bilgi eksikliğine sahip olan kara kutu yaklaşımlarının ve program analizi ve kısıtlama çözme nedeniyle yüksek ek masraflar içeren beyaz kutu yaklaşımlarının aksine, gri kutu yaklaşımları hafif kod enstrümantasyonu kullanır. Kod kapsamalı gri kutu test yaklaşımı, mutasyon yolunu kullanarak girdi alanı keşfi ile ilerler. İlk oluşturulan ve tohum girdi olarak adlandırılan girdilerden başlayarak, önceden tanımlanmış bir dizi genel mutasyon işlemleri (bitflip'ler gibi) kullanarak ilk girdileri değiştirir. Mutasyona uğramış girdiler tarafından uygulanan kontrol akışları, yeterince ilginç olup olmadıklarını belirlemek için incelenir. Hafif program enstrümantasyonu, test yapılırken karşılaşılan kontrol akışlarının yeterince ilginç, yani sistemdeki zafiyetleri ortaya çıkarmada ne kadar yeterli olup olmadığı kararının verilmesine yardımcı olur [46].

Kapsam yönlendirmeli fuzz testlerinin genel olarak iş akışı şu şekildedir;



Şekil 3.1. Kapsama Geri Bildirimli Fuzz Test Süreci

Kapsama yönlendirmeli fuzz testlerinin temel sorunlarından biri başlangıç girdilerini almaktır. Kapsama dayalı fuzz test araçlarının çoğu, ilk tohum girdilerinin kalitesine bağlı olan mutasyona dayalı bir test senaryosu oluşturma stratejisi kullanır. İyi başlangıç tohum girdileri, fuzz testlerinin verimliliğini ve etkinliğini iyi bir seviyede artırabilir. İyi biçimlendirilmiş tohum girdisine dayalı mutasyonun, daha derin ve ulaşılması zor yollara ulaşan test senaryoları oluşturmada daha olasıdır. Tohum girdilerini toplamak için yaygın olarak kullanılan yöntemler, standart karşılaştırma ölçütlerini kullanmak, internetten tarama yapmak ve mevcut POC örneklerini kullanmayı içerir. Hedef uygulama girişlerinin çeşitliliği göz önüne alındığında, internetten taramak en sezgisel yöntemdir. Belirli formatlardaki dosyalar kolayca indirilebilir ve kullanılabilir. Ayrıca yaygın olarak kullanılan bazı dosya formatları için ücretsiz test veri setleri sağlayan birçok açık test projesi mevcuttur [1].

Kapsama yönlendirmeli fuzz testlerinin ikinci anahtar sorusu, test senaryolarının nasıl oluşturulacağıdır. Test senaryolarının kalitesi, fuzz testinin verimliliğini ve etkinliğini etkilediği için önemlidir. İyi test senaryoları daha fazla program yürütme durumunu keşfeder ve daha kısa sürede daha fazla kodu kapsar [14]. Ayrıca iyi test senaryoları potansiyel savunmasız yerleri daha kolay bulabilir. Gri kutu test araçlarından biri olan Vuzzer, uygulamaya duyarlıdır ve statik ve dinamik analiz ile bütünleşir. Programın yürütülmesi sırasında Vuzzer, kontrol akışı dallarını etkileyen bilgileri toplamak için dinamik kusur analizi tekniğini kullanır [1].

Bununla birlikte test senaryolarını otomatik olarak oluşturmak için bazı makine öğrenme teknikleri kullanılabilir. Makine öğrenimi tekniklerini kullanarak bir dizi geçerli girdiden girdi biçimini sistematik olarak öğrenirler ve ardından öğrenilen

bilgileri test senaryosu oluşturmaya rehberlik etmek için kullanır. Örneğin sinir ağları, geçmişteki bulanık keşiflerden öğrenmek ve girdi dosyalarında hangi baytın değişeceğini tahmin etmek için kullanılır.

Kapsama rehberli fuzz testlerinin üçüncü anahtar sorusu, havuzdan tohum girdilerinin nasıl seçileceğidir. Test araçları, ana fuzz testi sürecinde yeni bir tur testinin başlangıcında mutasyona uğramak için tohum havuzundan sürekli olarak tohum seçer. Havuzdan tohumun nasıl seçileceği fuzz testlerinde açık olan bir diğer araştırma konusudur. İyi tohum seçim stratejileriyle test araçları daha fazla kodu kapsayan ve güvenlik açıklarını ortaya çıkarma olasılığı daha yüksek olan tohumlara öncelik verebilir. İyi bir seçim stratejisine sahip olmak, daha derin ve daha savunmasız kodları kapsayan ve gizli güvenlik açıklarını daha hızlı belirlemeye yardımcı olan tohumları seçmeye yarar. Havuzdan tohum seçerken ağırlıklı olarak statik ve dinamik analiz teknikleri ve bilinen güvenlik açıklarının özellikleri de kullanılabilir.

3.2. Otomat Tabanlı Sistemler

Gramatron, kapsama yönlendirmeli, dilbilgisine duyarlı nesil tabanlı bir test aracıdır. Gramatron girdi olarak hedef tarafından kabul edilen bir kontrol akış çizgesi alır. Ve sistemi çalışmaz hale getiren test senaryolarının çıktısını verir. Fuzz testi aşamasında, karmaşık hata tetikleyicilerini daha hızlı sentezlemek için agresif mutasyon operatörleriyle birlikte dilbilgisi otomatlarını kullanır. Bu yaklaşım, Gramatron'un karmaşık tetikleyicilerle hataları hızla bulmasını sağlar. Gramatron ile örnekleme önyargısını ele almak için dilbilgisi otomatları oluşturulur. Giriş durumu alanından tarafsız örnekleme için izin vermek için dilbilgisi yeniden yapılandırılır. Dilbilgisi farkında mutasyon operatörleri daha agresif olacak şekilde yeniden tasarlanır, böylece büyük ölçekli değişiklikler gerçekleştirilir.

Gramatron, geleneksel dilbilgilerini ayrıştırma ağaçlarıyla kullanmaya kıyasla etkili bir şekilde karmaşık hata tetikleyicilerini üretebilir. Gramatron tarafından sıfırdan üretilen girdiler, mevcut fuzz test araçlarına göre %24'e kadar daha fazla kapsama ulaştıklarından dolayı daha fazla çeşitliliğe sahiptir. Gramatron yaklaşımı, girdi üretimini %98 daha hızlı hale getirir ve girdiler %24 daha küçüktür. Gramatron tüm karmaşık hata tetikleyicilerini güvenilir ve daha hızlı bir şekilde bulur. Basit hata tetikleyicileri için Gramatron, Nautilus'u yedi adet yapılan çalışmanın dördünde geride bırakmıştır. Gramatron'un verimliliğini göstermek için, 10 günlük bir fuzz testi

kampanyası için üç popüler yorumlayıcıya yerleştirildi ve burada 10 yeni güvenlik açığı keşfedildi.

Gramatron, sonlu durum otomatlarını (FSA) kullanarak fuzz testleri gerçekleştirmek için dilbilgisi otomatları oluşturur. Bu, Gramatron'un hızlı ve agresif mutasyonları mümkün kılan hafif girdi temsilleri oluşturmasını sağlar. FSA, $M = (Q, \Sigma, \delta, q_0, F)$ olarak tanımlanır. Burada, Q , sonlu bir durum kümesidir. Σ , sonlu terminal kümesidir. δ , otomasyon durumları üzerindeki tüm geçişlerin kümesini tanımlamaktadır. q_0 ise, otomasyondaki gramerin başlangıç sembolüne karşılık gelen başlangıç durumudur. F de, otomasyondaki kabul durumuna karşılık gelir. Başlangıç durumundan son duruma geçiş kümesi, gramere ait bir giriş dizesini tanımlamaktadır.

Gramatron, dilbilgisi otomatları oluşturmak için aşağı itme otomatlarından (PDA) yararlanır. PDA, bir gramerden bağımsız bağlam için bir dil tanıyıcıdır. Gramatron bağlamında, boş bir yığın tarafından girdileri kabul eden 1 durumlu PDA'yı ele alır. PDA, $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ olarak tanımlanır. Burada, Q sonlu bir durum kümesidir. Σ sonlu terminal kümesidir. Γ , yığından itilebilen ve çıkarılabilen bir sembol kümesidir ve yığın sembolleri olarak da adlandırılır. Z ise yığına itilen başlangıç sembolüdür. q_0 başlangıç durumu iken ve F de kabul eden otomasyon durumudur. δ , otomasyon davranışını yöneten geçiş fonksiyonudur. Bağımsız değişken olarak $\delta(q, a, X)$ alır, burada $q \in Q$, $a \in \Sigma$ ve $X \in \Gamma$. δ 'nin çıktısı, p yeni bir durum ve γ , yığının tepesinde X 'in yerini ters yönde itilerek alan sembol dizisi olan sonlu bir (p, γ) çiftleri kümesidir. Eğer γ boşsa yığından yalnızca X çıkarılır.

Sonuç itibarıyla Gramatron, bir dilbilgisi otomasyonu oluşturmak için CFG üzerinde otomatik iki adımlı bir dönüşüm gerçekleştirir. İlk olarak, CFG'yi (içerikten bağımsız gramer) dilbilgisi yeniden yapılandırmasını gerçekleştiren GNF'sine (Greibach Normal Form) dönüştürür. İkinci olarak, dilbilgisinin GNF'sini bir otomasyona dönüştürür. Gramatron, dilbilgisine uyan herhangi bir girişi bir otomasyon gidişi olarak kodlayabilir. Dilbilgisi otomasyonları, fuzz test aracının giriş durumu alanından tarafsız örnekleme yapmasını sağlayarak dilbilgisini yeniden yapılandırır. Bu, test aracının daha yüksek çeşitliliğe sahip girdileri daha sık üretmesini sağlamaktadır.

Uygulama anlamında ise Gramatron, C ve Python'da geliştirilmiştir. Dilbilgisi ön işlemcisi Python'da geliştirilmiş, girdi üretici ve mutatorü ise C'de geliştirilmiştir. Giriş olarak, fuzz testi hedefi tarafından kabul edilen bir dilbilgisi alır. Gramatron, fuzz

testi sırasında girdi oluşturma ve mutasyon gerçekleştirmek için dilbilgisi otomatlarından yararlanmak üzere AFL++'ı temel alarak üzerinde değişiklik yapar. Bununla birlikte mutasyonunu yönlendirmek için test hedefinden gelen kod kapsamı geri bildirimini kullanır [47].

3.3. Yönlendirmeli Gri Kutu Sistemleri

Birçok gri kutu fuzz testi aracı, sınırlı bir zaman dilimi içinde mümkün olduğunca çok program yolunu kapsamayı amaçlayarak kapsam kılavuzlu olarak çalışmaktadır. Bunun nedeni, kod kapsamının sezgisel olarak hata kapsamıyla ilişkili olması ve daha yüksek kod kapsamına sahip fuzz test araçlarının daha fazla hata bulabilme yeteneğidir. Ancak diğer yandan, programın tüm kodlarını eşit bir şekilde ele almak uygun değildir çünkü kapsanan kodların çoğu hata içermeyebilir. Bu nedenle, kod kapsamını fazlasıyla genişleterek sistemi test etmek, özellikle köşe durumlar için az verimli sonuçlar üretir. Uygulamada tam kod kapsamına ulaşmak zor olduğundan, araştırmacılar verimliliği artırmak ve kaynakları korumak için koddaki savunmasız kısımları hedeflemişlerdir. Yönlendirilmiş fuzz testleri, bu amaca ulaşmanın bir metodu olarak önerilmektedir.

Yönlendirilmiş gri kutu fuzz testleri, güvenlik açığı keşfi alanında popüler hale gelmiştir. Kapsama dayalı fuzz testlerinden farklı olarak, programdaki tüm kodların zafiyete açık olmadığı üzerinden hareket ederler. Yönlendirilmiş gri kutu fuzz testinin amacı, potansiyel olarak hatalı kodun bir parçasının gerçekten bir hata içerip içermediğini kontrol etmektir. Bir kod parçasının potansiyel olarak hatalı kod olarak nasıl tanımlanacağı ve makine öğrenimine bir girdi olarak bir özellik vektörüne nasıl dönüştürüleceği zor bir araştırma problemidir. Yönlendirilmemiş bir şekilde kod kapsamını genişleten, kapsama yönlendirmeli gri kutu test araçlarının aksine, yönlendirilmiş bir gri kutu test aracı, zamanının çoğunu belirli hedeflere (hata olası bölgeler) yönlendirerek ilgisiz kısımlara gereksiz kaynak harcamaz. Bu nedenle, yönlendirilmiş gri kutu fuzz tekniği, yama testi, hata çoğaltma ve uzman hata yakalama gibi senaryolar için özellikle uygundur. Ayrıca bu bağlamda yalnızca belirli kod parçalarını hedefleyen konuma yönelik tür değil, aynı zamanda anormal program davranışlarını ortaya çıkarmayı hedefleyen davranışa yönelik testler de çalışmalar arasında yer almaktadır.

Yol kapsamını fazlasıyla genişleten kapsama tabanlı fuzz testlerinin aksine, yönlendirilmiş bir fuzz test aracı hedef konumlara odaklanır ve zamanın çoğunu, ilgisiz parçaları zorlayarak kaynakları boşa harcamadan bu konumlara ulaşmaya harcar. Başlangıçta yönlendirilmiş fuzz test araçları, farklı program yollarını uygulayan girdiler üretmek için program analizi ve kısıtlama çözme kullanan sembolik yürütmeye dayanmaktaydı. Bu tür yönlendirmeli araçlar, erişilebilirlik sorununu yinelemeli bir kısıtlama sorunu olarak göstermektedir. Ancak, yönlendirilmiş sembolik yürütme ağır program analizi ve kısıtlama çözmeye dayandığından ölçeklenebilirlik ve uyumluluk sınırlamalarına sahiptir.

Yol kapsamını artıran kapsama kılavuzlu fuzz testinin aksine, yönlendirmeli fuzz testleri kodda önceden belirlenmiş bir dizi konuma ulaşmayı hedefler (muhtemelen hatalı kısımlar) ve zamanının çoğunu kaynak israf etmeden hedef konumlara ulaşmak için harcar. Yönlendirmeli fuzz testlerinin ilkesini tanımlamanın bir yolu olarak, AFLGo, örnek olarak kullanılabilir. AFLGo, kapsama dayalı fuzz testleri ile aynı genel ilke ve mimariyi izler. Derleme zamanında, enstrümantasyon hariç, AFLGo ayrıca giriş ve önceden tanımlanmış hedefler arasındaki mesafeleri hesaplar. Daha sonra, çalışma zamanında AFLGo tohumları yeni yol kapsamı yerine mesafeye göre önceliklendirir ve temel blok düzeyindeki mesafede hedeflere daha yakın tohumlara öncelik verir. Böhme ve arkadaşları gri kutu fuzz testi sürecini, bir "güç çizelgesi" kullanılarak verimli bir şekilde gezinilebilen bir Markov zinciri olarak görmüştür. Hedeflere daha yakın olan bir tohuma, daha uzak olan bir tohumdan daha fazla enerji atamak için simüle edilmiş bir tavlama stratejisinden yararlanmışlardır.

Yönlendirilmiş gri kutu fuzz testleri, tohum ile hedef arasındaki mesafeyi tohum seçimine yardımcı olmak için uygunluk işlevi olarak kullanılabilir. Hedefe daha yakın olan tohumlara daha fazla mutasyon şansı vererek, gri kutu fuzz testini hedef konumlara kademeli olarak ulaşmaya yönlendirebilir. Geleneksel fuzz test tekniklerinin aksine, ulaşılabilirliği, üretilen tohumlar ile hedefleri arasındaki mesafeyi en aza indirmek için bir optimizasyon sorunu olarak kullanır. Yönlendirilmiş sembolik yürütmeyle karşılaştırıldığında, yönlendirmeli fuzz testi çok daha iyi ölçeklenebilirliğe sahiptir ve verimliliği daha çok artırır. Örneğin, Böhme ve arkadaşları Heartbleed güvenlik açığını 20 dakika içinde yeniden üretebilirken, yönlendirilmiş sembolik yürütme aracı KATCH, 24 saatten daha fazlasına ihtiyaç duyar. Şimdilik, yönlendirmeli gri kutu fuzz testleri tohumları önceliklendirmek için elle etiketlenen

hedef sitelere ve mesafe tabanlı metriklere dayanan orijinal örüntünün ötesine ulaşmıştır. Mevcut test araçları yalnızca hedefleri otomatik olarak belirlemekle kalmayıp aynı zamanda hedef program davranışını yönlendirilmiş bir şekilde açığa çıkarabilir. Yama testi, hatanın yeniden üretimi, sonuç doğrulama, enerji tasarrufu ve özel hata bulma gibi farklı senaryolar altında sistem testini artırmak için çok sayıda varyasyon kullanılmıştır [48].

DeepGo fuzz testi metodolojisi ile yürütülen tahmini yönlendirilmiş gri kutu fuzz testleri ile hedef siteye optimal bir yol üzerinden ulaşılması için tarihsel ve tahmin edilen bilgileri birleştirebilen öngörülü yönlendirilmiş gri kutu fuzz test aracı önerilmektedir. Bu çalışma ile yönlendirilmiş fuzz testi hedef siteye belirli yol geçiş dizileri aracılığıyla ulaşma süreci olarak modelleyen yol geçiş modeli önerilmiştir. Mutasyon tarafından üretilen yeni tohum yol geçişine neden olur ve yüksek ödüllü yol geçiş dizisine karşılık gelen yol, hedef siteye onun üzerinden ulaşma olasılığının yüksek olduğunu göstermiştir [49].

Büyük dil modeli aracılığıyla yönlendirilmiş gri kutu fuzz testleri çalışmasında büyük dil modelini (LLM) kullanan otomatik bir çerçeve olan HGFuzzer önerilmiştir. HGFuzzer, yol kısıtlaması sorunlarını hedeflenen kod oluşturma görevlerine dönüştürerek gereksiz keşif yollarını önemli ölçüde azaltmak için sistematik olarak test koşulları ve ulaşılabilir girdiler üretmiştir [50].

Tekrarlı önceliklendirme ile büyük ölçekli hedef kümeleri için yönlendirilmiş gri kutu fuzz testleri bağlamında büyük ölçekli hedef kümeleri senaryolarını ele alabilen yeni bir yönlendirmeli gri kutu test yaklaşımı olan Prospector sunulmuştur. Prospector, odaklanmış hedeflerden oluşan seçili bir gruba odaklanmak için yinelemeli bir süreç kullanır. Bu hedefleri dinamik olarak sürdürmek için Prospector, savunmasız kalıpları göz önünde bulunduran ve hedeflerin yeterliliğini test eden daha ayrıntılı bir strateji sunar. Ardından Prospector, keşif-istismar planlaması, tohum seçimi ve bayt planlamasında stratejileri iyileştirerek odaklanmış hedeflere yönelik fuzz testi yaklaşımını daha da belirginleştirir [51].

Kritik kod rehberli yönlendirilmiş gri kutu işlemler için fuzz testleri çalışmasında WAFLGO aracı, taahhütler tarafından tanıtılan güvenlik açıklarını etkili bir şekilde keşfetmek için önerilmiştir. WAFLGO, etkilenen kodu kapsamlı bir şekilde keşfetmek için yeni bir kritik kod rehberli girdi oluşturma stratejisi kullanır. Özellikle, iki tür

kritik kod tanımlanmıştır: pathprefix kodu ve data-suffix kodu. Kritik kod ilk olarak girdi oluşturmayı kademeli ve artımlı bir şekilde değişiklik sitelerine ulaştırmak için yönlendirir. Sonrasında kritik kodun erişilebilirliği korunurken, girdi oluşturma stratejisi etkilenen kodu keşfederken oluşturulan girdilerin çeşitliliğini daha da teşvik etmektedir [52].

3.4. Makine Öğrenmesi Tabanlı Sistemler

Makine öğrenimi teknolojisinin fuzz testleri üzerinde uygulanması güvenlik araştırmacılarının da ilgisini çekmiştir ve nedenleri şu şekilde özetlenebilir:

Fuzz test sürecinin birçok aşaması sınıflandırma problemi olarak ele alınabilir. Örneğin tohum dosyalarının ve test senaryolarının geçerli olup olmadığı, çalışmaz hala gelen sistemlerin güvenlik açısından zafiyete uğratıp uğratılamaması, hangi mutasyon operatörlerinin seçileceği gibi. Makine öğrenmesi teknikleri, sınıflandırma problemleriyle başa çıkmada nispeten mükemmel yeteneklere sahiptir [38]. Geleneksel fuzz testlerinde manuel tanımlama gerektiren birçok adım vardır ve bu süreç zaman alıcıdır. Ancak, makine öğrenimi modelinin tahmin süreci daha az zaman harcar. Bazı yöntemlerin veya adımların makine öğrenimi teknikleriyle değiştirilmesiyle fuzz testinde harcanan zaman azaltılabilir.

Fuzz testlerinde programın format kontrolünü atlamak, test işleminin daha derinlemesine kodu keşfetmesi için vazgeçilmez bir araçtır. Giriş sözdizimi biçiminin geleneksel manuel tanımlama genellikle zaman alıcıdır ve nispeten basittir. Makine öğrenimi, sözdizimi özelliklerine uyan geniş bir örnek havuzundan dil bilgisi kurallarını otomatik olarak öğrenmektedir [53].

Farklı fuzz testi adımlarına makine öğrenimi tekniklerinin uygulanması şu şekildedir;

- Tohum Girdi Dosyası Oluşturma
- Test Durumu/Senaryosu Oluşturma
- Test Durumu/Senaryosu Filtreleme
- Mutasyon Operatörü Seçimi
- İstismar Edilebilirlik/Kullanılabilirlik Analizi

Tohum Girdi Dosyası Oluşturma: Tohum girdi dosyası oluşturmada, makine öğrenme teknolojisi genel olarak girdi tohum dosyasının dilbilgisel ve anlamsal

bilgilerini öğrenmek için kullanılır. Tohum girdi dosyasının kalitesi, test etkinliğini etkileyen önemli bir faktördür. Bu belirtilen çözümde semantik bilgileri otomatik olarak ayıklamak ve tohum dosyaları oluşturmak için anlamsal kurallar ve dilbilgisi özellikleri içeren olasılıksal bağlama duyarlı dilbilgisinden faydalanılmaktadır.

Test Senaryosu Oluşturma: Yüksek kod kapsamı veya güvenlik açığı yönelimli bir test senaryosu oluşturmak, fuzz testlerinde güvenlik açığı tespitinin verimliliğini etkili bir şekilde artırmaktadır. Mutasyona dayalı yöntemde, makine öğrenimi teknolojisinin kullanımı temel olarak tohum dosyasının nerede mutasyona uğratılması gerektiğini çözmektir. Üretime dayalı yöntemde, makine öğrenimi teknolojisi, giriş dilbilgisi kurallarına uyan örnekler oluşturmak için temel olarak giriş örneklerinin dilbilgisel ve anlamsal bilgilerini öğrenmektedir.

Test Senaryosu Filtreleme: Test senaryosu filtrelemede, bir sınıflandırıcı oluşturmak için makine öğrenimi kullanılır, böylece hangi test senaryolarının daha fazla kod yürütmesinin daha kolay olduğunu ayırt edebilir ve programın çalışmaz hale geldiği durumları daha kolay tetikleyebilir. Test senaryosu filtrelemenin amacı, çok sayıda test senaryosundan yeni yolları veya güvenlik açıklarını tetikleme olasılığı daha yüksek olan test girdisini seçmektir. Giriş test senaryosu, makine öğrenimi teknikleri kullanılarak hangi test senaryosunun daha fazla yürütülmesi gerektiğini belirlemek için analiz edilebilir ve sınıflandırılabilir.

Mutasyon Operatörü Seçimi: Mutasyon operatörü seçiminde en yaygın olarak kullanılan makine öğrenme algoritması, fuzz testi sırasında mutasyon operatörlerinin seçimini yönlendirmek için tanımlanmış bir ödül mekanizması kullanan pekiştirmeli öğrenmedir.

İstismar Edilebilirlik/Kullanılabilirlik: Fuzz testlerinde büyük program çökmeleri ve hata mesajları vardır, ancak yalnızca birkaçı güvenlik açığı mevcut bulunmaktadır. Test edilen sistemlerde meydana gelen çökmelerden gerçek güvenlik açıklarının nasıl bulunacağı önemli bir araştırma problemi ve uygulama alanıdır.

Makine öğrenmesinin fuzz testlerinde en başarılı uygulamaları, girdi oluşturma aşamasında gerçekleşir. Genetik Algoritmalar girdi üretimi için en sık kullanılan makine öğrenmesi tekniğidir [11]. Genetik algoritmalara dayalı fuzz test araçları, objektif uygulamanın girdi formatını dikkate almaz. Genetik algoritmalara göre araçlar, bayt çevirme, çaprazlama vb. ile ayarlanan ilk girdi tohumunu değiştirir.

Ardından, programdaki çökmeleri keşfetmek için araçlar, uygulamaların girdileri olarak ilk girdi tohum kümesini ve mutasyona uğramış girdi dosyalarını alır [25].

Tekrarlayan sinir ağları (RNN), fuzz testleri için uygulanan en yaygın derin öğrenme yöntemidir. Özellikle, RNN'nin bir çeşidi olan Uzun-Kısa süreli bellek (LSTM) [24] ağları, girdi üretimi kullanımı için incelenmiştir. Yan ve arkadaşları hatadan yararlanılabilirliği belirlemek ve güvenilirliği artırmak için Bayes yöntemlerini ve sömürü tespit etme aracını kullanmıştır.

Potansiyel olarak savunmasız işlevleri ve konumları (yani savunmasız adresler) belirlemek için önceden eğitilmiş bir derin öğrenme tahmin modeli kullanılmıştır. Tam olarak, dikkat kelimelerini tanımlamak için Çift Yönlü-LSTM (BiLSTM) kullanılmakta ve güvenlik açıkları fonksiyonlardaki bu dikkat kelimeleriyle ilişkilendirilmektedir. Ve sonra, tahmin edilen konumlara ulaşma eğiliminde olan girdiler üretilerek potansiyel güvenlik açıklarını ortadan kaldırmak için doğrudan fuzz tekniği kullanılmaktadır [34].

Derleyiciler için test senaryosu oluşturma konusunda da makine öğrenmesi açısından bazı araştırmalar bulunmaktadır. Genellikle derleyicilerin girdileri oldukça yapılandırılmıştır. DeepSmith [54] ve DeepFuzz [55], sırasıyla, gerçek ortam kodunun gövdesinden programlama dilinin sözdizimini ve semantiğini öğrenmek için seq2seq ve LSTM modellerini kullanır, böylece programlama dilinin biçimine uyan çok sayıda test kodu oluşturur. JavaScript (JS) motoru için Lee ve arkadaşları [56], bir JS test senaryosunun hiyerarşik yapılarını ve bu yapılar arasındaki ilişkileri bir dizi parçaya modellemek için yeni bir algoritma önermiştir. Bu ilişki, daha yüksek kaliteli test senaryoları oluşturmak için orijinal alt ağacın yerini alacak yeni bir AST alt ağacı oluşturmak için kullanılmıştır.

Bazı çalışmalar uygunluk fonksiyonunu iyileştirerek test durumlarını filtrelemiştir. Sun ve arkadaşları [57], programcılar tarafından geliştirilen normal betiklerden oluşan bir senaryo kümesinden ortaklığı öğrenmek için Markov zinciri ve PCFG modelini birleştirmiş ve öğrenilen bilgileri, betiğin ortak bir betiğe sapmasını ölçerek betiğin alışılmadıklığını hesaplamak için kullanmıştır. Daha büyük sapmaya sahip betiklerin yorumlayıcılarda hata tetikleme olasılığı daha yüksek olabilir. Karamcheti ve arkadaşları [58], program girdilerini yürütme yörüngelerine eşlemiş ve yürütme yörüngesi dağılımının entropisini sıralamıştır. Bu, belirsizlik ne kadar yüksekse yeni

bir kod yolunu yürütme olasılığının o kadar yüksek olduğu varsayımına dayanır, bu nedenle en yüksek entropiye sahip girdi, bir sonraki girdi olarak seçilir.

Güçlendirme öğrenimine ek olarak, Karamcheti ve arkadaşları [59], tek bir programı fuzz testi sürecinde mutator dağılımını uyarlanabilir şekilde ayarlayabilen Thompson örnekleme optimizasyon yöntemi önermiştir. Her mutasyon operatörünün kod kapsamı üzerindeki etkisini öğrenerek hangi mutasyon operatörünün seçileceği belirlenmiştir. Raj ve arkadaşları [60] ise bilgisayarlı görü algoritmalarını test etmek için başka bir mutasyon tasarladılar, ilgisiz bir CNN'nin evrimsel filtre katmanından elde edilen bir deseni seçtiler ve bunu görüntü veya video karesindeki rastgele bir konuma eklediler.

Gömülü fuzz test araçlarında test girişi üretimini iyileştirmek için makine öğrenmesi tekniklerinin uygulanması çalışmasında gömülü cihazların fuzz testleri sırasında girdi oluşturma aşamasını artırmak için makine öğrenimi tabanlı bir yaklaşım önerilmiştir. Elde edilen sonuçlar, makine öğreniminin gömülü cihazların fuzz testlerine dahil edilebileceğini ancak bunun için önemli miktarda manuel çaba gerektiğini göstermiştir. Ayrıca genel kullanıma açık yöntemlerin mevcut durumunun henüz genel uygulamada kullanılabilecek kadar olgun olmadığı gösterilmiştir [61].

Büyük dil modelleri aracılığıyla fuzz testi tekniklerinin zorlukları üzerine yapılan çalışmada fuzz testi için büyük dil modellerini kullanan gelişmelerin sistematik bir genel bakışı sunulmuştur. Bu çalışma büyük dil modelleri, fuzz testi ve bu modellere dayalı olarak oluşturulan fuzz testi olmak üzere üç alanın kesişimini kapsayan ilk çalışmadır. Çalışma ayrıca, büyük dil modelleri tarafından oluşturulan fuzz test tekniklerinin gelecekte yaygın olarak dağıtılması ve uygulanması potansiyelini araştırarak, otomatik yazılım test uygulamalarını ilerletme konusundaki beklentileri vurgulamaktadır [62].

Büyük dil modelleriyle evrensel fuzz testleri çalışması, birçok farklı girdi dilini ve bu dillerin birçok farklı özelliğini hedefleyebilmesi anlamında evrensel olan ilk fuzz testi aracı olan Fuzz4All'ı sunmaktadır. Fuzz4All'ın ardındaki temel fikir, yaklaşımın pratik olarak ilgili herhangi bir dil için çeşitli ve gerçekçi girdiler üretmesini sağlayan bir girdi oluşturma ve mutasyon motoru olarak büyük dil modellerini (LLM'ler) kullanmaktır. Bu potansiyeli gerçekleştirmek amacıyla, fuzz testi için oldukça uygun olan LLM istemleri oluşturan yeni bir otomatik istem tekniği sunulmuştur [63].

LLM odaklı sentez ile verimli giriş üretimi fuzz testi aracı uzayı çalışması LLM tarafından yönlendirilen fuzz testi uzayı üzerinden sentez yoluyla test edilen bir sisteme göre uyarlanmış nesil tabanlı test araçlarını otomatik olarak sentezleyen yeni bir yaklaşım olan ELFUZZ'u önermektedir. Yüksek bir seviyede, minimal başlangıç fuzz araçlarıyla başlar ve sentezi, kapsam rehberliğiyle tamamen otomatik LLM tarafından yönlendirilen evrimle ilerletir. Önceki tekniklerle karşılaştırıldığında, ELFUZZ 1) gerçek dünya boyutlarındaki test edilen sistemlere sorunsuz bir şekilde ölçeklenebilir ve 2) ilginç dilbilgisi yapılarını ve anlamsal kısıtlamaları insan tarafından anlaşılabilir bir şekilde yakalayan verimli fuzz test araçlarını sentezleyebilir [64].

Fuzz testi saldırısı tespiti ile ağ güvenliğini iyileştirme çalışması model performansının hesaplama verimliliği ve yorumlanabilirlik ile dengelenmesinin önemini vurgulamaktadır. Gelecekteki çalışmalar, sınıf dengesizliğini ele almaya, gerçek zamanlı tespit için modelleri optimize etmeye, harici tehdit istihbaratını entegre etmeye ve model uyarlanabilirliğini geliştirmek için transfer ve sürekli öğrenme tekniklerini keşfetmeye odaklanmalıdır [65].

4. ÇALIŞMA BULGULARI

Tez çalışmamızın ana kapsamı, farklı seviyedeki mutasyon stratejileri açısından fuzz testi etkinliğinin değerlendirilmesidir. Fuzz testlerinde etkinliği değerlendirmek için kullanılan ölçütler kod kapsamı, benzersiz kod yolu, benzersiz çökme veya hata, geçiş oranı vb.'dir. Benzersiz çökme veya hata sayısı, fuzz testi sırasında tespit edilen benzersiz çökme ve hataların sayısıdır ve uygun bir değerlendirme ölçütü olarak kabul edilir. Benzersiz hata sayısı açısından, bir örnek vermek gerekirse, SmartSeed benzersiz hataları bulmak için diğer tohum stratejileriyle karşılaştırılır ve diğer tohum stratejilerinden daha iyi performans gösterdiği görülür. Benzersiz kod yolu, fuzz testi sırasında tetiklenen kod yolu sayısı anlamına gelir. Buna iyi bir örnek olarak, SmartSeed diğer tohum stratejilerine kıyasla daha fazla benzersiz yol tetikleyebilir. SmartSeed+AFL, mevcut stratejilerden %30,7 daha fazla benzersiz yol keşfeder. Geçiş oranı, test sırasında programın sözdizimi kontrolünden geçebilen test senaryolarının oranını gösterir. Skyfireda, bağlamdan bağımsız dilbilgisi tabanlı stratejiyle üretilen XML girdilerinin %34'ü anlamsal denetimi geçebilmiştir, Skyfire tarafından üretilen XSL ve XML girdilerinin ise sırasıyla %85'i ve %63'ü anlamsal denetimi geçip yürütme aşamasına ulaşabilmiştir. Kod kapsamı, temel blok, fonksiyon, satır, talimat ve dal kapsamından oluşan fuzz testi performansını değerlendirmek için en yaygın kullanılan ölçütlerden biridir. Wang ve arkadaşları, üç aylık bir taramadan toplanan örnekleri ve Skyfire tarafından üretilen iyi yapılandırılmış girdileri, açık kaynaklı XSL ve XML motorlarını test etmek için AFL tohumları olarak kullandılar.

Bizim tez çalışmamızda benzersiz hata sayısı, benzersiz kod yolu (kod kapsamı olarak da bilinir), hata tespit performansı değerlendirme ölçütlerini seçtik. Benzersiz hata sayısı, aynı hatayı tekrar eden hata örneklerinden ziyade, farklı nedenlere bağlı olarak ortaya çıkan özgün hata sayısıdır. Ne kadar fazla benzersiz hata bulunursa, fuzz testi o kadar verimli sayılmaktadır. Kod kapsamı veya bir diğer ifade ile benzersiz test senaryosu fuzz testleri sırasında uygulamanın farklı kontrol yollarını (örneğin if-else blokları) tetikleyen senaryoların oluşturduğu benzersiz yollardır. Kod kapsamı değerlendirmesindeki amaç uygulamanın mümkün olduğu kadar fazla alanının test

edilmesini sağlamak ve neticede hataların tespit edilme olasılığını artırmaktır. Bu yüzden kod kapsama değerinin yüksek olması genel itibarıyla avantajlı bir durum olarak değerlendirilmektedir. Buradan yola çıkarak benzersiz test senaryosunun bir diğer açıklaması da fuzz test araçları tarafından üretilen ve uygulamanın yeni yollarını tetikleyen, önceki senaryolardan farklı, yeni ve etkili test girdileridir. Bu yüzden benzersiz test senaryoları genel itibarıyla kod kapsamını genişleten girdilerdir.

Fuzz testi değerlendirme metrikleri açısından bakıldığında diğer çalışmalarda, hata sayısı, kod kapsamı, değerlendirme ölçütleri olarak seçilmiştir, bunlardan birkaçı geçme oranını içermektedir. Hata sayısı ve kapsam ölçütlerine ek olarak hata tespit performansını da dahil eden tek çalışma, bizim çalışmamızdır. Hata tespit performansı, belirli bir zaman diliminde tespit edilen hata sayısıdır. Bu metriğin kullanım amaçları; örneğin iki fuzz test mutasyon stratejisi karşılaştırılırken hangisinin aynı sürede daha fazla hata bulduğu, hangi test aracının daha kısa sürede daha çok hata bulduğu gibi kıyaslama ve analizlerin yapılmasına olanak sağlamaktadır. Öte yandan, literatür araştırmamızdaki çalışmaların çoğu, fuzz testlerini XML, JavaScript motorları gibi iyi biçimlendirilmiş girdiler alan hedef programlar üzerinde gerçekleştirmektedir. Birkaçı PDF ve ikili dosya formatları alan programlar için fuzz testi gerçekleştirirken, bizim çalışmamız yalnızca XML motorlarına/ayırıştırıcılarına odaklanmıştır.

Önceki çalışmalardan biri olan Deng. et. al'de görülebileceği gibi bayt düzeyinde ve ağaç düzeyinde mutasyonlar kullanan fuzz testi yaklaşımları arasında karşılaştırmalar yapılmaktadır. Ve değerlendirmelerini AFL ve Superion üzerinde gerçekleştirirler. Yeni bir ağaç düzeyinde mutasyon geliştirirken fuzz testinin kapsam açısından etkinliğini ölçmektedirler. Bu tez çalışmamızda, iki farklı mutasyon stratejisi arasında önemli ölçüde değişen hata bulma yeteneklerini incelemekteyiz. Test edilecek sistemler özellikle bu çalışma için XML dosyaları olmak üzere, iyi yapılandırılmış girdi dosyaları alan programlar olarak seçilmiştir. Doğası gereği iyi yapılandırılmış XML dosyaları AST'lere dönüştürülmek ve bunlar üzerinde ağaç düzeyinde mutasyonlar yapmak için daha uygundur. Fakat ağaç düzeyinde mutasyona dayalı fuzz testi yaklaşımlarının daha fazla hata tespit etme açısından iyi yapılandırılmış girdi dosyalarında iyi çalışıp çalışmadığı hala açık bir araştırma sorusudur. Bu nedenle iki farklı mutasyona dayalı fuzz testi tekniğinin hata tespit etkinliğini doğrudan değerlendireceğiz. Yani bayt düzeyinde ve ağaç düzeyinde mutasyon stratejisinin XML ayırıştırıcıları için daha fazla hata bulup bulamayacağını inceleyeceğiz. Ayrıca

her bir fuzz yürütmesindeki hata türleri hakkında bir araştırma yapacağız. Buradaki amaç, özellikle mutasyon yaklaşımlarının her biri tarafından hangi tür hataların bulunduğunu, bunun normal bir program çökmesi mi yoksa bir sistem güvenlik açığı mı olduğunu belirlemektir.

Tez çalışmamızın kapsamında ele aldığımız diğer önemli başlıkta ise akıllı tohumların her iki mutasyon tabanlı fuzz yaklaşımına olan etkilerini, akıllıca oluşturulan XML tohumlarının her bir fuzz türünün daha fazla sayıda hata bulmasına ne kadar yardımcı olduğunu bulmayı hedeflemekteyiz. Dilbilgisi farkında tohumları oluşturma ve genel tohumlarla karşılaştırma yapma zorluğuna yaklaşmak için, dilbilgisi ve anlambilim kuralları bilgisini otomatik olarak çıkarmak için çok sayıda örnekten yararlanan Skyfire metodolojisinden yararlanmaktayız. Skyfire yöntemi, iyi derece yapılandırılmış girdileri işleyen programlar için iyi dağıtılmış tohum girdileri oluşturmak için bilgiyi kullanır. Bu şekilde, yöntem mevcut dilbilgisi tabanlı fuzz test yaklaşımlarından bile daha ileri gider. Bunu sağlamanın yanı sıra fuzz test keşfini semantik kontrol aşamasının ötesine taşıyarak herhangi bir manuel çalışma veya spesifikasyon olmadan derin hataları bulmak için uygulama yürütme aşamasına ulaşır. Skyfireda fuzz testinin etkililiğini değerlendirmek için, yalnızca AFL fuzz test aracını kullanırlar ve hedef programlar olarak XML, XSL ve JavaScript motorlarını alırlar. Tohum oluşturma metodolojisini Skyfiredan alan tez çalışmamızda, testlerimizi yalnızca AFLde değil aynı zamanda Superionda da gerçekleştirmekteyiz. Çalışmamızda, özellikle dilbilgisi farkında algoritma yoluyla oluşturulan XML tohumlarının kullanımını hedefliyoruz ve etkinliğini genele açık XML dosyalarıyla karşılaştırıyoruz. Bunu başarmak için, öncelikle bunları XML ayrıştırıcıları için fuzz test sürecine veriyoruz ve ardından fuzz testinin hataları ne kadar bulabildiği konusunda etkinliğini ölçüyoruz. Bunu gerçekleştirmek ve oluşturulan XML dosyalarının fuzz testi etkililiğini ölçmek için, öncelikle genele açık XML dosyaları test için XML ayrıştırıcılarına verilmiş ve sonuçlar kaydedilmiştir. Daha sonra aynı prosedür, dilbilgisi öğrenme algoritmasına dayalı olarak oluşturulan XML tohum dosyaları için uygulanmıştır.

Dolayısıyla tez çalışmamızda kullanılan iki temel mutasyon stratejisi olan bayt düzeyi ve ağaç düzeyi mutasyonlar, test edilen yazılım sistemlerinde farklı seviyelerde davranış tetiklemekte ve bu sebeple hata tespitinde belirgin farklılıklara yol açmaktadır. Bayt düzeyindeki mutasyonlar, veri yapısının içeriğini doğrudan ve

yapısal fark gözetmeden deęiřtirirken, ağaç düzeyindeki mutasyonlar veri yapısının sentaktik ve yapısal bütünlüğünü daha çok dikkate alarak deęiřiklik yapmaktadır. Özellikle XML gibi hiyerarşik ve iyi biçimlendirilmiş veri formatlarında, yapısal bütünlük gözetilerek yapılan mutasyonların daha derin ve semantik açıdan anlamlı hataları ortaya çıkarma potansiyeli daha yüksektir. Bu durumun, yalnızca daha fazla hata bulma deęil, aynı zamanda hata türlerinin çeřitlilięi açısından da zenginlik oluřturması beklenmektedir.

Bununla birlikte, ağaç düzeyindeki mutasyon stratejilerinin etkinlięi yalnızca yapısal geçerlilikle sınırlı deęildir; aynı zamanda dilbilgisel ve anlamsal doęruluk da önemli olmaktadır. Bu noktada Skyfire tarafından önerilen dilbilgisi öğrenme temelli metodoloji önemli bir avantaj getirmektedir. Skyfire, büyük miktarda iyi yapılandırılmış örnekten otomatik olarak dilbilgisine dayalı kurallar çıkararak, bu kurallar doęrultusunda yeni test girdileri üretir. Tez çalışmamızda bu metodolojiyi kullanarak oluřturduğumuz XML tohumlarının, hem bayt düzeyinde hem de ağaç düzeyinde yürütölen fuzz test süreçlerinde daha yüksek hata tespit oranlarına neden olduęu gözlemlenmiştir. Özellikle anlamsal denetimden geçebilen girdiler sayesinde program yürütmesinin daha ileri aşamalarına ulařılması, yalnızca yüzeysel deęil, aynı zamanda derin yapısal hataların da ortaya çıkarılmasına olanak tanımıştır. Bu, klasik fuzz testi tekniklerinin ilerisine geçen bir test kapsamı sağlamaktadır.

Uygulamalı test aşamasında üç farklı XML ayrıştırıcısı (Xerces, Libxml2 ve Expat) üzerinde yapılan çalışmalarda, tohum stratejisinin etkinlięi, ölçölebilir metriklerle deęerlendirilmiştir. Benzersiz hata sayısı, kod kapsamı ve hata tespit performansı başlıca karşılařtırma ölçütleri olarak belirlenmiştir. Yapılan analizlerde, Skyfire tabanlı akıllı tohumlarla yapılan testlerin hem daha fazla benzersiz hata hem de daha yüksek kod kapsamı ortaya koyduęu saptanmıştır. Özellikle Superion gibi dilbilgisi farkında çalışan araçlarla entegre çalışıldığında, klasik tohumlara kıyasla %30'un üzerinde daha fazla benzersiz kod yolu tetiklenmiştir. Bu bağlamda, dilbilgisel farkındalıkla üretilmiş tohum dosyalarının yalnızca geçiř oranını artırmakla kalmayıp, aynı zamanda yazılım sisteminin daha karmařık davranışlarını da açığa çıkardıęı görölmüřtür.

Literatürdeki önceki çalışmalarda, çoęunlukla ya kod kapsamı ya da hata sayısı gibi tekil metriklerle odaklanılmışken, bizim çalışmamız bu iki ölçöte ek olarak zaman-temelli hata tespit performansı metrięini de deęerlendirmiştir. Bu yaklařım, yalnızca

toplam hata sayısını değil, aynı zamanda belirli bir zaman dilimi içerisinde ne kadar etkili hata tespiti yapılabildiğini de ortaya koymaktadır. Örneğin, iki farklı tohum kümesiyle yapılan 12 saatlik test sürecinde, Skyfire tabanlı tohumların hata tespit eğrisi, klasik tohumlara kıyasla daha hızlı yükselmiş ve daha erken safhalarda daha fazla hata tespiti sağlamıştır. Bu bulgu, özellikle sınırlı test süresine sahip güvenlik testlerinde, etkin tohum stratejisi seçiminin doğrudan sonuçları etkileyebileceğini göstermektedir.

Çalışmamız yalnızca sayısal metriklere dayalı bir analizle sınırlı kalmamış, aynı zamanda hata türlerinin niteliksel sınıflandırılmasını da yapmıştır. Elde edilen her bir hata örneği, istisna tipi, program davranışı ve potansiyel güvenlik riski açısından analiz edilmiş ve böylece mutasyon stratejisinin yalnızca hata sayısını değil, aynı zamanda hata kalitesini nasıl etkilediği de değerlendirilmiştir. Bu analiz neticesinde, bayt düzeyi mutasyonların genellikle sözdizimi hatalarına neden olduğu, buna karşın ağaç düzeyi mutasyonların mantıksal ve yürütme esnasında ortaya çıkan daha kritik hatalara yol açtığı belirlenmiştir. Bu bulgular, özellikle güvenlik odaklı yazılım testleri açısından ağaç düzeyi mutasyonların öncelikli değerlendirme stratejisi olması gerektiğini göstermektedir.

Tez çalışmamız kapsamında, yalnızca yapısal hata ve kapsam ölçütleriyle sınırlı kalmayarak, yazılım sistemlerinde ortaya çıkabilecek kritik güvenlik zafiyetlerinden biri olan XML Injection saldırılarına da odaklanılmıştır. XML Injection, kullanıcı girdilerinin kontrolsüz biçimde XML verileri içerisine dahil edilmesiyle, hedef sistemin beklenmeyen şekilde davranmasına yol açan önemli bir güvenlik açığıdır. Bu tür saldırılarla, saldırganlar XML belgesinin yapısını bozabilir, istenmeyen sorgular çalıştırabilir veya sistemden yetkisiz veri elde edebilir. Özellikle XML tabanlı iletişimin yaygın olduğu web servisleri ve yapılandırma sistemlerinde bu tür zafiyetler sistemin bütünlüğünü, gizliliğini ve güvenilirliğini tehdit eder.

Bu tür zafiyetleri keşfetmek ve analiz etmek amacıyla, Skyfire temelli öğrenme algoritmasından yararlanarak XML Injection karakteristikleri taşıyan tohum girdileri üretilmiştir. Skyfire'nin dilbilgisel ve anlamsal kuralları otomatik olarak öğrenme yeteneği, yalnızca iyi yapılandırılmış girdiler oluşturmakla kalmamış, aynı zamanda bu yapılar içerisine zafiyet oluşturabilecek yapıları da sistematik bir şekilde dahil etmemizi sağlamıştır. Özellikle `<!ENTITY>`, `<!DOCTYPE>`, karakter örnekleri

gibi potansiyel olarak istismar edilebilir XML bileşenleri, Skyfire tarafından oluşturulan örneklerde kontrollü biçimde yer almış ve test süreçlerinde kullanılmıştır.

Yürütülen çalışmalarda, özel olarak üretilmiş XML injection içeren girdiler, hem bayt düzeyinde hem de ağaç düzeyinde mutasyonlara maruz bırakılarak farklı test senaryoları oluşturulmuştur. Sonrasında, bu senaryolar Xerces, Libxml2 ve Expat gibi XML ayrıştırıcıları üzerinde fuzz test araçlarıyla çalıştırılmıştır. Bu testler sonucunda, Skyfire temelli öğrenilmiş tohumlardan türetilen enjeksiyon girdilerinin, klasik veya rastgele oluşturulmuş girdilere kıyasla daha yüksek oranda hata ve istisna tetiklediği görülmüştür. Bu durum, özellikle enjeksiyon türü güvenlik açıklarının daha sistematik ve etkili şekilde test edilmesini sağlamıştır.

Elde edilen sonuçlar göstermektedir ki, dilbilgisel farkındalıkla oluşturulmuş ve enjeksiyon semantiği içeren XML girdileri, yalnızca sistemin kod kapsamını artırmakla kalmayıp, aynı zamanda daha kritik güvenlik açıklarını ortaya çıkarabilmektedir. Özellikle Superion gibi yapı farkındalığı yüksek fuzz test araçlarıyla birlikte kullanıldığında, bu tür girdiler daha derin yürütme yollarını tetikleyebilmekte ve sistemin maruz kalabileceği saldırı yüzeyini ortaya koyabilmektedir. Bu bağlamda, çalışmamız, Skyfire'nin öğrenme algoritmasıyla sentezlenen XML enjeksiyon içeren tohum dosyalarının yalnızca yapısal hata tespitinde değil, aynı zamanda güvenlik açıklarının ortaya çıkarılmasında da kullanılabilecek bir araç olduğunu göstermektedir.

Araştırma yaklaşımımıza dayanarak, sorunu ve çalışmanın ihtiyaçlarını belirledikten sonra, sorunu bir vaka çalışması olarak formüle ettik ve ardından tasarım yönlerini tartıştık. Araştırma yaklaşımı açısından, çalışmamızın doğası ve bağlamı göz önünde bulundurulduğunda, vaka çalışmasını "in-vivo araştırma" [66] olarak adlandırılan gözlemsel, kontrolsüz bir bağlamda yürütülecek şekilde tasarladık.

4.1. Hedefler ve Araştırma Soruları

Tez çalışmamızdaki temel hedef, farklı mutasyon stratejileriyle gri kutu fuzz testi yaklaşımlarını karşılaştırmak ve değerlendirmektir. Bir diğer hedefimiz, öğrenme tabanlı üretilen tohumların her bir mutasyon tabanlı fuzz testi yaklaşımının etkinliğine olan etkisini analiz etmektir. GQM'in hedef şablonunu [67] kullanarak, projenin hedefini şu şekilde formüle ettik: GQM'in yaklaşımına dayanarak, bu çalışma için tanımladığımız hedefler şunlardır;

- İki farklı mutasyon tabanlı fuzz testi yaklaşımı için hata tespiti etkinliğinin değerlendirilmesi
- İki farklı mutasyon tabanlı fuzz testi yaklaşımı için hata tespiti performansının değerlendirilmesi
- İki farklı mutasyon tabanlı fuzz testi yaklaşımı için tespit edilen hata türlerinin değerlendirilmesi
- İki farklı mutasyon tabanlı fuzz testi yaklaşımı için akıllı tohumların fuzz testi etkinliğine olan etkilerinin bulunması

Araştırma sorularımız ise şu şekilde formüle edilmiştir;

- RQ1-Bayt düzeyindeki mutasyon stratejisi ve ağaç düzeyindeki mutasyon stratejisi, test etkinliği açısından nasıl karşılaştırılır?
- RQ2-PCSG öğrenme tabanlı üretilen tohumlar, genel geçer tohumlara kıyasla hataları bulmada ne kadar etkilidir?

4.2. Çalışmanın Temel Öğeleri

İki farklı fuzz testi mutasyon stratejisi arasında değerlendirme ve analiz yapmak için, bayt düzeyindeki mutasyonlar için bilinen fuzz testi aracı olan AFL kullanılmıştır. Ağaç düzeyindeki mutasyon stratejileri ve sonuçları için, AFLin uzantısı olan Superion adlı fuzz test aracı kullanılmıştır.

Öğrenmeye dayalı tohum dosyası oluşturma yönteminde, ANTLR, belirli bir dilin gramerinden sözcük çözücü ve ayrıştırıcı oluşturmak için kullanılır. Yaklaşımın etkinliğini ve genelliğini değerlendirmek için, tohum girdisi oluşturma için hedef dil olarak XML seçilmiştir. Seçilen fuzz testi araçları, bu tez çalışmasında değerlendirme ve deneyler için XML kullanmasına rağmen, belirli bir dili hedeflemez. XML, bilgileri yapılandırmak, depolamak ve iletmek için tasarlanmış bir işaretleme dilidir. SCSA (basitleştirilmiş sstemci sunucu mimarisi) uygulamaları arasında veri iletmek için en yaygın kullanılan dildir. Bu çalışmada yaygın olarak kullanılan Libxml2, Apache Xerces ve Expat olmak üzere gerçek dünyada çalışan XML ayrıştırıcıları seçilmiştir. Oluşturulan örnek dosyalar ve genel dosyalar, fuzz test edilebilmeleri için test edilen bir sistem altında yürütülmelidir. Bu amaçla Linux işletim sistemi altında çalışan

Libxml2, Apache Xerces, Expat isimli üç adet açık kaynaklı XML ayrıştırıcısı seçilmiştir.

Bu tez kapsamında gerçekleştirilen fuzz testleri, kontrollü ve tekrarlanabilir bir deneysel ortamda yürütülmüştür. Testlerin yürütüldüğü sistem, Ubuntu 18.04.6 LTS (64-bit) işletim sistemi üzerinde çalışmakta olup, sistem donanımı Intel(R) Core(TM) i3-7020U @ 2.30GHz işlemci, 12 GB DDR4 RAM ve 119 GB SSD depolama kapasitesine sahiptir. Tüm fuzz test araçları ve hedef XML ayrıştırıcıları bu sistem üzerinde derlenmiş ve yapılandırılmıştır.

Ayrıca, test süreçlerinin izlenebilirliğini sağlamak amacıyla loglama ve hata kayıtları sistematik şekilde toplanmış, ortaya çıkan hatalar analiz edilmek üzere ayrıntılı olarak kaydedilmiştir. Deneyler süresince sistemde başka yoğun işlem çalıştırılmamış olup, test sonuçlarının tutarlılığı sağlanmıştır.

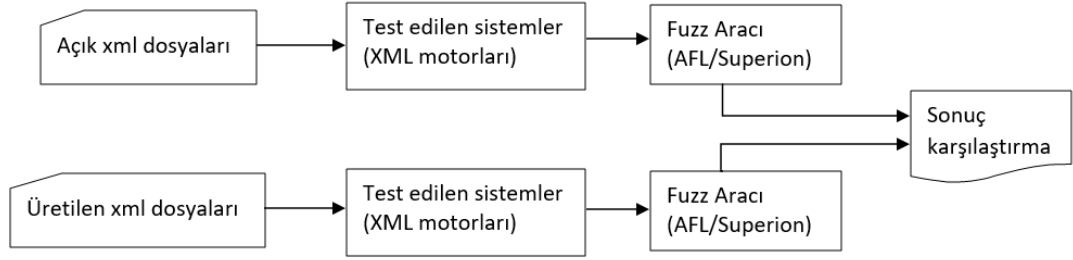
4.3. Kurulum

4.3.1. Test senaryosu hazırlığı

XML ayrıştırıcıları için fuzz testi yapmak amacıyla, testleri başlatacak tohum dosyaları iki şekilde hazırlanmıştır. İlk etapta, yaklaşık 100 tohumluk XML dosyalarının bir kısmı, farklı boyutlardaki genele açık web sitelerinden indirilmiştir. İkinci olarak, daha önce indirilen bu XML tohumlarının AST'lerinde ağaç düzeyinde mutasyonlar yapılarak PCSG tabanlı öğrenme yöntemi ile 300 adet yeni XML tohum dosyası üretilmiştir.

4.3.2. Fuzz proses akışı

Fuzz testleri sistemi iş akışının genel mimarisi, genel olarak seçilen ve öğrenmeye dayalı ilk XML tohumlarıyla ilgilenir ve bunları, bizim durumumuzda bayt düzeyinde mutasyonu destekleyen AFL ve ağaç düzeyinde mutasyonlar gerçekleştiren Superion olan iyi bilinen gri kutu fuzz test araçlarına verir. Ana sistem akış mimari diyagramı Şekil 4.1'de görülebilir.



Şekil 4.1. Fuzz testleri için ana mimari akış.

4.3.3. AFL (American Fuzzy Lop)

AFL test aracı, çok basit ama çok sağlam bir enstrümantasyon rehberliğinde genetik algoritma ile eşleştirilmiş bir kaba kuvvet ve gri kutu fuzz test aracıdır. Program kontrol akışındaki ince, yerel ölçekli değişiklikleri basitçe yakalamak için değiştirilmiş bir kenar kapsamı biçimi kullanır. Özellikle AFL, fuzz edilmiş programda yeni dahili durumları veya yolları tetikleyebilen ilginç test girdilerini keşfetmek için yeni bir derleme zamanı enstrümantasyonu türü kullanır ve bu da fuzz edilmiş programın kapsamını önemli ölçüde iyileştirir. AFL, giriş dosyalarıyla çalışırken bir kuyruk mekanizması kullanır. AFL ve AFL tabanlı test araçları, ilk tohum dosyalarını alır ve bunları kuyruğa koyar. Dosyalar veya sözde test senaryoları, test edilen sisteme karşı yürütülür ve algılanan hatalar saklanır. Sonra test senaryoları, bazı seçili mutasyon stratejilerine göre mutasyona uğratılır. Program içinde yeni durum geçişleri veya yollar üreten mutasyona uğramış test senaryoları, giriş kuyruğuna eklenir ve gelecekteki fuzz testi turları için bir başlangıç noktası olarak kullanılır.

```

american fuzzy lop ++4.08a {default} (./sax2print_cov) [fast]
- process timing
  run time : 0 days, 12 hrs, 41 min, 17 sec
  last new find : 0 days, 0 hrs, 2 min, 35 sec
  last saved crash : 0 days, 0 hrs, 39 min, 2 sec
  last saved hang : none seen yet
- cycle progress
  now processing : 944.7 (50.2%)
  runs timed out : 0 (0.00%)
- stage progress
  now trying : havoc
  stage execs : 46/69 (66.67%)
  total execs : 1.06M
  exec speed : 16.81/sec (zzzz...)
- fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 903/362k, 715/537k
  py/custom/rq : unused, unused, unused
  trim/eff : 13.87%/142k, disabled
  strategy: explore
- overall results
  cycles done : 3
  corpus count : 1881
  saved crashes : 37
  saved hangs : 0
- map coverage
  map density : 0.13% / 0.15%
  count coverage : 2.00 bits/tuple
- findings in depth
  favored items : 254 (13.50%)
  new edges on : 441 (23.44%)
  total crashes : 192 (37 saved)
  total tmouts : 1 (0 saved)
- item geometry
  levels : 17
  pending : 673
  pend fav : 0
  own finds : 1581
  imported : 0
  stability : 100.00%
[cpu000:200%]
state: in progress
  
```

Şekil 4.2. AFL test sonuçları ekranı

AFL aracı çalıştırıldığı zaman elde edilen örnek sonuç ekranından bir kesit Şekil 4.2’de görülmektedir. AFL çalıştırma ekranındaki bazı istatistiksel değerlerden bahsetmek gerekirse,

- “last new find” değeri, en son bulunan yeni bir yürütme yolunun veya yeni bir test senaryosunun ne zaman keşfedildiğini gösterir. Bu süre uzuyorsa yeni bulgular azalıyor demektir.
- “last saved crash” değeri, en son çökmenin ne zaman yaşandığını gösterir. Eğer uzun süredir çökme yoksa, testin daha az etkili hale geldiği anlamına gelir.
- “last saved hang” değeri, en son donma durumunun ne zaman kaydedildiğini belirtir. Donma, belirli bir test senaryosunun zaman sınırını aşarak programın yanıt vermemesi durumudur.
- “cycles done” değeri, girdi havuzundaki tüm test senaryolarının kaç kez tamamen döngüsel olarak test edildiğini gösterir. Her bir döngü, test edilen girdilerin bir tur tamamlamasıdır. Yüksek sayı, testin uzun süredir sürdüğünü ifade etmektedir.
- “corpus count” değeri, anlık kullanılan test senaryo sayısını belirtir. AFL zamanla bu sayıyı artırır; yeni yol bulan test senaryoları havuza eklenir.
- “saved crashes” değeri, anlık tespit edilerek kaydedilmiş benzersiz hata sayısıdır. AFL aynı hata yolunu farklı testlerle üretse bile sadece farklı olanları kaydeder.
- “saved hangs” değeri, kaydedilen benzersiz donma sayısıdır. Programın belirli test senaryolarında yanıt vermeyi durdurduğu durumların sayısıdır.

4.3.4. Superion

Superion, dilbilgisi odaklı mutasyona dayalı son teknoloji bir fuzz testi aracıdır ve hem kararlı sözdizimsel doğruluk hem de fuzz testi verimliliği elde edebilir. AFLdeki mutasyonlar dilbilgisi körüdür ve bayt düzeyinde çalışır, oysa Superiondaki mutasyonlar dilbilgisi farkındadır ve ağaç düzeyinde işlev görür. Superion, ayrıştırılan girdilerin AST'lerini kullanarak ağaç düzeyinde test girdilerini kırmak için dilbilgisi farkında bir kırpma stratejisi kullanır. Superion, iyi derece yapılandırılmış girdi dosyaları için oldukça uygundur, yani fuzz testi için geçerli girdiler üretebilir ve daha fazla sayıda hata tespit etme olasılığı yüksektir.

4.3.5. Akıllı tohum üretimi

Bu tez çalışmasının bir amacı, genel tohumlarla fuzz testi ile PCSG öğrenme tabanlı tohumlarla fuzz testi arasında karşılaştırma ve analiz yapmaktır. Öğrenme tabanlı tohumları üretmek için, tohumları PCSG metodolojisini kullanarak üreten Skyfire yöntemini kullandık. PCSG, girdi olarak bir gövde ve bir dilbilgisi alır ve tohum girişlerini iki adımda üretir. Bunu yaparken izlenen adımlar şunlardır;

- Öncelikle, toplanan örnekleri dilbilgisi sözcük çözücü ve ayrıştırıcı mekanizmalarına dayalı soyut sözdizimi ağaçlarına ayrıştırır. Daha sonra hem sözdizimi özelliklerini hem de anlamsal kuralları belirten bir PCSG öğrenir. PCSG bir üretim kuralları havuzu içerir.
- PCSG'de öğrenilen üretim kuralları, iyi dağıtılmış girdilerin üretilmesi sırasında kullanılır.

Yöntem, öğrenilen dilbilgisine dayalı bir üretim kuralını yinelemeli olarak seçip uygulayarak tohum girdileri üretir. Girişlerin oluşturulması sırasında, sonuç dizisinde terminal olmayan sembol kalmayana kadar bağlamı karşılamak için terminal olmayan bir sembol üzerinde işlemler gerçekleştirilir.

- Son adımda yöntem, AST'deki bir yaprak düzeyindeki düğümü, oluşturma kurallarına göre aynı türdeki düğümlerle rastgele değiştirerek kalan tohum girişlerini değiştirir. Bu süreçte dilbilgisi yapılarını korurken semantik değişikliklerdeki çeşitliliği daha fazla korumayı amaçlar.

Tohum dosyası oluşturma prosedüründeki ilk adım, ANTLR yaklaşımını kullanarak seçilen dil olan XML için bir sözcük çözücü ve ayrıştırıcı oluşturmaktır. İkinci adım olarak, XML dili için oluşturulan sözcük çözücüler ve ayrıştırıcılar ve ayrıca örnek olarak verilen bazı XML dosyaları kullanılarak bir AST oluşturulur. Daha sonra, oluşturulan AST üzerinde gezinerek, XML dosya biçimi için gereken üretim kuralları oluşturulur. Son olarak da, XML tohum dosyaları önceki adımda oluşturulan üretim kurallarından yararlanılarak oluşturulur.

4.3.6. Olasılıksal bağlam duyarlı dilbilgisi

Tohum oluşturma iş akışında, sistem XML dilinin AST'sini ve bazı örnek XML gövde dosyalarını alır ve ardından PCSG'yi kullanarak tohum oluşturma işlemi sırasında kullanılan üretim kurallarını çıktı olarak verir. Yaklaşım, bir üretim kuralları havuzuna

sahip olan bir PCSG öğrenir. PCSG hem sözdizimi özelliklerini hem de anlamsal kuralları belirtmek için önerilirken, CFG yalnızca sözdizimi özelliklerini açıklar. Genel olarak ifade edilecek olursa, dilbilgisi, programlama dillerinin yapısını tanımlayan etkili ön bilgidir. Birçok karmaşık girdi, CFG'leri kullanarak tanımlanabilir. Esas olarak bir CFG, dörtlü $G = (N, T, R, S)$ 'dir. Burada

- N, bir cümledeki farklı türdeki ifadeleri temsil eden sonlu bir sonlu sembol kümesidir.
- T, bir cümlenin gerçek içeriğini oluşturan sonlu bir sonlu sembol kümesidir.
- R, üyeleri bir dilbilgisinin üretim kuralları olan sonlu bir kümedir.
- S başlangıç sembolüdür ve tüm cümleyi (veya programı) temsil etmek için kullanılır.

CFG çoğu karmaşık girdiyi üretebilse de, bağlamdan bağımsız yapısı nedeniyle CFG tarafından üretilen girdilerin bir ayrıştırıcının semantik kontrollerini geçmesi zordur.

- Gcs, CFG temelinde Bağlam Duyarlı Dilbilgisi'dir (CSG). Bileşen öğeleri CFG ile aynı şekilde tanımlanan (N, T, Rc, S) dizisidir, ancak Rc hariç. Rc'yi oluşturan üretim kuralları $[c]A \rightarrow a$ biçimindedir, burada [c], A'nın büyük büyük ebeveyninin türü, A'nın büyük ebeveyninin türü, A'nın ebeveyninin türü, A'nın ilk kardeşinin değeri veya değer boşsa A'nın ilk kardeşinin türü gibi önceden tanımlanmış bir bağlamdır.

q, her üretim kuralına bir olasılık atar ve belirli bir terminal olmayan sembol için, tüm bağlamlar için uygulanabilir üretim kurallarının olasılığının bire eşit olmasını sağlar [39].

4.3.7. Fuzz testi çalıştırmaları ve sonuçları

XML tohum dosyalarını seçme ve üretme fuzz testi sürecinin ana iş akışında, test ve değerlendirme amacıyla iki ana yürütme gerçekleştirilir. Öncelikle, rastgele seçilen genel XML dosyaları Libxml2, Apache Xerces ve Expat olarak bilinen Linux tabanlı XML ayrıştırıcılarına verilir. Daha sonra, bu XML ayrıştırıcıları için iyi bilinen gri kutu fuzz testi araçları AFL ve Superion aracılığıyla testler yürütülür. İkinci olarak, seçilen genel dosyalar öğrenmeye dayalı yöntemle verilir ve PCSG öğrenme algoritması tarafından yeni XML tohum dosyaları üretilir. Genel XML dosyalarının toplam sayısı yaklaşık 100 iken, öğrenmeye dayalı tohum dosyalarının sayısı genel

olanların üç katı olup 300'dür. Genel XML dosyalarının boyutu, bayt seviyelerinden yaklaşık 300 KB'a kadar değişmektedir. Öğrenmeye dayalı üretim algoritmasının içindeki mantık nedeniyle, tohum dosyalarının boyutu sabittir ve sürekli 1K'dır. Fuzz test yürütmeleri 36 saat boyunca gerçekleştirilir ve her 12 saatte bir, hata sayısı ve hata sınıfları için ölçümler kaydedilir.

Hem genel olarak seçilen XML dosyaları hem de öğrenmeye dayalı XML dosyaları için iki farklı gri kutu fuzz testi aracı kullanılmıştır. İlk test aracı, yeni gövde dosyaları oluştururken genetik algoritma kullanan AFL adı verilen en iyi bilinen fuzz testi araçlarından biridir. Özellikle bu test aracı, bit ve bayt düzeyinde çalışan mutasyon stratejisinden, bit-bayt çevirmeleri, baytların eklenmesi ve çıkarılması, var olan tam sayı değerlerini artırma veya azaltma gibi basit aritmetiklerden yararlanır. Kullanılan ikinci fuzz testi yaklaşımı Superion olarak bilinmektedir. XML ve JavaScript gibi iyi yapılandırılmış girdileri fuzz testi yapmak için oldukça uygun olan, dilbilgisi farkında kapsama kılavuzlu bir gri kutu fuzz testi yaklaşımıdır. Superion, öncelikle tohum dosyalarını AST'ye dönüştüren ve ardından bir test girişinin AST'sindeki her alt ağacı yinelemeli olarak kaldıran ve kapsam farklılıklarını gözlemleyen yeni bir mutasyon stratejisiyle gelmektedir. Bunu başarmak için, AFLin sözlük tabanlı mutasyon stratejisini, dilbilgisi farkında bir şekilde belirteçleri ekleyerek/üzerine yazarak geliştirir ve ardından bir test girişinin AST'sindeki bir alt ağacı, kendisinden veya kuyruktaki başka bir test girişinden gelen alt ağaçla değiştiren ağaç tabanlı bir mutasyon stratejisi önerir. Hem bayt hem de ağaç düzeyinde iki farklı mutasyon stratejisi için fuzz test yürütmeleri gerçekleştirerek, tespit edilen hata sayısı, hata tespit performansı ve hata türleri açısından aralarındaki farkı değerlendirme şansı elde edebiliriz.

Sonuçlardaki gürültüden kaçınmak için, bu tez çalışmasında iyi bilinen istatistiksel testlerden biri olan ANOVA'yı uyguladık. Bu istatistiksel test yöntemi, gruplar arasındaki varyansın istatistiksel olarak anlamlı olup olmadığını belirlemek için kullanılan parametrik bir istatistiksel testtir. ANOVA, üç veya daha fazla bağımsız grup arasındaki farkı değerlendirir. ANOVA, gruplar arasında anlamlı bir fark gösteriyorsa, hangi grupların birbirinden farklı olduğunu belirlemeye yardımcı olmak için daha fazla post hoc testi yapılabilir. ANOVA testini, genele açık test durumları için çalışan fuzz testlerine uyguladık.



5. ÇALIŞMA BULGULARI

En genel anlamda iki tür mutasyon stratejisi için fuzz testi araştırmasını yürüttük. Bayt düzeyindeki mutasyon stratejileri için AFL kullanırken, ağaç düzeyindeki mutasyon stratejileri için Superior kullandık. Aşağıdaki metriklerle ilgili değerlendirmemizi yaptık; hata sayısı, kod kapsamı, hata tespit performansı. Öte yandan, tespit edilen hata türlerini de analiz ettik. Daha sonra, akıllı oluşturulmuş tohumların fuzz testi sürecine olan etkilerini inceledik.

5.1. RQ1- Bayt Seviyesinde Mutasyon Stratejisi İle Ağaç Seviyesinde Mutasyon Stratejisi Test Etkinliği Açısından Nasıl Karşılaştırılır?

Fuzz testi sürecinin ilk ve ana aşamasında, hata sayısını değerlendirmek için farklı mutasyon stratejileri için fuzz testi yürütmelerinin analizini ve araştırmasını yürüttük. Fuzz test aracı AFL, bayt düzeyine dayalı mutasyon stratejisinin sonuçlarını göstermek için kullanılırken, Superior adlı araç ağaç düzeyindeki mutasyon stratejisiyle bilinir. Her iki test aracı için de aynı sayıda tohum seçilmesine rağmen, ağaç düzeyindeki mutasyon stratejisine sahip araç, özellikle Apache Xerces için fuzz testinin başlangıcında tohum dosyalarından bazılarını reddetmiş ve "hatayla sonuçlanır" uyarı mesajını vermiştir. Mevcut dilbilgisi tabanlı test araçlarının bir dizi dezavantajı vardır. Örneğin, Superior, ürettiği test senaryoları için doğru bir sözdizimini korumayı başaramayan bazı mutasyonlar gerçekleştirir. Bu muhtemelen, fuzz testi başlamadan önce 16 test senaryosunun başarısız olduğu iddiasının doğrulanmasıdır. Bu nedenle, testler istenen bir şekilde başlayamamıştır. Bundan ötürü, fuzz testi yürütmesini başlatmak için bu hataya neden olan tohumları kaldırmamız gerekmiştir. Ve bu dosyalar, test başlangıcında tespit edilen hata sayısı için sayılır. Ek olarak, tasarım gereği, ağaç tabanlı mutasyon stratejisi, AST alt ağaçlarını işleyen olaylara dayandığı için dilin dilbilgisine özen gösterir. Ancak yukarıda belirttiğimiz gibi, ayrıştırmanın başarısız olacağı birçok durum vardır. Bu, diğer mutasyonlar test girdisini halihazırda ağaç tabanlı stratejinin şu anda dilbilgisi kurallarına uymayacak şekilde davrandığı şekilde değiştirmişse gerçekleşir. Bulgularımız AFLin hem hata ortaya çıkarma

yeteneđi, hem kod kapsamı hem de test girdisi oluřturma aısından diđer test aracı Superiondan daha iyi performans gsterdiđini ispat etmektedir.

Tespit edilen hataların sayısına bakıldığında AFL, Apache Xerces iin Superiondan daha fazla hata bulabilir. Apache Xerces'in fuzz testi yapılması durumunda, Superionun benzersiz mutasyon stratejisi ile fuzz testlerinde herhangi bir hata tespit edilmemiřtir. Tm hatalar fuzz testi ncesinde bulunur ve testin bařlatılmasını engeller. Bunun nedeni, ađa dzeyindeki mutasyonların yapılandırılmıř bir řekilde gerekleřtirilmesi, yani alt ađaların birbirleri arasında basite deđiřtirilmesi veya deđiřtirilmesinin hayata geirilmesi olabilir. Dosyanın yapısal kısmı bozuk olmadıđından ayrıřtırıcının iinde hataya neden olacak bir durumla karřılařamaz. te yandan, AFLdeki mutasyonlar bayt seviyesinde daha katıdır ve tohum dosyalarında daha fazla yapısal deđiřiklik yaparak daha fazla hataya neden olabilir. Sonularımız, Libxml2 ve Expat XML ayrıřtırıcısı iin herhangi bir hatanın tespit edilmediđini gsterdi. Bu alıřmayı gerekleřtirmeden nce Libxml2 ayrıřtırıcısının, AFL dahil diđer fuzz testi aralarıyla aylarca kapsamlı bir řekilde fuzz testine tabi tutulduđunu ve bu nceki fuzz testi sırasında bulunan tm hataların, bu alıřma iin kullandıđımız XML ayrıřtırıcısı srmnde dzeltildiđi bilgisi mevcuttur.

Xerces XML ayrıřtırıcısı zerinde yapılan testlerde, hem AFL hem de Superion belirli sayıda hata tespiti gerekleřtirmiřtir. zellikle AFL, 12 saatlik test sresinde 24 farklı hata tespit etmiř, bu sayı 36 saatte 38'e ulařmıřtır. Bu artıř, AFLin zamanla farklı mutasyonlar reterek yeni hata durumlarını keřfedebilme potansiyelini gstermektedir. Superion ise 12 saat iinde 15 hata tespit etmiř ve bu sayı test sresi uzasa dahi deđiřmemiřtir. Bu durum, Superionun yapı farkındalıklı mutasyonlarının belirli bir eřiđe ulařtıktan sonra ek hata tespitinde sınırlı kaldıđını gstermektedir.

AFLin, yapı farkındalıđı olmaksızın mutasyonlar uygulamasına rađmen zamanla daha fazla hata bulabilmesi, sistematik olmayan mutasyonların bazı senaryolarda daha etkili olabileceđini ortaya koymaktadır. Ayrıca bu aynı zamanda test sresinin uzatılmasının AFL iin dođrudan fayda sađladıđını, Superion iin ise erken dođgunluđa ulařıldıđını gstermektedir.

Her iki test aracı da Libxml2 ve Expat zerinde hibir hata tespit edememiřtir. Bu durum, birka farklı řekilde yorumlanabilir:

Kod Temelli Güvenlik: Libxml2 ve Expat ayrıştırıcılarının yapısal olarak daha güvenli ve olgun kod tabanlarına sahip olması muhtemeldir. Bu kütüphaneler daha yaygın şekilde kullanıldıkları için, uzun süre boyunca test edilip pek çok hata zaten giderilmiş olabilir. Libxml2 ve Expat ayrıştırıcılarında girdilere karşı kullanılan savunma mekanizmalarının (örneğin input validation, hata toleransı) oldukça gelişmiş olması, bu sistemlerin rastgele mutasyonlara karşı daha dirençli olmasına yol açmaktadır.

Test Kapsamı Kısıtlılığı: Kullanılan tohum girdileri ve mutasyon stratejileri, bu ayrıştırıcılar üzerinde hata üretmeye yeterli çeşitliliği sağlamamış olabilir. Özellikle öğrenmesiz stratejilerin, semantik olarak geçerli ama hata oluşturabilecek girdiler üretme başarısı sınırlı kalmaktadır.

Test sonuçları, öğrenmesiz mutasyon stratejilerinin XML gibi yapısal veri formatlarında sınırlı etkiye sahip olduğunu göstermektedir. AFL gibi geleneksel araçlar zamanla daha fazla varyasyon üretebilse de, bu varyasyonların ne ölçüde geçerli XML formatlarına yakın olduğu tartışmaya açıktır. Superion ise yapı farkındalığı sayesinde erken aşamada anlamlı hatalar bulsa da, bu avantajını zaman içinde sürdürememiştir. Bu bağlamda, XML gibi biçimsel olarak katı yapı gereksinimi olan veri formatları üzerinde öğrenmeye dayalı ve yapı farkındalıklı yaklaşımların daha verimli olabileceği düşünülmektedir.

Fuzz testlerinin etkinliğini belirlemede kullanılan en temel ölçütlerden biri benzersiz test senaryolarının sayısıdır. Bu senaryolar, hedef uygulamanın farklı dallarını, koşullarını ve kod yollarını tetikleyerek potansiyel hata durumlarını ortaya çıkarma amacı taşır. Bu bağlamda, üretilen benzersiz test senaryolarının sayısı arttıkça, sistemin daha geniş bir davranış kümesi test edilebilir. Bu durum doğrudan daha yüksek kod kapsama oranı anlamına gelir.

Kod kapsama (örneğin satır, dal, yol düzeyinde) doğrudan ölçülmediği durumlarda, benzersiz test senaryosu sayısı, önemli bir gösterge olarak kullanılır. Ancak her benzersiz senaryo potansiyel olarak bir hataya sebep olmayabilir; dolayısıyla test kapsamı ile hata tespiti arasında her zaman birebir doğrusal bir ilişki beklenmez. Buna rağmen genellikle söylenebilir ki; yüksek test senaryosu üretimi, hata tespiti olasılığını artırır. Ampirik olarak incelendiği üzere, kod kapsamındaki %1'lik artışın, bulunan hataların yüzdesini %0,92 oranında artırabileceği ortaya çıkartılmıştır [68]. Bu

bağlamda, hata bulma yeteneğinin yanı sıra, fuzz testi ile elde edilen kod kapsamı da bu tez çalışmasında ölçülmüştür. Ayrıca her stratejinin ürettiği benzersiz test senaryolarının sayısını da karşılaştırdık; 36 saatlik bir pencerede yeni yolları tetikleyen test senaryoları incelendi. Test senaryolarını karşılaştırmanın stratejileri değerlendirmenin standart bir yolu olduğuna dikkat etmek önemlidir [69]. Ayrıca tohumların 36 saat boyunca toplanması, deneyler için zaman aralığının kabaca 24 saat olduğu önceki çalışmayla tutarlıdır [70]. Ayrıca stratejileri etkililik açısından karşılaştırdık. Bir mutasyon stratejisinin etkinliğini, benzersiz test senaryolarının ürettiği toplam test senaryosu sayısına oranı olarak tanımlıyoruz. Sonuçlarımız tüm mutasyon stratejilerinin farklı katsayılarla doğrusal bir büyüme gösterdiğini göstermektedir. Sonuçlarımız, stratejilerimizin giderek artan oranda dikkat çekici test senaryoları ürettiğini gösteriyor. Başka bir gözlem, dilbilgisi körü stratejilerin fuzz testleri sırasında verimli olabileceğidir; bu, farklı yorumlayıcıları inceleyen Superion araştırmacıları tarafından da yapılan bir gözlemdir.

Xerces için, AFL tarafından üretilen test senaryosu sayısı oldukça yüksektir ve zamanla önemli ölçüde artış göstermektedir. 12 saatlik sürede 2.176 örnek üretilmişken bu sayı 36. saatin sonunda 2.773'e ulaşmıştır. Aynı zaman aralıklarında Superion ise 378 ile başlayıp sadece 406'ya ulaşmıştır. Bu fark, Superionun yapı farkındalığı nedeniyle daha az ama daha anlamlı test örneği üretme stratejisine işaret etmektedir.

Bu test senaryo sayıları ile daha önceki hata tespit verileri birlikte incelendiğinde, AFLin artan sayıda senaryo üretmesiyle birlikte hata sayısında da (24 → 33 → 38) bir artış sağladığı görülmektedir. Buna karşılık, Superionun ürettiği senaryo sayısı sabit kalmakta ve bunun sonucunda hata tespiti de aynı şekilde sabit kalmaktadır. Bu durum Superionun erken dönemde yüksek kaliteli örneklerle kritik hataları yakaladığını; ancak test süresi uzadığında yeni senaryo üretiminde kısıtlı kaldığını ve dolayısıyla yeni hata bulamadığını göstermektedir.

Her iki araç da Libxml2 ve Expat için binlerce test senaryosu üretmiş olmasına rağmen, bu iki ayrıştırıcı üzerinde hiçbir hata tespiti yapılamamıştır. AFL, Expat üzerinde 36 saatte 6.791 senaryo üretmiş ama hata bulamamıştır. Superion, Libxml2 üzerinde 36 saatte 885 senaryo üretmiş ama yine hata tespit edememiştir.

Bu durum, yüksek senaryo sayısının tek başına yeterli olmadığını, aynı zamanda bu örneklerin hedef uygulamanın savunmasız kod yollarına erişim sağlayabilmesi gerektiğini ortaya koymaktadır. Diğer bir ifadeyle, bu sistemler ya daha güvenlidir ya da oluşturulan senaryolar bu sistemlerin hata üretmesini tetikleyecek nitelikte değildir. Kod kapsama potansiyeli yüksek olan araçlar daha fazla hata bulabilmektedir, ancak bu etki yalnızca savunmasız uygulamalarda (Xerces) belirgindir. Bu nedenle test senaryosu üretim kapasitesi, sadece uygulamanın test edilebilirliğiyle değil, aynı zamanda uygulamanın iç güvenlik mekanizmalarıyla da ilişkilidir.

Bu analizde, öğrenmesiz fuzz test araçları tarafından üretilen test senaryosu miktarı ile hata tespiti performansı arasında anlamlı bir ilişki olduğu gözlemlenmiştir. Ancak bu ilişkinin uygulama bağımlı olduğu anlaşılmıştır. AFL gibi daha agresif mutasyon stratejisi kullanan araçlar daha fazla test senaryosu üretebilmekte ve bu durum bazı uygulamalarda (Xerces) daha fazla hata ile sonuçlanabilmektedir. Öte yandan, yapı farkındalığına sahip Superior, sınırlı sayıda fakat daha anlamlı senaryo üretmesiyle erken evrede yüksek verim göstermekte, fakat bu verim zamanla doygunluğa ulaşmaktadır. Bu bağlamda, sistemli ve etkili bir fuzz test süreci için hem senaryo üretim yoğunluğu hem de üretim kalitesi dikkate alınmalı; ideal durumda bu iki yaklaşım birlikte kullanılmalıdır.

Bayt düzeyindeki mutasyonlar, hata tetikleyici test senaryolarının üretilmesinde lider konumdadır. Bunun nedeni, öncelikle bit ve bayt değişimlerinin çoğunlukla girdi yapısını tahrip etmesi ve daha önce görülmemiş hata işleme yollarını tetiklemesidir; ve ikinci olarak, bit ve bayt çevirmeler sırayla uygulanacak ilk mutasyon stratejisidir, böylece diğer mutasyon stratejileri tarafından da tetiklenebilecek yeni kapsamı ilk kez tetikleme fırsatına sahip olurlar. Dilbilgisine duyarlı mutasyon stratejileri tarafından üretilen ilginç test senaryolarının sayısı giderek diğer mutasyon stratejilerinden daha iyi performans göstermektedir.

Fuzz testlerinde seçtiğimiz hata tespit performans metriği olarak, belirli bir sürede (örneğin 12 saatte) tespit edilen hata sayısını süreye bölmek, birim zaman başına bulunan hata sayısını verir ve hata tespit hızı ya da hata üretkenliği olarak adlandırılır. Bu tür oranlar, özellikle farklı araçların etkinliklerini kıyaslamak için oldukça geçerli bir yöntemdir. Her iki mutasyon stratejisi için hata tespit performansı analizini yaptığımızda, bayt seviyesi mutasyon stratejisi için hata tespit performansının ağaç seviyesi mutasyon stratejisindeki performanstan çok daha büyük olduğu sonucunu

bulduk. Bu hesaplama göre bayt düzeyinde mutasyon için hata tespit maliyetleri ilk 12 saat için 2, ikinci 12 saat için 0,75'tir. Ve üçüncü 12 saat için 0,41. Bu sonuçlar, hata tespit maliyetinin zaman içinde giderek azaldığını göstermektedir. Öte yandan, ağaç düzeyinde mutasyon stratejisinde hata sayısı zamanla değişmediğinden, hata tespit performansı sırasıyla birinci, ikinci ve üçüncü 12 saatte 1,25, 0 ve yine 0'da sabit kalmıştır. Buradan çıkarılacak yorumlara göre; byte mutasyon stratejisine göre çalışan AFL, rastgele fakat yoğun test senaryosu üretimi sayesinde daha fazla varyant oluşturarak erken dönemde daha çok hata bulmuştur. Ağaç mutasyonu uygulayan Superion ise anlamlı fakat sınırlı sayıda senaryo üretmiş ve dolayısıyla daha yavaş bir hata tespit performansına sahip olmuştur. Libxml2 ve Expat üzerindeki hata tespit performansını inceleyecek olursak her iki araç da bu iki ayrıştırıcı üzerinde hiçbir hata tespit edemediği için hata tespit performansı 0 hata/saat olarak hesaplanmıştır. Bu sonuç, ya test senaryolarının yeterince etkili olmadığını ya da ayrıştırıcıların daha güvenli bir yapıya sahip olduğunu göstermektedir. Bu durum ayrıca, hata tespit performansının sadece araca değil, hedef sistemin güvenliği ve dayanıklılığına bağlı olduğunu ortaya koymaktadır.

Bu sonuçlar aynı zamanda AFLin bu tür karmaşık ve yapılandırılmış girdileri etkili bir şekilde işleyemeyen dilbilgisi körlüğünü de kanıtlıyor. XML gibi nispeten basit bir dilbilgisi için AFL, dilbilgisi tabanlı fuzz testinin çok gerisinde kalmamıştır. Ayrıca, karmaşık dilbilgisi nedeniyle AST'leri geçmek çok daha fazla zaman almış ve bu da her mutasyon turunun maliyetini artırmıştır. AFL ile karşılaştırıldığında Superion, dilbilgisine göre mutasyona uğrayabilir ve AFLin dilbilgisi körü stratejisinden daha güvenilir tohumlar üretme kapasitesine sahiptir. AFLin kendisi dilbilgisini hesaba katmaz, bu nedenle doğru şekilde ayrıştırılamayan önemli sayıda tohum olacaktır. Dolayısı ile hata tespit performansı, farklı fuzz araçlarının verimliliğini karşılaştırmak için anlamlı ve uygulanabilir bir metriktir. Bu metrik, özellikle kısa süreli testlerde hangi aracın daha hızlı geri bildirim verdiğini görmek açısından değerlidir. Ancak bu metrik tek başına yeterli değildir. Bazı araçlar düşük başlangıç hızına sahip olsa da daha yüksek toplam hata sayısına ulaşabilir.

Bayt ve ağaç düzeyinde mutasyon stratejileri için hata değerlendirme ölçütü sınıfı olarak, tespit edilen tek hata sınıfı "geçersiz talimat hatası"dır. Bu, 36 saatlik yürütme süresine kadar her iki mutasyona dayalı fuzz testi için açığa çıkan güvenlik açığı türünde hiçbir hata olmadığı sonucunu ortaya koymuştur. Bu bulgular, sistemin belirli

bir kararsızlık gösterdiğini ancak klasik bellek güvenliği zafiyetleri (örneğin buffer overflow, heap overflow, use-after-free) gibi daha ciddi ve sömürülebilir zafiyetler çıkarmadığını göstermektedir. Geçersiz talimat hatası, programın işlemci tarafından tanınmayan bir talimat çalıştırmaya çalışması durumunda meydana gelir. Bu durumun olası nedenleri:

Bozulmuş Kod Yolu: Bellekte geçersiz bir adrese atlama (örneğin, işaretçilerin yanlış yönlendirilmesi) sonucu geçerli olmayan bir komutun yürütülmesi.

Yanlış Formatta Girdi: Girdi tarafından tetiklenen hatalı bir durumun, yürütme akışını geçersiz bir noktaya taşıması.

Bu hata genellikle programın çökmesine neden olur ancak her zaman sömürülebilir bir zafiyet anlamına gelmez. Geçersiz talimat hatası, çoğu zaman sömürülemeyen bir duruma işaret eder. Yani bu çökme, sistemin kontrolünün kötü amaçlı yazılım tarafından ele geçirilmesine yol açmayabilir. Dolayısıyla, bu hatalar sistem kararlılığı açısından risk teşkil etse de, doğrudan güvenlik açığı sınıfına girmez.

Test süresince bellek taşması, yığın taşması gibi yaygın bellek tabanlı zafiyetlerin tespit edilmemesi şu olasılıkları gündeme getirir; ayrıştırıcıların bellek yönetimi güçlüdür, test edilen XML ayrıştırıcılarının olgun ve dikkatli şekilde geliştirilmiş olduğu, belleği etkin şekilde sınırladığı ve güvenlik odaklı tasarlandığı düşünülebilir. Diğer yandan fuzz testlerinin yetersizliği olarak düşünülürse uygulanan mutasyon stratejileri kritik bellek hatalarını tetikleyecek kadar karmaşık olmayabilir. Ayrıca 36 saatlik test süresi anlamlıdır; ancak bazı karmaşık zafiyetlerin ortaya çıkması için daha uzun süreli, öğrenen veya model tabanlı fuzz tekniklerine ihtiyaç olabilir. Öğrenmesiz testler, örneklerin sistematik şekilde zenginleştirilmesini sağlayamadığı için bazı derin zafiyetler görünür hale gelmemiş olabilir.

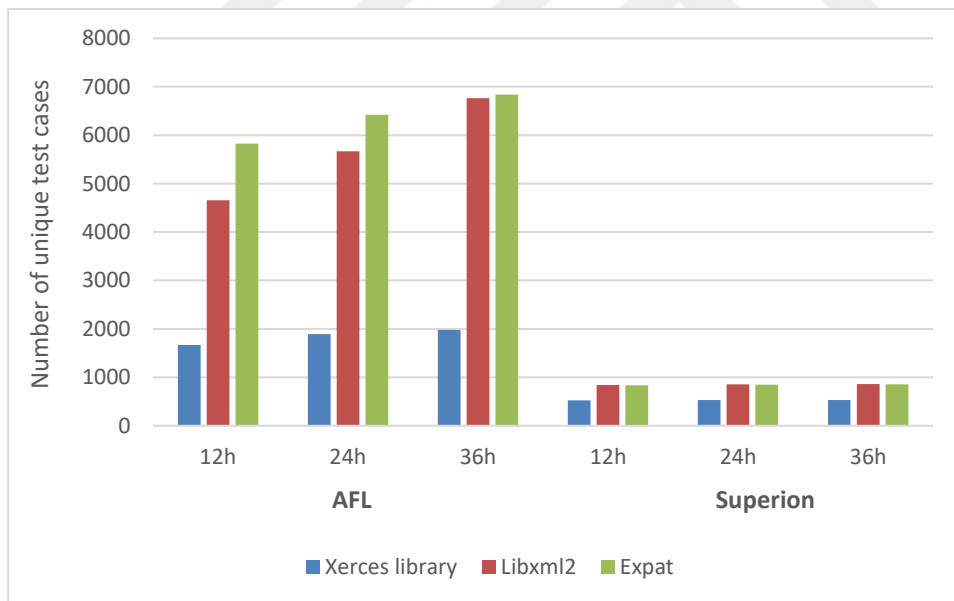
Sonuçlardaki rastgele gürültüyü önlemek için ANOVA adı verilen istatistiksel test yöntemi kullanılmıştır. Yapılan fuzz testlerinde hata sayılarının zaman (12h, 24h, 36h) ve test turları (1., 2., 3. tur) boyunca nasıl değiştiğini değerlendirmek için birden fazla grup arasındaki ortalama farklarını test etme ihtiyacı doğmuştur. Bu durumda klasik iki grup karşılaştırması yapan t-testleri yetersiz kalmaktadır.

Tek yönlü ANOVA (Analysis of Variance) testi, üç veya daha fazla grup ortalaması arasında anlamlı bir fark olup olmadığını test etmek için kullanılır. Bu bağlamda;

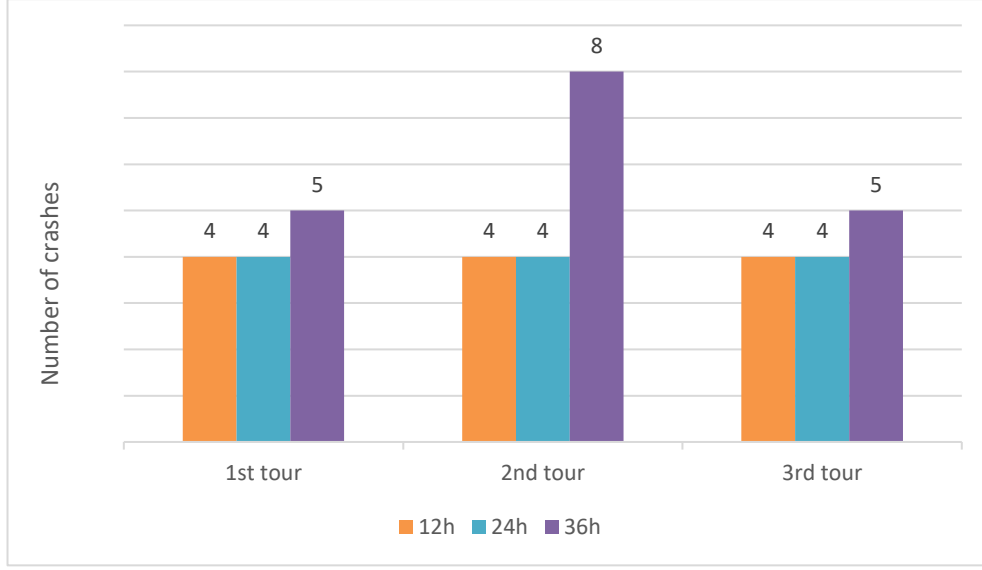
- Bağımlı değişken: Hata sayısı

- Bağımsız değişken: Zaman (12h, 24h, 36h)
- Alternatif olarak, bir diğer ANOVA bağımsız değişkeni: Tur (1., 2., 3.)

Motivasyon olarak ANOVA testi, hem zamana bağlı hata sayısı değişkenliğini hem de tekrar eden deneysel turlarda bu değişimin tutarlılığını istatistiksel olarak değerlendirmeyi mümkün kılar. Böylece yalnızca sayısal farklılıklar değil, bu farkların istatistiksel anlamlılığı da ortaya konmuş olur. Grafikte, üç farklı test turu boyunca (1. tur, 2. tur, 3. tur) farklı çalışma sürelerine göre (12 saat, 24 saat, 36 saat) elde edilen hata sayıları karşılaştırılmıştır. Her bir test turunda gözlenen hata sayıları oldukça benzer olup tutarlılık göstermektedir. Ağaç düzeyindeki mutasyonlarda görülen hiçbir hataya yol açmayan test yürütmeler için her seferinde sonuçlar değişmemiş, bu da onların hala hiçbir hata tespit edemediği anlamına gelmiştir. Ancak bayt düzeyindeki mutasyonlar için Apache Xerces platformunun test sonuçları biraz değişiklik göstermiştir. Fuzz testi için benzersiz test senaryolarının sayısı Şekil 5.1'de, Apache Xerces için, ANOVA baz alınarak her 12 saatlik, 24 saatlik ve 36 saatlik periyotlardaki hata sayısı ise Şekil 5.2'de görülmektedir.



Şekil 5.1. Fuzz testi için oluşturulan benzersiz test senaryoları



Şekil 5.2. Üç defa yapılan test için ANOVA sonuçları

5.2. RQ2-Öğrenme Tabanlı Oluşturulan Tohumlar, Genel Seçilmiş Tohumlara Göre Hata Bulmada Ne Kadar Etkili?

Tohum dosyası, fuzz testlerinin etkinliğini etkileyen kritik faktörlerden biridir. Fuzz testlerinin ikinci aşamasında genele açık olarak seçilmiş XML girdi dosyaları ile öğrenme tabanlı oluşturulan XML girdi dosyaları arasındaki analiz ve değerlendirmenin gerçekleştirilmesinin nedeni budur. Bunu gerçekleştirmek için Linux ortamında çalışan fuzz test araçları AFL ve Superion genele açık olarak seçilmiş ve öğrenmeye dayalı XML girdileri verilmiş ve ardından Libxml2, Apache Xerces ve Expat için 36 saat boyunca yürütmeler gerçekleştirilmiştir. Bu arada, her 12 saatte bir fuzz test uygulamalarının her biri için tespit edilen hata sayısı ölçülmüştür. Sonuçlar, Apache Xerces ayrıştırıcısı için hem öğrenme tabanlı hem de genele açık olarak seçilmiş girdilerle bayt düzeyinde mutasyon tabanlı testlerde tespit edilen hata sayısının 30'dan fazla olduğunu göstermektedir. Öğrenme tabanlı girdilerle tespit edilen hataların sayısı, genele açık olarak seçilmiş girdi dosyalarıyla tespit edilen hata sayısından daha fazladır.

Öğrenmesiz test senaryoları kapsamında, genele açık XML belgeleri kullanılarak gerçekleştirilen fuzz testlerinde en fazla hata tespit edilen ayrıştırıcı Xerces olmuştur. Bu ayrıştırıcı üzerinde AFL, 12., 24. ve 36. saatlerde sırasıyla 24, 33 ve 38 hata tespit etmiş; Superion ise her üç zaman diliminde sabit şekilde 15 hata üretmiştir. Bu bulgular, byte düzeyindeki mutasyonların zamanla daha fazla varyasyon üreterek yeni

hatalar keşfedebildiğini, buna karşın ağaç tabanlı mutasyonlarda hata üretiminin belirli bir noktada doygunluğa ulaştığını göstermektedir.

Libxml2 ve Expat ayrıştırıcıları değerlendirildiğinde, her iki test aracının da herhangi bir zaman diliminde hata tespit edemediği gözlemlenmiştir. Bu durum, iki olasılıkla açıklanabilir: (1) bu ayrıştırıcıların genele açık XML girdilerine karşı daha dirençli ve hata toleranslı olması, veya (2) kullanılan test girdilerinin, bu ayrıştırıcıların daha az karmaşık veya daha sınırlı özelliklerini tetikleyebilmesi nedeniyle etkili olamaması.

Skyfire temelli öğrenmeli test girdileri ile gerçekleştirilen fuzz testlerinde dikkat çekici bir artış görülmüştür. Xerces ayrıştırıcısı üzerinde AFL, sırasıyla 39, 50 ve 57 hata tespit etmiş; bu, öğrenmesiz testlere kıyasla ciddi bir artış demektir. Bu sonuç, Skyfire'nin yapısal olarak daha çeşitli ve hedefe yönelik test girdileri üretmesinin, byte düzeyinde mutasyon yapan araçlar için bile daha fazla hata tespit edebildiğini ortaya dökmektedir.

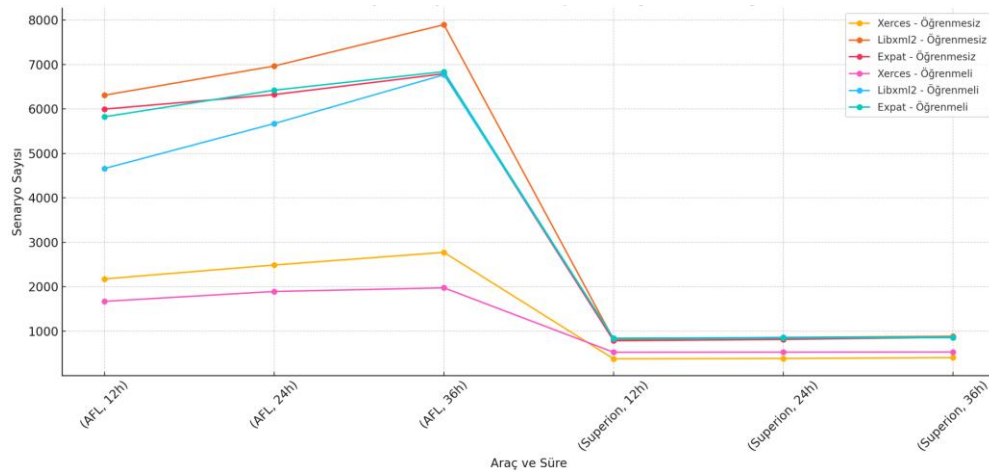
Diğer yandan, Superion aracıyla yapılan testlerde Xerces üzerinde yalnızca 5 hata tespit edilmiş ve bu sayı zamanla artmamıştır. Bu sonuç, Skyfire çıktılarının yapısal olarak Superion ile birlikte kullanılmasına rağmen yeterli çeşitlilikte mutasyon üretilmemiş olabileceğini göstermektedir. Veya başka bir seçenek olarak, Superion'un öğrenmeli girdiler üzerindeki mutasyon stratejisinin, zaten optimize edilmiş Skyfire çıktılarında etkin bir fark oluşturmaması olabilir. Libxml2 ve Expat için yine her iki araçla da herhangi bir hata tespit edilememiştir. Bu durum, öğrenmeli test verilerinin dahi bu ayrıştırıcıların hataya açık alanlarını ortaya çıkaramadığını veya kullanılan araçların bu sistemlerin karmaşık hata özelliklerini ortaya çıkaracak mutasyonları yeterince etkili biçimde üretmediğine işaret etmektedir.

Elde edilen bulgular genel olarak değerlendirildiğinde, test girdilerinin yapısal kalitesi ve mutasyon stratejisinin birlikte değerlendirilmesinin önemli olduğu görülmektedir. AFL, öğrenmeli senaryolarda yüksek performans göstererek, öğrenme algoritması ile üretilen girdilerde rastgele mutasyonlarla dahi ciddi sayıda hata tespit edebildiğini göstermiştir. Superion ise, öğrenmesiz testlerde belirli bir başarı yakalasa da, öğrenmeli testlerde görece düşük hata üretimi ile sınırlı kalmıştır. Bu durum, yapı öğrenmeye dayalı giriş üretim tekniklerinin sadece test girdisinin kalitesini değil, aynı zamanda kullanılan mutasyon stratejisinin etkisini de belirleyici kıldığını ortaya koymaktadır. Özellikle yüksek yapı farkındalığı gerektiren ağaç tabanlı mutasyon

tekniklerinin, etkili olabilmesi için test girdilerinin yapısal çeşitliliğini artıracak ilave mekanizmalarla desteklenmesi gerektiği öngörülmektedir.

Ek olarak, öğrenme tabanlı tohum dosyalarının sayısı genele açık tohum dosyalarından çok daha fazla olmasına rağmen, öğrenme tabanlı tohumlarla fuzz testi başlangıcından önce tespit edilen hataların sayısı, genele açık olarak seçilen tohumlarla başlatılanlardan daha azdır. Dolayısıyla bu bize, tohum dosya sayısı fazla olmasına ve öğrenme algoritmasına dayalı olarak oluşturulmasına rağmen, ağaç düzeyinde mutasyon stratejisi kullanıldığında testlerin daha fazla hata bulamadığı sonucunu veriyor. Bunun nedeni, genele açık tohum dosyalarında hem içerik hem de boyut olarak çok daha fazla çeşit olması ve ardından fuzz test sürecinin ilk kontrol aşamasında başarısız olma ihtimalinin daha yüksek olmasıdır. Bu sonuçlar, akıllı tohumların bir fuzz test aracı için iyi tohum girdileri sağlayabildiğini, dolayısıyla fuzz test aracından sürekli yürütme ve mutasyon yoluyla iyi tohum bulma yükünü ortadan kaldırdığını ve XML girdilerini işleyen programlar için hataları keşfetme yeteneğini önemli ölçüde geliştirebildiğini göstermektedir.

Fuzz testlerinin başarısı, yalnızca tespit edilen hata sayıları ile değil, aynı zamanda test süresi boyunca üretilen test senaryolarının sayısı ile de doğrudan ilişkilidir. Çünkü yüksek sayıda ve benzersiz test senaryosu üretimi, sistemin farklı kod yollarına erişimi artırarak hata tespit oranını yükseltebilir. Bu kapsamda, hem öğrenmesiz hem de öğrenmeli XML test girdileri kullanılarak, 12, 24 ve 36 saatlik zaman dilimlerinde AFL ve Superion araçlarının toplam kaç adet test senaryosu ürettiği ölçülmüştür. Fuzz test araçlarının verilen süreler doğrultusunda XML ayrıştırıcıları için oluşturduğu benzersiz test senaryolarını gösteren grafik aşağıdaki şekilde verilmiştir.



Şekil 5.3. XML ayrıştırıcıları için üretilen test senaryoları

Sonuçlardan anlaşılacağı üzere genele açık girdi dosyalarındaki test senaryolarının sayısı, öğrenme tabanlı girdi dosyalarındaki test senaryo sayısından daha fazladır. Buna karşılık, genele açık olarak seçilen girdi dosyaları için hata sayısı daha azdır. Bu bize artan test senaryo sayısının doğrudan tespit edilen hata sayısında artışa neden olmadığı içgörüsünü vermiştir. Ancak girdi ve test senaryo dosyalarının kalitesi veya akıllılığı, hataların bulunmasında bu dosyaların miktarından çok daha önemli bir rol oynar. Bu sonuçlardan, akıllı tohumların fuzz aracı için iyi dağıtılmış tohum girdileri sağlayabildiğini ve böylece kod kapsamının büyük ölçüde iyileştirilebildiğini çıkarabiliriz.

AFLin byte düzeyinde çalışması ve rastgele mutasyonlarla hızlı şekilde test senaryoları üretme kabiliyeti, yüksek sayıda üretilen test senaryo sayılarının temel nedenidir. Özellikle Libxml2 ve Expat ayrıştırıcıları üzerinde 36 saatlik sürede neredeyse 7–8 bin test senaryosuna ulaşılmış olması, AFLin bu sistemlere karşı oldukça agresif bir test üretim stratejisi izlediğini göstermektedir.

Bununla birlikte Superion, öğrenmesiz tohumlarla daha sınırlı sayıda test senaryosu üretmiştir. Bunun nedeni, Superionun XML'in ağaç yapısını korumaya çalışması ve yapısal bütünlüğü bozmadan mutasyon üretmeye çalışması nedeniyle üretim hızının düşmesidir. Ancak bu durum, üretilen test senaryolarının yapısal olarak daha geçerli ve analiz açısından daha anlamlı olabileceğine de ışık tutmaktadır.

Öğrenme ile oluşturulan tohumlarla yapılan testlerde ise AFLin üretim kapasitesi kısmen azalmış, Superionun ise daha dengeli sonuçlar verdiği görülmüştür. Bu sonuçlar, öğrenmeli tohumlar kullanıldığında AFLin test üretim hızı biraz düşse de hâlâ yüksek kaldığını, Superionun ise benzer düzeyde üretim gerçekleştirmeye devam ettiğini göstermektedir. Özellikle Superionun 36 saatlik sürede her ayrıştırıcı için ortalama 850 civarında test senaryosu üretmesi, yapısal olarak anlamlı mutasyonların üretiminde tutarlı bir davranış gösterdiğini ortaya koymaktadır. AFLin öğrenmeli senaryolarda test üretim hızının nispeten azalması, Skyfire tarafından üretilen girdilerin daha “geçerli” ve karmaşık olması nedeniyle rastgele mutasyonların daha az etkili hale gelmesi ile açıklanabilir. Superionun yapı bazlı olması nedeniyle, Skyfire girdileri üzerinde daha kolay çalışabildiği ve üretim kapasitesini koruyabildiği gözlemlenmiştir.

Daha fazla sayıda oluşturulan test senaryoları, kod kapsamı açısından ayrıştırıcıda daha fazla dallara ulaşabilir. Ancak bu, otomatik olarak daha fazla hata tespit edebileceği sonucuna varmaz. Dolayısıyla, hataları açığa çıkarabilecek test senaryolarının kalitesi veya akıllılığı, test senaryolarının miktarından ziyade etkinliğini artırır. Bu bize, öğrenmeye dayalı tohum üretiminin, tohum dosyalarının kalitesi açısından, genele açık tohum dosyalarıyla karşılaştırıldığında daha fazla hata bulabilecek kadar akıllı olduğuna dair bir ipucu vermiştir.

Her iki mutasyon stratejisi için hata tespit performansı analizi yaptığımızda, bayt seviyesi mutasyon stratejisi için hata tespit performansının ağaç seviyesi mutasyon stratejisindeki performanstan daha büyük olduğu sonucunu bulduk. Bu hesaplama göre bayt düzeyinde mutasyon için hata tespit performansları ilk 12 saat için 3,25, ikinci 12 saat için 0,91, üçüncü 12 saat için ise 0,58'dir. Bu sonuçlar, hata tespit performansının zaman içinde giderek azaldığını göstermektedir. Diğer taraftan ağaç düzeyinde mutasyon stratejisinde hata sayısı zaman içinde değişmediğinden, hata tespit performansı da sabit olup en başta 0,41 değerine eşittir, ve ikinci ve sonraki aşamalarda 0'a eşit kalmıştır. Buradan çıkarılacak yoruma göre byte mutasyon stratejisine göre çalışan AFL, rastgele fakat yoğun test senaryosu üretimi sayesinde daha fazla varyant oluşturarak erken sürede daha çok hata bulmuştur. Ağaç mutasyonu uygulayan Superion ise anlamlı fakat sınırlı sayıda senaryo üretmiş ve bu sebepten ötürü daha yavaş bir hata tespit performansına sahip olmuştur. Libxml2 ve Expat üzerindeki hata tespit performansını inceleyecek olursak her iki araç da bu iki ayrıştırıcı üzerinde hiçbir hata tespit edemediği için hata tespit performansı 0 olarak hesaplanmıştır. Bu sonuç, ya test senaryolarının yeterince etkili olmadığını ya da ayrıştırıcıların daha güvenli bir yapıya sahip olduğunu göstermektedir. Bu durum ayrıca, hata tespit performansının sadece araca değil, hedef sistemin güvenlik ve dayanıklılık düzeyine bağlı olduğunu göstermektedir. Xerces ayrıştırıcısının fuzz testleri sonucu elde edilen hata tespit performans değerleri Tablo 5.1'de görülmektedir.

Tablo 5.1. Xerces için hata tespit performanları.

	AFL			Superion		
	12h	24h	36h	12h	24h	36h
Öğrenmesiz	2	0.75	0.41	1.25	0	0
Öğrenmeli	3.25	0.91	0.58	0.41	0	0

Diğer yandan fuzz testlerinde oluşturulan test senaryoları ile tespit edilen hatalar arasındaki korelasyon incelendiğinde fuzz test performansını etkileyen iki temel faktör üzerinde durulabilir;

- Test senaryosu üretim hacmi
- Bu senaryoların hata üretme etkinliği

Byte tabanlı mutasyon stratejisi uygulayan AFL aracının test senaryosu üretimi ve hata etkinliği incelendiğinde öğrenmesiz tohum dosyaları için Xerces ayrıştırıcısında üretilen senaryo sayısı 2773, tespit edilen hata sayısı 38, buradan yola çıkarak elde edilen verimlilik oranı da 1.37%'dir (her 73 testte 1 hata). Libxml2 ve Expat'ta üretilen test senaryo sayıları 7895 ve 6791 iken hiç hata tespit edilememiştir. Bu durum, yüksek hacimli test üretiminin her zaman hata tespitine yol açmadığını göstermektedir. Buradan ayrıştırıcıların ya oldukça sağlam olduğu ya da AFLin bu sistemlerin kırılabilirlik noktalarını ortaya çıkaramadığı anlaşılmaktadır.

Öğrenmeli tohum dosyaları ile gerçekleştirilen testlerde Xerces için üretilen test senaryo sayısı 1976, tespit edilen hata sayısı 57'dir. Bunu baz alarak hesaplanan verimlilik oranı 2.88%'dir (her 35 testte 1 hata). Bu durum, öğrenmesiz durumun yaklaşık iki katı kadar daha verimli olduğunu göstermektedir. Skyfirein ürettiği yapısal olarak anlamlı girdiler, AFLin mutasyonlarının daha hedefli olmasını sağlamış ve hata üretimini artırmıştır.

Ağaç tabanlı mutasyon stratejisi uygulayan Superion aracının test senaryosu üretimi ve hata etkinliği incelendiğinde öğrenmesiz tohum dosyaları için Xerces ayrıştırıcısında üretilen senaryo sayısı 406, tespit edilen hata sayısı 15, buradan yola çıkarak elde edilen verimlilik oranı da 3.69%'dur (her 27 testte 1 hata). Superion, çok daha az sayıda test üretmiş olsa da, AFL'e göre daha yüksek verimlilik oranı

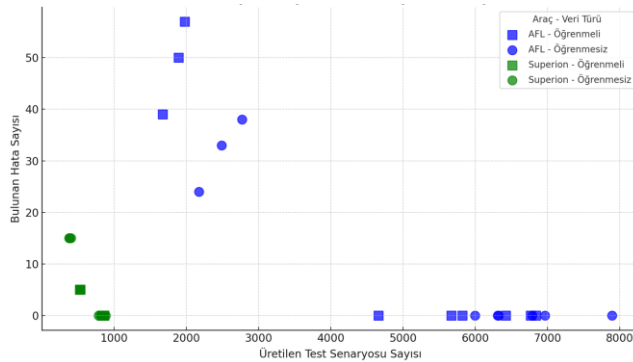
göstermektedir. Bu durum, yapıya duyarlı mutasyonların daha “anlamlı” hata düzeylerine eriştiğini göstermektedir.

Öğrenmeli tohum dosyaları ile gerçekleştirilen testlerde Xerces için üretilen test senaryo sayısı 532, tespit edilen hata sayısı 5’dir. Bunu baz alarak hesaplanan verimlilik oranı 0.94%’dür (her 106 testte 1 hata). Görüleceği üzere, öğrenmeli durumda Superionın hata üretme etkinliği düşmüştür. Skyfire çıktıların zaten yapısal olarak optimize edilmiş olması, Superionın bu yapılar üzerinde efektif mutasyon yapmasını zorlaştırmış olması muhtemeldir. Üretilen test senaryoları ile tespit edilen hatalar arasındaki etkinlik değerlerini Xerces ayrıştırıcısı için gösteren tablo aşağıdadır;

Tablo 5.2. Xerces için test senaryo ve hata korelasyonu.

Araç	Tohum Türü	Üretilen Test	Hata Sayısı	Hata için gerekli test
AFL	Öğrenmesiz	2773	38	73
AFL	Öğrenmeli	1976	57	35
Superion	Öğrenmesiz	406	15	27
Superion	Öğrenmeli	532	5	106

Tüm bu çıkarımları özetleyen bir şekil çizilmesi gerekirse bu aşağıdadır ve bu şekil en yüksek hata verimliliğinin Superion + öğrenmesiz senaryoda elde edildiğini, ancak en fazla sayıda hata tespiti açısından AFL + öğrenmeli kombinasyonun daha güçlü olduğunu göstermektedir.



Şekil 5.4. Test Senaryosu ile hata arasındaki korelasyon

Sonuç olarak, tüm fuzz testi süreci boyunca oluşan hata sayısı ve oluşturulan test senaryolarının sayısı Tablo 5.3 ve Tablo 5.4'de gösterilmiştir.

Tablo 5.3. Test edilen üç sistem için hata tespit sonuçları.

Öğrenmesiz	AFL			Superion		
	12h	24h	36h	12h	24h	36h
Xerces	24	33	38	15	15	15
Libxml2	0	0	0	0	0	0
Expat	0	0	0	0	0	0
Öğrenmeli	AFL			Superion		
	12h	24h	36h	12h	24h	36h
Xerces	39	50	57	5	5	5
Libxml2	0	0	0	0	0	0
Expat	0	0	0	0	0	0

Tablo 5.4. Test edilen üç sistem için benzersiz test senaryoları.

Öğrenmesiz	AFL			Superion		
	12h	24h	36h	12h	24h	36h
Xerces	2176	2489	2773	378	386	406
Libxml2	6309	6965	7895	812	861	885
Expat	5996	6323	6791	786	815	863
Öğrenmeli	AFL			Superion		
	12h	24h	36h	12h	24h	36h
Xerces	1671	1893	1976	526	529	532
Libxml2	4659	5670	6768	842	858	864
Expat	5823	6421	6839	832	845	856

5.3. XML zafiyetlerine yönelik çalışma

Fuzz testleri kapsamında yalnızca yazılımın kararlılığı ve çökme odaklı hataların ötesinde, potansiyel güvenlik açıklarını da ortaya çıkarma amacıyla özel bir yol izlenmiştir. Bu amaçla, Skyfire tabanlı tohum üretme algoritması üzerinde özelleştirmeler yapılarak, bilinen XML güvenlik zafiyetlerini taşıyan XML dosyaları üretilmiş ve bu dosyalar test giriş verileri olarak kullanılmıştır.

Çalıştırması yapılan testlerin detayını ve sonuçlarını yorumlamadan önce, XML güvenlik açıklığına sebebiyet verebilecek zafiyetli XML dosya tiplerini açıklamak gerekmektedir;

XML Injection: Kötü niyetli kullanıcıların, gönderilen XML verisi içine özel yapılandırılmış girdiler enjekte ederek XML işleme mantığını manipüle etmeye çalıştığı bir saldırı türüdür. Bu zafiyet, özellikle XPath veya XQuery ifadelerinin kullanıcı girdisiyle oluşturulduğu durumlarda ortaya çıkabilir. Hedef sistemde kimlik doğrulama bypass, yetkisiz veri erişimi ya da mantıksal sistem hatalarına yol açabilir.

XML External Entity (XXE): XXE zafiyeti, XML dosyasının içinde tanımlanan harici varlıkların (external entity) kullanılması ile, sistemin yerel dosya sistemine erişim, SSRF (Server-Side Request Forgery) gibi saldırılara maruz kalmasına neden olabilir. Bu zafiyet, özellikle DTD (Document Type Definition) tanımlarının etkin olduğu XML ayrıştırıcılarda ortaya çıkmaktadır.

Billion Laughs (XML Bomb): Bu saldırı türü, küçük XML varlıklarının üstel biçimde genişleyerek sistem kaynaklarını tüketmesini sağlayan, klasik bir DoS (Denial of Service) saldırısıdır. Ayrıştırıcı belleği veya işlemci gücünü aşırı kullanıma zorlayarak sistemi çökertmeyi hedefler.

36 saat süresince hem American Fuzzy Lop (AFL) hem de Superion aracı kullanılarak yapılan testlerde, özel olarak üretilmiş güvenlik zafiyeti içeren XML dosyaları ile Xerces, Libxml2 ve Expat ayrıştırıcılarının dayanıklılığı test edilmiştir. Giriş verisi olarak kullanılan dosyalar, farklı biçim ve varyasyonlarda XXE, XML injection ve Billion Laughs gibi tehditleri barındıracak şekilde yapılandırılmıştır. Ancak yapılan tüm testlerin sonucunda:

- Herhangi bir bellek taşması (heap/stack overflow),
- Doğrudan sömürülebilir bir arabellek aşımı (buffer overflow),
- Yasaklı sistem kaynaklarına erişim (örneğin /etc/passwd sızıntısı),

gibi kritik güvenlik zafiyetleri tespit edilememiştir.

Yukarıdaki sonuçlar, test edilen ayrıştırıcıların yaygın XML güvenlik zafiyetlerine karşı büyük ölçüde dirençli olduğunu göstermektedir. Özellikle Libxml2 ve Xerces gibi olgun kütüphanelerin, XXE ve Billion Laughs gibi klasik saldırı vektörlerine karşı güncel sürümlerinde gerekli güvenlik önlemlerini içerdiği anlaşılmıştır.

Ancak burada dikkat çekici olan bir diğer husus, kullanılan fuzz test araçlarının (özellikle AFL) bu tip semantik açıdan sofistike saldırıları tespit etmede sınırlı etkili olabileceğidir. Bu tür saldırılar, yalnızca programın çökmesine değil, mantıksal veya yetkisiz işlem yürütmesine neden olabileceğinden, tespit edilmesi için dinamik analiz ya da davranışsal analiz tabanlı yöntemlerin eklenmesi gerekebilir.

Buna ek olarak, Skyfirein içerik tabanlı tohum üretme algoritmasının özelleştirilmiş sürümü sayesinde belirli semantik yapıların test senaryolarında yer alması sağlanmış olsa da, bu tür hedefe yönelik güvenlik testleri için klasik mutasyon tabanlı fuzz testlerinin tamamlayıcı araçlar ile desteklenmesi gerektiği sonucuna ulaşılmıştır.

Diğer yandan fuzz testlerinin daha derinlemesine incelenmesiyle, özellikle Billion Laughs saldırı vektörlerini içeren XML tohumları ile yapılan testlerde, sistemin çökmesine neden olan klasik "crash" durumları yerine, ayrıştırıcıların uzun süre işlemeyi tamamlayamaması şeklinde gözlemlenen "hangs" (kilitlenme) vakalarında belirgin bir artış yaşandığı tespit edilmiştir. Bu durum, söz konusu saldırının doğası gereği gayet tutarlıdır.

Billion Laughs saldırısı, XML içinde tanımlanan iç içe geçmiş ve üstel olarak büyüyen varlık tanımlamaları aracılığıyla XML ayrıştırıcıyı yoğun işlem ve bellek tüketimi ile karşı karşıya bırakır. Ancak bu saldırı çoğu zaman sistemin doğrudan çökmesine neden olmaz; bunun yerine, ayrıştırıcının uzun süre boyunca yanıt verememesine, işlem süresinin aşırı uzamasına ve CPU kaynaklarının tükenmesine neden olur. Bu tür etkiler, özellikle fuzz test araçlarında "hang" olarak sınıflandırılmakta ve sistemin stabilite, kullanılabilirlik ve performans açısından risk altında olduğunu göstermektedir.

Sonuç olarak bu testler, ayrıştırıcıların Billion Laughs saldırılarına karşı doğrudan güvenlik açığı vermemelerine rağmen, hizmetin kesintiye uğramasına neden olabilecek performans temelli saldırılara karşı kırılgan olabileceğini göstermektedir. Bu durum, XML işlemeye dair yapılan tasarımlarda yalnızca güvenlik açıkları değil, aynı zamanda kaynak tüketimi temelli DoS risklerinin de dikkate alınması gerektiğini ortaya koymaktadır.

5.4. Sonular ve gelecek alıřmalar

Bu tez alıřmasında, farklı mutasyon stratejilerini kullanan fuzz test yöntemleri arasında kapsamlı bir vaka alıřması araştırması yapılmıřtır. Sadece hatalar deęil, aynı zamanda mutasyona dayalı farklı fuzz testi yaklařımları iin akıllıca üretilen tohumların mutasyon stratejileri ve etkileri üzerine de detaylı bir analiz yaptık. Öncelikle tespit edilen hata sayısı, hata tespit performansı ve hata sınıflarına iliřkin farklı mutasyon stratejileri hakkında bir araştırma yürüttük. XML ayrıştırıcıları iin yaptığımız fuzz testi yürütmelerinden sonra, bayt düzeyindeki mutasyonların, ağaç düzeyindeki mutasyonlardan daha fazla hata bulabileceęi sonucuna vardık. İkinci olarak, öğrenme tabanlı XML tohumlarıyla bařlayan fuzz testinin hata tespit yeteneklerini, genele açık XML tohumlarından beslenen fuzz testi ile karşılařtırdık ve öğrenme tabanlı tohumların, genel olanlardan daha fazla hatayı ortaya ıkarabildięini gördük. Ayrıca hataların bulunmasında test senaryolarının miktarından ziyade, akıllılıęının ok daha önemli bir rol oynadıęını ve bunun, PCSG öğrenme algoritması aracılıęıyla oluřturulan daha akıllı tohum dosyaların daha fazla hataya neden olabileceęini de gösterdik. Derinlemesine alıřmamız bir karşılařtırma kriterini, yani üç XML ayrıştırıcısını, Libxml2, Apache Xerces ve Expat'ı ele alıyor. Dięer girdi formatları ve dięer ayrıştırıcılar iin sonular farklılık gösterebilir ve spesifik deneysel sonularımız hakkında yapılan ayrıntılı gözlemlerin dięer bağlamlara tařınacaęına dair bir iddiamız yoktur. Bu belirsizlięin oęu, belirsiz araştırma alanlarındaki herhangi bir ampirik alıřmanın doęasında bulunmaktadır.

Verimlilik aısından bakıldıęında, ok sayıda test üretmek deęil, doęru yapıda ve hedefli mutasyonlar yapmak daha önemlidir. Özellikle Superion gibi yapı tabanlı mutasyon araçları, düşük hacimli ama etkili test senaryoları üretebilmektedir.

Skyfire tarafından oluřturulan öğrenmeli tohumlar, bazı durumlarda test verimlilięini artırırsa da, Superion gibi yapı-koruyucu araçlar bu tohumlarda sınırlı mutasyon alanı bulmuř olabilir.

AFL, özellikle yüksek hacimli ve agresif test üretimiyle dikkat ekmekte; ancak verimlilik aısından her zaman en iyi sonuları üretmemektedir.

Ayrıştırıcı baęımlılıęı ok yüksektir. Libxml2 ve Expat, hem öğrenmesiz hem öğrenmeli testlerde hiçbir araçla hata vermemiřtir. Bu ayrıştırıcıların derinlemesine analizine yönelik özel hedefli grammar tabanlı teknikler önerilmektedir.

Gelecekteki bir çalışma olarak, ağaç düzeyinde mutasyon stratejisinin daha az hata bulma yetenekleri daha ayrıntılı olarak araştırılabilir. Bu, hataların fuzz testinin başlangıcında meydana gelmesi nedeniyle, test esnasında hataların nasıl tespit edilebileceğine dair bazı araştırmalar yapılabilecek ve ağaç düzeyinde mutasyon stratejisinin geliştirilmesi veya yeni bir çözüm bulunması için hata sayısına ilişkin fuzz testi etkinliğini artırmak için mutasyon stratejisi konusunda yapılabilecek şekilde gerçekleştirilebilir.



KAYNAKLAR

- [1] Li, J., Zahao, B., & Zhang, C. (2018). Fuzzing: a survey. *Cybersecurity* 1.1: 1-13. <https://doi.org/10.1186/s42400-018-0002-y>
- [2] Chen, C., Cui B., Ma J., Wu R., Guo J., & Liu, W. (2018). A systematic review of fuzzing techniques. *Computers Security*, 36, vol. 75, pp. 118-137. <https://doi:10.1016/j.cose.2018.02.002>
- [3] Payer, M. (2019). The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security Privacy*, 38, vol. 17, no. 1, pp. 78-82. <https://doi:10.1109/MSEC.2018.2889892>
- [4] Izumi, F., de Lima Filho, E. B., Cordeiro, L. C., Maia, O., Fabricio, R., Farias, B., & Silva, A. (2023). A fuzzing-based test-creation approach for evaluating digital TV receivers via transport streams. *Software Testing, Verification and Reliability*, vol. 33, no. 1, p. e1833. <https://doi:10.1002/stvr.1833>
- [5] Görz, P., Mathis, B., Hassler, K., Güler, E., Holz, T., Zeller, A., & Gopinath, R. (2023). Systematic assessment of fuzzers using mutation analysis. 32nd *USENIX Security Symposium (USENIX Security 23)*, pp. 4535-4552
- [6] Böhme, M., Cadar, C., Roychoudhury, A. (2020). Fuzzing: Challenges and reflections. *IEEE Software*, vol. 38, no. 3, pp. 79-86. <https://doi:10.1109/MS.2020.3016773>
- [7] Wang, J., Chen, B., Wei, L., Liu, Y. (2019). Superion: Grammar-aware greybox fuzzing. *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, pp. 724-735. <https://doi.org/10.1109/ICSE.2019.00081>
- [8] Güler, E., Schumilo, S., Schloegel, M., Bars, N., Görz, P., Xu, X., ...& Holz, T. (2024). Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In 33rd *USENIX Security Symposium (USENIX Security 24)*, pp. 4765-4782
- [9] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11), 2312-2331. <https://10.1109/TSE.2019.2946563>
- [10] Godefroid, P. (2020). Fuzzing: Hack, art, and science. *Communications of the ACM* 63.2, 70-76. <https://doi.org/10.1145/3363824>
- [11] Saavedra, G. J., Rodhouse, K. N., Dunlavy, D. M., & Kegelmeyer P. W. (2019). A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*
- [12] Liang, H., Pei, X., Jia, X., Shen, W., & Zhang J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability* 67.3, 1199-1218. <https://10.1109/TR.2018.2834476>

- [13] Ognawala, S.(2020). Scalable Greybox Fuzzing for Effective Vulnerability Management [Doctorate thesis]. Technische Universität München
- [14] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In NDSS, volume 16, pages 1–16. <http://dx.doi.org/10.14722/ndss.2016.23368>
- [15] Godefroid P., Peleg H., Singh R. (2017). Learnfuzz: Machine learning for input fuzzing. Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp. 50-59. <https://doi:10.1109/ASE.2017.8115618>
- [16] Kim M., Park S., Yoon J., Noh, B. N. (2014). File analysis data auto-creation model for peach fuzzing. Journal of the Korea Institute of Information Security Cryptology, vol. 24, no.2, pp. 327-333. <http://dx.doi.org/10.13089/JKIISC.2014.24.2.327>
- [17] Biyani A., Sharma G., Aghav J., Waradpande P., Savaji P., & Gautam, M. (2011). Extension of SPIKE for encrypted protocol fuzzing. Third International Conference on Multimedia Information Networking and Security, IEEE, pp. 343-347. <https://10.1109/MINES.2011.143>
- [18] Yang X., Chen Y., Eide E., Regehr J. (2011). Finding and understanding bugs in C compilers. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, pp. 283-294
- [19] Holler C., Herzig K., Zeller A. (2012). Fuzzing with code fragments. 21st USENIX Security Symposium (USENIX Security 12), pp. 445-458
- [20] Veggalam S., Rawat S., Haller I., Bos H. (2016). Ifuzzer: An evolutionary interpreter fuzzer Using genetic programming. European Symposium on Research in Computer Security, Springer, pp. 581-601
- [21] Viide, J., Helin, A., Laakso, M., Pietikäinen, P., Seppänen, M., Halunen, K., ... & Röning, J. (2008). Experiences with Model Inference Assisted Fuzzing. WOOT, 2, pp. 1-2
- [22] Wu, Z. (2022). A Study of Grammar-Based Fuzzing Approaches [Master's thesis]. California Polytechnic State University
- [23] Vimpari, M. (2015). An evaluation of free fuzzing tools [Master's thesis]. University of Oulu
- [24] Song, J., & Alves-Foss, J. (2024). A Fuzzing Tool Based on Automated Grammar Detection. Software, 3(4), 569-586. <https://doi.org/10.3390/software3040028>
- [25] Zhao, Y., Gao, L., Wan, Q., & Zhao, L. (2024). Grammar-aware test case trimming for efficient hybrid fuzzing. Journal of King Saud University-Computer and Information Sciences, 36(1), 101920. <https://doi.org/10.1016/j.jksuci.2024.101920>
- [26] Müller, S., Sparka, J. A., Kuban, M., Draxl, C., & Grunske, L. (2024). Grammar-based fuzzing of data integration parsers in computational materials science. Software: Practice and Experience, 54(2), 208-224. <https://doi.org/10.1002/spe.3266>

- [27] Zamudio Amaya, J. A., Smytcek, M., & Zeller, A. (2025). FANDANGO: Evolving Language-Based Testing. *Proceedings of the ACM on Software Engineering*, 2(ISSTA), 894-916. <https://doi.org/10.1145/3728915>
- [28] Barranca F. P. (2020). Machine learning for fuzzing: State of art
- [29] Lyu C., Ji S., Li Y., Zhou J., & Chen J. (2018) Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606*
- [30] Swiecki, R. (2017). Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. URL: <https://github.com/google/honggfuzz> (visited on 06/21/2017)
- [31] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). VUzzer: Application-aware evolutionary fuzzing. *NDSS*. Vol. 17. <https://doi.org/10.14722/ndss.2017.23404>
- [32] Ye, A., Wang, L., Zhao, L., Ke, J., Wang, W., & Liu, Q. (2021). RapidFuzz: Accelerating fuzzing via generative adversarial networks. *Neurocomputing* 460, 195–204. <https://doi.org/10.1016/j.neucom.2021.06.082>
- [33] Chaffjiri, S. B., Legg, P., Hong, J., & Tsompanas, M. A. (2024). Vulnerability Detection through machine learning based fuzzing: A systematic review. *Computers Security*, 103903. <https://doi.org/10.1016/j.cose.2024.103903>
- [34] Daniele, C., Andarzian, S. B., & Poll, E. (2024). Fuzzers for stateful systems: Survey And Research Directions. *ACM Computing Surveys*, 56(9), 1-23. <https://doi.org/10.1145/3648468>
- [35] Harzevili, N. S., Mohajer, M. M., Wei, M. O. S. H. I., Pham, H. V., & Wang, S. O. N. G. (2024). History-driven fuzzing for deep learning libraries. *ACM Trans. Softw. Eng. Methodol*
- [36] Ollando, R., Shin, S. Y., & Briand, L. C. (2025). Learning-Guided Fuzzing for Testing Stateful SDN Controllers. *ACM Transactions on Software Engineering and Methodology*. <https://doi.org/10.1145/3733717>
- [37] Parikh, D. P. (2025). Adaptive Reinforcement Learning-Based Fuzzer for 5G RRC Security Evaluation [Master's thesis]. Virginia Polytechnic Institute and State University
- [38] Wei, Z., Wei, X., Zhao, X., Hu, Z., & Xu, C. (2024). SGANFuzz: A Deep Learning-Based MQTT Fuzzing Method Using Generative Adversarial Networks. *IEEE Access*, 12, 27210-27224. <https://doi.org/10.1109/ACCESS.2024.3365712>
- [39] Wang J., Chen B., Wei L., & Liu Y. (2017). Skyfire: Data-driven seed generation for fuzzing. *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 579-594. <https://doi.org/10.1109/SP.2017.23>
- [40] Aschermann C., Frassetto T., Holz T., Jauernig P., Sadeghi A. R., & Teuchert, D. (2019). NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*. <https://dx.doi.org/10.14722/ndss.2019.23xxx>
- [41] Xu, H., Jiang, Z., Wang, Y., Fan, S., Xu, S., Xie, P., ... & Payer, M. (2024, December). Fuzzing JavaScript Engines with a Graph-based IR. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 3734-3748. <https://doi.org/10.1145/3658644.3690336>

- [42] Wang, J., Ma, Y., Xie, X., Du, X., & Zhang, X. (2025). PatchFuzz: Patch Fuzzing for JavaScript Engines. arXiv preprint arXiv:2505.00289
- [43] Zou, Y., Zhai, J., Fang, C., Liu, J., Zheng, T., & Chen, Z. (2024). Mutation-Based Deep Learning Framework Testing Method in JavaScript Environment. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 970-981
- [44] Wang, J., Qian, P., Huang, X., Ying, X., Chen, Y., Ji, S., ... & Liu, L. (2024, September). Tacoma: Enhanced Browser Fuzzing with Fine-Grained Semantic Alignment. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1174-1185. <https://doi.org/10.1145/3650212.368035>
- [45] Deng J., Zhu X., Xiao X., Wen S., Li Q., & Xia, S. (2021). Fuzzing With Optimized Grammar-Aware Mutation Strategies. IEEE Access, vol. 9, pp. 95061-95071. <https://doi:10.1109/ACCESS.2021.3093904>
- [46] Zhu, X., Liu, S., Li, X., Wen, S., Zhang, J., Seyit, C., & Xiang, Y. (2020). DeFuzz: Deep Learning Guided Directed Fuzzing. arXiv Preprint arXiv:2010.12149
- [47] Pham, V. T., Böhme, M., Santosa, A. E., Căciulescu, A. R., & Roychoudhury, A. (2019). Smart greybox fuzzing. IEEE Transactions on Software Engineering, 47(9), 1980-1997. <https://doi.org/10.1109/TSE.2019.2941681>
- [48] Srivastava, P., & Mathias P. (2021). Gramatron: Effective Grammar-Aware Fuzzing. In Proceedings of the 30th acm sigsoft international symposium on software testing and analysis, (pp. 244-256). <https://doi.org/10.1145/3460319.3464814>
- [49] Lin, P., Wang, P., Zhou, X., Xie, W., Zhang, G., & Lu, K. (2024). DeepGo: Predictive Directed Greybox Fuzzing. In Proceedings of the Network and Distributed System Security Symposium. <https://dx.doi.org/10.14722/ndss.2024.2451>
- [50] Xu, H., Zhao, Y., & Wang, H. (2025). Directed Greybox Fuzzing via Large Language Model. arXiv preprint arXiv:2505.03425
- [51] Zhang, Z., Chen, L., Wei, H., Shi, G., & Meng, D. (2024). Prospector: Boosting Directed Greybox Fuzzing for Large-Scale Target Sets with Iterative Prioritization. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1351-1363. <https://doi.org/10.1145/3650212.3680365>
- [52] Xiang, Y., Zhang, X., Liu, P., Ji, S., Liang, H., Xu, J., & Wang, W. (2024). Critical code guided directed greybox fuzzing for commits. In 33rd USENIX Security Symposium (USENIX Security 24), pp. 2459-2474
- [53] Wang, P., Zhou, X., Yue, T., Lin, P., Liu, Y., & Lu, K. (2024). The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. Software Testing, Verification and Reliability, 34(2), e1869. <https://doi.org/10.1002/stvr.1869>
- [54] Wang, Y., Jia, P., Liu, L., Huang, C., & Liu, Z. (2020). A systematic review of fuzzing based on machine learning techniques. PloS one, 15.8, e0237749. <https://doi.org/10.1371/journal.pone.0237749>

- [55] Cummins C., Petoumenos P., Murray A., Leather H. (2018). Compiler fuzzing through deep learning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 95–105. <https://10.1145/3213846.3213848>
- [56] Liu X., Li X., Prajapati R., Wu D. (2019). DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33, No. 01, pp. 1044-1051
- [57] Lee, S., Han, H., Cha, S. K., & Son, S. (2020). Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In 29th USENIX Security Symposium, USENIX Security 20, pp. 2613-2630
- [58] Sun X., Fu Y., Dong Y., Liu Z., Zhang Y. (2018). Improving Fitness Function for Language Fuzzing with PCFG Model. In: COMPSAC. IEEE, p. 655–660
- [59] Karamcheti S., Mann G., Rosenberg D. (2018). Improving Grey-Box Fuzzing by Modeling Program Behavior. arXiv preprint arXiv181108973
- [60] Karamcheti S., Mann G., Rosenberg D. (2018). Adaptive Grey-Box Fuzz-Testing with Thompson Sampling. In: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, p. 37–47
- [61] Stuby, J. (2024). Applying machine learning techniques for improving test input generation in embedded fuzzers [Bachelor's thesis]. University of Stuttgart
- [62] Huang, L., Zhao, P., Chen, H., & Ma, L. (2024). Large language models based fuzzing techniques: A survey. arXiv preprint arXiv:2402.00350
- [63] Xia, C. S., Paltenghi, M., Le Tian, J., Pradel, M., & Zhang, L. (2024). Fuzz4all: Universal fuzzing with large language models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1-13. <https://doi.org/10.1145/3597503.3639121>
- [64] Chen, C., Dolan-Gavitt, B., & Lin, Z. (2025). ELLFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. arXiv preprint arXiv:2506.10323
- [65] Abuzir, Y., & Khalil, D. R. A. (2025). Improving Network Security through Fuzzing Attack Detection: Machine Learning on the Kitsune Dataset. <https://doi.org/10.13189/csit.2025.130102>
- [66] Jha, S. R. S. K., & Pullum, A. R. L. (2017). Work-in-progress: testing autonomous cyber-physical systems using fuzzing features from convolutional neural networks. In: International Conference on Embedded Software (EMSOFT), IEEE, 1–2
- [67] Perry D. E., Porter A. A., Votta L. G. (2000). Empirical studies of software engineering: a roadmap. Proceedings of the conference on The future of Software engineering, pp. 345-355
- [68] Basili V. R. (1992). Software modeling and measurement: the Goal/Question/Metric paradigm. University of Maryland at College Park
- [69] Charlie, M. (2008). Fuzz by number. In CanSecWest Conference

- [70] Mitropoulos, C., Sotiropoulos, T., Ioannidis, S., & Mitropoulos, D. (2023). Syntax-Aware Mutation for Testing the Solidity Compiler. In European Symposium on Research in Computer Security, pp.327-347



ÖZGEÇMİŞ

Ad-Soyad : Şerafettin ŞENTÜRK

ÖĞRENİM DURUMU:

- **Lisans** : 2002, Yıldız Teknik Üniversitesi, Elektrik-Elektronik, Bilgisayar Mühendisliği
- **Yükseklisans** : 2005, Stuttgart Uygulamalı Bilimler Üniversitesi, Bilgisayar Mühendisliği, Yazılım Teknolojileri

MESLEKİ DENEYİM VE ÖDÜLLER:

- Kalyon Holding (2023- ...)
- TÜBİTAK Bilgem (2019-2023)
- Kuveyt Türk Katılım Bankası (2014-2019)
- ATOS (2012-2014)
- Seri Bilgi Teknolojileri (2011-2012)
- Hayat Holding (2007-2010)

TEZDEN TÜRETİLEN ESERLER:

- ŞENTÜRK, ŞERAFETTİN; GAROUSI, VAHİD; and YUMUŞAK, NEJAT (2025) "A case study of gray-box fuzzing with byte- and tree-level mutation strategies in XML-based applications for exposing security vulnerabilities," Turkish Journal of Electrical Engineering and Computer Sciences: Vol. 33: No. 2, Article 2.
- Şentürk Ş., Garousi V., Yumuşak N. (2025, 13-15, December). Comparison of Fuzz Tests According To Different Approaches. II. International Marmara Scientific Studies Congress.

DİĞER ESERLER:

- Şentürk, Şerafettin, Talha Apaydın, and Hasan Yaşar. "Image and File System Support Framework for a Digital Mobile Forensics Software." 2020 Turkish National Software Engineering Symposium (UYMS). IEEE, 2020.

- Şerafettin Şentürk, Hasan Yaşar, and İbrahim Soğukpınar. "Model Driven Security in a Mobile Banking Application Context". Proceedings of the 14th International Conference on Availability, Reliability and Security. 2019
- Sentürk Serafettin, Abdurrahman Akin, Ayşe Betül Karagöz and Vahid Garousi, "Model Based Testing In Practice: An Experience Report From The Banking Domain". Enstitüsü, İzmir Yüksek Teknoloji. "13. Ulusal Yazılım Mühendisliği Sempozyumu." (2019).
- Akin Abdurrahman, Serafettin Sentürk, and Vahid Garousi. "Transitioning from manual to automated software regression testing: Experience from the banking domain." 2018 25th Asia- Pacific Software Engineering Conference (APSEC). IEEE, 2018.
- Şerafettin Şentürk, Elif Yerli, and İbrahim Soğukpınar. "Email phishing detection and prevention by using data mining techniques." Computer Science and Engineering (UBMK), 2017 International Conference on. IEEE, 2017
- Kerem Ok, Şerafettin Şentürk, Serdar Aktaş, Cem Çevikbaş. "Secure Cryptographic Operations on SIM card for Mobile Financial Services", Financial Cryptography and Data Security, 2016
- Kalkan Kübra, Gürkan Gür, and Şerafettin Şentürk. "Dağıtık Servis Dışı Bırakma Saldırılarına Karşı Filtreleme Tabanlı Savunma Mekanizmaları", Akademik Bilişim, 2016
- Şerafettin Şentürk, Ahmet Bekir Bakkal, and Gürkan Gür. "Bulut Tabanlı Uçtan Uca Çevrimiçi İşlem İmzalama", Akademik Bilişim, 2016