ANDROID UYGULAMA KURULUM SÜRECİ İÇİN VERİMLİ BİR EVRİMSEL TABANLI FUZZ TESTİ

AN EFFICIENT EVOLUTIONARY-BASED FUZZING FOR ANDROID APPLICATION INSTALLATION PROCESS

VEYSEL HATAŞ

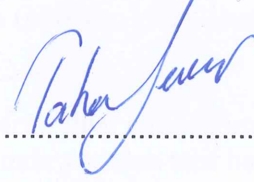
Doc. Dr. SEVİL ŞEN AKAGÜNDÜZ
Tez Danışmanı

Hacettepe Üniversitesi
Lisansüstü Eğitim-Öğretim ve Sınav Yönetmenliğinin
Bilgisayar Mühendisliği Anabilim Dalı için Öngördüğü
YÜKSEK LİSANS tezi olarak hazırlanmıştır.

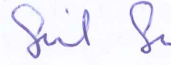
2018

Veysel HATAŞ'ın hazırladığı **"Android Uygulama Kurulum Süreci için Verimli Bir Evrimsel Tabanlı Fuzz Testi"** adlı bu çalışma aşağıdaki jüri tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Doc. Dr. Hüsrev Taha SENCAR
Başkan


....

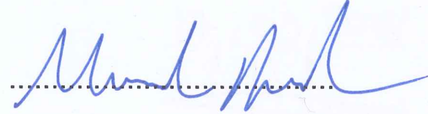
Doc. Dr. Sevil ŞEN AKAGÜNDÜZ
Danışman


....

Doc. Dr. Süleyman TOSUN
Üye

....

Dr. Murat AYDOS
Üye


....

Dr. Ayça TARHAN
Üye

....

Bu tez Hacettepe Üniversitesi Fen Bilimleri Enstitüsü tarafından **YÜKSEK LİSANS TEZİ** olarak onaylanmıştır.

Prof. Dr. Menemşe GÜMÜŞDERELİOĞLU
Fen Bilimleri Enstitüsü Müdürü

YAYINLAMA VE FİKRİ MÜLKİYET HAKLARI BEYANI

Enstitü tarafından onaylanan lisansüstü tezimin / raporumun tamamını veya herhangi bir kısmını, basılı (kağıt) ve elektronik formatta arşivleme ve aşağıda verilen koşullarla kullanıma açma iznini Hacettepe Üniversitesine verdiğimi bildiririm. Bu izinle Üniversiteye verilen kullanım hakları dışındaki tüm fikri mülkiyet haklarım bende kalacak, tezimin tamamının ya da bir bölümünün gelecekteki çalışmalarda (makale, kitap, lisans ve patent vb.) kullanım hakları bana ait olacaktır.

Tezin kendi orijinal çalışmam olduğunu, başkalarının haklarını ihlal etmediğimi ve tezimin tek yetkili sahibi olduğumu beyan ve taahhüt ederim. Tezimde yer alan telif hakkı bulunan ve sahiplerinden yazılı izin alınarak kullanılması zorunlu metinlerin yazılı izin alınarak kullandığımı ve istenildiğinde suretlerini Üniversiteye teslim etmeyi taahhüt ederim.

Yükseköğretim Kurulu tarafından yayınlanan **" Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge"** kapsamında tezim aşağıda belirtilen koşullar haricinde YÖK Ulusal Tez Merkezi / H. Ü. Kütüphaneleri Açık Erişim Sisteminde erişime açılır.

o Enstitü / Fakülte yönetim kurulu kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren 2 yıl ertelenmiştir. (1)

o Enstitü / Fakülte yönetim kurulunun gerekçeli kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren .... Ay ertelenmiştir. (2)

o Tezimle ilgili gizlilik kararı verilmiştir. (3)

17/09/2018

(İmza)

Öğrencinin Adı SOYADI

"Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge"

(1) Madde 6. 1. Lisansüstü tezle ilgili patent başvurusu yapılması veya patent alma sürecinin devam etmesi durumunda, tez danışmanının önerisi ve enstitü anabilim dalının uygun görüşü üzerine enstitü veya fakülte yönetim kurulu iki yıl süre ile tezin erişime açılmasının ertelenmesine karar verebilir

(2) Madde 6. 2. Yeni teknik, materyal ve metotların kullanıldığı, henüz makaleye dönüşmemiş veya patent gibi yöntemlerle korunmamış ve internetten paylaşılması durumunda 3. Şahıslara veya kurumlara haksız kazanç imkanı oluşturabilecek bilgi ve bulguları içeren tezler hakkında tez danışmanının önerisi ve enstitü anabilim dalının uygun görüşü üzerine enstitü ve fakülte yönetim kurulunun gerekçeli kararı ile altı ayı aşmamak üzere tezin erişime açılması engellenebilir.

(3) Madde 7. 1. Ulusal çıkarları veya güvenliği ilgilendiren, emniyet, istihbarat, savunma ve güvenlik, sağlık vb. konulara ilişkin lisansüstü tezlerle ilgili gizlilik kararı, tezin yapıldığı kurum tarafından verilir*. Kurum ve kuruluşlarla yapılan işbirliği protokolü çerçevesinde hazırlanan lisansüstü tezlere ilişkin gizlilik kararı ise, ilgili kurum ve kuruluşun önerisi ile enstitü veya fakültenin uygun görüşü üzerine üniversite yönetim kurulu tarafından verilir. Gizlilik kararı verilen tezler Yükseköğretim Kuruluna bildirilir.
Madde 7. 2. Gizlilik kararı verilen tezler gizlilik süresince enstitü veya fakülte tarafından gizlilik kuralları çerçevesinde muhafaza edilir, gizlilik kararının kaldırılması halinde Tez Otomasyon Sistemine yüklenir.

* Tez danışmanının önerisi ve enstitü anabilim dalının uygun görüşü üzerine enstitü veya fakülte yönetim kurulu tarafından karar verilir.

ETİK

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü, tez yazım kurallarına uygun olarak hazırladığım bu tez çalışmasında,

- tez içindeki bütün bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- görsel, işitsel ve yazılı tüm bilgi ve sonuçları bilimsel ahlak kurallarına uygun olarak sunduğumu,
- başkalarının eserlerinden yararlanılması durumunda ilgili eserlere bilimsel normlara uygun olarak atıfta bulunduğumu,
- atıfta bulunduğum eserlerin tümünü kaynak olarak gösterdiğimi,
- kullanılan verilerde herhangi bir tahrifat yapmadığımı,
- ve bu tezin herhangi bir bölümünü bu üniversitede veya başka bir üniversitede başka bir tez çalışması olarak sunmadığımı

beyan ederim.

17/09/2018

VEYSEL HATAŞ

ÖZET

Android Uygulama Kurulum Süreci için Verimli Bir Evrimsel Tabanlı Fuzz Testi

Veysel HATAŞ
Yüksek Lisans, Bilgisayar Bölümü
Tez Danışmanı: Doc. Dr. Sevil ŞEN AKAGÜNDÜZ
Haziran 2018, 90 sayfa

Uygulama güvenliğini sağlamak için uygulama geliştirme ve test aşamalarında kullanılan kaynak kod analizi temelli otomatik araçlar yetersiz kalmaktadır. Bu nedenle araştırmacılar, uygulamalar üzerinde zafiyet analizi yapmak ve yazılım hatalarını ortaya çıkarmak için fuzz testi yöntemlerine başvurmaktadırlar. İyi konfigüre edilmiş fuzz testi güvenlik araçları ile yapılan açık kutu ve/veya kapalı kutu fuzz testleri ve tersine mühendislik analizleri sayesinde birçok ciddi güvenlik açığı ortaya çıkartılıp, gerekli yamalar zamanında yapılabilmektedir.

Son yıllarda, mobil platformun gelişmesi ile mobil uygulamalar ve Android işletim sistemi üzerinde fuzz testleri yapılmaya başlanmıştır. Şu ana kadar yapılmış olan fuzz testi çalışmalarının çoğu uygulama seviyesinde ve Android uygulamalarının grafik arayüzleri hedef alınarak gerçekleştirilmişlerdir. Daha geniş kullanıcı kitlesini etkileyen ve kritik olan işletim sistemi seviyesindeki kütüphanelerdeki zafiyetlerin tespiti için yapılan çalışmalar çok yenidir.

Bu tez çalışmasında, Android işletim sistemindeki güvenlik zafiyetleri bulmak için yeni bir fuzz testi yöntemi önerilmiştir. Önerilen genetik algoritma tabanlı kapalı kutu fuzz testi yöntemi ile Android sistemindeki zafiyetleri otomatik olarak bulmak için bir platform sunulmuş ve benzer

çalışmalar ile karşılaştırılmıştır. Önerilen yöntem ile yeni güvenlik zafiyetlerinin, diğer çalışmalardan çok daha kısa sürede bulunabildiği gösterilmiştir.

Anahtar Kelimeler: Android, zafiyet tespiti, fuzz testi, sıfırıncı gün saldırısı, evrimsel hesaplama, genetik algoritma

ABSTRACT

An Efficient Evolutionary Based Fuzzing for Android Application Installation Process

Veysel HATAŞ
Master, Department of Computer Engineering
Supervisor: Doc. Dr. Sevil ŞEN AKAGÜNDÜZ
June 2018, 90 pages

Automated tools based on source code analysis used in application development and test phases are insufficient to ensure application security. For this reason, researchers also resort to fuzz testing methods to analyze vulnerability in applications and to reveal software faults. With a well-configured fuzz test security tools and white-box and/or black-box fuzz tests and reverse engineering analysis, many serious security vulnerabilities can be uncovered and the required patches made in a timely manner.

In recent years, with the development of mobile platform, fuzz tests have been started on mobile applications and Android operating system. Much of the fuzz testing work that has been done so far is implemented at the application level and GUI based. The work done to determine vulnerabilities in operating system level libraries that affect and critical the wider user base is very new.

In this thesis, a new fuzzy test method is proposed to find the security vulnerabilities in the Android operating system. A proposed genetic algorithm-based black-box fuzz test method is presented as a platform for automatically detecting vulnerabilities in the Android system and compared with similar studies. The proposed method has shown that new

security vulnerabilities can be found in a much shorter time than other studies.

Keywords: Android, vulnerability detection, fuzzing, zero day attack, evolutionary computation, genetic algorithm

TEŞEKKÜR

Tez çalışmam boyunca kıymetli desteklerini esirgemeyen aileme ve tez hazırlığımın bütün aşamalarına değerli bilgi ve deneyimleriyle ışık tutan danışmanım Doc. Dr. Sevil ŞEN AKAGÜNDÜZ'e sonsuz teşekkürlerimi sunuyorum.

İÇİNDEKİLER

Sayfa

ÖZET i
ABSTRACT.... iii
TEŞEKKÜR....v
İÇİNDEKİLER vi
ŞEKİLLER.... viii
TABLOLAR ix
ÇİZELGELER x
KISALTMALAR....xi
1. GİRİŞ 1
2. ÖN BİLGİ 4
2.1. Android İşletim Sistemi 4
2.2. Uygulama Kurulum Süreci 6
2.2.1. DEX Dosya Formatı 7
2.3. Zafiyetler....9
2.3.1. Bellek Taşması....10
2.3.2. Katar Dizgisi Biçimlendirme 12
2.4. Fuzz Testi....13
2.4.1. Fuzz Testi Sınıflandırması 13
2.4.1.1. Açık Kutu....13
2.4.1.2. Kapalı Kutu....14
2.4.1.3. Gri Kutu 14
2.4.2. Fuzz Testi Yöntemleri....14
2.4.2.1. Mutasyon Tabanlı Fuzz Testi....17
2.4.2.2. Üretim Tabanlı Fuzz Testi 18
2.4.2.3. Vekil tabanlı Fuzz Testi 19
2.4.2.4. Evrimsel Tabanlı Fuzz Testi 20
2.5. Evrimsel Hesaplama 20
2.5.1. Genetik Algoritma 20
2.5.1.1. Seçim....22
2.5.1.2. Çaprazlama 23
2.5.1.3. Mutasyon....23
2.5.1.4. Uygunluk Fonksiyonu....24
3. İLGİLİ ÇALIŞMALAR....25
3.1. Evrimsel Hesaplama Tabanlı Fuzz Testleri 26
3.2. Android Platformunda Fuzz Testleri 28
3.2.1. Android'te Evrimsel Hesaplama Tabanlı Çalışmalar 32
3.2.2. Uygulama Kurulum Sürecine Yönelik Çalışmalar 32
4. GADROID 37
4.1. İlk Veri Üretimi....38
4.2. Bozulmuş DEX Dosya Formatı Tamiri 38

4.3. Fuzz Testi Süreci 40
4.4. Kayıtların Toplanması ve Analizi 40
4.5. Uygunluk Fonksiyonu Bölümleri 42
5. DENEY SONUÇLARI 46
6. SONUÇ 51
7. KAYNAKLAR 53
8. EKLER 60
8.1. Ek 1 60
8.2. Ek 2 68
ÖZGEÇMİŞ 69

ŞEKİLLER

Sayfa

Şekil 1. Android mimarisi [5] 6
Şekil 2. APK içeriği [8] 8
Şekil 3. DEX dosya formatı bölümleri [9][10] 9
Şekil 4. Harvard Mark II (1947) - raporlanan ilk hata [11] 10
Şekil 5. Örnek bir bellek taşması zafiyeti 11
Şekil 6. Örnek bir katar dizgisi biçimlendirme zafiyetinin görünümü 12
Şekil 7. Fuzz testi tarihi [20] 15
Şekil 8. Fuzz testi fazları 16
Şekil 9. Charlie Miller'ın Python fuzz testi kodu 17
Şekil 10. Bit kaydırma yöntemi örneği 18
Şekil 11. Üretim tabanlı fuzz testi örneği 19
Şekil 12. Genetik algoritma gösterimi ve işleçleri 21
Şekil 13. Genetik algoritma akış şeması 22
Şekil 14. Genetik çaprazlama örneği 23
Şekil 15. Mutasyon örneği 24
Şekil 16. GAdroid'in adımları 38
Şekil 17. Değiştirilmemesi ve tamir edilmesi gereken DEX bölümleri 39
Şekil 18. Bir bireye ilişkin örnek bir kayıt 41
Şekil 19. Markov kontrol akış grafiği 44
Şekil 20. DEX dosyaları üzerinde çaprazlama işleci 45

TABLOLAR

Sayfa

Tablo 1. Toplam program çakılma sayısına göre karşılaştırma....................47
Tablo 2. Tekil program çakılma sayısına göre karşılaştırma........................48
Tablo 3. Sıfırıncı-gün zafiyeti keşfine göre literatür karşılaştırması..........49
Tablo 4. Zamana göre grogram çakılması analizi grafiği 49

ÇİZELGELER

Sayfa

Çizelge 1. Sunulan literatür taramasının özeti 35
Çizelge 2. Literatürün strateji olarak karşılaştırılması 36

KISALTMALAR

Kısaltmalar	
GA	Genetik Algoritma
Fuzzing	Fuzz Testi
PC	Program Counter
AOSP	Android Open Source Project
ART	Android Run Time
AFL	American Fuzzy Loop
d’ART	Digging Into the Android L Runtime Internals
CVE	Common Vulnerabilities and Exposures
APK	Android Application Package
DEX	Dalvik Executable
Bug	Yazılım Hatası
EFS	Evolutionary Fuzzing System
UI	User Interface
GAdroid	Genetic for Android
OSVDB	Open Sourced Vulnerability Database
NVD	National Vulnerability Database
HTML	HyperText Markup Language
RFC	Request For Comments

1. GİRİŞ

Video görüşmeleri, internet kullanımı, mesajlaşma gibi iletişim ve haberleşmeye olan talebin artmasıyla teknolojinin de mobilleşmeye doğru evrildiği görülmektedir. 2017 başlarında yayınlanan *"We Are Social"* araştırma verilerine göre [1] Dünya'da internete bağlı mobil cihaz sayısı 8 milyar iken, Türkiye'deki mobil cihaz sayısı 71 milyonu aşmıştır. Gelişen teknoloji ile birlikte akıllı telefon, saat, tablet, vb. gibi mobil cihazların kullanımındaki hızlı artış, bu cihazlardaki verilerin güvenliğinin sağlanması için çalışmaları da hızlandırmıştır.

Mobil cihazların artması ile bu cihazlar kötü niyetli yazılım korsanları ve gizli istihbarat örgütlerinin de hedefi haline gelmiştir. Varolan işletim sistemlerine ilişkin kişisel verilerin sızdırılması, yetkisiz erişim, hizmet dışı bırakma, izlenme vb. gibi güvenlik risklerinin hepsi mobil işletim sistemleri için de geçerlidir. Kodlama hataları nedeniyle oluşan güvenlik zafiyetleri, zararlı yazılımlar tarafından sömürülerek mobil cihazların farklı amaçlar doğrultusunda kullanılmasına, ele geçirilmesine ve bilgi sızıntılarına sebep vermektedir. Bunun neticesinde de can, mal ve itibar kayıplarına neden olunmaktadır. Dolayısıyla, milyarlarca insanı etkileyen bu yazılım hatalarının yazılım korsanlarından önce tespit edilmesi ve kapatılması önem kazanmaktadır. Bu konuda son yıllarda araştırmalar yapılmaya başlanmıştır.

Mobil işletim sistemlerinin en popüleri olan ve 2005 yılında ortaya çıkan Android, ilk çıkışında dijital fotoğraf makinaları için geliştirilmiştir. Sonrasında, modifiye edilmiş Linux 2.6.25 çekirdeği üzerinde tasarlanarak mobil işletim sistemi olarak kullanılmaya başlanmıştır. Günümüzde 8.0 *(Android O)* beta versiyonuna ulaşan Android işletim sistemi için Google Play'de bulunan 1,5 milyon uygulama geliştirilmiş ve bu uygulamaları indiren kullanıcı sayısı 2 milyarı geçmiştir [2]. Bu kadar yaygın kullanılmasına rağmen, her geçen gün Android işletim sistemine ilişkin yeni zafiyetler ortaya çıkmaktadır. 2016 yılı Android Güvenlik

Bültenine göre [3]; son bir yılda 655 adet Android sistemine ilişkin güvenlik zafiyeti bulunmuştur. Bu zafiyetlerin tespiti için fuzz testi en çok kullanılan yöntemlerden birisidir.

Fuzz testleri, dizüstü bilgisayarlardan gömülü akıllı batarya sistemlerine kadar çok çeşitli platformlar üzerindeki güvenlik açıklıklarının tespiti için uygulanırlar [3]. Android sistemi üzerinde de son yıllarda fuzz testleri gerçekleştirilmektedir. Android'in geniş kitlelere yayılmasıyla birlikte güvenlik araştırmacıları da son 5 yılda yoğun bir şekilde zafiyet araştırma çalışmalarıyla bu alana yönelmişlerdir. Android platformunda ilk çalışmalardan birisi olan Sulley fuzzer [4], bozulmuş SMS mesajlarıyla stres testlerini gerçekleştiren, ilginç fuzz testi uygulamalarından birisidir. Fakat, Android kurulum süreci üzerinde çok az sayıda fuzz testi gerçekleştirilmiştir. Android kurulum süreci üzerinde fuzz testi yapılmasının en önemli avantajı; Android uygulamalarından bağımsız olması ve uygulamalar üzerinde yapılan fuzz testi çalışmalarına göre çok daha kritik olan daha alt seviyelerde zafiyetlerin tespitini gerçekleştirebilme potansiyelidir. Böylece çok daha geniş kitlelerin etkilendiği Android çekirdeğinde zafiyetlerin bulunması ile işletim sisteminin versiyonundan bağımsız zafiyetlerin tespitinin yapılması sağlanabilmektedir.

Bu tez çalışmasında, Android kurulum süreci üzerinde fuzz testleri gerçekleştirebilmek için genetik algoritma tabanlı bir yaklaşım önerilmiş ve başarımı değerlendirilmiştir. GAdroid ismi verilen bu yaklaşımda, genetik algoritmadan faydalanılarak üretilen girdilerle test süreçleri kısaltılmış ve hedef uygulamada daha önce keşfedilmemiş zafiyetlerin tespitine odaklanılmıştır. Android emülatörü kullanılarak hazırlanan test platformu üzerinde 2 haftalık test süreci sonucunda 20.452 tane yazılım çakılması elde edilmiş, bunlardan 305 tanesinin tekil olduğu görülmüştür. Bu yazılım çakılmaları arasında kritik olarak tespit edilen 11 zafiyetten bir tanesinin sıfırıncı gün açıklığı olduğu belirlenmiş ve bu zafiyetin bildirimi için uluslarası zafiyet veri tabanlarına başvuruda bulunulmuştur. Ayrıca

literatürdeki benzer çalışmalara göre daha kısa sürede daha çok güvenlik zafiyeti tespit edebildiği gösterilmiştir.

Bölüm 2'de, zafiyet tespit yöntemleri ve bu amaç doğrultusunda kullanılan fuzz testi türleri ve araçları hakkında genel bilgi verilecek ve genetik algoritmaya girdi olarak verilen DEX dosya formatı açıklanacaktır. Ayrıca tezde kullanılan evrimsel hesaplama ve genetik algoritma yöntemleri ile zafiyet tespiti hakkında temel bilgiler verilecektir. Bölüm 3'te, literatürde bulunan ilişkili çalışmalar özetlenmiştir. Bölüm 4'te ise bu çalışma kapsamında sunulan GAdroid'in yapısı ve temel adımları açıklanmış, diğer çalışmalara göre benzer ve farklı yönleri vurgulanmıştır. Bölüm 5'te, önerilen yöntemin değerlendirilmesi için yapılan deneyler açıklanmış ve deney sonuçları sunulmuştur. Bölüm 6'da bu tez çalışması kapsamında yapılan çalışmalar özetlenerek tezin literatüre katkısı sunulmuştur.

2. ÖN BİLGİ

Bu bölümde, tez çalışmasında önerilen GAdroid yaklaşımının daha iyi anlaşılması için bazı temel bilgiler verilecektir. Öncelikle, Android'in temel yapısı ve GAdroid'e girdi olarak verilen DEX dosyasının formatı açıklanacaktır. Sonrasında, fuzz testi ve farklı fuzz testi yöntemlerinden kısaca bahsedilecektir. Son olarak, bu tez çalışmasında önerilen evrimsel-tabanlı fuzz testinin kullandığı, en popüler evrimsel hesaplama yöntemlerinden birisi olan genetik algoritmaların temel adımları ve temel yapı taşları tanıtılacaktır.

2.1. Android İşletim Sistemi

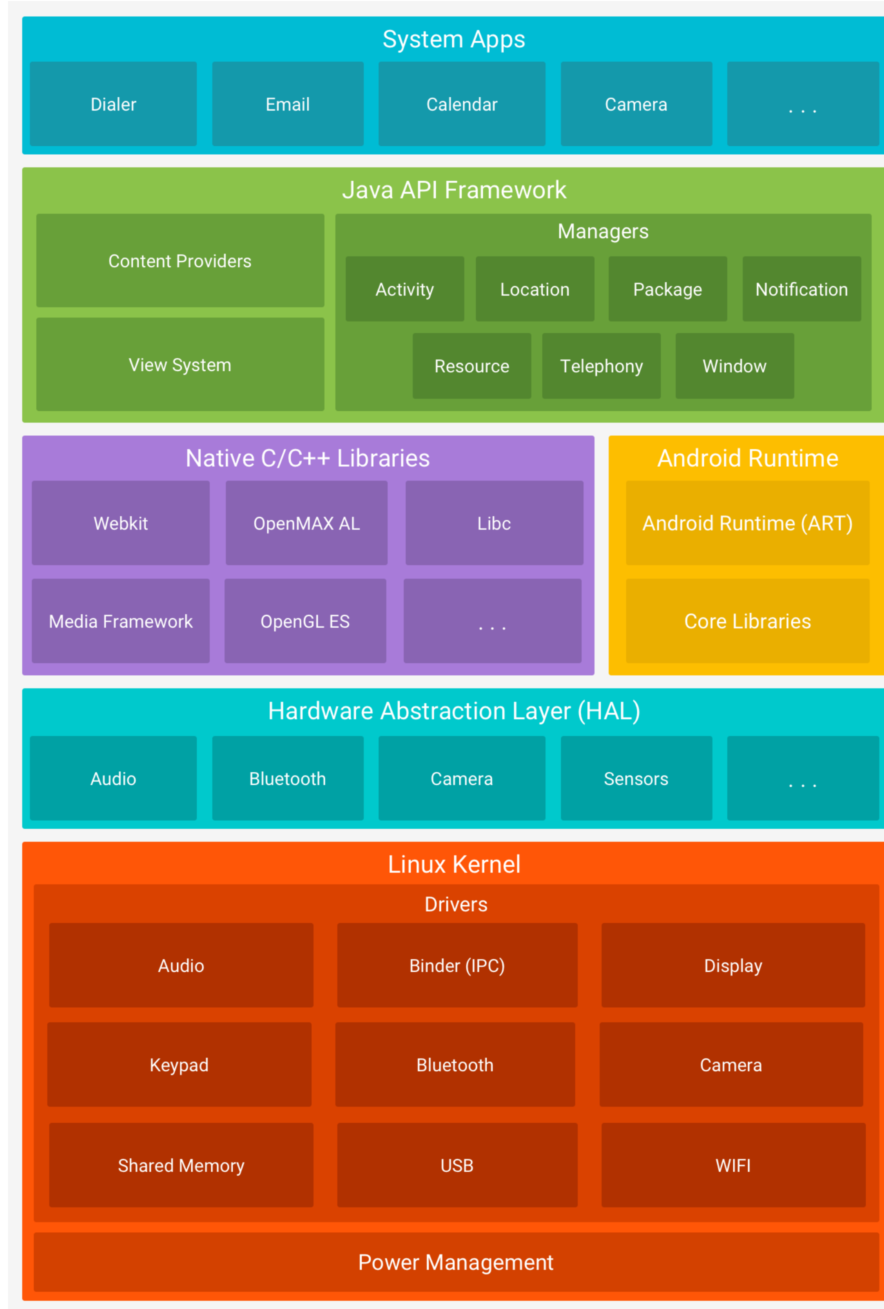
Bu tez çalışmasında, Android uygulamalarının kurulumu aşamasında kullanılan derleyici üzerinde güvenlik riskleri oluşturabilecek zafiyetlerin tespit edilmesi amaçlanmıştır. Android DEX dosyaları üzerinden oluşturulan test durumlarının daha iyi anlaşılması için bu bölümde Android işletim sistemi hakkında ön bilgi verilecektir.

Mobil işletim sistemlerinin en popüleri olan ve 2005 yılında ortaya çıkan Android, ilk çıkışında dijital fotoğraf makinaları için geliştirilmiştir. Daha sonra modifiye edilmiş Linux 2.6.25 çekirdeği üzerinde tasarlanarak mobil işletim sistemi olarak kullanılmaya başlanılmıştır. Günümüzde 8.0 *(Android O)* beta versiyonuna ulaşan Android işletim sistemi üzerinde Google Play'de bulunan 1,5 milyon uygulama geliştirilmiş ve bu uygulamaları kullanan sayısı 2 milyarı geçmiştir [2].

Temelinde modifiye edilmiş Linux çekirdeği üzerine inşa edilmiş olan Android, her yeni versiyonda az da olsa üst katmanlarında da değişiklikler yapmaktadır. Bu kod eklentileri veya yama işlemi ile bir yandan kodun işlevselliği arttırılırken, diğer bir yandan yazılım hatalarından kaynaklanabilecek yeni güvenlik zafiyetleri oluşturulabilmektedir. Araştırmacılar ise daha önce keşfedilmemiş bu zafiyetleri zamanında ortaya çıkararak, güvenlik risklerini engellemeye çalışmaktadırlar.

Android’in temel mimarisi Şekil *1*’de verilmiştir. Android’in alt katmanında Linux çekirdeği üzerinde, çoğunlukla C programlama dili ile yazılmış yerel kodların yer aldığı *Android Runtime* bulunmaktadır. Bu katman, üst seviyedeki komutları donanım seviyesindeki *HAL* katmanına aktarmaktadır. Burada temel kütüphaneler ve her uygulamanın birbirinden izole edilerek ayrı çalıştırıldığı sanal makineler yer almaktadır. Android JAVA katmanında ise Android uygulamaları ile çekirdek arasında haberleşmeyi sağlayan JAVA kütüphaneleri ve API’leri yer almaktadır [5].

Şekil 1. Android mimarisi [5]



2.2. Uygulama Kurulum Süreci

Android'in uygulamalarının kurulumundan sorumlu varsayılan uygulaması *PackageInstaller*'dır. Bu uygulama, kullanıcıdan kurulum komutlarını almak için *InstallAppProgress* aktivitesini çağırır. Bu aktivite de *Package Manager Service* isimli servisi çağırmaktadır. Android kurulum süreci, gelişen teknoloji ile birlikte zamanla değişim göstermektedir.

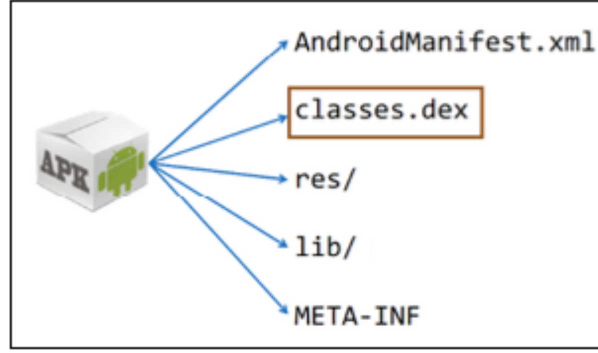
2015 yılında Androd 5.0 ile birlikte gelen ve halen kullanılmakta olan ART *(Android Runtime),* düşük hafızalı cihazlarda birden çok sanal makinenin çalıştırılabilmesi için geliştirilmiştir. ART, uygulamalara ilişkin kaynak kodları, Android platformunda çalışabilen DEX bayt koduna çevirmektedir [6]. Uygulamaların kurulum aşamasında ise *dex2oat* aracı kullanılır. *dex2oat*, .dex uzantılı DEX dosyalarını girdi olarak kabul eder ve uygulamaları derleyerek .oat uzantılı bir dosya oluşturur. Bu çevirim işlemi ile, işlemci tarafından doğrudan yerel olarak çalıştırılabilecek yerel kodlar oluşturulur. Bu çalışmada, 2013 yılında yayınlanan *Android 4.4 KitKat* ile Dalvik sanal makinesinde kullanılan dexopt yerine, daha güncel olduğu için 2015 yılında yayınlanan *Android 5 Lollipop* ile ART yapısında kullanılan *dex2oat* metodu tercih edilmiştir. Bunun en önemli nedeni, ART ile DEX dosyalarının komut satırı üzerinden adb [7] aracılığı ile doğrudan çalıştırılabilir olmasıdır.

2.2.1. DEX Dosya Formatı

Bu tez çalışmasında, genetik algoritmaya girdi olarak DEX dosyaları verilmektedir. Önerilen yöntemde, genetik algoritma ile evrimleştirilen DEX dosyaları ile farklı fuzz testlerinin uygulanması sağlanmış ve bu testler ile yeni güvenlik zafiyetlerinin keşfedilmesi amaçlanmaktadır. Bu nedenle, bu bölümde DEX dosya formatı anlatılacaktır.

Android DEX dosya formatı, 2015 yılında Intel tarafından gerçekleştirilen bir proje kapsamında geliştirilmiştir. *Şekil 2.* APK içeriği görüldüğü üzere Android paketlenmiş uygulamaları, bir APK paketinin içerisinde yer almaktadır. *AndroidManifest.xml* isimli manifest dosyasında, uygulama tarafından kullanılan izinler, uygulamanın bileşenleri vb. gibi bilgiler tanımlanmaktadır. *META-INF* klasörü, APK'ya ilişkin özel imza bilgisini içerir. *lib* dizini, eğer var ise, uygulamaya has derlenmiş yerel Android kütüphanelerini içermektedir. *res* klasörü, uygulamaya ait olan her türlü kaynağı (resimler, dil dosyaları, vb. gibi) içermektedir. Son olarak *classes.dex* dosyası ise derlenmiş uygulama kodlarını barındırmaktadır.

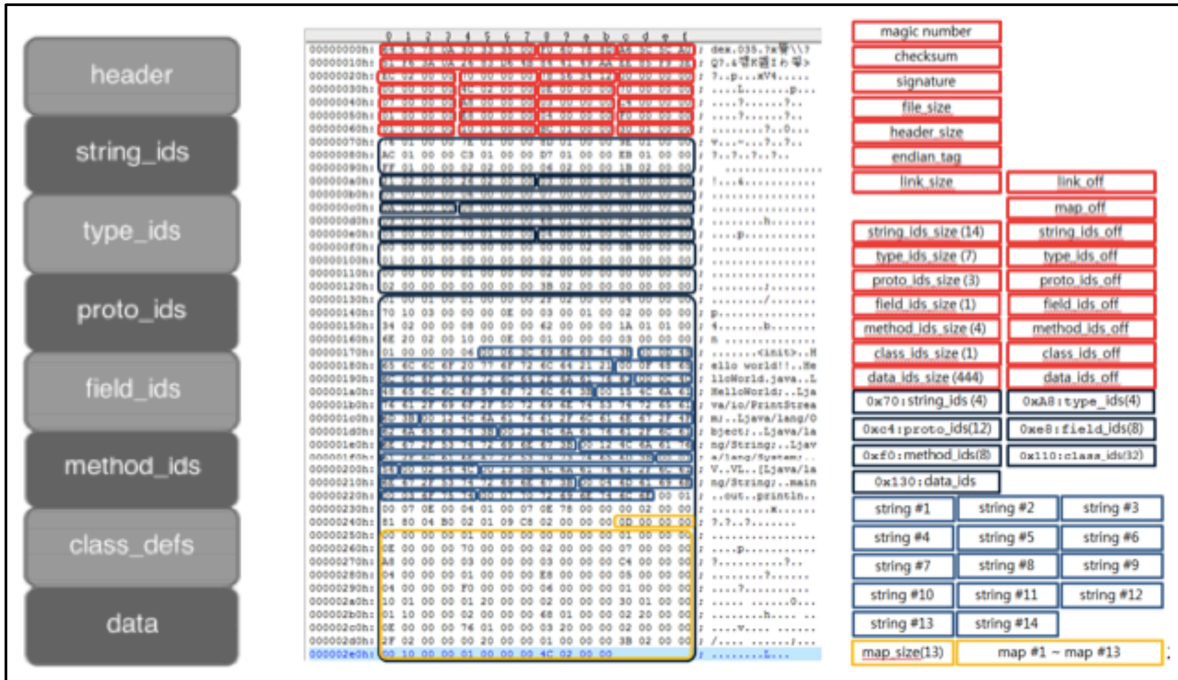
Şekil 2. APK içeriği [8]



Android platformu için yazılan bir kod, .dex uzantılı bir dosya *(Dalvik Executable)* olarak derlenmelidir. DEX dosyaları, sanal bir makine içinde kullanılırlar. Şekil *3*. DEX dosya formatı bölümleri [9-10]'te gösterilen DEX dosya formatının başlık alanında yer alan bölümler aşağıda açıklanmıştır [9];

- string_ids – DEX dosyası içerisinde kullanılan bütün katar dizgilerinin listesini içerir ve referanslarını tutar.
- type_ids – DEX dosyası içerisinde kullanılan bütün türlerin *(sınıflar, diziler, temel türler)* listesini tutar.
- proto_ids – DEX dosyası tarafından referans edilen bütün prototiplerin tanımlarını içerir. *(Örn; int fn(double), void fn())*
- field_ids –DEX dosyası tarafından belirtilen tüm alanlar için tanımlayıcıların listesini tutar. *(Örn; integer.MAX_VALUE)*
- method_ids – DEX dosyası tarafından referans edilen metodları içerir.
- class_defs –hem DEX dosyasının erişebileceği metod ve kodlar için hem de tanımlı DEX dosyası için bütün sınıf tanımlarını içerir.

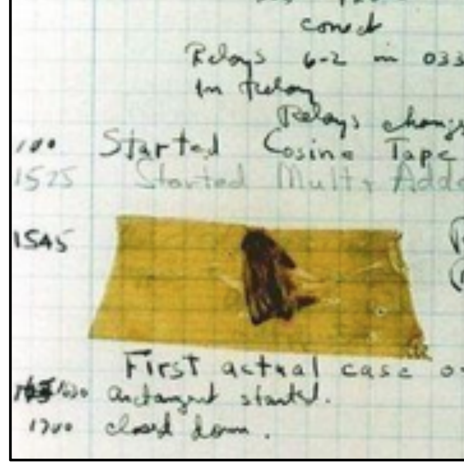
Şekil 3. DEX dosya formatı bölümleri [9-10]



2.3. Zafiyetler

Yazılımcılar tarafından geliştirilen uygulamalar üzerinde yazılım hatalarından dolayı birçok zafiyet bulunabilmektedir. Hata anlamında kullanılan *bug* kavramı, Harvard Mark II (1947) tarafından gerçek bir böceğin neden olduğu sistem odasındaki sunucuların kısa devre ile erişim kısıtı hatasıyla ortaya çıkmış ve günümüzde bilişim termolojisi olarak kullanılmaktadır [11].

Şekil 4. Harvard Mark II (1947) - raporlanan ilk hata [11]

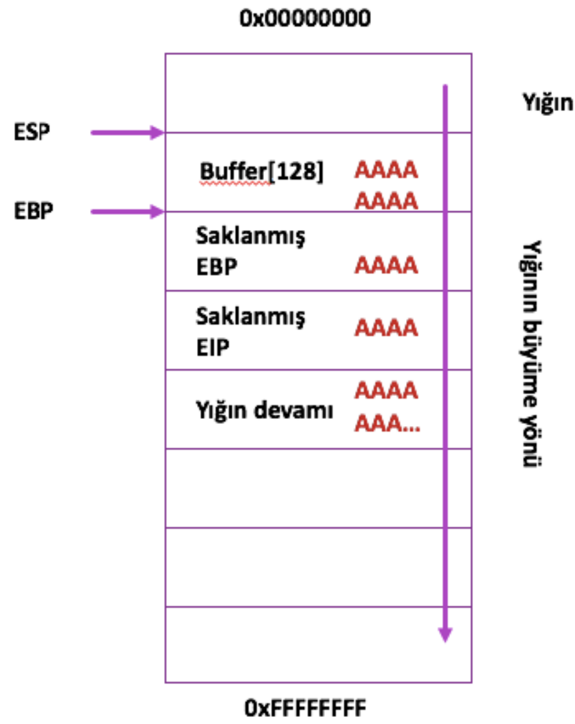


Bu yazılım hatalarının bazıları, saldırganlar tarafından zararlı kod çalıştırılabilmesine neden olan güvenlik zafiyetlerine neden olmaktadırlarlar. Bu güvenlik zafiyetlerinden en yaygın olanlarını, işletim sisteminden bağımsız olarak iki gruba ayırabiliriz: bellek taşması ve katar dizgisi biçimlendirme *(format string).*

2.3.1. Bellek Taşması

Bellek taşması, bir yazılımın erişmeye yetkisinin olmadığı bellek alanlarına veri yazmasıdır. Bunun temel nedeni, yazılım geliştirilirken gerekli sınır kontrollerinin yapılmamasıdır. Örneğin kullanılan bir dizinin sınır kontrolünün yapılmaması sonucu, dizinin bulunduğu bellek dışına veri yazılmasının yolu açılmış olur. Bu tür bir zafiyet içeren yazılım sömürülerek, kritik yazmaçlara veri yazılıması ve program akışının değiştirilmesi sağlanabilir. Yazılımların bellek taşma zafiyetlerinden faydalanılarak, yetki yükseltme, arka kapı oluşturma, vb. gibi saldırılar gerçekleştirilebilir.

Şekil 5. Örnek bir bellek taşması zafiyeti



Şekil *5*. Örnek bir bellek taşması zafiyeti örnek bir bellek taşması gösterilmektedir. *buffer()* fonksiyonun çağrıldıktan sonraki bellek görüntüsü gösterilmektedir. Bu fonksiyonda sınır kontrolü yapılmadığı için, aynı bellek bölgesi A harflerinden oluşan katar dizgileri ile doldurulmuştur. Böylelikle geri gönüş adresi ve EBP yazmaçlarının üzerlerine yazılmıştır. EIP, bellekteki çalıştırılacak bir sonraki komut satırını göstermektedir. Bu değerin üzerine yazılması, program akışının değiştirilebileceği, yani kontrol altına alınabileceği anlamına gelmektedir. Dahası, bu değerler uygun adres değerleri ile güncellenebilirse, program akışı zararlı kodun yer aldığı bellek bölgesine yönlendirilebilir.

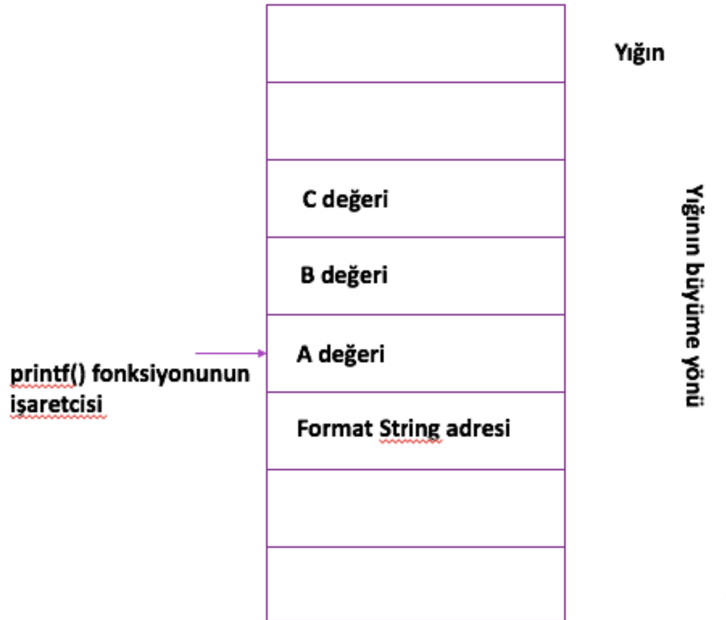
Bellek taşması, gerçekleştiği bellek bölümüne göre yığıt ve yığın olmak üzere ikiye ayrılır. Yığıt taşması, yığıt adı verilen bellek bölgesinde sınır kontrolü yapılmayan dizi ya da girdi bölümlerine yapılan uygun olmayan girdilerin verilmesi ile, program akış kontrolünün ele geçirilmesidir. Yığın taşması, benzer şekilde yığın bölgesinde olmasıyla beraber, dinamik

değişkenlerin, dinamik bellek alınmasını sağlayan *malloc()* ve *calloc()* fonksiyonlarının kullanılmasıyla tetiklenebilmektedir [12-14].

2.3.2. Katar Dizgisi Biçimlendirme

C programlama dilinde, ekrana çıktı basılmasını sağlayan *printf* ve benzeri fonksiyonların amacı dışında kullanılmasıyla bellekteki bilginin ifşa edilebilmesi zafiyetidir. Bu zafiyet zararlı kod çalıştırılabilmesine neden olabilmektedir [15]. *printf()* ailesinde yer alan *printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf* fonksiyonlarda *%x, %s, %d* gibi formatlı girdi gönderilidiğinde zafiyet tetiklenebilmektedir [16]. Şekil *6*. Örnek bir katar dizgisi biçimlendirme zafiyetinin görünümü görüldüğü gibi örnek bir katar dizgisi biçimlendirme zafiyeti yer almaktadır. Bellek alanında yığının büyümesi yönünde printf() işaretçisinin gösterdiği yerden itibaren illegal karakter girdisi ile diğer bellek alanlarına müdahale edilebilmektedir. Böylece EIP göstergecine müdahale ile program akışı kontrol altına alınılabilmektedir.

Şekil 6. Örnek bir katar dizgisi biçimlendirme zafiyetinin görünümü



2.4. Fuzz Testi

Mevcut uygulamalar içerisinde bulunan bellek bozulması, bellek taşması, kod enjeksiyonu, bellek sızıntıları, vb. gibi zafiyetlerin tespit edilebilmesi için, literatürde birçok teknik ve yöntem kullanılmaktadır. Birçok çeşit zafiyeti ortaya çıkarabilmek için tek bir yöntem kullanılmamaktadır. Zafiyetlerin ortaya çıkarılması için kullanılan yöntemlere genel olarak fuzz testi denilmektedir.

Bunun dışında uygulama geliştiriciler fuzz testi dışında kaynak kod analizine de başvurmaktadırlar. Kaynak kod analizi, uygulamaların güvenlik bakış açısıyla derinlemesine incelenmesidir. Kaynak kod analizi, kod denetimiyle benzerlik gösterse de temel fark; kod denetiminde genelde yazılımın üretimi sonrası yazılım hatalarının gözden geçirilmesi için başvurulan yöntem iken, kaynak kod analizi ise yazlımın kullanıcıya sunulması sonrasında dahi başvurulabilen, genelde ortaya çıkan yazılım çakılmalarının tespiti ve kök nedeninin ortaya çıkarılabilmesi için başvurulan daha ayrıntılı tersine mühendislik yöntemidir. Genelde kapalı kutu uygulamaların geliştiricilerinin veya kaynak kodlu uygulamaların geliştiricilerinin ve güvenlik araştırmacılarının güvenlik açıklıklarını ortaya çıkarabilmek için başvurduğu bir yöntemdir. Kod geliştiricilerinin yaptıkları tasarım ve gerçekleştirdikleri yazılımından kaynaklanan kod hatalarına ait kök nedenlerin ortaya çıkarılmasındaki yöntemlerden birisidir. Açık kaynak kodlu ve ticari birçok kaynak kod analizi aracı geliştirilmiştir [17].

2.4.1. Fuzz Testi Sınıflandırması

Fuzz testleri gerçekleştirilirken, uygulandıkları uygulama ya da sistemin kaynak koduna sahip olunup olunmamasına göre ikiye ayrılır: açık kutu ve kapalı kutu fuzz testleri. Aşağıda her iki yöntemden kısaca bahsedilecektir.

2.4.1.1. Açık Kutu

Bu fuzz testi yöntemi için kaynak kodun varlığına ihtiyaç duyulmaktadır. Genelde kaynak kod geliştiricilerinin rahatlıkla başvurabildiği,

uygulanabilirlik açısından daha kolay bir yöntemdir. Genellikle, açık kutu yöntemi ile kapalı kutu yönteminin bulduğu zafiyetler farklılık göstermektedirler. Bu nedenle, genelde Google, Microsoft gibi büyük firmalar açık kaynak kodlu analizi yapsalar da aynı zamanda büyük yatırımlarla bulut ortamında kapalı kutu fuzz testleri yapmaktadırlar.

2.4.1.2. Kapalı Kutu

Kaynak kod erişiminin mümkün olmadığı durumlarda, sadece girdi noktalarının bilindiği ve çıktının yorumlanabildiği durumlarda tercih edilen yöntemdir. Bu yöntem ile gerçekleştirilen fuzz testleri, kaynak kod bilinmediğinden genelde rasgele girdiler üretilerek ya da sezgisel yaklaşımlar ile gerçekleştirilir. Bu yöntemde herhangi bir formatlı girdi söz konusu değildir.

2.4.1.3. Gri Kutu

Kapalı kutu tekniği ile birlikte sistemin kaynak kodunun da kullanılmasıyla, hem açık kutu hem de kapalı kutu yöntemlerin avantajlarından faydalanmak üzere ortaya çıkan fuzz testi yöntemidir. İkili kodların daha ayrıntılı analizi için tersine mühendislik yöntemlerine başvurulmaktadır. Aslında temel olarak kapalı kutu fuzz testi yöntemine yardım amacıyla kullanılmaktadır. Kapalı kutu fuzz testi ile hedef yazılım üzerinde kapsanan kodun büyüklüğünün belirlenebilmesini sağlar.

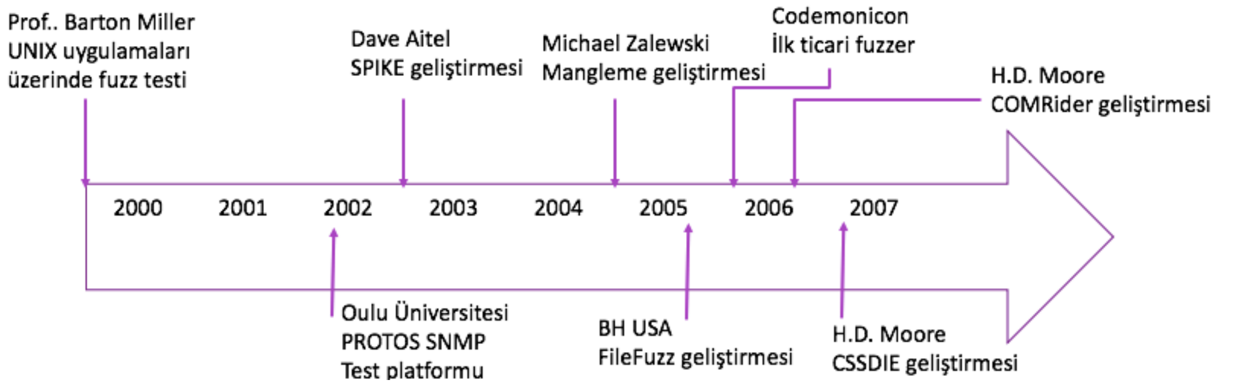
2.4.2. Fuzz Testi Yöntemleri

Fuzz testi; tam geçerli olmayan, beklenmedik, rasgele girdileri üretip farklı platformlardaki *(UNIX, Windows, Android, IOS)* hedef uygulamalar üzerinde güvenlik zafiyetlerinin tespitini sağlayan bir otomatik yazılım test tekniğidir. Kaynak kod analizinin yetersiz olduğu için günümüzde birçok büyük yazılım ve uygulama geliştirici kurum ve kuruluşlar bu yönteme başvurmaktadırlar.

Fuzz testi terimi, Wisconsin Üniversitesi’nde Barton Miller tarafından ortaya atılmıştır [18]. Fuzz testi ismi, 1988/89 yılında yağmur damlalarının telefon hatlarında oluşturduğu sinyal bozulmalarından ve oluşturduğu *fuzz* sesinden esinlenerek verilmiştir. 1988/89 yılında,

Borton Miller ve öğrencileri UNIX uygulamaları üzerinde hataların tespiti için ilk fuzz testini geliştirmişlerdir. Bu test için, temel yöntem olarak rasgele yarı geçerli ya da anlamlı bozulmuş girdiler üretip hedefe göndermişler ve hata ayıklama ile program çakılmalarını analiz etmişlerdir [19], [20]. Bu tarihten beri, fuzz testi üzerindeki çalışmalar devam etmektedir. Fuzz testinin tarihsel evrimi Şekil *7*'de görülmektedir.

Şekil 7. Fuzz testi tarihi [20]



1999 yılında Oulu Üniversitesi'nde ilk protokol tabanlı fuzz testi aracı olan *PROTOS* geliştirilmiştir. Bu araç ile, rasgele girdiler üretilirken, girdilerin protokole uygunluğu da ele alınarak yarı anlamlı girdiler üretilmiştir. 2004 yılında ise ilk web tarayıcı tabanlı uygulamalara yönelik fuzz testi gerçekleştirilmiştir. *HTML* dosya formatını kullanarak bazı web tarayıcılarında güvenlik zafiyetlerinin tespiti hedeflenmiştir. Yine aynı yılda geliştirilen ve fuzz testi tarihinde önemli bir gelişme olan dosya formatlı fuzz testi yöntemi geliştirilmiştir. Bu yöntem ile Microsoft işletim sistemlerinde tespit edilen, MS04-028 numaralı, uzaktan kod çalıştırılabilmesine neden olan bellek taşması zafiyeti tespit edilmiştir [21]. 2006 yılında *Explorer* tabanlı olan *ActiveX* dosya formatında zafiyetler tespit edilirken, 2007 yılında da ilk ortadaki adam tabanlı, istemci ile sunucu arasındaki iletişime girilerek protokollerin fuzz edilmesini sağlayan *ProxyFuzz* aracı geliştirilmiştir. Bu araç, TCP ve UDP

protokollerini desteklemekte ve bu protokollere ait paketlerin değiştirilmesi ile ağ iletişiminin fuzz edilebilmesi sağlamaktadır [22].

Fuzz testi, yeni bir alan olmasına rağmen araştırmacılar ve yazılım geliştiriciler tarafından yoğun ilgi görmüştür. Fuzz testi konusunda literatürde yapılmış birçok akademik çalışma vardır. Bu çalışmaların birçoğu Linux ve Windows platformlarını hedeflemektedir. Android gibi mobil platformları hedef alan az sayıda çalışma vardır. Bu çalışmalar, Bölüm 3'te ayrıntılı bir şekilde incelenmiştir.

Fuzz testi araçlarının ortak benimsediği fazlar bulunmaktadır [20]. Bu fazlar, Şekil 8'te gösterildiği gibi, hedefin belirlenmesi, hedef için uygun girdilerin tanımlanması, fuzz verilerinin üretilmesi ve hedef üzerinde çalıştırılması, en sonunda ise çıktıların izlenmesidir [23].

Şekil 8. Fuzz testi fazları

Fuzz testleri genellikle oynatıcı, okuyucu gibi uygulamaları, Linux, Windows, OSX ve Android gibi işletim sistemlerini, mobil cihaz, modem gibi araçları tercih etmektedirler. Fuzz testleri için girdi olarak; *.swf, .pdf, .png, .jpeg, .m3u*, vb. gibi uzantılı dosya formatları, *ftp, http, arp, ssl* gibi ağ protokolleri paketleri, değişkenler gibi çok çeşitli girdiler oluşturulabilir. Bu girdilerden fuzz test verilerinin üretilmesi için genellikle mutasyon ve üretim tabanlı yöntemler tercih edilmektedir. Sonrasında hata ayıklama ile yarı uygun girdilerin hedef sistemde çalıştırılması, çıktı ve kayıtlarının incelenmesi ile zafiyetler bulunabilmektedir. Fuzz testi

yöntemleri; mutasyon, üretim, vekil ve evrimsel tabanlı olmak üzere dörde ayrılmaktadır.

2.4.2.1. Mutasyon Tabanlı Fuzz Testi

Mutasyon tabanlı fuzz testi, var olan örnek bir girdiyi çeşitli şekillerde bozarak test verilerini üretmektedir. Bu yöntemde, girdinin formatının ya da protokolünün bilinmesi gerekmez. Çekirdek dosya üzerinde yapılan bozulma tamamen rasgele ya da sezgisel olarak gerçekleştirilebilir. Çıktıları oldukça fazladır, tekrar eden yazılım hataları fazla olabilmektedir. Temelde, rasgele bit/bayt değiştirme ve bit/bayt kaydırma olmak üzere ikiye ayrılır.

Rasgele bit değiştirme yöntemine, Şekil 9'da verilen Charlie Miller'ın geliştirdiği meşhur Python kodu örnek verilebilir. Girdi dosyası, ikili dosya olarak açılır ve bu dosyada rasgele seçilen bit'ler 1 ise 0, 0 ise 1 olarak değiştirilirler. Sonrasında, girdi dosyası paketlenip hedef uygulamaya gönderilir.

Şekil 9. Charlie Miller'ın Python fuzz testi kodu

```
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
  rbyte = random.randrange(256)
  rn = random.randrange(len(buf))
  buf[rn] = "%c"%(rbyte)
```

Bit kaydırma yönteminde ise çekirdek dosya üzerinde önceden belirlenmiş bir *örüntü* kaydırılarak yeni test dosyaları üretilir ve hedef dosyaya gönderilir. Örneğin, Şekil 10'da verilen çekirdek dosya üzerinde önceden belirlenmiş "*FF FF FF FF*" örüntüsü sırayla kaydırılarak birçok test dosyası üretilmektedir. Şekil 10'da birçok üretilen test dosyalarından bir tanesi görülmektedir. Bu test dosyasında "*FF FF FF FF*" örüntüsü öyle bir yere gelmiştir ki dosya içerisinde sunulan bilgi ifşası zafiyetine neden olmuştur.

Şekil 10. Bit kaydırma yöntemi örneği

```
151.mov

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00004760  CF 0A 87 7E 79 24 56 2A A2 8B 3F A7 6B B8 5D A8   Ï.‡~y$V*¢‹?§k¸]¨
00004770  13 3C FF E1 79 01 6E 4F 9D 81 3E 7D FF E9 BF A7   .<ÿáy.nO..>}ÿé¿§
00004780  92 0D 2C FC EC 02 71 D5 AA 06 53 F9 E6 15 71 B1   ’.,üì.qÕª.Sùæ.q±
00004790  AC 4F 07 09 CC 6F E7 89 E4 2B 09 14 FD B4 F3 F5   ¬O..Ìoç‰ä+..ý´óõ
000047A0  D9 BB AA 7A FD 18 4E FA 79 06 46 63 3C 32 41 CF   Ù»ªzý.Núy.Fc<2AÏ
000047B0  68 CF 9D 03 39 06 04 E2 FE A3 30 A0 CF 0B BE B4   hÏ..9..âþ£0 Ï.¾´
000047C0  6F DD 9E CA 70 8B D5 3C 74 EA 8F D3 6A 51 DF F0   oÝžÊp‹Õ<tê.ÓjQßð
000047D0  AB AD AD A8 16 54 AA 28 54 35 47 39 FC 1C CE 3F   «--¨.Tª(T5G9ü.Î?
000047E0  E7 90 BC F0 94 7F A3 94 CB 97 B4 73 22 54 E6 6D   ç.¼ð”.£”Ë—´s"Tæm
000047F0  ED F2 5C B9 3D 1C 95 68 FA C6 67 4B FC 74 67 92   íò\¹=.•húÆgKütg’
00004800  B2 E7 D3 6B 7D 2E 7E 7F 09 E8 EF A3 D3 DF F6 E1   ²çÓk}.~..èï£Óßöá
00004810  C5 63 EA B5 2F 2F ED 39 ED 34 03 56 7F F8 9D 27   Åcêµ//í9í4.V.ø.'
00004820  E5 B6 3B 9F B7 7E 9E 4B 98 AF 5F 41 CF 7B DF FF   å¶;Ÿ·~žK˜¯_AÏ{ßÿ
00004830  5B FB 5A FA 7D 2D DD BE D5 FC BD AC 35 7F 64 0D   [ûZú}-Ý¾Õü½¬5.d.
00004840  E4 D2 5E BF 2D 92 92 7E BF 76 61 FB DF A7 BD AE   äÒ^¿-’’~¿vaûß§½®
00004850  84 7B AA 7D 7E 10 92 4B BB B4 67 B4 15 33 E3 B9   „{ª}~.’K»´g´.3ã¹
00004860  3A E0 A5 5D DA CB DB 45 02 5E DA AB B6 8B A3 E5   :à¥]ÚËÛE.^Ú«¶‹£å
00004870  D2 5E B5 5D 18 F0 D2 5E B5 5D 18 F0 D2 5E B5 5D   Ò^µ].ðÒ^µ].ðÒ^µ]
00004880  18 F0 D2 5E B5 5D 18 F0 D2 5E B5 5D 18 F0 D2 5E   .ðÒ^µ].ðÒ^µ].ðÒ^
00004890  B5 FD 0F 72 65 DF 9B 9D FF 64 AF 00 00 00 00 49   µý.reß›.ÿd¯....I
000048A0  45 4E 44 AE 42 60 82 00 00 07 00 00 FF FF FF FF   END®B`‚.....ÿÿÿÿ
000048B0  00 01 FF FF 0C 03 00 03 00 04 00 00 00 0D 00 0C   ..ÿÿ............
000048C0  00 10 00 A0 0C A0 00 BA 00 BA 00 28 00 32 00 22   .......º.º.(.2."
000048D0  10 51 75 69 63 6B 54 69 6D 65 AA 20 61 6E 64 20   .QuickTimeª and 
000048E0  61 00 00 28 0C 3C 00 1F 10 50 4E 47 20 64 65 63   a..(.<...PNG dec
000048F0  6F 6D 70 72 65 73 73 6F 72 00 00 28 00 46 00 00   ompressor..(.F..
00004900  1F 61 72 65 20 6E 65 65 64 65 64 20 74 6F 20 73   .are needed to s
00004910  65 65 20 74 68 69 73 20 70 69 63 74 75 72 65 2E   ee this picture.
00004920  00 00 00 FF                                       ...ÿ
```

2.4.2.2. Üretim Tabanlı Fuzz Testi

Üretim tabanlı fuzz testi yönteminde, hedeflenen protokolün ya da dosyanın formatının bilinmesi gerekmektedir. Hedef protokol ya da dosya formatının spesifikasyonlarından faydalanarak yeni test durumları üretilmektedir. Rasgele tabanlı fuzz testi yöntemine göre daha uzun ve zor bir hazırlık aşaması gerektirmektedir. Hedef odaklı yani belli bir formata özel tasarlanmış olan fuzz testi aşamalarında tercih edilmektedirler. Bu test ile, tek tek bilinen format aşamalarının fuzz

testinde tanımlanması yapıldığı için, format dışı sıfırıncı gün güvenlik açıklarının bulunması ihtimali düşüktür [24].

Şekil 11’de bir üretim tabanlı fuzz testi yöntemi ile bir pdf dosyasının başlık bölümünün dosya formatına uygun bir şekilde değiştirildiği görülmektedir. Bu örnek, *PDFuzzer* aracı [25] ile yaratılmıştır.

Şekil 11. Üretim tabanlı fuzz testi örneği

```
1  #!/usr/bin/perl
2  use PDF::Create;
3  use Getopt::Std;
4
5  @overflow = ('A' x 8200, 'A' x 11000, 'A' x 110000, 'A' x 550000, 'A' x 1100000, '
6  @fmtstring = ("%n%n%n%n%n", "%p%p%p%p%p", "%s%s%s%s%s", "%d%d%d%d%d", "%x%x%x%x%x",
7               "%s%p%x%d", "%.1024d", "%.1025d", "%.2048d", "%.2049d", "%.4096d", "
8               "%99999999999s", "%08x", "%%20n", "%%20p", "%%20s", "%%20d", "%%20x",
9               "%#0123456x%08x%x%s%p%d%n%o%u%c%h%l%q%j%z%Z%t%i%e%g%f%a%C%S%08x%%", "
10 @numbers = ("0", "-0", "1", "-1", "32767", "-32768", "2147483647", "-2147483647", "
11             "4294967294", "4294967295", "4294967296", "357913942", "-357913942", "
12             "1.79769313486231E+308", "3.39519326559384E-313", "99999999999", "-999
13             "0x3fffffff", "0x7ffffffe", "0x7fffffff", "0x80000000", "0xffff", "0xf
14             "0x10000", "0x100000", "0x99999999", "65535", "65536", "65537", "16777
15 @miscbugs = ("test|touch /tmp/ZfZ-PWNED|test", "test`touch /tmp/ZfZ-PWNED`test", "
16              "test&&touch /tmp/ZfZ-PWNED&&test", "test|C:/WINDOWS/system32/calc.exe
17              "test'C:/WINDOWS/system32/calc.exe'test", "test;C:/WINDOWS/system32/ca
18
19 getopts('t:o:', \%opts);
20 $target = $opts{'t'};
21 $pdfdoc = $opts{'o'};
22
23 foreach(@numbers) { $fuzz = $_;
24 $pdf = new PDF::Create('filename' => $pdfdoc,
25                        'Version' => $fuzz,
26                        'Author' => 'pdfUZZ',
27                        'Title' => 'pdfUZZ',
28                        'CreationDate' => [localtime],
29                        'Subject' => 'pdfUZZ',
30                        'Keywords' => 'pdfUZZ');
31 $main = $pdf->new_page('MediaBox' => $pdf->get_page_size('A4'));
32 $pdf->close;
33 system $target $pdfdoc; }
```

2.4.2.3. Vekil tabanlı Fuzz Testi

Vekil tabanlı fuzz testi, genelde istemci ve sunucu uygulamalar arasındaki iletişimin arasına girilmesi *(ortadaki adam)* ve girdilerin bozulması ile gerçekleştirilir. Protokol tabanlı bir yöntemdir. *ProxyFuzz* aracı, bu yöntemi kullanan bilinen fuzz testi araçlarından birisidir [22].

2.4.2.4. Evrimsel Tabanlı Fuzz Testi

Evrimsel tabanlı fuzz testi yöntemi, diğer yöntemlerde farklı olarak, bir geri bildirim mekanizması ile çıktıya göre test girdilerinin üretilmesine dayanmaktadır. Autodafe [26], EFS [27], AFL [28] araçları, bu yöntemi kullanmaktadır. Bu yöntem ile daha kısa sürede daha çok tekil ve yeni güvenlik zafiyetlerin tespit edilebilmesi amaçlanmaktadır. Bu tez çalışmasında önerilen yöntemde olduğu gibi, genetik algoritma, vb. gibi evrimsel hesaplama tabanlı yöntemler ile gerçekleştirilen fuzz testleri, bu gruba girmektedir.

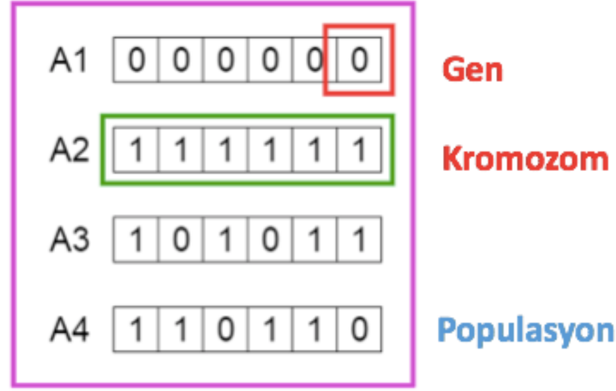
2.5. Evrimsel Hesaplama

Tez kapsamında daha kısa sürede daha çok yeni tekil güvenlik zafiyeti tespit edebilmek için, yeni girdi test dosyalarının üretilmesinde evrimsel hesaplama yöntemlerine başvurulmuştur. Bu başlık altında, bu tez kapsamında kullanılan ve en popüler evrimsel hesaplama yöntemlerinden birisi olan genetik algoritma hakkında bilgi verilecektir.

2.5.1. Genetik Algoritma

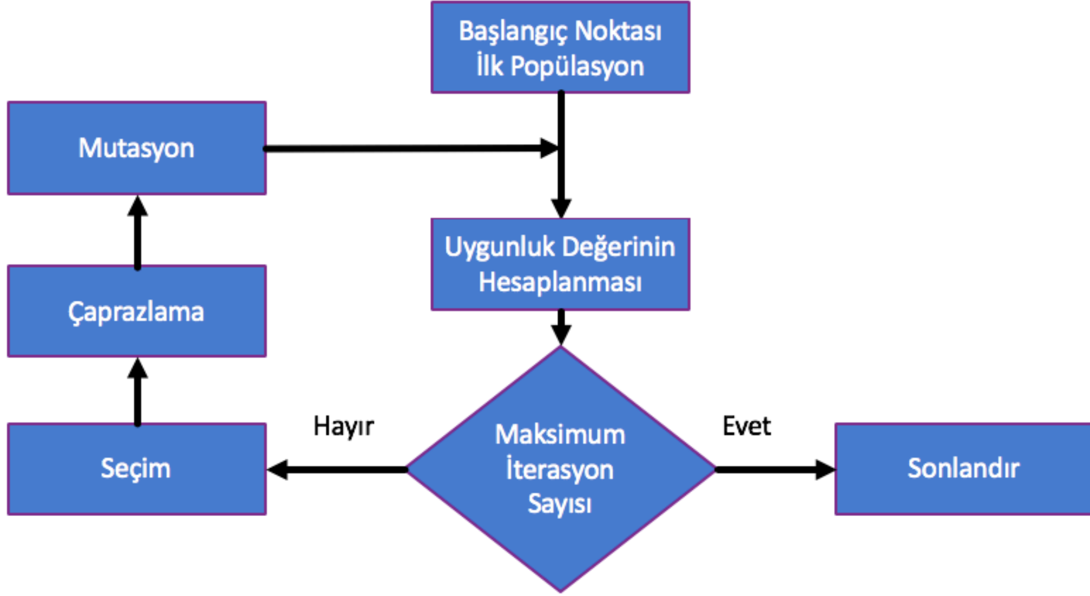
Genetik Algoritma, 1975 yılında John Holland tarafından tanıtılmıştır [29]. Genetik Algoritma *(GA),* tabiatta izlenen evrimsel akışa benzer bir şekilde çalışan, arama ve en iyileme yöntemi olarak tanımlanmaktadır [30]. GA, bir problemin ideal ya da ideale yakın bir çözümünün verimli bir şekilde bulunması için doğadaki evrim mekanizmasından esinlenmiş bir algoritmadır [31]. Şekil *12*. Genetik algoritma gösterimi ve işleçleri ’de doğadaki şekli gibi dijital ortama uyarlanması gösterilmiştir. Diğer bir tanımla, Genetik algoritmalar *(GA),* Darwin'in en güçlü olanın hayatta kalması ilkesine dayanan bir algoritmadır [32], [33].

Şekil 12. Genetik algoritma gösterimi ve işleçleri



Genetik Algoritma akış şeması *Şekil 13'te* verilmiştir. Öncelikle, ilk popülasyon yaratılır. Popülasyon, problemin çözümü için önerilen bireylerden oluşmaktadır. Örneğin, yazılım testi veya fuzz testi amaçlı bir genetik algoritma uygulamasında bireyler, test durumlarını temsil etmektedirler. İlk popülasyon, genetik algoritmaya girdi olarak verilebilir, ya da rasgele oluşturulabilir. Daha sonra, popülasyondaki her bireyin uygunluk değeri hesaplanır. Bu değer, bir bireyin problemin çözümüne ne kadar yakın olduğunu ifade eder. Eğer bireylerden en az birisi ideal çözüme ulaşmamışsa, bu bireyler üzerine genetik operatörler uygulanır ve yeni popülasyon oluşturulur. Bu işlemlerin tümü bir iterasyonu ya da nesli tanımlar. GA'da, nesil değiştikçe kötü çözümlerin yok olması ve iyi çözümlerin daha fazla kullanılması amaçlanır [34]. Genetik algoritma, genellikle manuel olarak çözümü zor olan karmaşık problemler üzerine uygulanmaktadır. Bu nedenle, genellikle algoritmanın sonlandırılma kriteri, ideal çözümün bulunmasından ziyade, belirlenen iterasyon sayısı kadar algoritmanın koşup koşmadığıdır. Bir problemin ideal çözüme ulaşamayacağı ya da ulaşmasının çok uzun süreceği varsayımından yola çıkarak, iterasyon sayısı algoritmanın önemli parametrelerinden birisi olarak kullanılır. Aşağıda, genetik algoritmada kullanılan temel bazı genetik operatörler tanıtılmaktadır.

Şekil 13. Genetik algoritma akış şeması



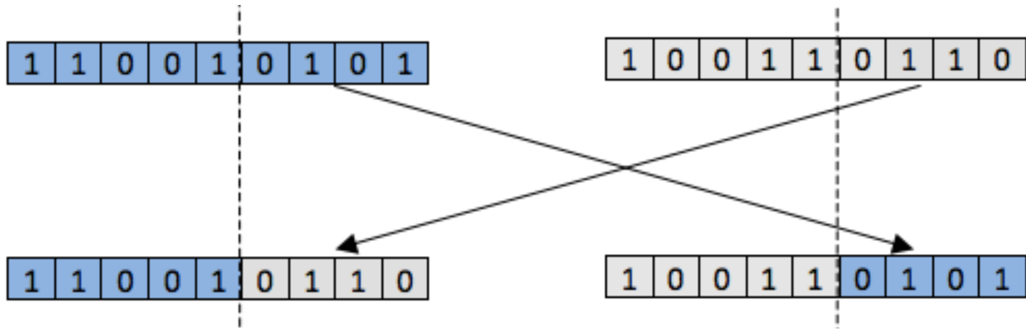
2.5.1.1. Seçim

Seçim ya da seçilim, gelecek nesilleri üretmek için kullanılacak bireylerin seçildiği genetik algoritma aşamasıdır. Yeni neslin (popülasyonun) üretilebilmesi için, varolan popülasyonun içerisinden seçim operatörüne göre bireyler seçilir. Seçilen bu bireyler üzerinde çaprazlama, mutasyon gibi genetik operatörler uygulanarak yeni bireyler oluşturulur. Rulet çamberi, turnuva ve lineer sıralama yöntemleri seçim yöntemlerindendirler. Rulet çemberi, nesildeki bireylerin uygunluk değerinin toplam uygunluk değere oranlamasıyla seçim oranının belirlenmesidir. Lineer sıralı seçim, en iyi uygunluk değerinden en kötü uygunluk değerli bireye doğru sıralanması yöntemidir. En düşükten en yüksek uygunluk değerli bireye doğru 1, 2, 3 şeklinde değerler verilir. Son olarak turnuva seçimi de nesilden seçilen rasgele sayıdaki bireyin içinden uygunluk değeri en iyi olan bireyin seçilmesi yöntemidir.

2.5.1.2. Çaprazlama

Popülasyondan seçilen ve ebeveyn bireyler olarak isimlendirilen iki birey üzerinde uygulanır ve bu bireylerden yeni iki birey elde edilir. Değişik türleri bulunmaktadır. Örneğin *Şekil 14'*te tek noktalı çaprazlama mutasyona bir örnek verilmektedir. Burada ebeveyn bireyler üzerinde ortak bir çaprazlama noktası belirlenir ve bireylerin bu noktadan sonraki kromozomları yer değiştirilerek yeni bireyler elde edilir. Çaprazlama oranı ile popülasyondaki ne kadar kromozomun çaprazlamaya gireceğini belirlenmektedir.

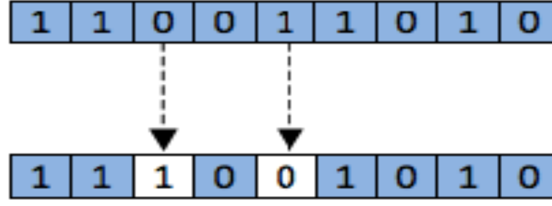
Şekil 14. Genetik çaprazlama örneği



2.5.1.3. Mutasyon

Mutasyon işleci, bir popülasyondaki genetik çeşitliliğinin sürdürebilmesine yardımcı olmaktadır. Mutasyon için değişik yöntemler kullanılmaktadır. Şekil *15*'te bit katar dizgisi mutasyonu örnek olarak gösterilmektedir. Bu yöntemde, bir bireyde rasgele seçilmiş bitlerin değerleri değiştirilir. Seçilen bit 1 ise 0'a, 0 ise 1'e değiştirilir.

Şekil 15. Mutasyon örneği



2.5.1.4. Uygunluk Fonksiyonu

Bir bireyin uygunluk değerinin, ya da o bireyin çözüme ne kadar yaklaştığını ifade eden değerinin hesaplanması için uygulanan fonksiyondur. Bir bireyin uygunluk değeri yüksek olması, o bireyin ideal çözüme ulaşmış ya da yakın olduğunu göstermektedir.

3. İLGİLİ ÇALIŞMALAR

Fuzz güvenlik testleri, ilk olarak 2009 yılında uygulanmaya başlanmıştır. Günümüze kadar yapılan çalışmaların çoğu Windows, IOS ve UNIX işletim sistemi platformlarını hedef alarak gerçekleştirilmiştir. Mobil cihazların yaygınlaşması ile, mobil platformlar için fuzz testleri son yıllarda popülarite kazanmaya başlamıştır. Mobil alanda gerçekleştirilen ilk testler, Android işletim sistemini hedef almaktadır. Bu alanda gerçekleştirilen ticari ya da açık kaynak kodlu fuzz testi araçlarının yanında, az sayıda akademik çalışma bulunmaktadır. Miller [35], fuzz testlerinin Android platformunda kullanımının ilk örneklerinden birisini sunmuştur. Bu çalışmada, akıllı telefonlar üzerinde açık kaynak kodlu Sulley fuzz testi aracı [4] kullanılarak, bozulmuş SMS mesajları ile stres testleri gerçekleştirilmiştir ve uygulamaların çakılmalarına neden olan tespit ettikleri güvenlik zafiyet kayıtları incelenmiştir. Günümüze kadarki fuzz testi çalışmaların çoğu uygulama seviyesinde ya da kullanıcı arayüzü/grafiksel kullanıcı arayüzü bazlı gerçekleştirilmiştir [36-45]. Daha geniş kullanıcı kitlesini etkileyen ve kritik olan işletim sistemi seviyesindeki kütüphanelerde zafiyet tespiti çalışmaları ise çok azdır.

Bu bölümde, öncelikle bir platformdan bağımsız olarak evrimsel hesaplama tabanlı fuzz testi uygulayan çalışmalar hakkında bilgi verilecektir. Sonrasında, Android işletim sistemi üzerinde yapılan fuzz testi çalışmalarına yer verilecektir. Android platformunda farklı girdi noktaları ve yöntemleri kullanılarak yapılan testlerden bahsedilecektir. Android platformu üzerindeki evrimsel hesaplama tabanlı çalışmalar detaylandırılacaktır. Son olarak da Android üzerinde, özellikle DEX dosya formatının modifiye edilmesi ile gerçekleştirilen fuzz testi çalışmalari ayrıntılı açıklanacaktır.

3.1. Evrimsel Hesaplama Tabanlı Fuzz Testleri

Evrimsel hesaplama yöntemleri, probleme uygunluğu açısından oldukça kullanılmıştır. Şu ana kadar farklı uygulama ve farklı platformlara uygulanmıştır. Bu alanda yapılan bir çalışmada Alimi ve diğerleri [46], genetik algoritma temelli önerdikleri yaklaşım ile probleme neden olan komut kümelerinin tespitinde optimizasyon yapılmasını hedeflemişlerdir. Test ortamı olarak akıllı kartlar üzerindeki *applet* olarak adlandırılan birkaç basit fonksiyondan oluşmuş küçük kod parçalarından oluşan gömülü uygulamalar tercih edilmiştir. Fuzz testi platformu olarak WinSCard [47] seçilmiştir. Popülasyon boyutu olarak 10.000, iterasyon olarak 5.000 değerleri seçilmiştir. 6 bayttan oluşan *MasterCard M / Chip CVR* değerinin son 5 bit'i uygunluk fonksiyonunda kullanılmıştır. Mutasyon ve seleksiyon olarak rasgele metodları tercih edilmiştir. JCOP simülasyonu aracı [48] ile *MasterCard* üzerinde onlarca saat yapılan testlerde önerilen yaklaşımla sonuç alınamamış, fakat ardışık çevrimdışı işlemlerin sınırının üzerinde birçok işlemin oylanması gibi anormal bazı işlemler tespit edilmiştir.

Liu ve diğerleri tarafından yayınlanan çalışmada [49], x86 platformunda bulunan popüler programlar üzerindeki güvenlik kod hatalarının tespit edilmesi hedeflenmiştir. Zafiyetlerin tespiti, statik ve dinamik analizleri için *GAFuzzing* adı altında yeni bir fuzz testi aracı tanımlanmıştır. Analizler, diğer fuzz testi araçlarından ayrı olarak kaynak kod kullanmadan yapılmaktadır. Fuzz testi metodu olarak da rasgele fuzz testi seçilmiştir. Statik analiz aşamasında, çalıştırılabilir kod üzerindeki *strcpy(), sprintf(), syslog()* gibi C programları seçilmiştir. Dinamik analizde kontrol akış grafiklerinin analizleri için açık kaynak kodlu IDA Pro [50], [51] kullanılmıştır. Dinamik analiz tarafında çalıştırılması ve izlenebilmesi için ise, açık kaynak kodlu Pin [52] aracı kullanılmıştır. Genetik algoritmada ise %10 mutasyon ve %70 çaprazlama oranı kullanılmıştır. Bellek taşması zafiyeti olduğu bilinen *VulPro* çalıştırılabilir

dosyası üzerinde yapılan testte, rasgele fuzz testi yönteminin daha efektif olduğu görülmüştür.

Sparks ve diğerleri [53] tarafından yapılan çalışmada *Dynamic Markov Model* [54] uygunluk fonksiyonuna dayanan, kaynak koda ihtiyaç duymadan kapalı kutu fuzz testi uygulaması yapılmıştır. Kontrol akış grafiği için burada da IDA Pro [50] kullanılmıştır. Mutasyon oranı %90, çaprazlama oranı olarak da %75 kullanılmıştır. Çalıştırma ortamı olarak *PAIMEI* fuzz testi platformu tercih edilmiştir [55]. *Tftpd.exe* çalıştırılabilir dosyası üzerinde yapılan testlerde, %84.81 kod kapsama oranı elde edilmiştir. Çalışmalar, Windows işletim sistemi platformunda gerçekleştirilmiştir.

Seagle [56] tarafından yazılan bir doktora tezi çalışmasında, Mac OS X işletim sisteminde, statik ve dinamik analize dayalı, konfigüre edilebilir uygunluk fonksiyonları ile birlikte fuzz testi platformu sunulmuştur. Mamba adı verilen fuzz testi platformunda; *Mangle, Basit Genetik Algoritma (SGA), Bayt Genetik Algoritma (BGA), Basit Genetik Algoritma Mangle Mtantörü (SGA-MM), ve Çapraz Kuşak Elitist Seçimi, Heterojen Rekombinasyon ve Cataclysmic Mutasyon Genetik Algoritması (CHC-GA)* olarak 5 farklı fuzz testi algoritması kıyaslanmıştır. BGA algoritması ile en fazla tekil program hatası tespit edilmiştir. Genetik algoritma parametreleri için; çaprazlama oranı 0.2, mutasyon oranı 0.5, iterasyon sayısı 100, popülasyon büyüklüğü 250 değerleri uygulanmıştır. Gri kutu fuzz testi uygulanmıştır.

Evrimsel algoritmaları kullanarak fuzz testi yapan bir diğer çalışmada, üretim tabanlı ve evrimsel tabanlı fuzz testi yöntemleri karşılaştırılmıştır. [57]. Karşılaştırma sonucunda evrimsel tabanlı fuzz testi yönteminin üretim tabanlı fuzz testine göre daha fazla kod kapsama sağladığı ve kısa sürede daha fazla performans göstererek daha fazla sömürülebilir güvenlik zafiyeti tespit edebildiği görülmüştür. Evrimsel fuzz testi için *EFS* fuzz testi aracı [27] kullanılmıştır. Üretim tabanlı fuzz testi için ise *FTPStress* fuzz testi aracı [58] kullanılmıştır. *EasyFTP server 1.7.0.1*

hedefinde yapılan testlerde zaman, kod kapsama ve program hataları tespitinde evrimsel tabanlı fuzz testi yönteminin daha iyi olduğu görülmüştür.

Beterke tarafından 2016'da yayınlanan bir araştırmada [59], yapılan fuzz testi çalışmalarında süreyi kısaltmak için dağıtık yapıda fuzz testi platformu geliştirilmiştir. *Evofuzz* adının verildiği bu platformda bulut ile birlikte eş zamanlı testler gerçekleştirilmektedir. Android uygulama programlama arayüzleri üzerinde fuzz testleri gerçekleştirilmiştir. Dağıtık yapıda eş zamanlı olarak birden fazla hedef çalıştırılabilir dosya üzerinde fuzz testi gerçekleştirilebilmektedir. Ayrıca geliştiriciler tarafından ihtiyaca özgü farklı güvenlik testleri için kullanılabilir hale getirilmiştir.

2014 yılında Duchene ve diğerleri [60] tarafından sunulan bir çalışmada, önerilen *KameleonFuzz* fuzz testi aracı ile *XSS* web zafiyeti tespiti üzerine çalışılmıştır. Evrimsel tabanlı fuzz testi çalışmaları gerçekleştirildiği görülmüştür.

3.2. Android Platformunda Fuzz Testleri

Android işletim sisteminde yapılan fuzz güvenlik testi çalışmaları, genellikle girdi noktaları ve kullandıkları yöntem açısından farklılık göstermektedir. Mahmood ve diğerleri [61], Android uygulamalarına ait güvenlik açıklıklarının periyodik olarak tespitinin Android marketler tarafından otomatik yapılamaması sorununu ele almışlardır. Android uygulamalarına ilişkin eş zamanlı fuzz testlerinin, otomatik olarak belli periyotlarla bulut ortamında gerçekleştirilebilmesini sağlamışlardır. Beterke [59] de, bulut ortamında birçok Android emülatör üzerinde eş zamanlı fuzz testi gerçekleştirilebilmesi için bir yöntem önermiştir. Öncelikle, uygulamaların *manifest* dosyalarından ilgili uygulamanın girdi noktaları tespit edilmekte ve girdi noktalarına göre test durumları üretilmektedir. Sonrasında bu üretilen test durumları, arka planda birçok Android emülatörün bulunduğu sanal bağlantı noktalarından herbirine farklı girdi olarak verilerek tek tek çalıştırılması sağlanmaktadır. Derleyici altında çalışan bu emülatörler izlenebilmekte ve herhangi bir yazılım

hatası, program çakılması, aykırı durum tespit edildiğinde çıktı klasöründe kayıt altına alınmaktadır. Bu bir döngü halinde devam etmektedir. Elde edilen çıktılar üzerinden aykırı durum analizi yapılmaktadır. Farklılıklara göre belirlenen aykırı durum analisti ile yanlış-pozitifler ayıklanarak, potansiyel güvenlik zafiyetleri bulunmaktadır. Android uygulamalar için bulut tabanlı bir fuzz testi platformu, 2012 yılında Malek ve diğerleri [62] tarafından önerilmiştir. Bu çalışmada, ölçeklenebilir bir test platformu sunulmakta ve hızlı olarak güvenlik zafiyetlerinin tespiti amaçlanmaktadır. Benzer bir çalışmada, bulut ortamının yüksek depolama ve elastik arama [63] gücü kullanılarak fuzz testleri gerçekleştirilmiştir [64]. Önerilen platform ile birçok fuzz testi aracının aynı anda çalışması ve bu araçların çıktılarının ortak analiz edilmesi sağlanmıştır.

Diğer bir araştırmada Iannillo ve diğerleri [65], *Chizpurfle* olarak isimlendirdikleri fuzz testi aracı ile kaynak kod erişiminin sınırlandırıldığı geliştiriciler için emülatörler üzerinde çalıştırılamayan Android servisleri için geliştirilmiştir. Samsung Galaxy S6 Edge Android v7 üzerinde yapılan testte *privilaged* servislerde iki adet yazılım hatası tespit edilmiştir. Bu süre zarfında Samsung özelinde 2272 servis tespit edilmiş ve toplam 34645 test gerçekleştirilmiştir. İlk yazılım hatası *spengestureservice* servisinde, ikincisi ise *voip* servisinde tespit edilmiştir. Bir başka çalışma olan Dynodroid'te [37] de Java üzerinde kod kapsama hedeflenmiş ve 12 Android uygulaması üzerinde 15 yazılım hatası bulunmuştur.

Gu ve diğerleri [66], ele aldıkları çalışmada Android uygulamalarının otomatik test işlemlerinin gerçekleştirilmesi esnasında aktivite bileşenleri [67] arasındaki geçişlerin yönetilememesi sorununu ele almışlardır. Bu kapsamda sunulan Aimdroid fuzz testi aracı ile, test esnasında *aktivite* bileşenleri arasındaki geçişleri minimuma indirilerek, zaman kayıpları önlenmiştir. Aimdroid fuzz testi aracının diğer model tabanlı çalışan fuzz testi araçları ile karşılaştırılması sonucunda; Aimdroid'in hedef olarak

seçilen aktivite, metod ve kod kapsamada daha iyi performans gösterdiği ve daha fazla program hatası tespit edebildiği görülmüştür [68].

Farklı bir araştırmada ise Android uygulamaları ile haberleşmede kullanılan *Intent'lerle* fuzz testi platformu geliştirilmiştir [69]. BIFUZ adı verilen fuzz testi platformu ile, *Intent'lerin* nasıl oluşturulduğu, hangi parametreleri kabul edip etmedikleri dikkate alınarak *Inter Process Communication* protokolündeki güvenlik zafiyetleri tespit edilebilmektedir. Hala geliştirme aşamasında olan bu model tabanlı fuzz testi aracında gelecek çalışma olarak *Intent* ve *Broadcast_Intent* fuzz testlerinin ayrı ayrı ele alınması ve kategorize edilmesi hedeflenmektedir.

Ye ve diğerleri [70] tarafından yapılan çalışmada, Android uygulamalarındaki MIME *(AVI, HTML, MP3, M3U, BMP gibi)* veri dosyalarını kabul eden aktiviteleri hedef alan fuzz testleri gerçekleştirilmiştir. Buradaki zafiyetleri tespit etmek için geliştirilen DroidFuzzer, *manifest* dosyasında yer alan *Intent_filter* bölümlerini kullanarak bir uygulamanın kullandığı aktiviteleri tespit etmektedir. Fuzz testi aşamasında kaynak koda ihtiyaç duyulmamaktadır. Uygulama özelinde kullanılan aktiviteye özgü önceden hazırlanmış çekirdek dosya seçilerek bozulup uygulamaya gönderilebilmektedir. *QQ Browser* ve *UC Browser* üzerinde tespit ettikleri *XSS güvenlik zafiyeti* [71] için 2 adet CVE numarası alınmıştır. Bu güvenlik zafiyetleri, *WebView* sınıfı üzerinde bozulmuş URL'i yorumlarken program çakılması meydana gelmesiyle tespit edilmişlerdir.

Fuzz testi yönteminin popüler olması ardından Android sistemler üzerinde kullanılır hale gelmesiyle birçok araç üretilmiştir. Araştırmacılar da Android sistemler için geliştirilmiş fuzz testi araçlarının etkinliklerini karşılaştırma ihtiyacı duymuşlardır. Javaid [72] tarafından yayınlanan bir tezde, 2013 yılına kadar çıkmış olan fuzz testi araçları karşılaştırılmıştır. Android platformu için geliştirilen farklı amaçlar için oluşturulmuş bu fuzz testi araçları; kolay kullanım, kod kapsama, program hataları tespiti ve farklı platformlarda çalışabilme kategorileri altında karşılaştırılmıştır. 60

Android uygulaması üzerinde popüler 3 fuzz testi olan Monkey [36], Dynodroid [37] ve A3E [73] kıyaslanmıştır. Genel olarak en iyi performans, rasgele fuzz testi gerçekleştirdiği için Monkey tarafından gerçekleştirilmiştir. Bu konuda en güncel karşılaştırma, Choudhary, Gorla ve Orso [74] tarafından 2015 yılında yapılmıştır. Bu çalışmada, Android sistemler için geliştirilmiş fuzz testi araçları, dört karşılaştırma metriği altında incelenmiştir. Bunlar; kod kapsama, hata tespit etme yeteneği, birden fazla platformda çalışma yeteneği ve kullanım kolaylığıdır. Android sistemler için geliştirilmiş 14 fuzz testi aracından belirlenen 7 tanesi; rastgele, sistematik ve model tabanlı olmak üzere 3 farklı kategoride incelenmiştir. Kod kapsamı açısından, rasgele fuzz testi yöntemlerinin daha başarılı oldukları ve uygulama kodlarının yaklaşık olarak %80'ini kapsayabildikleri gösterilmiştir. Hata tespit etme yeteneği konusunda, topladıkları 68 Android uygulaması üzerinde onar kez yapılan testlerde en çok program çakılma hataları yine rasgele fuzz testi yöntemini kullanan araçlarla yakalanmıştır. Bu açıdan en düşük başarımı model tabanlı yaklaşımlar göstermiştir. Kullanım kolaylığı kategorisinde rasgele fuzz testi güvenlik yöntemini kullanan araçların daha iyi oldukları, birden fazla platformda çalışma yeteneği göz önünde bulundurulduğunda da yine rasgele fuzz testi yöntemini kullanan araçlarının güçlü oldukları görülmüştür.

Diğer bir çalışmada Blanda [10], Android üzerinde genel olark fuzz testi yöntemlerini ele almıştır. Android sistem komponentlerine fuzz testleri gerçekleştirmiştir. Yöntem olarak, dosya-tabanlı fuzz testi yöntemini tercih etmiştir. MP3, MPEG4 dosyaları gibi dosya formatına özgü oluşturulan bir taslak ile belirlenmiş bölümlere kurallar çerçevesinde değiştirilerek Android sistem komponentleri hedef alınmıştır. Genetik algoritma kullanımı tercih edilmemiş, kapalı kutu yaklaşımı gerçekleştirilmiştir. Bu çalışmada, bir adet CVE-2014-7918 numaralı güvenlik zafiyeti tespit edilmiştir.

3.2.1. Android'te Evrimsel Hesaplama Tabanlı Çalışmalar

Bir diğer benzer çalışmada yine Android uygulamaları üzerinden otomatik olarak rastgele tabanlı, sistematik ve araştırma tabanlı fuzz testi yöntemlerinin kombinasyonu ile minimum test yaparak maksimum kod kapsamayı sağlama ve yazılım hatalarının tespiti hedeflenmiştir. Genetik algoritma olarak ile çok amaçlı optimizasyon amaçlanmıştır [75]. Testler, uygulama seviyesinde gerçekleştirildiği için oldukça fazla programlama hatası tespit edilmiştir. Bunların bir bölümü için uygulama geliştiricileri ile iletişime geçip yama yapılmasını sağlamışlardır. 1000 Android uygulaması üzerinde yapılan testlerde, tespit edilen 558 tekil programlama hatasından 14 tanesinin gerçekliği doğrulanmıştır. Bu testlerde çaprazlama olasılığı 0.7 ve mutasyon olasılığı 0.3 olarak seçilmiştir. Popülasyon 50, maksimum iterasyon 100 ve her birey için test durumu 5 olarak belirlenmiştir. *Dynodroid* [37] ve *Monkey* [36] fuzz testi araçları ile kıyaslandığında kod kapsama ve programlama hataları tespiti açısından daha iyi performans gösterdiği görülmüştür.

3.2.2. Uygulama Kurulum Sürecine Yönelik Çalışmalar

Mayıs 2016'da Android çalıştırılabilir dosyalar için *Droid-FF* fuzz testi platformu geliştirilmiştir [76]. Bu platform, genel amaçlı geliştirilen ilk Android fuzz testi platformudur. Açık kaynak kodlu zzuf fuzz testi aracı [77] üzerine inşa edilmiş olan bu platform, Android işletim sistemi üzerinde dosya girdisi kabul edilen bütün çalıştırılabilir dosyaları kapsamaktadır. Mutasyon ve üretim tabanlı fuzz testi yöntemleri farklı fuzz araçları entegrasyonu ile desteklenebilmektedir. Bu test platformu ile .dex uzantılı dosyalar üzerinde mutasyon tabanlı fuzz testi çalışması yapılabilmektedir. DEX dosyaları için yapılan test sonuçları bilinmemekle birlikte, MP4 dosya formatı kullanılarak *stagefright* çalıştırılabilir dosyası üzerinde yaptıkları 14 saat testin sonucunda 3 adet program çakılması elde edilmiştir.

Kyle ve diğerleri tarafından Java ile geliştirilen DexFuzz aracında [78], yapısal .dex dosya formatı kullanılarak fuzz testi gerçekleştirilmesi

amaçlanmıştır. Bu alanda yapılan açık kaynak kodlu ilk yapısal rasgele fuzz testi aracıdır. Google tarafından sunulan ve temel geliştirici araçlarının yer aldığı *master development branch* platformu kapsamı altında yer almaktadır [79]. Android için yeni geliştirilen 4.4 versiyonu ile birlikte gelen ve 5.0 versiyonu ile birlikte yaygın olarak kullanılan ART uygulama sanal makinesi, hala geliştirme aşamasında olduğu için bünyesinde birçok güvenlik açıklıkları ve programlama hatalarını barındırmaktadır. DexFuzz güvenlik aracı, özellikle bu açık ve hataların tespiti için tasarlanmıştır. ART sanal makinesi altında çalışan *Interpreter*, *Quick derleyici* ve *Optimizing derleyici* [80] çalıştırılabilir metodlar hedeflenmiştir. Arka planda çalışan bu çalıştırılabilir metodlar üzerindeki yazılım hatalarının bir kısmı çalışma anında, bir kısmı da derleme anında tespit edebilmiştir. Fuzz testi yöntemi olarak ikili mutasyon yöntemi kullanılmıştır. 5000 iterasyonun her bir aşamasında, bir çekirdek dosya üzerinden üretilen bozulmuş test dosyaları ile 189 program çakılması tespit edilmiş ve bunlardan 15 tanesinin biricik olduğu belirtilmiştir. Bu çalışmada, yazılım hatası bulmak için ART'ın varsayılan modu olan *quick* derleyicisine yoğunlaşılmış. Birçok yazılım hatası tespit edilmiş ve bunlardan 4 tanesi açıklanmış ve Android açık kaynak kod geliştirme projesi olan AOSP'a [81] gönderilerek yamalarının yayınlanması sağlanmıştır. Ayrıca DexFuzz, AFL *(American Fuzzy Loop)* aracı [78] ile kıyaslanmış ve AFL'nin aynı zamanda çok daha fazla test iterasyonu gerçekleştirdiği görülmüştür. Bunun nedeni olarak, AFL'nin C programlama dili ile, DexFuzz'ın ise JAVA programlama dili ile yazılmasının olduğu belirtilmiştir. Ayrıca AFL fuzz testi aracı yaklaşık 20 program çakılması yakalarken, DexFuzz ise 1 tane yakalamıştır. Sonuç olarak, AFL fuzz testi aracının, ART'ın yapısal doğrulaması üzerinde yapılan testlerde daha etkili ve iyi olduğu görülmüştür. Ayrıca, bu çalışmada 11 farklı mutasyon fuzz yöntemi *(BranchShifter, ComparisonBiasChanger, ConstantValueChanger, InstructionDeleter, InstructionDuplicator, InstructionSwapper, OperationChanger, PoolIndexChanger, RandomInstructionGenerator, TryBlockShifter ve*

VirtualRegisterChanger) belirlenmiştir. Bu mutasyon fuzz yöntemleri, DexFuzz altında kullanılarak 6 farklı kategoride *(Doğrulama, Hata, Mutasyon Hatası, Zaman Aşımı, Başarı, Program Çakılması ve Farklılık)* kıyaslanmıştır. Yapılan çalışmalarda sadece mutasyon tabanlı fuzz testine yer verilmiş, girdi optimizasyonuna yer verilmemiştir. Evrimsel tabanlı çalışma uygulanmamıştır.

Anestis [82], ART derleyicisi üzerindeki *Quick* ve *Optimizing* çalıştırılabilir metodlarını hedef almıştır. Bu kapsamda d'ART *(Digging Into the Android L Runtime Internals)* fuzz testi aracı geliştirilmiştir. Oluşturulan kural kümeleri ile akıllı fuzz testi uygulanmıştır. 4 kategoride toplanmış olan *(Index, Relative Offset, Data Placehorders ve Attribute Metadata)* ve 18 temel bölümden oluşan .dex dosya formatına odaklanılmıştır. Geliştirilen veri karıştırma algoritması ile akıllı mutasyon kural kümeleri oluşturulmuştur. Oluşturulan yapıda kullanılan manuel geri bildirimler ile de iyileştirmeler uygulanmaktadır. Geri bildirimlerle bir kural kümesi üzerinden 16 farklı kural kümesi türetilmiştir. *Quick* ve *Optimizing* çalıştırılabilir metodları üzerinde her bir kural kümesi için 5000 iterasyonlu test gerçekleştirilmiştir. *Runtime Initialization* ve *Runtime Execution* metodları üzerinde çalışma yapılmamıştır. Nexus 4 Android 5.1 üzerinde *Optimizing Major* çalıştırılabilir metodu üzerinde 17, *Quick Major* çalıştırılabilir metodu üzerinde 22 tekil yazılım hatası tespit edilmiştir.

Sonuç olarak bu bölümde literatürde geçen çalışmalar kategorize edilerek kısaca açıklanmıştırlar. Henüz yeni olan Android platformunda fuzz testleri, güvenlik araştırmacıları tarafından farklı hedef ve yöntemlerle yapılmaktadır. Android platformunda fuzz testleri için genetik algoritmaların kullanıldığı birkaç örnek çalışma bulunmakla beraber, bu çalışmalar uygulama ya da API düzeyinde kalmıştır. Android işletim sistemi seviyesinde bir çalışma yoktur.

Çizelge 1’ de görüldüğü üzere DEX dosya formatı üzerinde 3 farklı fuzz testi çalışması yapılmıştır. Bunlardan sadece DexFuzz [78], akademik bir çalışmadır. Fakat DEX dosya formatı üzerinde genetik algoritma tabanlı fuzz testi yöntem ve araçları bulunmamaktadır. Bu tezde sunulan GAdroid aracı bu alanda yapılacak çalışmalar için bir ilktir. Çizelge 2’de ise literatürde Android için önerilen fuzz testi araçlarının fuzz testi yöntemine, hedef türüne ve kullandıkları zafiyet testi yöntemine göre kıyaslamaları verilmiştir.

Çizelge 1. Sunulan literatür taramasının özeti

Fuzz Testi Aracı	Android	GA Kullanımı	Kurulum Süreci
Dynodroid [37]	X		
DroidFuzzer [70]	X		
Monkey [36]	X		
AFL [28]	X		
Aimdroid [66]	X		
Bifuz [69]	X		
Chizpurfle [65]	X		
Evofuzz [59]		X	
KamelonFuzz[60]		X	
Mamba [56]		X	
GAFuzzing [49]		X	
DexFuzz [78]	X		X
Droid-FF [76]	X		X
d’ART [82]	X		X
GAdroid	X	X	X

Çizelge 2. Literatürün strateji olarak karşılaştırılması

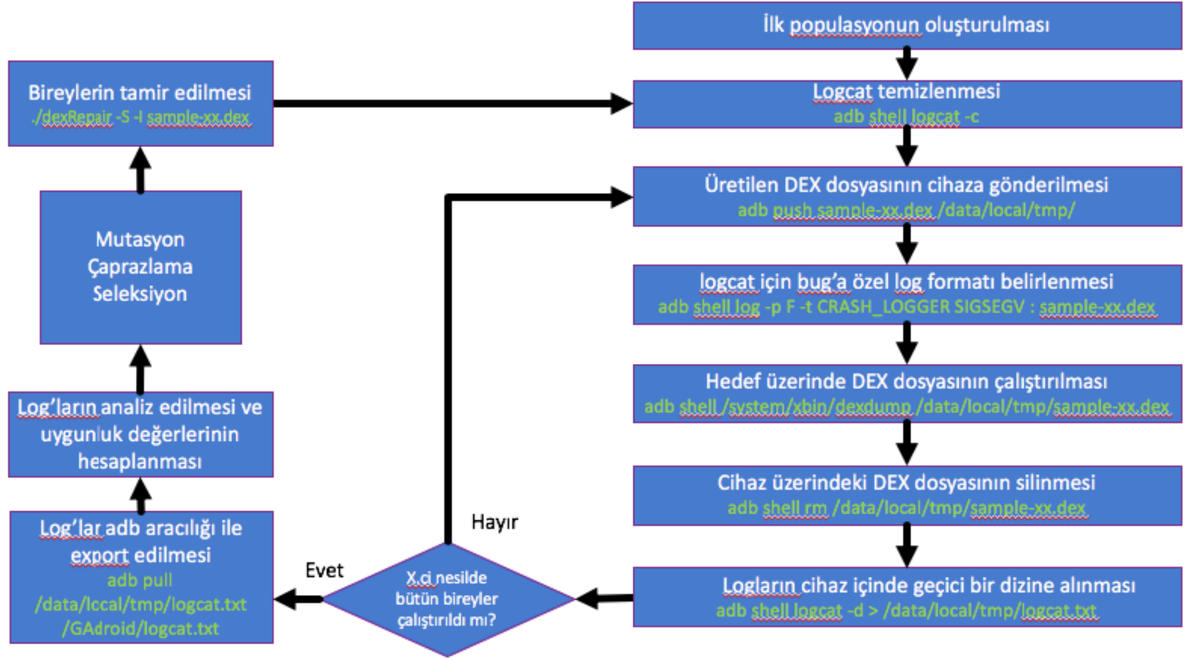
Fuzz Testi Aracı	Zafiyet Tespit Yöntemi	Hedef Türü	Fuzz Testi Yöntemi
Dynodroid [37]	Kapalı Kutu	UI/Sistem	Mutasyon Tabanlı
DroidFuzzer [70]	Kapalı Kutu	Sistem	Mutasyon Tabanlı
Monkey [36]	Kapalı Kutu	UI	Mutasyon Tabanlı
AFL [28]	Açık Kutu	UI/Sistem	Mutasyon Tabanlı
Aimdroid [66]	Açık Kutu	Sistem	Model Tabanlı
Bifuz [69]		Sistem	Model Tabanlı
Chizpurfle [65]	Gri Kutu	Sistem	Mutasyon Tabanlı
Evofuzz [59]			Evrimsel Tabanlı
KamelonFuzz [60]	Kapalı Kutu	UI	Evrimsel Tabanlı
Mamba [56]	Gri Kutu	Sistem	Model Tabanlı
GAFuzzing [49]			Evrimsel Tabanlı
DexFuzz [78]	Kapalı Kutu	Sistem	Mutasyon Tabanlı
Droid-FF [76]	Kapalı Kutu	Sistem	Mutasyon Tabanlı
d'ART [82]	Açık Kutu	Sistem	Model Tabanlı
GAdroid	Kapalı Kutu	Sistem	Evrimsel Tabanlı

4. GADROID

Fuzz güvenlik testlerinin mobil platformdaki uygulamalar üzerinde kullanılması yeni bir konudur. Bu alanda kısıtlı çalışmalar gerçekleştirilmiştir. Bu tez çalışmasında, Android uygulamalarının kurulumu aşamasında kullanılan derleyici üzerinde güvenlik riskleri teşkil edecek zafiyetleri tespit etme amaçlanmıştır. Önerilen yaklaşım ile DEX dosyası üzerinden etkili test durumlarının verimli bir zamanda üretilmesi hedeflenmektedir.

Bu tez kapsamında geliştirilen genetik algoritma tabanlı fuzz testi platformu GAdroid olarak adlandırılmıştır. GAdroid oluşturulurken, birden çok platformda çalışılabilmesi için Python programlama dili tercih edilmiştir. GAdroid ile hata ayıklayıcı altında adb [7] komutları ile emülatördeki */system/xbin/dexdump* çalıştırılabilir dosyasına bozulmuş DEX dosyaları gönderilerek fuzz testi yapılmaktadır. dexdump, DEX dosyası üzerinden Dalvik VM ikili kod'unu üretmektedir. Çalıştırılabilir dosyaları hazır hale getirmekiçin derleme işlemini sağlayan çalıştırılabilir dosyadır. Bozulmuş DEX dosyaları çalıştırılması sonrasında üretilen çıktılar analiz edilerek, genetik algoritmanın her iterasyonunda fuzz testi için daha etkin dosyalar oluşturulması hedeflenmektedir. Testlerin gerçekleştirilmesi için Android emulator [83] kullanılmış ve 2 GB belleğe sahip, Android 5.1.1 sürümünün koştuğu Nexus 4 cihazının benzetimi oluşturulmuştur. Android emulator [83], komut satırı üzerinden çalıştırılmaktadır. Komutlar, Android cihaza adb *(Android Debugger Bridge)* [7] kullanılarak gönderilmektedir. Komutların adb aracılığı ile doğrudan Android işletim sistemi çekirdeğine gönderilmesinin en büyük avantajı, Android uygulama paketlenmesi/açılması işlemlerine gerek kalmaması ve Android işletim sisteminde daha alt seviyelerde zafiyetlerin tetiklenme olasılığını arttırmasıdır. Emulatorun adb üzerinden istemci ile haberleşmesi için haberleşme protokolü olarak TCP *(kapı: 5037)* tercih edilmiştir.

Şekil 16. GAdroid’in adımları



4.1. İlk Veri Üretimi

Şekil *16*’da GAdroid’in temel adımları verilmiştir. Öncelikle çözüm için önerilen bireylerden ilk nesil oluşturulmuştur. Bu tez çalışmasında, fuzz testine girdi olarak DEX dosyaları tercih edilmiş, bu dosyalar üzerinde değişiklik yapılarak dexdump üzerindeki zafiyetler bulunmaya çalışılmıştır. Önceden rasgele toplanmış 20 Android uygulama örneklerine ait *classes.dex* dosyaları kullanılarak ilk nesil oluşturulmuştur.

4.2. Bozulmuş DEX Dosya Formatı Tamiri

Genetik işleçler ile DEX dosyaları üzerinde değişiklikler yapılırken, geçerli DEX dosyaların oluşabilmesi için bazı hususlara dikkat edilmesi gerekmektedir. DEX dosyasının başlık bölümünde birkaç kontrol mekanizması bulunmaktadır. Fuzzing işleminin geçerli olması ve derleme işleminin reddedilmemesi için DEX dosya bölümlerinin tutarlı olması gerekmektedir.

Şekil 17. Değiştirilmemesi ve tamir edilmesi gereken DEX bölümleri

struct header_item dex_header		0h	70h	Dex file header
struct dex_magic magic	dex 035	0h	8h	Magic value
uint checksum	B3D20217h	8h	4h	Alder32 checksum of rest of file
SHA1 signature[20]	6DB8EDA774	Ch	14h	SHA-1 signature of rest of file
uint file_size	1430508	20h	4h	File size in bytes
uint header_size	112	24h	4h	Header size in bytes
uint endian_tag	12345678h	28h	4h	Endianness tag
uint link_size	0	2Ch	4h	Size of link section
uint link_off	0	30h	4h	File offset of link section
uint map_off	1430336	34h	4h	File offset of map list
uint string_ids_size	11029	38h	4h	Count of strings in the string ID list
uint string_ids_off	112	3Ch	4h	File offset of string ID list
uint type_ids_size	2068	40h	4h	Count of types in the type ID list
uint type_ids_off	44228	44h	4h	File offset of type ID list
uint proto_ids_size	2592	48h	4h	Count of items in the method prototype ID list
uint proto_ids_off	52500	4Ch	4h	File offset of method prototype ID list
uint field_ids_size	5335	50h	4h	Count of items in the field ID list
uint field_ids_off	83604	54h	4h	File offset of field ID list
uint method_ids_size	12925	58h	4h	Count of items in the method ID list
uint method_ids_off	126284	5Ch	4h	File offset of method ID list
uint class_defs_size	1427	60h	4h	Count of items in the class definitions list
uint class_defs_off	229684	64h	4h	File offset of class definitions list
uint data_size	1155160	68h	4h	Size of data section in bytes

Şekil *17*'de verilen DEX dosya formatı üzerinde sabit alanlar bulunmaktadır. Bunlar *magic*, *endian tag* ve *header size* alanlarıdır. *Magic* değeri DEX dosya formatı olduğunu, *endian tag* verilerin sıralanma şeklini, *header size* ise DEX dosya başlık boyutunu gösterir. Başlık bölümündeki bu alanların değiştirilmemesi gerekmektedir. Bunun dışında *file size*, *checksum* ve *SHA-1* bölümleri ise her DEX dosyasında farklı olan ve hesaplanması gereken kontrol bölümleridir. *file size* dosya boyutunu, *checksum* Adler-32 algoritması ile oluşturulmuş dosyanın imzası, *SHA-1* ise dex dosyanın 160 bitlik kriptografik özet bilgisidir. Fuzz testi aracı DEX dosyası üzerinde değişiklikler yaptığında bu kontrol bölümleri tutarsız olacaktır. Dolayısıyla yeni üretilen DEX dosyasında bu bölümlerin de tekrardan hesaplanması ve değiştirilmesi gerekmektedir. Bozulmuş DEX dosyalarının tamiri için açık kaynak kodlu dexRepair [84] aracı kullanılmıştır. DEX tamir süreci sonrasında fuzz testi süreci başlamaktadır. Genetik algoritma ile elde edilen DEX dosyaları, Android emulator üzerinde çalıştırılır ve varsa hata kayıtları izlenir. Sonrasında dex dosyaları ile bu dosyaya ilişkin kayıtlar incelenerek bu dosyalara

ilişkin bireylerin uygunluk değerleri hesaplanır ve genetik işlemlere tabi tutulurlar.

4.3. Fuzz Testi Süreci

Şekil *16*'da GAdroid'in temel adımları verilmektedir. Algoritma, ilk popülasyonun oluşturulması ile başlar. Rasgele oluşturulan ilk popülasyondaki bireyler, yani DEX dosyaları Android cihaza gönderilerek tek tek fuzz testine tabi tutulurlar.

Öncelikle cihaz üzerinde önceden oluşmuş bütün kayıtlar temizlenir. Her yeni birey, DEX cihaza gönderildiğinde kayıt çıktılarının ayırt edilebilirliği için kayıt çıktısı formatı belirlenir. DEX dosyası, adb aracılığı ile TCP Port 5037 kapısı üzerinden lokaldeki cihazdan emulatöre gönderilir. Dex dosyası, erişebileceğimiz ve yazma hakkımızın da bulunduğu *tmp* dizininin altına yerleştirilir ve *dexdump* uygulamasına gönderilir. Çalıştırma süreci sonrasında *tmp* dizini altındaki DEX dosyası silinir ve elde edilen kayıtlar, aynı dizin altında saklanır. Popülasyondaki bütün bireyler çalıştırıldığında elde edilen çıktılar incelenir. Öncelikle çıktılar üzerinden tekillik analizi yapılır, eğer yeni bir güvenlik açıklığı tespit edilmişse, bu açığa neden olan Gadroid ile elde edilen DEX dosyası ve bu dosyanın çalıştırılması sonucu elde edilen kayıtlar kaydedilir. Sonrasında, bu kayıtlara göre her bir bireyin uygunluk değeri hesaplanır ve uygunluk değerlerine göre bireyler üzerinde genetik operatörler uygulanır.

4.4. Kayıtların Toplanması ve Analizi

Kayıtların toplanması ve analizi süreci ile yeni fuzz testlerinin gerçekleştirilmesi için genetik algoritmanın yeni nesillerinin oluşturulması sağlanır. Fuzz testi sonrası her nesilde oluşan çakılma kayıtları, cihazdan lokale adb aracılığıyla çıkarılmaktadır. Dışa aktarılan bu kayıtlar, yeni neslin üretilebilmesi için analiz edilerek uygunluk değerlerinin oluşturulmasını sağlamaktadırlar.

Dışa aktarılan kayıtlar, kritik olup olmadıkları ve sıralarının belirlenebilmesi için hata sinyallerine göre analiz edilmektedirler. Eğer *SIGSEGV*, *SIGABRT*, *SIGFPE*, *SIGILL* sinyallerinden birisini içeriyorsa ve

F/libc ile *F/CRASH_LOGGER* kelime dizgilerine sahipse yeni bir program hatası tespit edilmiş demektir. *SIGSEGV* segmantasyon hatası alındığında, *SIGABRT* libc kütüphanesi yanlış kullanıldığında, *SIGFPE* hatalı aritmetik kullanımında, *SIGILL* ise geçersiz komut kümesi kullanıldığına karşımıza çıkmaktadır. Hata kaydı içerisinde *F/libc* ile *F/CRASH_LOGGER* kelimeleri geçiyorsa da kritik kütüphane veya sistem seviyesinde bir program hatası elde edilmiş demektir. Bu otomatik inceleme sonrasında ilgili program hatası, fuzz testi süresince daha önce tespit edilen program hataları ile karşılaştırılır. Eğer tekil bir yazılım hatası tespit edilmişse kaydedilir, tekrardan ibaret ise silinmektedir. Örnek bir kayıt, *Şekil 18'de* verilmektedir.

Şekil 18. Bir bireye ilişkin örnek bir kayıt

```
Muh4f1Z-MacBook-Air:droid-ff root# more logcat.txt
--------- beginning of /dev/log/system
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2279
D/MobileDataStateTracker(  356): default: setPolicyDataEnable(enabled=true)
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2287
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2294
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2308
D/MobileDataStateTracker(  356): default: setPolicyDataEnable(enabled=true)
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2326
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2333
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2340
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2354
D/MobileDataStateTracker(  356): default: setPolicyDataEnable(enabled=true)
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2369
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2379
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2386
D/MobileDataStateTracker(  356): default: setPolicyDataEnable(enabled=true)
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2394
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2408
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2422
D/MobileDataStateTracker(  356): default: setPolicyDataEnable(enabled=true)
W/NativeCrashListener(  356): Couldn't find ProcessRecord for pid 2433
--------- beginning of /dev/log/main
I/DEBUG   (   54):          bec15950  b6e69f28  /system/lib/liblog.so
I/DEBUG   (   54):          bec15954  b6e669d1  /system/lib/liblog.so
I/DEBUG   (   54):          bec15958  b6f10678  /system/bin/linker
I/DEBUG   (   54):          bec1595c  b6f0be81  /system/bin/linker
I/DEBUG   (   54):          bec15960  b6f0c9e8  /system/bin/linker
I/DEBUG   (   54):          bec15964  00000004
I/DEBUG   (   54):          bec15968  b6f0be81  /system/bin/linker
I/DEBUG   (   54):     #00  bec1596c  b6dbf000  /data/local/tmp/sample275921.dex
I/DEBUG   (   54):          bec15970  00086ba3
I/DEBUG   (   54):          bec15974  14f42474
I/DEBUG   (   54):          bec15978  bec15998  [stack]
I/DEBUG   (   54):          bec1597c  00000000
I/DEBUG   (   54):          bec15980  00000000
I/DEBUG   (   54):          bec15984  b6ed635c
I/DEBUG   (   54):     #01  bec15988  00086ba3
I/DEBUG   (   54):          bec1598c  b6f101d0  /system/bin/linker
I/DEBUG   (   54):          bec15990  b6f10670  /system/bin/linker
I/DEBUG   (   54):          bec15994  b6f02959  /system/bin/linker
I/DEBUG   (   54):          bec15998  00000000
I/DEBUG   (   54):          bec1599c  b6f10ccc  /system/bin/linker
I/DEBUG   (   54):          bec159a0  b6f12000  /system/xbin/dexdump
```

4.5. Uygunluk Fonksiyonu Bölümleri

Fuzz testi döngüsü sonrası elde edilen kayıtlara göre uygunluk değerleri belirlenmiştir. Uygunluk fonksiyonunda öncelikle, ilgili bireyin program çakılmasına neden olup olmadığıdır. Bazı bireyler, program çakılmasına neden olmazken, bazıları olabilir. Eğer bir birey program çakılmasına neden olmazsa, o bireye en düşük uygunluk değeri atanır. Sonrasında, uygunluk fonksiyonunda üretilen hata sinyallerine bakılır. *SIGSEGV*, *SIGABRT*, *SIGFPE*, *SIGILL* sinyallerinden birisi tetiklenmiş ise, bu çakılmanın kritik olduğunu gösterir. Yani ilgili çakılma, bir güvenlik zafiyeti ile ilişkili olabilir. Diğer dikkat edilen husus, ilgili çakılmanın hangi kütüphaneden kaynaklandığıdır. Eğer çakılma, bir yerel kütüphanede tetiklendi ise, bir JAVA kütüphanesinde tetiklenen çakılmalardan daha kritik bir güvenlik zafiyetini işaret eder. Diğer önemli bir kriter, elde edilen çakılmanın, tekil bir program hatası olup olmadığı, o ana kadar genetik evrim ile elde edilen diğer hatalardan farklı olup olmadığıdır. Eğer yeni bir çakılma ise, bu bireyin uygunluk değeri yüksek olacak, sonraki iterasyonlarda yeni test durumlarının oluşmasında etkili olabilme olasılığı yüksek olacaktır. Her bir bireye bu değerlerin anlamlandırdığı, 7 elemanlı bir dizi atanır. Bu dizi baz alınarak her bir kriter ağırılık değerine göre o bireyin uygunluk değeri hesaplanmaktadır. İlk eleman program çakılması olup olmadığıdır. Eğer çakılma olduysa, değeri 1, değilse 0 yapılır. 2-5. elemanlar ise sırasıyla *SIGSEGV*, *SIGABRT*, *SIGFPE*, *SIGILL* sinyallerinden birisi ile tetiklenip tetiklenmediği kontrolüdür. 6. eleman ise Android yerel (NDK) bir kütüphanede bir program çakılması olup olmadığıdır. Son eleman ise elde edilen program çakılmasının daha önce de üretilip üretilmediğidir.

$$X.Birey\ Gösterimi = [1100010]$$

Yukarıda, bir X bireyi için fuzz testi sonrası elde edilen kayıtın düzenlenmiş hali gösterilmektedir. Bu değere göre, bu birey ile yerel bir kütüphanede program çakılması gerçekleştirilmiş ve SIGSEGV sinyali üretilmiştir. Bu çakılma, daha önce evrimleştirilmiş bireyler ile daha önce

de gerçekleştirilmiştir. Burada, o ana kadar GAdroid algoritmasının farklı çalıştırılmaları sonucu oluşturulan tüm çakılmalara bakılarak bir değer atanır, sadece o andaki çalıştırılmasına bakılmaz.

Uygunluk değeri hesaplamasında; kaynak koda ihtiyaç duymadan kapalı kutu fuzz testi uygulamaları için önerilen bir çalışmadan esinlenerek [53] uygun olan *Markov Model* [54] tercih edilmiştir.

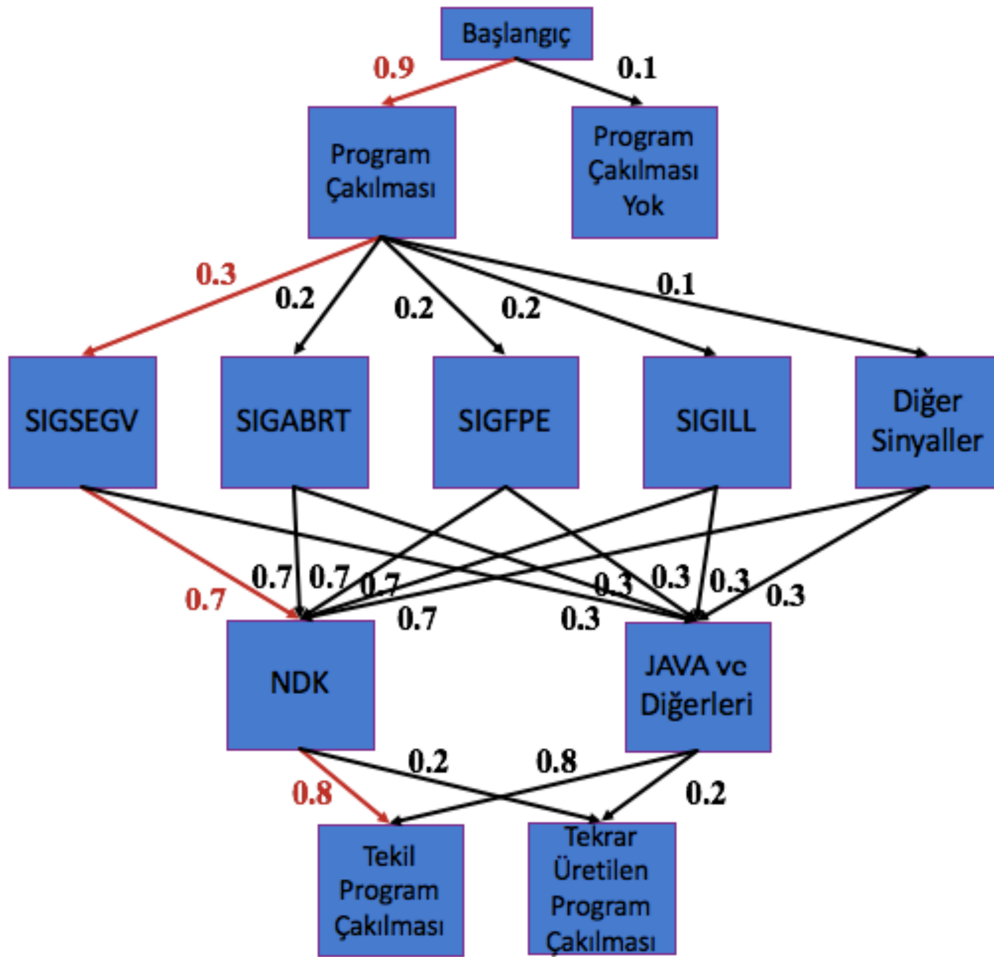
$$Uygunluk\ Fonksiyonu(X.Birey) = \frac{1}{\Pi P_i}$$

P_i değeri yukarıda akışta verilen ilerleme adımlarına göre oluşan her biri 0 ile 1 arasındaki olasılıksal değerler zinciridir. Belirlenen değerler ile ilgili bireyin yukarıdaki örnekte verilen ağırlık dizisindeki değerlerlere göre P_i hesaplamasına dahil olur. Bir program çakılması varsa 0.9, yoksa 0.1 seçilir. *SIGSEGV*, *SIGABRT*, *SIGFPE*, *SIGILL* sinyallerinden birisi tetiklenmiş ise sırasıyla 0.3, 0.2, 0.2, 0.2 değerleri, başka bir sinyal var ise 0.1 değeri verilir. Android yerel bir kütüphanede (NDK) bir program çakılması var ise 0.7 değeri, yoksa 0.3 seçilir. Son olarak program çakılmasının daha önce de üretilmiş ise 0.2, değilse 0.8 değeri verilir.

Şekil *19*. Markov kontrol akış grafiği'da görülen Markov kontrol akış grafiğinde uygunluk değeri katsayıları verilmiştir. Örneğin; $X.Birey$=[1100010] dizisine sahip bir bireyin uygunluk değeri, kırmızı oklar üzerinden ilerlendiğinde bu katsayıların çarpımı ile aşağıdaki gibi oluşacaktır.

Uygunluk Değeri = 1 / (0.9*0.3*0.7*0.8) = 6.613

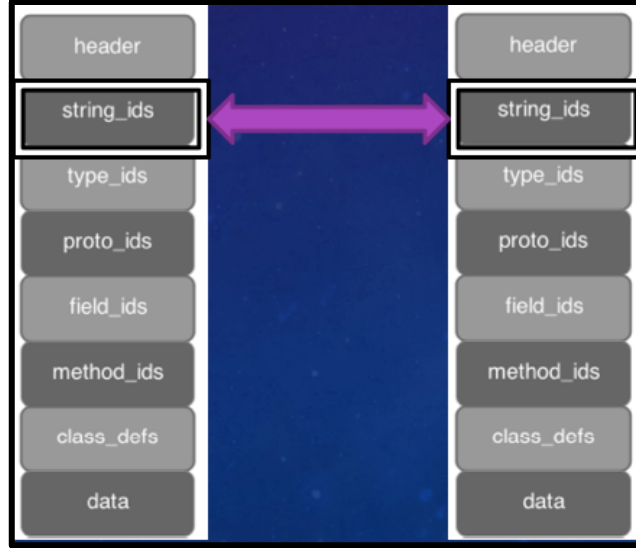
Şekil 19. Markov kontrol akış grafiği



Uygunluk değerlerine göre seçilen bireyler üzerinde çaprazlama ve mutasyon işleçleri uygulanarak yeni bireyler elde edilir. Şekil *20*. DEX dosyaları üzerinde çaprazlama işleci gibi bireylerin, yani DEX dosyalarında katar dizgisi değerlerinin yazıldığı *string_ids* alanı üzerinden çaprazlama işleci uygulanır. İki bireydeki aynı alanda ve aynı noktada yer alan bit örüntüleri yer değiştirilir. Mutasyon işlecinde ise, DEX dosyasında rasgele bit sayısında *string_ids* bölümünde rasgele yerde bit bazında değişiklik yapılır. Bu işleçler uygulandıktan sonra classes.dex dosyası tekrar paketlenir ve tamir sürecinden geçirilir. Yapılan değişiklikler sonucu etkilenen dosya boyutu, sağlama, özetleme gibi alanlar güncellenir. Bu güncelleme ile geçerli bireyler oluşturulmuş olur. *string_ids* alanı, katar dizgilerinden oluştuğu için tercih edilmiştir. Diğer

bir tercih sebebi ise katar dizgilerinin olduğu bölümler, bellek taşması gibi daha kritik ve sömürülebilir zafiyetleri içerme olasılığı çok yüksek olmasıdır. Gelecek çalışmalarda diğer katar dizgilerinden oluşan örneğin *method_ids* ve *class_ids* alanlarına da fuzz testi gerçekleştirilebilir.

Şekil 20. DEX dosyaları üzerinde çaprazlama işleci



Genetik algoritma için Clinton Sheppard tarafından geliştirilen Phyton kütüphanesi [85] kullanılmış ve probleme göre modifiye edilmiştir. Genetik algoritmanın parametleri ise aşağıda verilmiştir.

Parametre Adı	Değeri
Çaprazlama oranı	0.2
Mutasyon oranı	0.5
İterasyon sayısı	5000
Popülasyon büyüklüğü	20
Seçim yöntemi	turnuva
Çaprazlama yöntemi	tek nokta
Mutasyon yöntemi	bit katar dizgisi

5. DENEY SONUÇLARI

Bu bölümde, öncelikle deneyler için hazırlanan test ortamı anlatılacak. Sonrasında deney sonuçları verilip, analiz edilecektir.

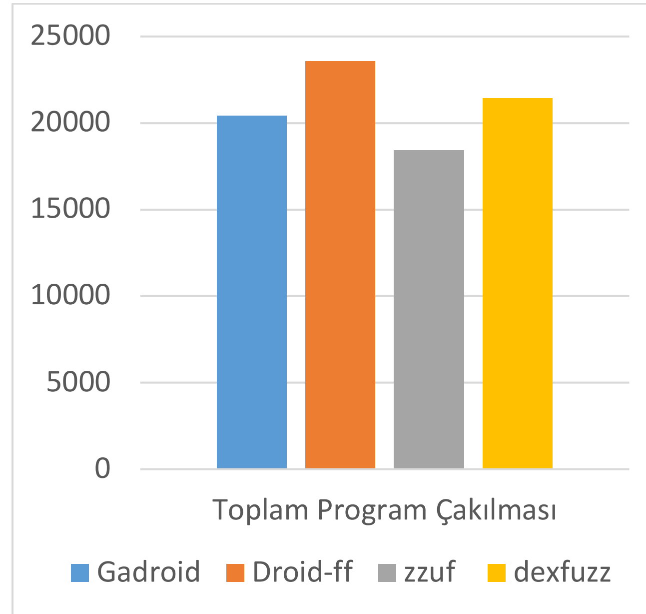
Genetik algoritma ile evrimleştirilen fuzz testleri, Android emülator üzerinde gerçekleştirilmiştir. Simüle edilen cihaz, 2GB belleğe sahip, üzerinde Android 5.1.1 koşan bir Nexus 4 cihazıdır. Android işletim sistemi çakılma tespiti yapılabilmesi için testler adb hata ayıklayıcı [7] altında gerçekleştirilmiştir. Testlerde, Gadroid algoritması ardarda durmaksızın iki hafta boyunca yaklaşık olarak 80 kez çalıştırılmış, 400.000 iterasyon gerçekleştirilmiştir.

Yapılan deneyler sonucunda, GAdroid ile toplam 20.452 tane program çakılması elde edilmiştir. Bunlardan çoğu tekrar eden çakılmalar iken, 305 tanesinin tekil olduğu görülmüştür. Bu tekil zafiyetlerden 11 tanesi kritik, bunlardan birisinin ise daha önce tespit edilmediği görülmüştür. Sıfırıncı gün zafiyeti olarak düşünülen bu zafiyetin, Ek 2' deki gibi uluslarası zafiyet veri tabanlarına eklenmesi için başvurularda bulunulmuştur. Böylelikle ilgili zafiyetin uluslarası düzeyde kabul görmesi için CVE ve OSVDB kimlik numaraları alınabilecektir. Ayrıca, uluslarası kabul görmüş zafiyetlerin herkese açıklandığı zafiyet bildirim siteleri olan Packet Storm ve Security Focus *(Seclist - Bugtraq Digest)* adreslerine bildirimi yapılmıştır.

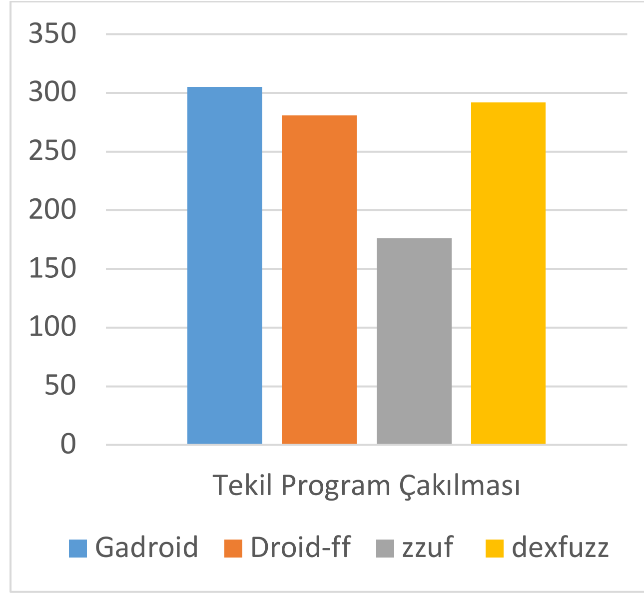
GAdroid, Droid-ff [76], DexFuzz [79] ve zzuf [77] fuzz testi araçları ile karşılaştırılmıştır. Droid-ff [76] ve DexFuzz [79] araçları, özellikle GAdroid gibi DEX dosyası üzerinden fuzz testi yapıldığı için tercih edilmiştir. Android kurulum sürecisini hedef alan diğer bir araç d'ART'tır, fakat ilgili çalışmanın koduna, erişim olmadığı için ulaşılamamıştır. d'ART içerisinde ve birçok fuzz testi içerisinde zzuf fuzz testi aracı kullanıldığı için karşılaştırmalarda zzuf da tercih edilmiştir. Tüm araçlar aynı test ortamında iki hafta boyunca çalıştırılmışlardır.

Tablo 1. Toplam program çakılma sayısına göre karşılaştırma*'de* fuzz testi araçları tespit ettikleri toplam program çakılmalarına göre incelenmiştir. Görüldüğü üzere, en çok sayıda program çakılması Droid-ff tarafından üretilmiştir. GAdroid 20.452, Droid-ff 23.565, DexFuzz 21.430, zzuf ise 18.342 program çakılması üretmiştir. Gadroid, Droid-ff [76], zzuf [77] ve DexFuzz [79] ile karşılaştırılabilir sonuçlar vermiştir. Fakat, *Tablo 2.* Tekil program çakılma sayısına göre karşılaştırma*'de* gösterildiği gibi, elde edilen tekil program çakılmaları incelendiğinde en yüksek başarımı GAdroid üretmiştir. Bu değer, toplam program çakılma sayısından çok daha önemlidir. Bu açıdan verimlilik oranları şöyle verilebilir : GAdroid %1.5, Droid-ff %1.1, zzuf %1 ve DexFuzz %1.3. Genetik algoritmanın uygunluk değerinde, tekrarlı program çakılmaları geldiğinde, bu durum uygunluk değerinin azaltılması ile cezalandırılmakta, bu da evrimi yeni program çakılmalarının tespitine yönlendirmektedir. Bu durum, GAdroid'in aynı sürede en fazla tekil program çakılmasını elde etmesini sağlamıştır.

Tablo 1. Toplam program çakılma sayısına göre karşılaştırma



Tablo 2. Tekil program çakılma sayısına göre karşılaştırma



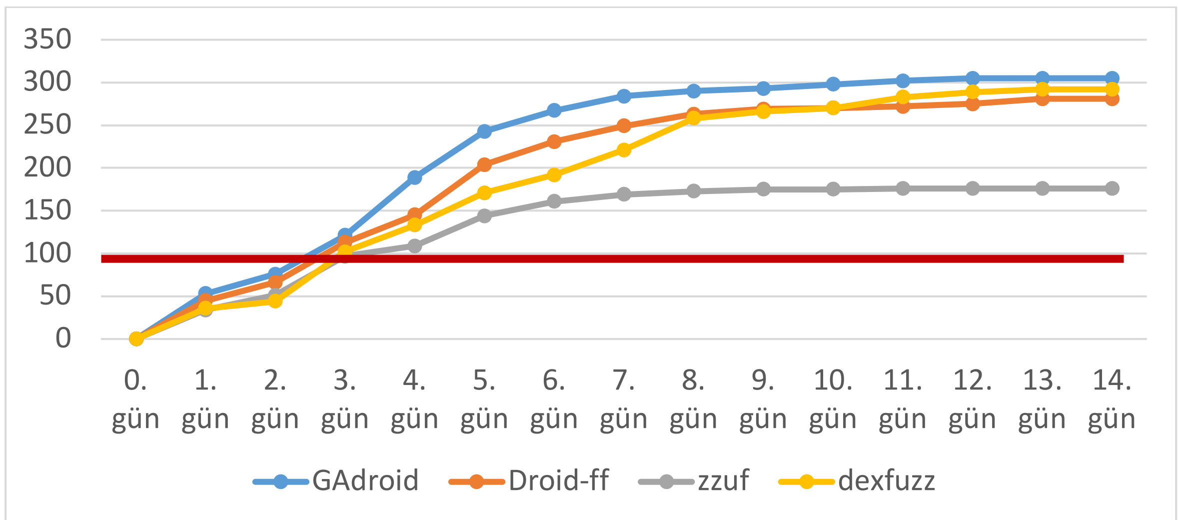
Tablo 3. Sıfırıncı-gün zafiyeti keşfine göre literatür karşılaştırması'*te* ise daha önce keşfedilmemiş ve bu araçlar tarafından bulunmuş yeni zafiyetlerin karşılaştırması yapılmıştır, Droid-ff [76] tarafından keşfedilen üç adet sıfırıncı gün zafiyeti, GAdroid ve DexFuzz [79] tarafından da keşfedilmiştir. Ama bu üç zafiyet öncelikle Droid-ff [76] tarafından bulunduğu için, tabloda sadece Droid-ff [76] için belirtilmiştir. Dolayısıyla, GAdroid'in bu zafiyetleri bulma potansiyeli olmasına rağmen, Droid-ff [76]'den sonra önerilen bir sistem olduğu için bu zafiyetleri de daha geç bulmuştur. GAdroid, bunlara ek olarak yeni bir sıfırıncı gün zafiyeti daha bulmuştur. Aynı süre zarfında zzuf [77] tarafından herhangi bir sıfırıncı gün zafiyeti keşfedilememiştir.

Tablo 3. Sıfırıncı-gün zafiyeti keşfine göre literatür karşılaştırması

Fuzz Testi Araçları	Diğerleri ile ortak bulunan tekil güvenlik zafiyeti sayısı	Yeni bulunan sıfırıncı gün zafiyeti sayısı
GAdroid	305	1
Droid-ff [76]	281	3
zzuf [77]	176	-
DexFuzz [79]	292	-

Tablo 4. Zamana göre grogram çakılması analizi grafiği'te ise günlere göre tespit edilmiş tekil program çakılması analizi yapılmıştır. İlk 5 güne baktığımızda 5. gün sonunda GAdroid aracının aynı süre içerisinde daha çok tekil program çakılması tespit ettiği görülmüştür. Bunun dışında 150 tekil program çakılmasına GAdroid ilk 3. günde ulaşırken, Droid-ff [76] ve DexFuzz [79] 4. günde, zzuf [77] aracı ise 6. günde ulaşabilmiştir. Bu da GAdroid aracı ile aynı tekil program çakılması sayısına daha kısa sürede ulaşabildiğini göstermiştir.

Tablo 4. Zamana göre grogram çakılması analizi grafiği



Günün sonunda bu tez kapsamında elde edilen keşfedilmiş sıfırıncıgün zafiyetine ilişkin kayıt, Ek 1'de verilmiştir. Bu güvenlik zafiyetinin ayrıntılarına baktığımızda *SIGSEGV* sinyalinin üretildiğini görmekteyiz. Bu da bellek taşması zafiyetinin tetiklendiğini göstermektedir. İşlem ağacı *(Backtrace)* çıktısı bölümüne baktığımızda, bulunan güvenlik açığının *"/system/lib/libz.so"* kütüphanesinde keşfedildiğini ve bu kütüphanenin Android alt katmanlarında bulunan yerel bir Android çekirdek kütüphanesi (NDK) olduğu anlaşılmaktadır. Dolayısıyla bu zafiyet, Android uygulamalarından bağımsız olan bu kütüphanenin kullanıldığı bütün Android cihaz ve sürümlerini etkilemektedir.

6. SONUÇ

Günümüzde, saldırganların en çok hedef aldığı platform mobil cihazlardır. Android, mobil cihazlarda en yaygın olarak kullanılan işletim sistemidir. Bir yandan bu sistemin güvenliği arttırılırken, diğer bir yandan bu sisteme ilişkin birçok yeni zafiyet bulunabilmektedir. Dolayısıyla mobil cihazlardaki bu güvenlik zafiyetlerinin, saldırganlardan önce tespit edilmesi önem kazanmaktadır. Fuzz testlerinin mobil ortamda kullanılması ile daha öncesinde tespit edilememiş birçok güvenlik zafiyeti tespit edilip kapatılabilmektedir. Bu tez çalışmasında, Android sistemini hedef alan yeni zafiyetlerin tespiti için genetik algoritma tabanlı bir fuzz testi yaklaşımı önerilmiştir. Bu yaklaşım ile yeni DEX dosyaları evrimleştirilmiş ve Android kurulum sürecine ilişkin zafiyetlerin bulunması amaçlanmıştır. Böylelikle, Android uygulamalardan bağımsız, ilgili Android versiyonunu kullanan tüm cihaz ve kullanıcıları etkileyen zafiyetler bulunmuştur.

GAdroid ismi verilen bu yaklaşım ile, iki hafta boyunca fuzz testleri gerçekleştirilmiştir. Bu testler sonucunda toplam 20.452 program çakılması elde edilmiş ve bunlardan 305 tanesinin biricik olduğu görülmüştür. Bu tekil zafiyetlerden 11 tanesinin kritik olduğu belirlenmiştir. Bu kritik zafiyetler ayrıntılı incelendiğinde, bir tanesinin sıfırıncı gün zafiyeti olduğu tespit edilmiştir. Bu zafiyete ilişkin, uluslarası zafiyet veri tabanlarına zafiyet bildirimlerinde bulunulmuştur.

GAdroid, aynı test ortamında Droid-ff [76], DexFuzz [79] ve zzuf [77] fuzz testi araçları ile karşılaştırılmıştır. Özellikle Droid-ff [76] ve DexFuzz [79] araçları, GAdroid gibi DEX dosyası üzerinden fuzz testi yapıldığı için tercih edilmiştir. Tüm araçlar aynı süre boyunca çalıştırılmıştır. Yapılan karşılaştırmalarda, GAdroid'in aynı sürede daha çok biricik çakılma elde

ettiği gözlemlenmiştir. Ayrıca, GAdroid biricik çakılmaları, diğer araçlara kıyasla çok daha kısa sürede elde etmiştir.

Gelecekte, GAdroid aracı geliştirilebilir. Örneğin, DEX dosyasındaki *string_ids* alanı dışındaki alanlar üzerinde de değişiklikler yapılarak, yeni zafiyetlerin tespit edilmesi hedeflenebilir. Ayrıca, bu fuzz testlerinin bulut ortamına taşınması üzerine çalışılabilir. Böylelikle fuzz testlerinin eş zamanlı olarak gerçekleştirilmesi ve birçok kaydın depolanmasından faydalanılabilir. Dahası ileride, evrimleştirilen zafiyetler ile Android'in sömürülüp sömürülemeyeceğini analiz eden otomatik bir araç geliştirilip, bu yaklaşıma entegre edilebilir.

7. KAYNAKLAR

[1] W. A. Social, 2017 Digital Yearbook, https://www.slideshare.net/wearesocialsg/2017-digital-yearbook (Ağustos, **2018**).

[2] AppBrain, Number of Android applications, https://www.appbrain.com/stats/number-of-android-apps (Ağustos, **2018**).

[3] A. Ludwig, *Android Security 2016 Year in Review*, Google Rep. 2016, no. March, p. 71, **2016**.

[4] A. Pedram, A. Portnoy, *Sulley fuzzing framework*. **2010**.

[5] Google, Android Architecture, https://developer.android.com/guide/platform/ (Ağustos, **2018**).

[6] Google, ART ve Dalvik, https://source.android.com/devices/tech/dalvik/ (Ağustos, **2018**).

[7] Google, Android Debug Bridge (adb), https://developer.android.com/studio/command-line/adb (Ağustos, **2018**).

[8] S. Tajbakhsh, The Android Structure, https://www.google.com.tr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjqrqSB2ZfdAhVIaFAKHf2OBfAQjRx6BAgBEAU&url=https%3A%2F%2Fmstajbakhsh.ir%2Fsmali-code-injection%2F&psig=AOvVaw3y1arXv2IPb4uNscGXFKvL&ust=1535818797536699 (Ağustos, **2018**).

[9] Google, Dalvik Executable format, https://source.android.com/devices/tech/dalvik/dex-format (Ağustos, **2018**).

[10] A. Blanda, *Fuzzing Android: a recipe for uncovering vulnerabilities inside system components in Android*, p. 16, **2015**.

[11] Anonim, First actual case of bug being found, https://www.webdevelopersnotes.com/the-first-computer-

bug-was-a-moth (Ağustos, **2018**).

[12] S. Mittal, Heap overflow and Stack overflow, https://www.geeksforgeeks.org/heap-overflow-stack-overflow/ (Ağustos, **2018**).

[13] Anonim, Heap-based Buffer Overflow, https://cwe.mitre.org/data/definitions/122.html (Ağustos, **2018**).

[14] Anonim, Heap overflow, https://webcache.googleusercontent.com/search?q=cache:7EWJe76CBjIJ:https://en.wikipedia.org/wiki/Heap_overflow+&cd=1&hl=tr&ct=clnk&gl=tr (Ağustos, **2018**).

[15] Anonim, Format String Vulnerability, http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf, (Ağustos, **2018**).

[16] Koziol, Jack, *The shellcoder's handbook: discovering and exploiting security holes*, New York: Wiley, **2004**.

[17] Anonim, OWASP-Source Code Analysis Tools, https://www.owasp.org/index.php/Source_Code_Analysis_Tools (Ağustos, **2018**).

[18] C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, Boston, London: Artech House, **2008**.

[19] Anonim, Fuzzing, https://webcache.googleusercontent.com/search?q=cache:08wBD2qnpjkJ:https://en.wikipedia.org/wiki/Fuzzing+&cd=12&hl=tr&ct=clnk&gl=tr (Ağustos, **2018**).

[20] M. Sutton, *Fuzzing Brute Force Vulnerability Discovery.pdf*, Addison-Wesley, **2007**.

[21] Anonim, Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution, https://docs.microsoft.com/en-us/security-updates/securitybulletins/2004/ms04-028 (Ağustos, **2018**).

[22] R. Marcos, ProxyFuzz - On-The-Fly Proxy Fuzzer, http://www.securiteam.com/tools/5LP021FL5M.html (Ağustos, **2018**).

[23] G. Evron, N. Rathaus, *Open Source Fuzzing Tools*, **2007**.

[24] Anonim, Zero-day, https://webcache.googleusercontent.com/search?q=cache:gobsKmwyJ_YJ:https://en.wikipedia.org/wiki/Zero-day_(computing)+&cd=4&hl=tr&ct=clnk&gl=tr (Ağustos, **2018**).

[25] J. Brown, PDFuzzer, http://www.securiteam.com/tools/6R0052KN5I.html (Ağustos, **2018**).

[26] M. Vuagnoux, Autodafe, http://autodafe.sourceforge.net/tutorial/index.html (Ağustos, **2018**).

[27] J. Godinez, EFS, https://javiergodinez.com/wp/?p=16 (Ağustos, **2018**).

[28] M. Zalewski, AFL (american fuzzy lop), http://lcamtuf.coredump.cx/afl/README.txt (Ağustos, **2018**).

[29] J. Holland, https://en.wikipedia.org/wiki/John_Henry_Holland (Ağustos, **2018**).

[30] R. R. Martin, An Overview of Genetic Algorithms, http://mat.uab.cat/~alseda/MasterOpt/Beasley93GA1.pdf (Ağustos, **2018**).

[31] R. R. Hill, G. A. Mcintyre, *Genetic Algorithms for Model Optimization*, **2000**.

[32] Anonim, Genetic algorithm, https://webcache.googleusercontent.com/search?q=cache:BEv5N4rZj90J:https://en.wikipedia.org/wiki/Genetic_algorithm+&cd=2&hl=tr&ct=clnk&gl=tr (Ağustos, **2018**).

[33] H. Sthamer, P. Morgannwg, *The Automatic Generation of Software Test Data Using Genetic Algorithms*, no. November, **1995**.

[34] V. Mallawaarachchi, Introduction to Genetic Algorithms, https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 (Ağustos, **2018**).

[35] C. Miller, *Battery Firmware Hacking*, System, **2011**.

[36] Anonim, UI/Application Exerciser Monkey,

https://developer.android.com/studio/test/monkey.html (Ağustos, **2018**).

[37] A. Machiry, R. Tahiliani, M. Naik, *Dynodroid: an input generation system for Android apps*, Proc. 2013 9th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2013, p. 224, **2013**.

[38] R. Mahmood, N. Mirzaei, S. Malek, *EvoDroid: segmented evolutionary testing of Android apps*, Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. - FSE 2014, pp. 599–609, **2014**.

[39] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De, U. Federico, and I. I. Napoli, *Using GUI Ripping for Automated Testing of Android Applications*, pp. 258–261, **2012**.

[40] Yang, Wei, Mukul R. Prasad, Tao Xie, *A grey-box approach for automated GUI-model generation of mobile applications*, International Conference on Fundamental Approaches to Software Engineering. Springer, Berlin, Heidelberg, **2013**.

[41] Azim, Tanzirul, Iulian Neamtiu, *Targeted and depth-first exploration for systematic testing of android apps*, Acm Sigplan Notices. Vol. 48. No. 10. ACM, **2013**.

[42] Choi, Wontae, George Necula, Koushik Sen, *Guided gui testing of android apps with minimal restart and approximate learning*, Acm Sigplan Notices. Vol. 48. No. 10. ACM, **2013**.

[43] Hao, Shuai, *PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps,* Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, **2014**.

[44] Anand, Saswat, *Automated concolic testing of smartphone apps*, Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, **2012**.

[45] Y. Liu, C. Xu, S. C. Cheung, *Verifying Android Applications Using Java PathFinder State Key Laboratory for Novel Software Technology Table of Contents*, **2012**.

[46] Alimi, Vincent, Sylvain Vernois, C. Rosenberger, *Analysis of embedded applications by evolutionary fuzzing*, High Performance Computing & Simulation (HPCS), 2014 International Conference on. IEEE, **2014**.

[47] Vernois, Sylvain, V. Alimi, *WinSCard Tools: a software for the development and security analysis of transactions with smartcards*, The third Norsk Information Security Conference (NISK), **2010**.

[48] Anonim, JCOP Simulator Proxy Tool, https://sites.google.com/site/bondhannovandy/Home/smart-card/jcop-simulator-proxy-tool (Ağustos, **2018**).

[49] G. H. Liu, G. Wu, T. Zheng, J. M. Shuai, Z. C. Tang, *Vulnerability analysis for X86 executables using Genetic Algorithm and Fuzzing*, Proc. - 3rd Int. Conf. Converg. Hybrid Inf. Technol. ICCIT 2008, vol. 2, no. 1, pp. 491–497, **2008**.

[50] DataRescue, IDA Pro disassemble, http://www.datarescue.com/idabase (Ağustos, **2018**).

[51] C. Eagle, *The IDA PRO Book, 2nd Edition*, **2011**.

[52] C.-K. Luk, *Pin: building customized program analysis tools with dynamic instrumentation*, Proc. 2005 ACM SIGPLAN Conf. Program. Lang. Des. Implement. - PLDI ’05, vol. 40, no. 6, p. 190, **2005**.

[53] S. Sparks, S. Embleton, R. Cunningham, C. Zou, *Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting*, *Proc. - Annu. Comput. Secur. Appl. Conf. ACSAC*, pp. 477–486, **2007**.

[54] S. Sparks, Dynamic Markov Model, https://ieeexplore.ieee.org/document/4413013/ (Ağustos, **2018**).

[55] A. Pedram, *PaiMei - Reverse Engineering Framework*, RECON Conference, **2016**.

[56] R. S. Jr, *A Framework for File Format Fuzzing with Genetic Algorithms*, **2012**.

[57] R. Mohsen, *Automated Vulnerability Analysis using Advanced Fuzzing : Generation Based and Evolutionary Fuzzers*, Security, no. October, pp. 1–108, **2016**.

[58] Infigo, Ftpstress, https://www.ee.oulu.fi/roles/ouspg/FTPStress (Ağustos, **2018**).

[59] F. Beterke, *Distributed Evolutionary Fuzzing with Evofuzz*, pp. 23–32, **2016**.

[60] F. Duchene, S. Rawat, J.-L. Richier, R. Groz, *KameleonFuzz: Evolutionary Fuzzing for Black-box XSS Detection,* Proc. 4th ACM Conf. Data Appl. Secur. Priv., vol. 1, pp. 37–48, **2014**.

[61] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, A. Stavrou, *A whitebox approach for automated security testing of Android applications on the cloud*, Ast, pp. 22–28, **2012**.

[62] S. Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei, A. Stavrou, *A framework for automated security testing of android applications on the cloud,* Proc. 2012 IEEE 6th Int. Conf. Softw. Secur. Reliab. Companion, SERE-C 2012, pp. 35–36, **2012**.

[63] Anonim, Elasticsearch, https://www.elastic.co/ (Ağustos, **2018**).

[64] J. Wu, Y. Wu, M. Yang, Z. Wu, Y. Wang, *Vulnerability Detection of Android System in Fuzzing Cloud*, 2013 IEEE Sixth Int. Conf. Cloud Comput., pp. 954–955, **2013**.

[65] Iannillo, Antonio Ken, *Chizpurfle: A gray-box android fuzzer for vendor service customizations*, Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on. IEEE, **2017**.

[66] T. Gu, *AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications*, 2017 IEEE Int. Conf. Softw. Maint. Evol., pp. 103–114, **2017**.

[67] Anonim, Activity, https://developer.android.com/reference/android/app/Activity.html (Ağustos, **2018**).

[68] Anonim, Method, https://developer.android.com/reference/java/lang/reflect/Method.html (Ağustos, **2018**).

[69] I. Costin, *BIFUZ – Broadcast Intent Fuzzing Framework for Android*, **2015**.

[70] H. Ye, S. Cheng, L. Zhang, and F. Jiang, *DroidFuzzer*, Proc. Int. Conf. Adv. Mob. Comput. Multimed., pp. 68–74, **2013**.

[71] OWASP, XSS, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (Ağustos, **2018**).

[72] T. Javaid, *Testing of Android testing tools: development of a benchmark for the evaluation*, no. July, **2015**.

[73] Gonzalez-Sanchez, Alberto*, Prioritizing tests for fault localization through ambiguity group reduction*, Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, **2011**.

[74] S. R. Choudhary, A. Gorla, A. Orso, *Automated test input generation for android: Are we there yet?*, Proc. - 2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2015, pp. 429–440, **2016**.

[75] K. Mao, M. Harman, Y. Jia, *Sapienz: multi-objective automated testing for Android applications*, Proc. 25th Int. Symp. Softw. Test. Anal. - ISSTA 2016, pp. 94–105, **2016**.

[76] AntoJoseph, *Droid-FF – the Android Fuzzing Framework*, **2016**.

[77] S. Hocevar, zzuf - Multiple Purpose Fuzzer, http://caca.zoy.org/wiki/zzuf (Ağustos, **2018**).

[78] S. Kyle, H. Leather, B. Franke, D. Butcher, S. Monteith, *Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing,* Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Exec. Environ. - VEE ’15, pp. 121–132, **2015**.

[79] Google, dexfuzz, https://android.googlesource.com/platform/art/+/master/tools/dexfuzz/README (Ağustos, **2018**).

[80] L. Jordan, P. Greyling, *Android Projects*. **2011**.

[81] Anonim, AOSP, https://source.android.com/ (Ağustos, **2018**).

[82] A. Bechtsoudis, *Fuzzing Objects d ’ ART Digging Into the New Android L Runtime Internals*, **2015**.

[83] Anonim, Android Emulator, https://developer.android.com/studio/run/emulator (Ağustos, **2018**).

[84] A. Bechtsoudis, dexRepair, https://github.com/anestisb/dexRepair (Ağustos, **2018**).

[85] C. Sheppard, genetic.py, https://github.com/handcraftsman/GeneticAlgorithmsWithPython/blob/master/ch01/genetic.py (Ağustos, **2018**).

8. EKLER

8.1. Ek 1

```
Muh4f1Z-MacBook-Air:droid-ff muh4f1z$ more
/Users/muh4f1z/GAdroid/confirmed_crashes/sample0-137960-275921.dex
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint:
'Android/sdk_phone_armv7/generic:5.1.1/LMY48X/3079158:userdebug/te
st-keys'
Revision: '0'
ABI: 'arm'
pid: 2621, tid: 2621, name: dexdump  >>> /system/xbin/dexdump <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0xb6c00000
    r0 00008651  r1 00000000  r2 ff778564  r3 b6c0000c
    r4 05ce5d54  r5 b6c00a9c  r6 b6c00a6c  r7 80078071
    r8 000015af  r9 00010ca2  sl 00008651  fp 00000000
    ip 00000000  sp be8b4960  lr b6fa0d2b  pc b6ebb10c  cpsr
80000010
    d0  0000000000000000  d1  0000000000000000
    d2  0000000000000000  d3  0000000000000000
    d4  0000000000000000  d5  0000000000000000
    d6  0000000000000000  d7  0000000000000000
    d8  0000000000000000  d9  0000000000000000
    d10 0000000000000000  d11 0000000000000000
    d12 0000000000000000  d13 0000000000000000
    d14 0000000000000000  d15 0000000000000000
    d16 0000000000000000  d17 0000000000000000
    d18 0000000000000000  d19 0000000000000000
    d20 0000000000000000  d21 0000000000000000
    d22 0000000000000000  d23 0000000000000000
    d24 0000000000000000  d25 0000000000000000
    d26 0000000000000000  d27 0000000000000000
    d28 0000000000000000  d29 0000000000000000
    d30 0000000000000000  d31 0000000000000000
    scr 00000000

backtrace:
    #00 pc 0000210c  /system/lib/libz.so (adler32+196)
    #01 pc 00006d27  /system/xbin/dexdump
    #02 pc 00003d57  /system/xbin/dexdump
    #03 pc 000039b5  /system/xbin/dexdump
    #04 pc 000019d7  /system/xbin/dexdump
    #05 pc 00012dc9  /system/lib/libc.so (__libc_init+44)
```

```
    #06 pc 00001aa4  /system/xbin/dexdump

stack:
         be8b4920  b6f9ba44  /system/xbin/dexdump
         be8b4924  b6f9a000  /system/xbin/dexdump
         be8b4928  be8b499c  [stack]
         be8b492c  b6f8dae7  /system/bin/linker
(__dl___linker_init+1650)
         be8b4930  00000000
         be8b4934  00000000
         be8b4938  00000000
         be8b493c  b6f8502c  [anon:linker_alloc]
         be8b4940  b6f86154  [anon:linker_alloc]
         be8b4944  b6f862a4  [anon:linker_alloc]
         be8b4948  b6f863f4  [anon:linker_alloc]
         be8b494c  b6f86544  [anon:linker_alloc]
         be8b4950  b6f86694  [anon:linker_alloc]
         be8b4954  b6f867e4  [anon:linker_alloc]
         be8b4958  b6f85024  [anon:linker_alloc]
         be8b495c  41cdd377
    #00  be8b4960  b6379000  /data/local/tmp/sample0-137960-
275921.dex
         be8b4964  00086ba3
         be8b4968  14f42474
         be8b496c  be8b4990  [stack]
         be8b4970  00000000
         be8b4974  00000000
         be8b4978  b6f7bd94
         be8b497c  be8b49bc  [stack]
    #01  be8b4980  00086ba3
         be8b4984  b6f9873c  /system/bin/linker
         be8b4988  00000000
         be8b498c  00000000
         be8b4990  be8b4970  [stack]
         be8b4994  b6f980e4  /system/bin/linker
         be8b4998  00000000
         be8b499c  00000000
         be8b49a0  00000000
         be8b49a4  00000002
         be8b49a8  be8b4b34  [stack]
         be8b49ac  be8b4b40  [stack]
         be8b49b0  be8b4b7c  [stack]
         be8b49b4  b6f98510  /system/bin/linker
         be8b49b8  00000000
         be8b49bc  00000000
         ........  ........
    #02  be8b4a38  00000002
         be8b4a3c  41cdd377
         be8b4a40  be8b4b34  [stack]
         be8b4a44  b6f7bd94
         be8b4a48  be8b4a8c  [stack]
         be8b4a4c  b6fa40d6  /system/xbin/dexdump
         be8b4a50  b6f7a400  /system/lib/libc.so
         be8b4a54  be8b4a8c  [stack]
```

```
         be8b4a58  0000003a
         be8b4a5c  b6f791d4  /system/lib/libc.so
         be8b4a60  b6fa42e4  /system/xbin/dexdump
         be8b4a64  b6f44413  /system/lib/libc.so (vfprintf+30)
         be8b4a68  00000002
         be8b4a6c  41cdd377
         be8b4a70  be8b4c36  [stack]
         be8b4a74  00000002
         ........  ........
    #03  be8b4a98  b6379000  /data/local/tmp/sample0-137960-
275921.dex
         be8b4a9c  00086ba3
         be8b4aa0  b6379000  /data/local/tmp/sample0-137960-
275921.dex
         be8b4aa4  00086ba3
         be8b4aa8  00000000
         be8b4aac  00000002
         be8b4ab0  00000000
         be8b4ab4  be8b4b34  [stack]
         be8b4ab8  b6f791d4  /system/lib/libc.so
         be8b4abc  b6fa42e4  /system/xbin/dexdump
         be8b4ac0  00000000
         be8b4ac4  b6f9b9db  /system/xbin/dexdump
    #04  be8b4ac8  0000000c
         be8b4acc  00000000
         be8b4ad0  b6fa9424  /system/xbin/dexdump
         be8b4ad4  b6fa9424  /system/xbin/dexdump
         be8b4ad8  b6f234d1  /system/lib/libc.so (__libc_fini)
         be8b4adc  be8b4b34  [stack]
         be8b4ae0  be8b4b40  [stack]
         be8b4ae4  00000002
         be8b4ae8  b6f9b87d  /system/xbin/dexdump
         be8b4aec  00000000
         be8b4af0  00000000
         be8b4af4  00000000
         be8b4af8  be8b4b2c  [stack]
         be8b4afc  b6f21dcb  /system/lib/libc.so (__libc_init+46)
    #05  be8b4b00  be8b4b18  [stack]
         be8b4b04  00000000
         be8b4b08  00000000
         be8b4b0c  00000000
         be8b4b10  00000000
         be8b4b14  b6f9baa8  /system/xbin/dexdump
    #06  be8b4b18  b6fa8dc4  /system/xbin/dexdump
         be8b4b1c  b6fa8dcc  /system/xbin/dexdump
         be8b4b20  b6fa8dd4  /system/xbin/dexdump
         be8b4b24  be8b4b30  [stack]
         be8b4b28  00000000
         be8b4b2c  b6f8b123  /system/bin/linker
(__dl__ZN6soinfoD2Ev+20)
         be8b4b30  00000002
         be8b4b34  be8b4c21  [stack]
         be8b4b38  be8b4c36  [stack]
         be8b4b3c  00000000
```

```
        be8b4b40  be8b4c60  [stack]
        be8b4b44  be8b4c77  [stack]
        be8b4b48  be8b4c8a  [stack]
        be8b4b4c  be8b4ca3  [stack]
        be8b4b50  be8b4cb8  [stack]
        be8b4b54  be8b4ccb  [stack]

memory near r0:
    00008630 -------- -------- -------- --------
    00008640 -------- -------- -------- --------
    00008650 -------- -------- -------- --------
    00008660 -------- -------- -------- --------
    00008670 -------- -------- -------- --------
    00008680 -------- -------- -------- --------
    00008690 -------- -------- -------- --------
    000086a0 -------- -------- -------- --------
    000086b0 -------- -------- -------- --------
    000086c0 -------- -------- -------- --------
    000086d0 -------- -------- -------- --------
    000086e0 -------- -------- -------- --------
    000086f0 -------- -------- -------- --------
    00008700 -------- -------- -------- --------
    00008710 -------- -------- -------- --------
    00008720 -------- -------- -------- --------

memory near r3:
    b6bfffec 00000000 00000000 00000000 00000000
    b6bffffc 00000000 -------- -------- --------
    b6c0000c -------- -------- -------- --------
    b6c0001c -------- -------- -------- --------
    b6c0002c -------- -------- -------- --------
    b6c0003c -------- -------- -------- --------
    b6c0004c -------- -------- -------- --------
    b6c0005c -------- -------- -------- --------
    b6c0006c -------- -------- -------- --------
    b6c0007c -------- -------- -------- --------
    b6c0008c -------- -------- -------- --------
    b6c0009c -------- -------- -------- --------
    b6c000ac -------- -------- -------- --------
    b6c000bc -------- -------- -------- --------
    b6c000cc -------- -------- -------- --------
    b6c000dc -------- -------- -------- --------

memory near r4:
    05ce5d34 -------- -------- -------- --------
    05ce5d44 -------- -------- -------- --------
    05ce5d54 -------- -------- -------- --------
    05ce5d64 -------- -------- -------- --------
    05ce5d74 -------- -------- -------- --------
    05ce5d84 -------- -------- -------- --------
    05ce5d94 -------- -------- -------- --------
    05ce5da4 -------- -------- -------- --------
    05ce5db4 -------- -------- -------- --------
    05ce5dc4 -------- -------- -------- --------
```

```
    05ce5dd4 -------- -------- -------- --------
    05ce5de4 -------- -------- -------- --------
    05ce5df4 -------- -------- -------- --------
    05ce5e04 -------- -------- -------- --------
    05ce5e14 -------- -------- -------- --------
    05ce5e24 -------- -------- -------- --------

memory near r5:
    b6c00a7c -------- -------- -------- --------
    b6c00a8c -------- -------- -------- --------
    b6c00a9c -------- -------- -------- --------
    b6c00aac -------- -------- -------- --------
    b6c00abc -------- -------- -------- --------
    b6c00acc -------- -------- -------- --------
    b6c00adc -------- -------- -------- --------
    b6c00aec -------- -------- -------- --------
    b6c00afc -------- -------- -------- --------
    b6c00b0c -------- -------- -------- --------
    b6c00b1c -------- -------- -------- --------
    b6c00b2c -------- -------- -------- --------
    b6c00b3c -------- -------- -------- --------
    b6c00b4c -------- -------- -------- --------
    b6c00b5c -------- -------- -------- --------
    b6c00b6c -------- -------- -------- --------

memory near r6:
    b6c00a4c -------- -------- -------- --------
    b6c00a5c -------- -------- -------- --------
    b6c00a6c -------- -------- -------- --------
    b6c00a7c -------- -------- -------- --------
    b6c00a8c -------- -------- -------- --------
    b6c00a9c -------- -------- -------- --------
    b6c00aac -------- -------- -------- --------
    b6c00abc -------- -------- -------- --------
    b6c00acc -------- -------- -------- --------
    b6c00adc -------- -------- -------- --------
    b6c00aec -------- -------- -------- --------
    b6c00afc -------- -------- -------- --------
    b6c00b0c -------- -------- -------- --------
    b6c00b1c -------- -------- -------- --------
    b6c00b2c -------- -------- -------- --------
    b6c00b3c -------- -------- -------- --------

memory near r7:
    80078050 -------- -------- -------- --------
    80078060 -------- -------- -------- --------
    80078070 -------- -------- -------- --------
    80078080 -------- -------- -------- --------
    80078090 -------- -------- -------- --------
    800780a0 -------- -------- -------- --------
    800780b0 -------- -------- -------- --------
    800780c0 -------- -------- -------- --------
    800780d0 -------- -------- -------- --------
    800780e0 -------- -------- -------- --------
```

```
    800780f0 -------- -------- -------- --------
    80078100 -------- -------- -------- --------
    80078110 -------- -------- -------- --------
    80078120 -------- -------- -------- --------
    80078130 -------- -------- -------- --------
    80078140 -------- -------- -------- --------

memory near r8:
    0000158c -------- -------- -------- --------
    0000159c -------- -------- -------- --------
    000015ac -------- -------- -------- --------
    000015bc -------- -------- -------- --------
    000015cc -------- -------- -------- --------
    000015dc -------- -------- -------- --------
    000015ec -------- -------- -------- --------
    000015fc -------- -------- -------- --------
    0000160c -------- -------- -------- --------
    0000161c -------- -------- -------- --------
    0000162c -------- -------- -------- --------
    0000163c -------- -------- -------- --------
    0000164c -------- -------- -------- --------
    0000165c -------- -------- -------- --------
    0000166c -------- -------- -------- --------
    0000167c -------- -------- -------- --------

memory near r9:
    00010c80 -------- -------- -------- --------
    00010c90 -------- -------- -------- --------
    00010ca0 -------- -------- -------- --------
    00010cb0 -------- -------- -------- --------
    00010cc0 -------- -------- -------- --------
    00010cd0 -------- -------- -------- --------
    00010ce0 -------- -------- -------- --------
    00010cf0 -------- -------- -------- --------
    00010d00 -------- -------- -------- --------
    00010d10 -------- -------- -------- --------
    00010d20 -------- -------- -------- --------
    00010d30 -------- -------- -------- --------
    00010d40 -------- -------- -------- --------
    00010d50 -------- -------- -------- --------
    00010d60 -------- -------- -------- --------
    00010d70 -------- -------- -------- --------

memory near sl:
    00008630 -------- -------- -------- --------
    00008640 -------- -------- -------- --------
    00008650 -------- -------- -------- --------
    00008660 -------- -------- -------- --------
    00008670 -------- -------- -------- --------
    00008680 -------- -------- -------- --------
    00008690 -------- -------- -------- --------
    000086a0 -------- -------- -------- --------
    000086b0 -------- -------- -------- --------
    000086c0 -------- -------- -------- --------
```

```
    000086d0 -------- -------- -------- --------
    000086e0 -------- -------- -------- --------
    000086f0 -------- -------- -------- --------
    00008700 -------- -------- -------- --------
    00008710 -------- -------- -------- --------
    00008720 -------- -------- -------- --------

memory near sp:
    be8b4940 b6f86154 b6f862a4 b6f863f4 b6f86544
    be8b4950 b6f86694 b6f867e4 b6f85024 41cdd377
    be8b4960 b6379000 00086ba3 14f42474 be8b4990
    be8b4970 00000000 00000000 b6f7bd94 be8b49bc
    be8b4980 00086ba3 b6f9873c 00000000 00000000
    be8b4990 be8b4970 b6f980e4 00000000 00000000
    be8b49a0 00000000 00000002 be8b4b34 be8b4b40
    be8b49b0 be8b4b7c b6f98510 00000000 00000000
    be8b49c0 00000000 00000000 00000000 00000000
    be8b49d0 00000000 00000000 00000000 be8b4a98
    be8b49e0 00000003 be8b4a98 00000000 b6f281e9
    be8b49f0 00000003 00000000 00000000 00000000
    be8b4a00 00000000 00000000 00086ba3 b6fa1cb1
    be8b4a10 00000003 00000000 b6fa8f14 be8b4a98
    be8b4a20 be8b4c36 00000000 00000000 b6f7bd94
    be8b4a30 00000003 b6f9dd5b 00000002 41cdd377

code around pc:
    b6ebb0ec e5d31000 e2833010 e553900f e0800001
    b6ebb0fc e553c00e e553b00d e080a009 e08a9000
    b6ebb10c e553100c e08a000c e080c00b e089a000
    b6ebb11c e553b00b e08c1001 e08a900c e553000a
    b6ebb12c e553c009 e089a001 e081b00b e5539008
    b6ebb13c e08a100b e08b0000 e080b00c e081a000
    b6ebb14c e5531007 e08a000b e08b9009 e553c006
    b6ebb15c e089b001 e080a009 e5530005 e08a900b
    b6ebb16c e5531004 e08bc00c e553a003 e089b00c
    b6ebb17c e08c0000 e08b9000 e080c001 e553b002
    b6ebb18c e08c100a e089000c e553a001 e0809001
    b6ebb19c e081c00b e089b00c e08c000a e1530005
    b6ebb1ac e08b1000 e0844001 1affffcc e0853097
    b6ebb1bc e1520008 e08a3497 e2863030 e1a067a5
    b6ebb1cc e0669606 e1a0b7aa e086c209 e06b160b
    b6ebb1dc e06c0000 e08b5201 e0654004 8affffbb

code around lr:
    b6fa0d08 44799500 f7fa447a 2000ed1a 46024601
    b6fa0d18 ed26f7fa f1046a22 68a6010c f7fa3a0c
    b6fa0d28 42b0ed20 d0084603 200649b3 44794ab3
    b6fa0d38 447a9600 ed02f7fa 1963e7c1 4bb0637b
    b6fa0d48 46212000 63f86438 0270f104 447b6478
    b6fa0d58 465864b8 63bd633c fdc4f7fd d0ae2800
    b6fa0d68 489c6aa3 42836ae2 6ea66b21 d09b6ee5
    b6fa0d78 200649a4 44794aa4 e026447a 19aa6b3b
    b6fa0d88 19594658 4ba1441a f7fd447b 2800fdab
    b6fa0d98 6a63d095 2b7062fc 499dd207 4a9d2570
```

```
    b6fa0da8 95002006 447a4479 d011e7c4 4a9a2170
    b6fa0db8 20059100 447a4999 f7fa4479 e007ecc0
    b6fa0dc8 20064997 44794a97 f7fa447a e776ecb8
    b6fa0dd8 627a6b62 6a7eb162 44266a78 623e4b7e
    b6fa0de8 58261d31 429e6afd 9068f8d5 e1e3d908
    b6fa0df8 2006498d 44794a8d f7fa447a e75eeca0

memory map: (fault address prefixed with --->)
    b6379000-b63fffff rw-   552960  /data/local/tmp/sample0-
137960-275921.dex
    b6400000-b6bfffff rw-  8388608  [anon:libc_malloc]
--->Fault address falls at b6c00000 between mapped regions
    b6de3000-b6de3fff r-x     4096  /system/lib/libnetd_client.so
    b6de4000-b6de4fff ---     4096
    b6de5000-b6de5fff r--     4096  /system/lib/libnetd_client.so
    b6de6000-b6de6fff rw-     4096  /system/lib/libnetd_client.so
    b6de7000-b6e06fff r--   131072  /dev/__properties__
    b6e07000-b6e07fff r--     4096  [anon:linker_alloc]
    b6e08000-b6e09fff r-x     8192  /system/lib/libunwind-
ptrace.so
    b6e0a000-b6e0afff r--     4096  /system/lib/libunwind-
ptrace.so
    b6e0b000-b6e0bfff rw-     4096  /system/lib/libunwind-
ptrace.so
    b6e0c000-b6e17fff r-x    49152  /system/lib/libunwind.so
    b6e18000-b6e18fff r--     4096  /system/lib/libunwind.so
    b6e19000-b6e19fff rw-     4096  /system/lib/libunwind.so
    b6e1a000-b6e5ffff rw-   286720
    b6e60000-b6e63fff r-x    16384  /system/lib/libgccdemangle.so
    b6e64000-b6e64fff ---     4096
    b6e65000-b6e65fff r--     4096  /system/lib/libgccdemangle.so
    b6e66000-b6e66fff rw-     4096  /system/lib/libgccdemangle.so
    b6e67000-b6e9dfff r-x   225280  /system/lib/libstlport.so
    b6e9e000-b6ea0fff r--    12288  /system/lib/libstlport.so
    b6ea1000-b6ea1fff rw-     4096  /system/lib/libstlport.so
    b6ea2000-b6eacfff r-x    45056  /system/lib/libcutils.so
    b6ead000-b6eadfff r--     4096  /system/lib/libcutils.so
    b6eae000-b6eaefff rw-     4096  /system/lib/libcutils.so
    b6eaf000-b6eb4fff r-x    24576  /system/lib/libbacktrace.so
    b6eb5000-b6eb5fff ---     4096
    b6eb6000-b6eb6fff r--     4096  /system/lib/libbacktrace.so
    b6eb7000-b6eb7fff rw-     4096  /system/lib/libbacktrace.so
    b6eb8000-b6eb8fff r--     4096
    b6eb9000-b6ecffff r-x    94208  /system/lib/libz.so
    b6ed0000-b6ed0fff ---     4096
    b6ed1000-b6ed1fff r--     4096  /system/lib/libz.so
    b6ed2000-b6ed2fff rw-     4096  /system/lib/libz.so
    b6ed3000-b6ee7fff r-x    86016  /system/lib/libutils.so
    b6ee8000-b6ee8fff ---     4096
    b6ee9000-b6ee9fff r--     4096  /system/lib/libutils.so
    b6eea000-b6eeafff rw-     4096  /system/lib/libutils.so
    b6eeb000-b6eecfff r-x     8192  /system/lib/libstdc++.so
    b6eed000-b6eedfff r--     4096  /system/lib/libstdc++.so
    b6eee000-b6eeefff rw-     4096  /system/lib/libstdc++.so
```

```
    b6eef000-b6f05fff r-x    94208  /system/lib/libm.so
    b6f06000-b6f06fff r--     4096  /system/lib/libm.so
    b6f07000-b6f07fff rw-     4096  /system/lib/libm.so
    b6f08000-b6f0cfff r-x    20480  /system/lib/liblog.so
    b6f0d000-b6f0dfff r--     4096  /system/lib/liblog.so
    b6f0e000-b6f0efff rw-     4096  /system/lib/liblog.so
    b6f0f000-b6f73fff r-x   413696  /system/lib/libc.so
    b6f74000-b6f74fff ---     4096
    b6f75000-b6f77fff r--    12288  /system/lib/libc.so
    b6f78000-b6f7afff rw-    12288  /system/lib/libc.so
    b6f7b000-b6f83fff rw-    36864
    b6f84000-b6f84fff r--     4096  [anon:linker_alloc]
    b6f85000-b6f85fff rw-     4096  [anon:linker_alloc]
    b6f86000-b6f86fff r--     4096  [anon:linker_alloc]
    b6f87000-b6f87fff r--     4096
    b6f88000-b6f89fff rw-     8192
    b6f8a000-b6f96fff r-x    53248  /system/bin/linker
    b6f97000-b6f97fff r--     4096  /system/bin/linker
    b6f98000-b6f98fff rw-     4096  /system/bin/linker
    b6f99000-b6f99fff rw-     4096
    b6f9a000-b6fa7fff r-x    57344  /system/xbin/dexdump
    b6fa8000-b6fa8fff r--     4096  /system/xbin/dexdump
    b6fa9000-b6fa9fff rw-     4096  /system/xbin/dexdump
    be894000-be8b4fff rw-   135168  [stack]
    ffff0000-ffff0fff r-x     4096  [vectors]
--------- tail end of log main
```

8.2. Ek 2

☆ 113754619 ▾ Security Report - Android Dexdump Buffer Overflow Vulnerability

Public Trackers ▸ Android External Security Reports

Veysel hataş <vhatas@gmail.com> #1

Reported issue.

Title : Android Dexdump Buffer Overflow Vulnerability
Discoverer: Veysel HATAS (vhatas@gmail.com)
Web page : wise.cs.hacettepe.edu.tr
Test: Nexus 4 Android 5.1.1
Status: Not Fixed
Severity : High

Discovered: 04 February 2018
Reported: 03 August 2018
Published: -

Description : dexdump contains a flaw that is triggered as user-supplied input is not properly sanitized when handling a specially crafted dex file. This bug is triggered in "/system/lib/libz.so" native library. This may allow a context-dependent attacker to corrupt memory and cause a denial of service or potentially execute arbitrary code.

tombstone_sample0-137960-275921.dex
19 KB Download

log.txt
20 KB View Download

ÖZGEÇMİŞ

Kimlik Bilgileri

Adı Soyadı : Veysel HATAŞ

Doğum Yeri : Kırıkkale

Medeni Hali : Evli

E-posta : vhatas@gmail.com

Adresi : Ovacık Mah. Hürriyet Cd. No:102/8 Merkez/Kırıkkale

Eğitim

Lisans : Ege Üniversitesi Elektrik Elektronik Mühendisliği

Yüksek Lisans : Hacettepe Üniversitesi Bilgisayar Mühendisliği

Yabancı Dil ve Düzeyi

İngilizce : Çok iyi

Deneyim Alanları

Tersine Mühendislik

Fuzz Testi

Sızma Testi

Zararlı Kod Analizi

OSCP, OSCE

Tezden Üretilmiş Projeler ve Bütçesi

Yok.

Tezden Üretilmiş Yayınlar

Hazırlık aşamasında.

Tezden Üretilmiş Tebliğ ve Poster Sunumla Katıldığı Toplantılar

Yok.

HACETTEPE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
YÜKSEK LİSANS/DOKTORA TEZ ÇALIŞMASI ORJİNALLİK RAPORU

HACETTEPE ÜNİVERSİTESİ
FEN BİLİMLER ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI BAŞKANLIĞI'NA

Tarih: 17/05/2018

Tez Başlığı / Konusu: Android Uygulama Kurulum Süreci için Verimli Bir Evrimsel Tabanlı Fuzz Testi

Yukarıda başlığı/konusu gösterilen tez çalışmamın a) Kapak sayfası, b) Giriş, c) Ana bölümler d) Sonuç kısımlarından oluşan toplam 73 sayfalık kısmına ilişkin, 17/05/2018 tarihinde ~~şahsım~~/tez danışmanım tarafından *Turnitin* adlı intihal tespit programından aşağıda belirtilen filtrelemeler uygulanarak alınmış olan orijinallik raporuna göre, tezimin benzerlik oranı % 2 'dir.

Uygulanan filtrelemeler:

1- Kaynakça hariç
2- Alıntılar hariç/~~dâhil~~
3- 5 kelimeden daha az örtüşme içeren metin kısımları hariç

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü Tez Çalışması Orijinallik Raporu Alınması ve Kullanılması Uygulama Esasları'nı inceledim ve bu Uygulama Esasları'nda belirtilen azami benzerlik oranlarına göre tez çalışmamın herhangi bir intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.

Gereğini saygılarımla arz ederim.

Tarih ve İmza
17.05.2018

Adı Soyadı:	Veysel HATAŞ
Öğrenci No:	N14120787
Anabilim Dalı:	Bilgisayar Mühendisliği
Programı:	Yüksek Lisans
Statüsü:	☑ Y.Lisans ☐ Doktora ☐ Bütünleşik Dr.

DANIŞMAN ONAYI

UYGUNDUR.

Doç. Dr. Sevil Şen Akagündüz

(Unvan, Ad Soyad, İmza)